WEBINAR

How Careem achieved running container workloads at scale using Amazon EKS and KEDA

Talal Shobaita

Sr Solutions Architect
AWS

Sami Shabaneh

Snr SRE Engineer
Careem

Agenda

Why are customers choosing Amazon EKS

Scaling with Kubernetes

Serving millions of users with Careem

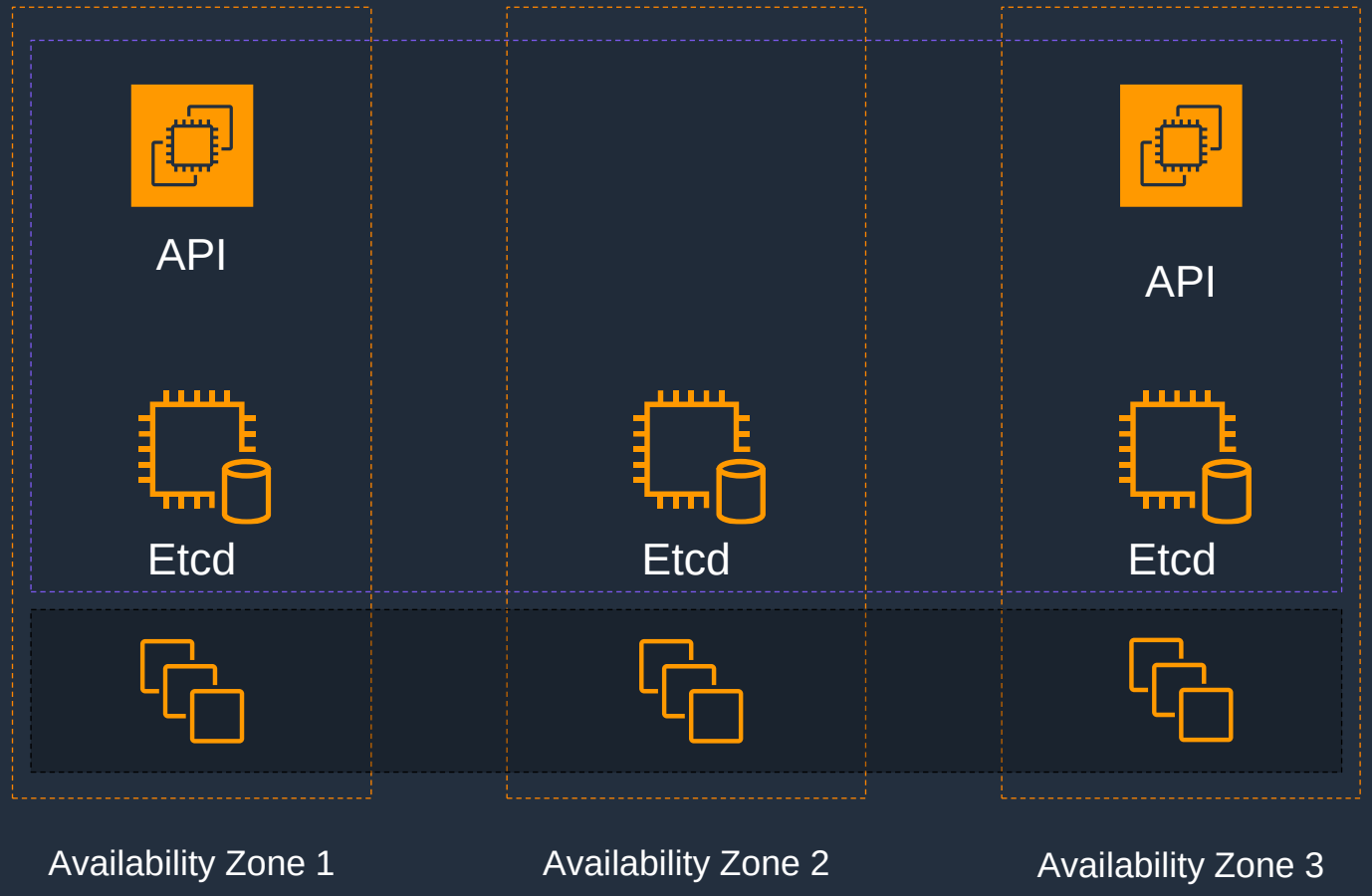
Careem technical implementation

Key takeaways

Why modernize with Kubernetes?

Ease	Declarative and self-reconciling Flexible and extensible
Consistency	Same API, regardless of where you run or at what scale
Ecosystem	Hundreds of solutions across the CNCF ecosystem
Community	De facto standard with numerous enterprises helping chart the future

Amazon EKS in the cloud



- Single tenant
- Highly available cluster API endpoint
- 99.95% SLA
- 24x7x365 support
- Instances scaled up/down seamlessly
- Upgrade and patching

- Focus on apps

Kubernetes with AWS

HOW YOU WANT IT, WHERE YOU NEED IT

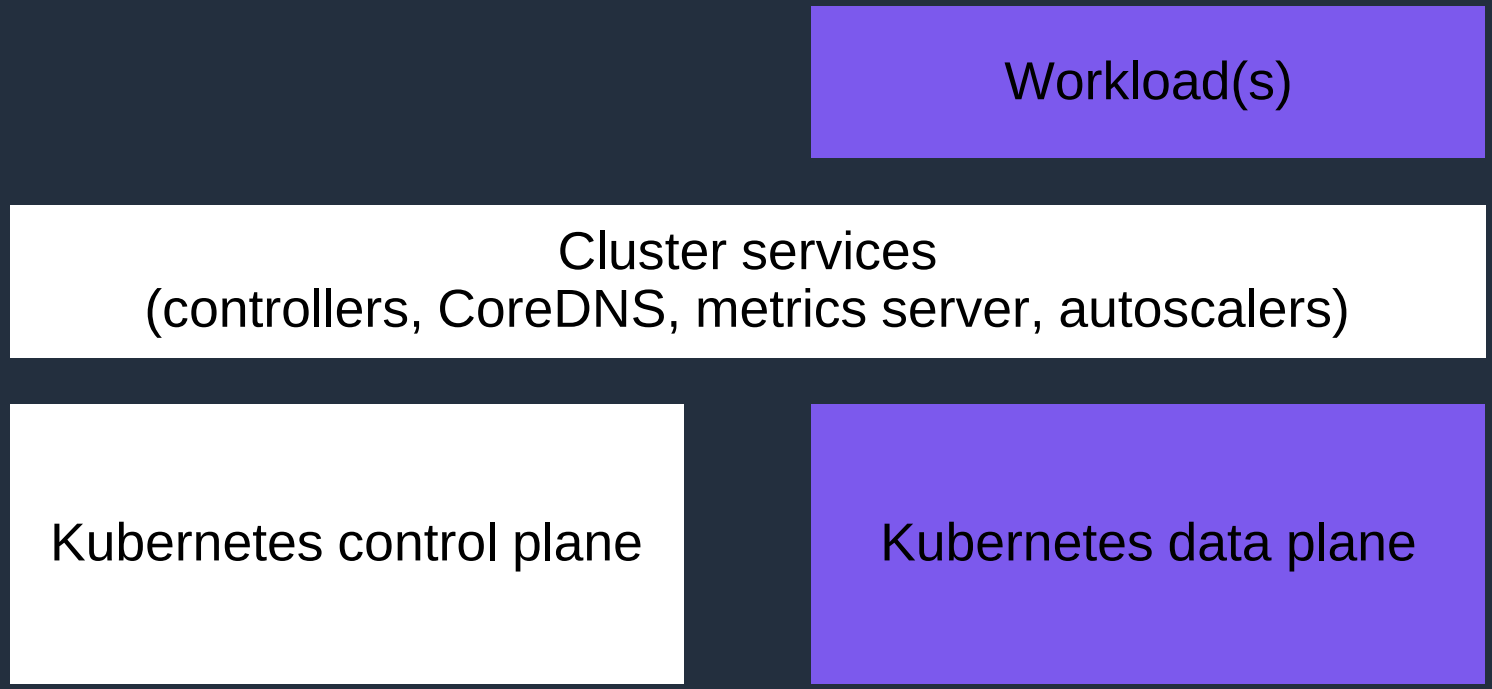
AWS is pushing the boundaries with AWS Outposts, AWS Wavelength, AWS Local Zones, and now on-premises, edge, and hybrid capabilities

Amazon EKS Anywhere

Amazon EKS

Kubernetes scaling mechanisms

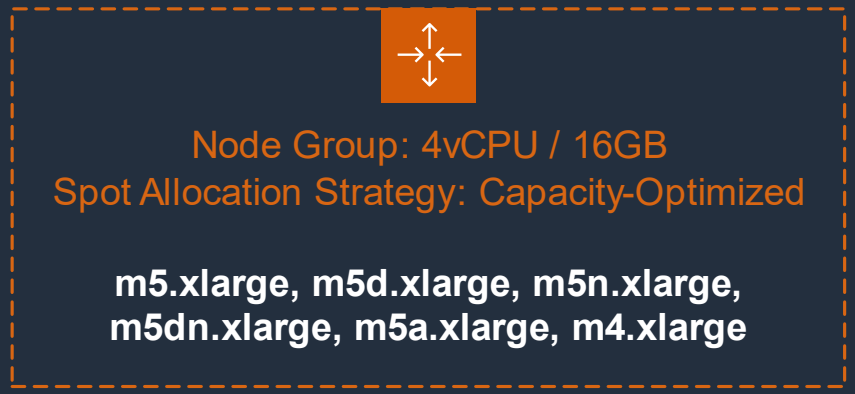
Amazon EKS scaling dimensions



Kubernetes data plane scaling mechanisms

Managed node groups or Karpenter is recommended for large-scale clusters

+



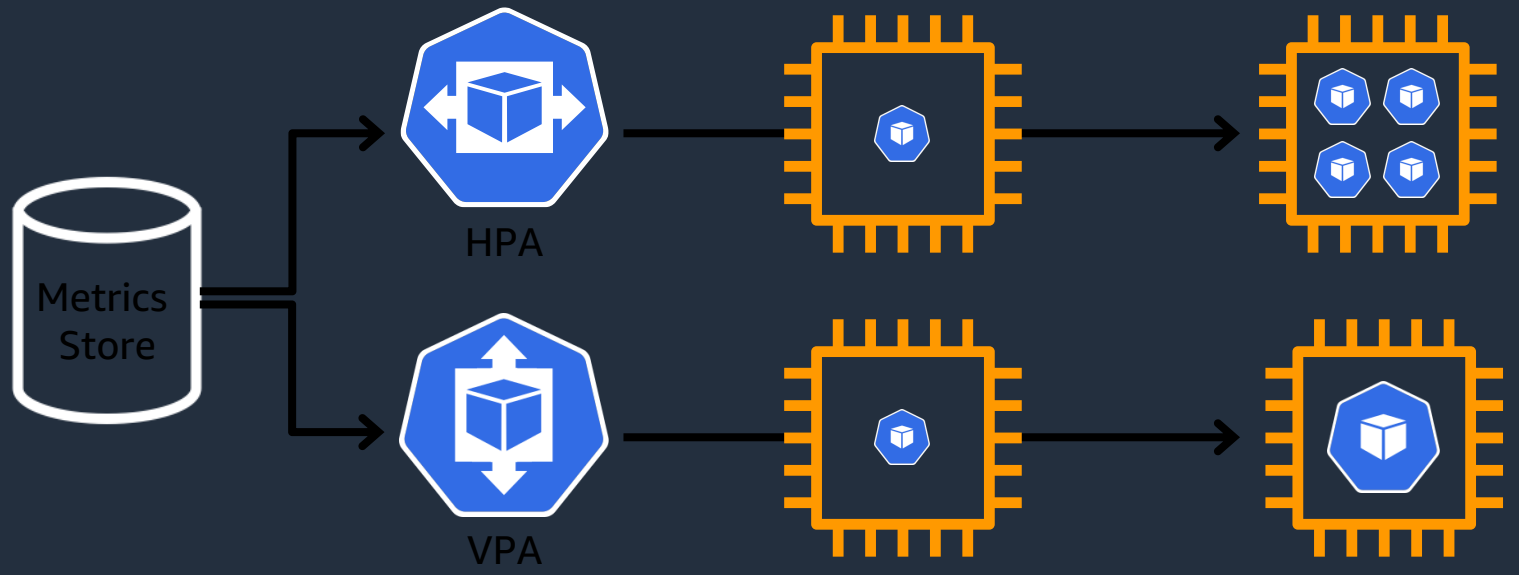
Managed node groups with Kubernetes Cluster Autoscaler

Karpenter

Kubernetes pod scaling

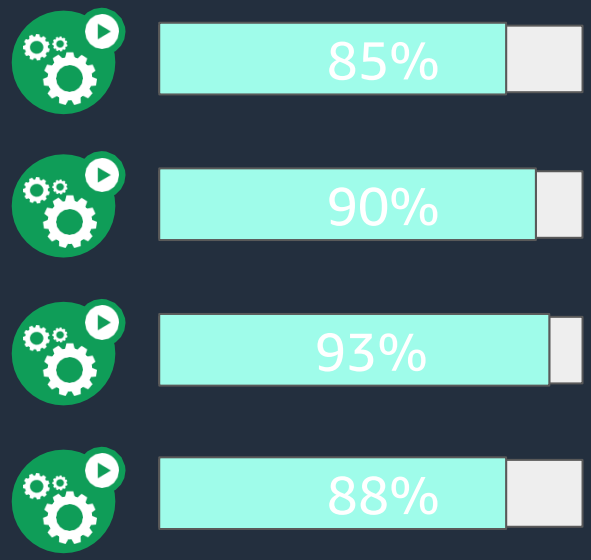
Kubernetes Workload Autoscaling

1. Horizontal Pod Autoscaling (HPA)
2. Vertical Pod Autoscaling (VPA)



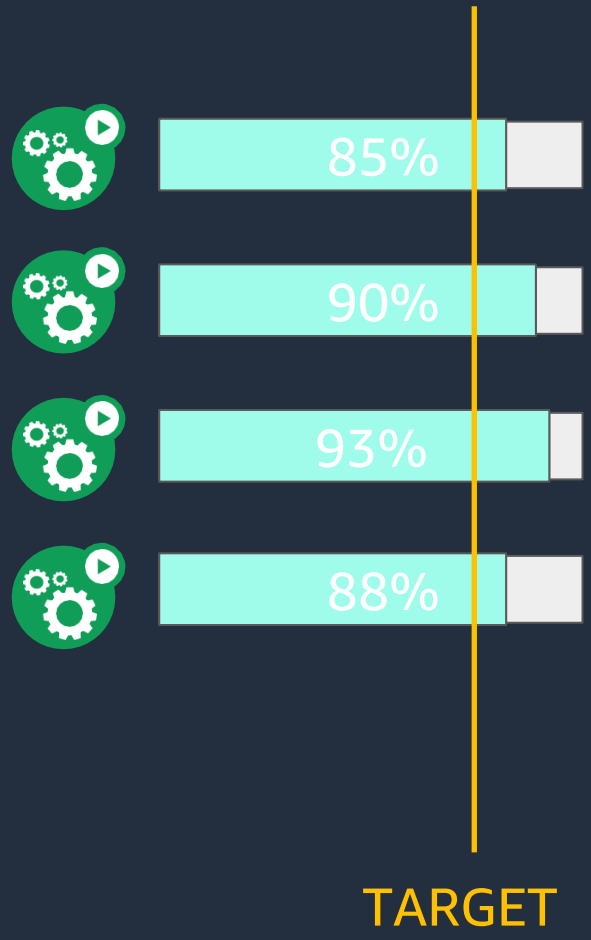
Scaling workloads with HPA

- If pods are heavily loaded, then starting new pods may bring average load down
- If pods are barely loaded, then stopping pods will free resources



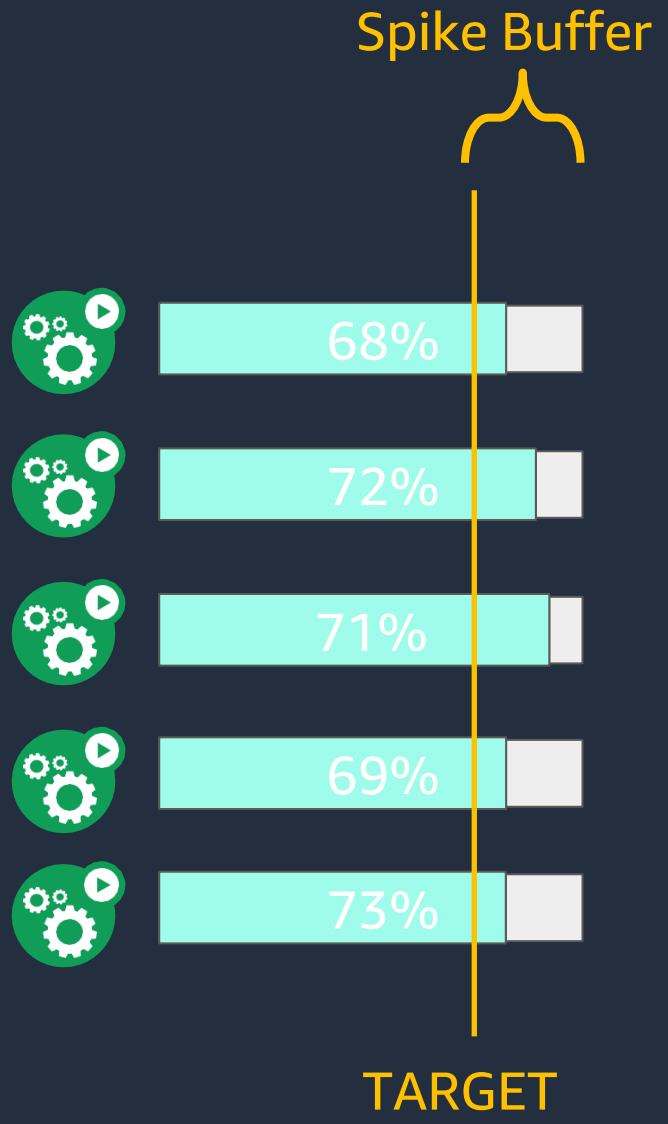
Scaling workloads with HPA

- If pods are heavily loaded, then starting new pods may bring average load down
- If pods are barely loaded, then stopping pods will free resources
- Specify the target for the load
  - e.g. target = cpu utilization 70%



Scaling workloads with HPA

- If pods are heavily loaded, then starting new pods may bring average load down
- If pods are barely loaded, then stopping pods will free resources
- Specify the target for the load
  - e.g. target = cpu utilization 70%
  - Too small spike buffer may overload your replicas
  - Too big buffer causes resource waste



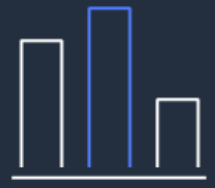
KEDA (Kubernetes-based Event driven Autoscaler)

- Works alongside with HPA to support event-driven scale
- HPA is CPU based. Using Custom Metrics is available but complicated
- Keda does not scale a cluster. It auto scales in/out Kubernetes deployments
- KEDA acts as a Metrics server that exposes rich event data

8 best practices to consider

Workload

Right metrics



Set CPU and memory

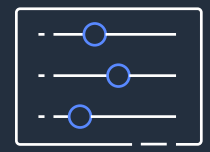
Pod

Spread constraints

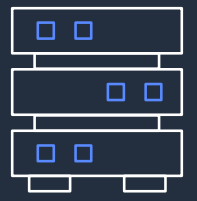
Set disruption budgets

Node

Broad on compute type



Similar node = consistent performance



Consider fewer, bigger nodes

Automate node updates

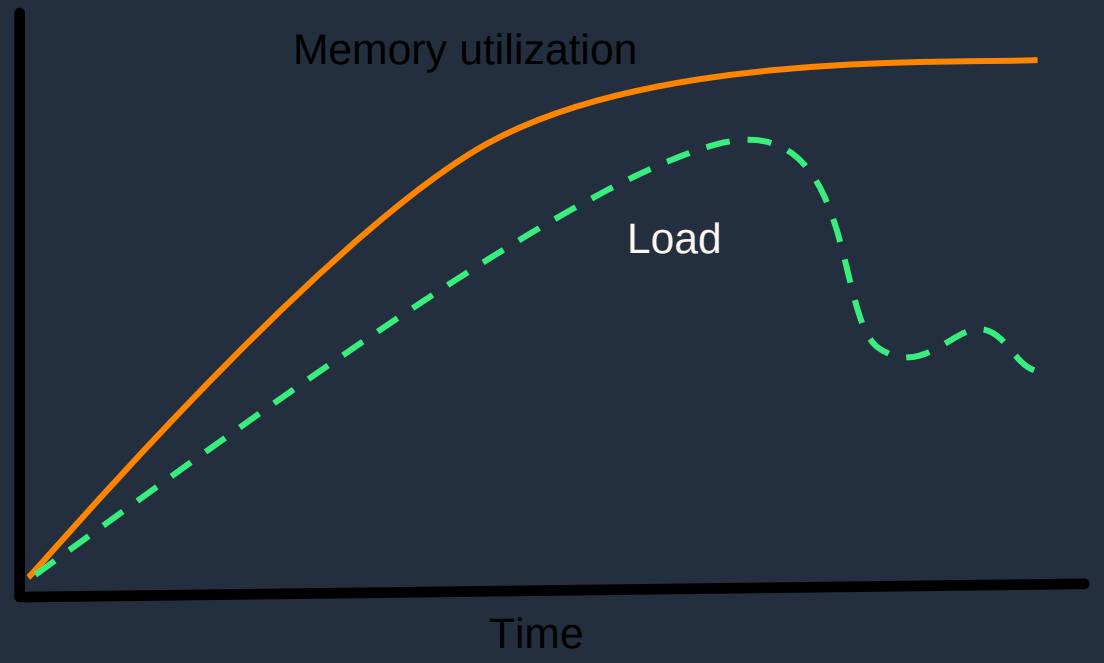
Choose appropriate scaling metrics when using HPA

It should be a **proportional** metric

- Average CPU utilization
- Average queue depth

Generally not as good metrics

- Average response time
- Memory utilization
- Max (*N*)
- P-values (p95, p99, etc.)



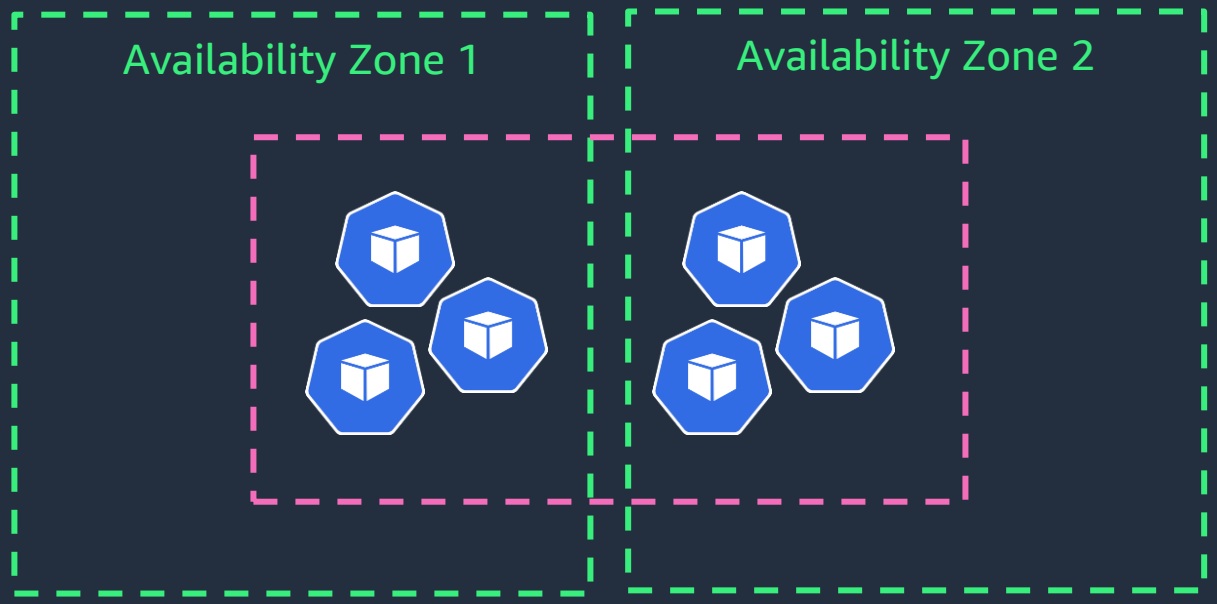
Configure and resource requests and limits for workloads

Non-compressible resources
during exhaustion are terminated (e.g., OOM)

Compressible resources
during contention things work more slowly

```yaml
apiVersion: v1
kind: Pod
metadata:
name: resources-pod
spec:
  containers:
    - name: container
      image: <image>
      resources:
        limits:
          memory: "200Mi"
        requests:
          cpu: 500m
          memory: "200Mi"
```

Use Kubernetes pod topology spread constraints or pod anti-affinities




```
topologyKey: "topology.kubernetes.io/zone” | "kubernetes.io/hostname”
             karpenter.sh/capacity-type
```

Use pod disruption budgets and consider pod readiness gates

Control pod termination during **voluntary disruptions**

Pod readiness gates to avoid timeouts with AWS Load Balancer Controller during target registration

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: my-app
spec:
  minAvailable: 20
  selector:
    matchLabels:
      app: my-scaled-app
```

Avoid overly constraining instance type selection and compute purchase options

Attribute based approach over instance configuration

```
…
managedNodeGroups:
  - name: my-managed-node-group
    desiredCapacity: 2
    spot: true
    instanceSelector:
        vCPUs: 2
        memory: 2GiB
        cpuArchitecture: x86_64
```

Karpenter Provisioner

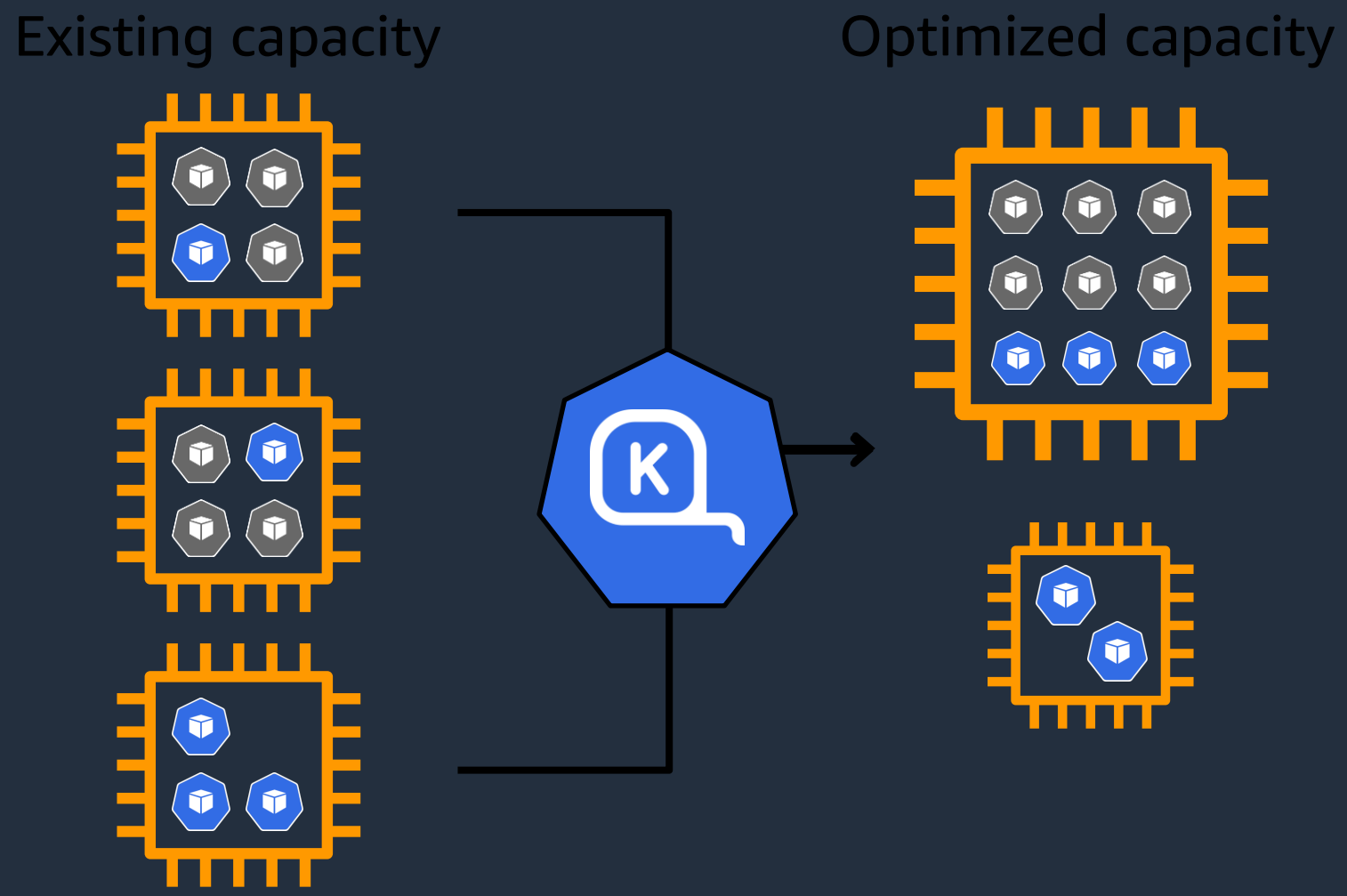
```
…
requirements:
  - key: karpenter.sh/capacity-type
    operator: In
    values: ["on-demand","spot"]
  - key: kubernetes.io/arch
    operator: In
    values: ["amd64","arm64"]
```

Use similar nodes for consistent compute performance

```
kind: deployment
...
spec:
  containers:
    nodeSelector:
      karpenter.k8s.aws/instance-size: 8xlarge
```

```
spec:
  affinity:
   nodeAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
     nodeSelectorTerms:
     - matchExpressions:
       - key: eks.amazonaws.com/nodegroup
         operator: In
         values:
         - 8-core-node-group
```

Use compute resources more efficiently



Automate Kubernetes node updates

- Treat infrastructure as immutable

- If speed is a necessity
  - AWS Systems Manager Patch Manager for in-place patching
  - For OS with a read-only root file system, consider update operators for the OS (e.g., Bottlerocket update operator)

```
apiVersion:
karpenter.sh/v1alpha5
kind: Provisioner
metadata:
  name: default
spec:
  requirements:
....
  limits:
  resources:
    cpu: 1000
ttlSecondsUntilExpired: 1800
```

Careem: Scaling Workloads on EKS with KEDA

Agenda

- Careem SuperApp & Cloud footprint
- Scaling Challenges in Careem
- Community’s scaling options and challenges
- KEDA architecture and how it works
- How do we use KEDA
- Challenges
- Best practices

Who we are?

- Our mission is to simplify the life of people.
- Founded in 2012, operating in >70 cities and 10 countries.
- Ride Hailing, Food & Groceries delivery, Payments solutions , Bike & Car renting and more…
- Acquired by Uber for $3.1B in 2020.
- 2.5m+ registered captains.
- 50m+ total registered users.

Careem Cloud Footprint

1. Careem is leveraging the power of AWS with more than 50+ AWS accounts.

2. With over 800+ services, Careem utilizes a wide range of AWS offerings to support its operations EBS , ECS, Lambda ,EKS ,SQS ,Kinesis,DynamoDb, RDS etc…

3. We handle more than ~13k Requests per seconds on our APIs or 1 billion requests a day.

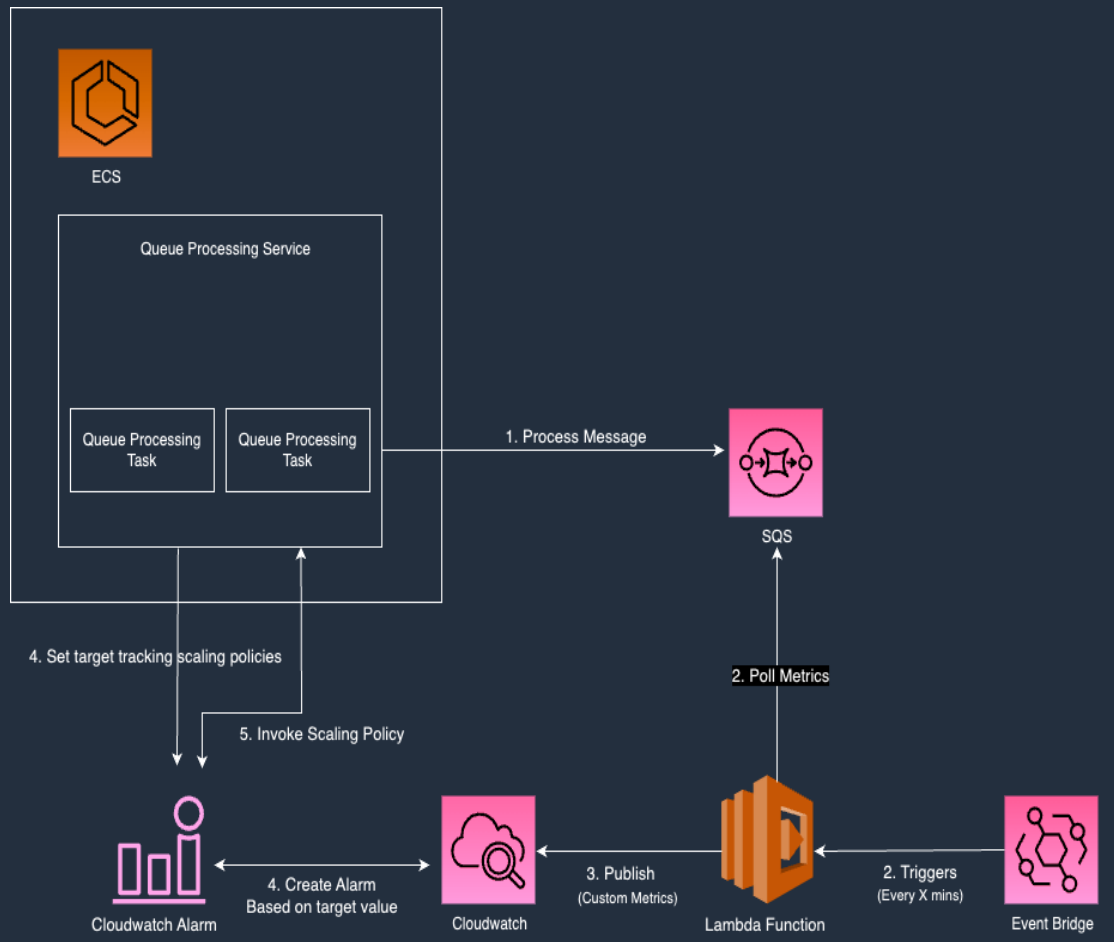
Scaling Challenges in Careem

Sudden Increases in Traffic	Careem receives sudden increases in traffic due to promo codes or random events/discounts
CPU/Memory Limitations	The CPU/Memory often falls short for most 'consumers' workloads, creating scalability challenges in ECS/Beanstalk based on SQS/Kinesis/Kafka load.
Near Real-Time Scaling	We cannot afford delays due to the critical nature of our operations.

Community's Scaling Solution

ECS Autoscaling with SQS & Custom Metrics

- EventBridge Scheduled Event:
  - Triggers Lambda periodically (e.g., every 5 minutes).
  - Lambda polls ApproximateNumberOfMessages from SQS.
  - Calculates backlog per task based on current ECS tasks.
- CloudWatch Custom Metric:
  - Monitors messages in the queue per task.
  - Metric value determined by Lambda.
- Target Tracking Policy:
  - Scales ECS service based on custom metric & set target value.
  - Analogous to a thermostat: set desired value, system adjusts.
- CloudWatch Alarms:
  - Invokes scaling policy based on backlog per task measurements.



Challenges with Community AutoScaling Methods

- The costs associated with solutions like Lambda and custom metrics can be significant.
- Publishing custom metrics often requires altering application logic, hindering cloud nativity and increasing effort.
- Customizing these solutions for developers, especially when integrating with Kinesis/Kafka/Prometheus or non-AWS services, can be challenging.
- We require near real-time scaling , cannot afford timeouts due to the critical nature of our operations.

EKS & KEDA scaler to the Rescue

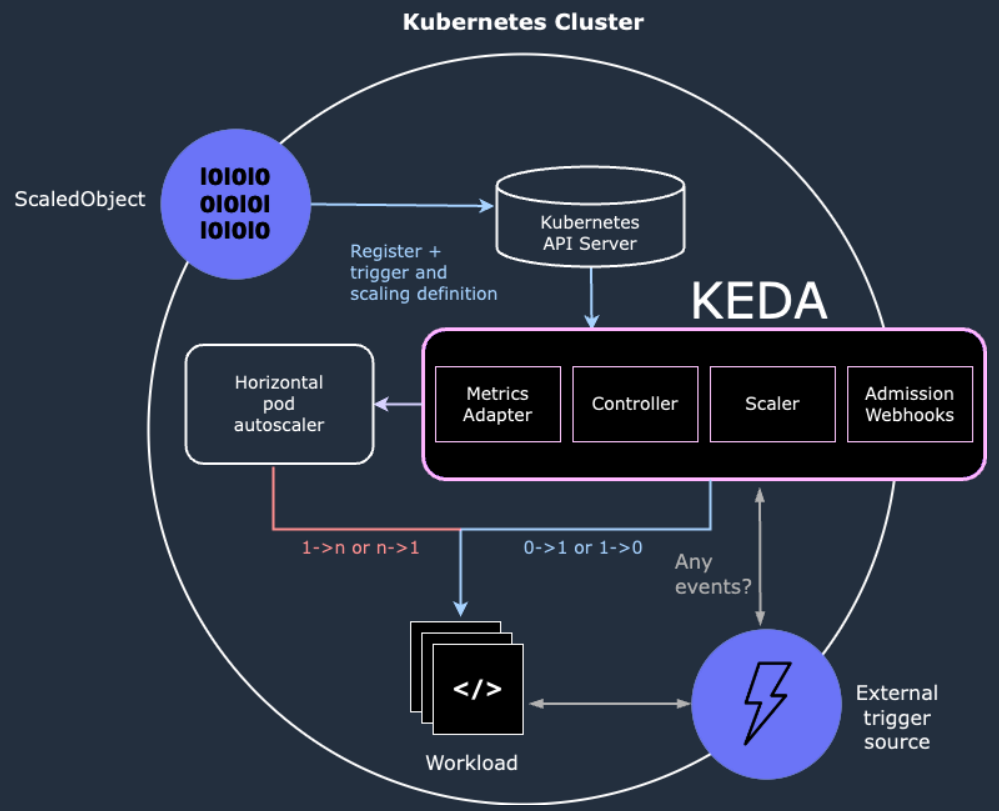
- KEDA is a Kubernetes-based event-driven autoscaler that aims to make Kubernetes event driven scaling very simple.
- Keda allows you to scale any deployment resource or job based on events, not only CPU / Memory
- KEDA integrates with popular event sources like SQS, Prometheus, Kinesis, Kafka and much more.

How Keda Works?

- Create ScaledObject/ScaledJob
- You can use multiple scaler under triggers.
- KEDA creates the HPA and provide required metrics to it.
- You can tweak HPA related settings

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
 name: sqs-consumer
spec:
 scaleTargetRef:
   name: example-sqs-consumer
 pollingInterval:  30
 minReplicaCount:  1
 maxReplicaCount:  100
 fallback:
   failureThreshold: 3
   replicas: 6
 triggers:
     - type: aws-sqs-queue
       metadata:
         queueURL: https://sqs...com/account_id/QueueName
         queueLength: "5"
         awsRegion: "eu-west-1"
```

KEDA architecture



source: KEDA docs

- The KEDA-operator poll the metrics from external sources and stores it, it also activates and deactivates Kubernetes Deployments to scale 0←→1.
- Metric-adapter: acts as a Kubernetes metrics server that exposes rich event data like queue length or stream lag to the Horizontal Pod Autoscaler to drive scale out.
- Admission-controller: ensure best practices, preventing issues like multiple ScaledObjects targeting the same scale point.

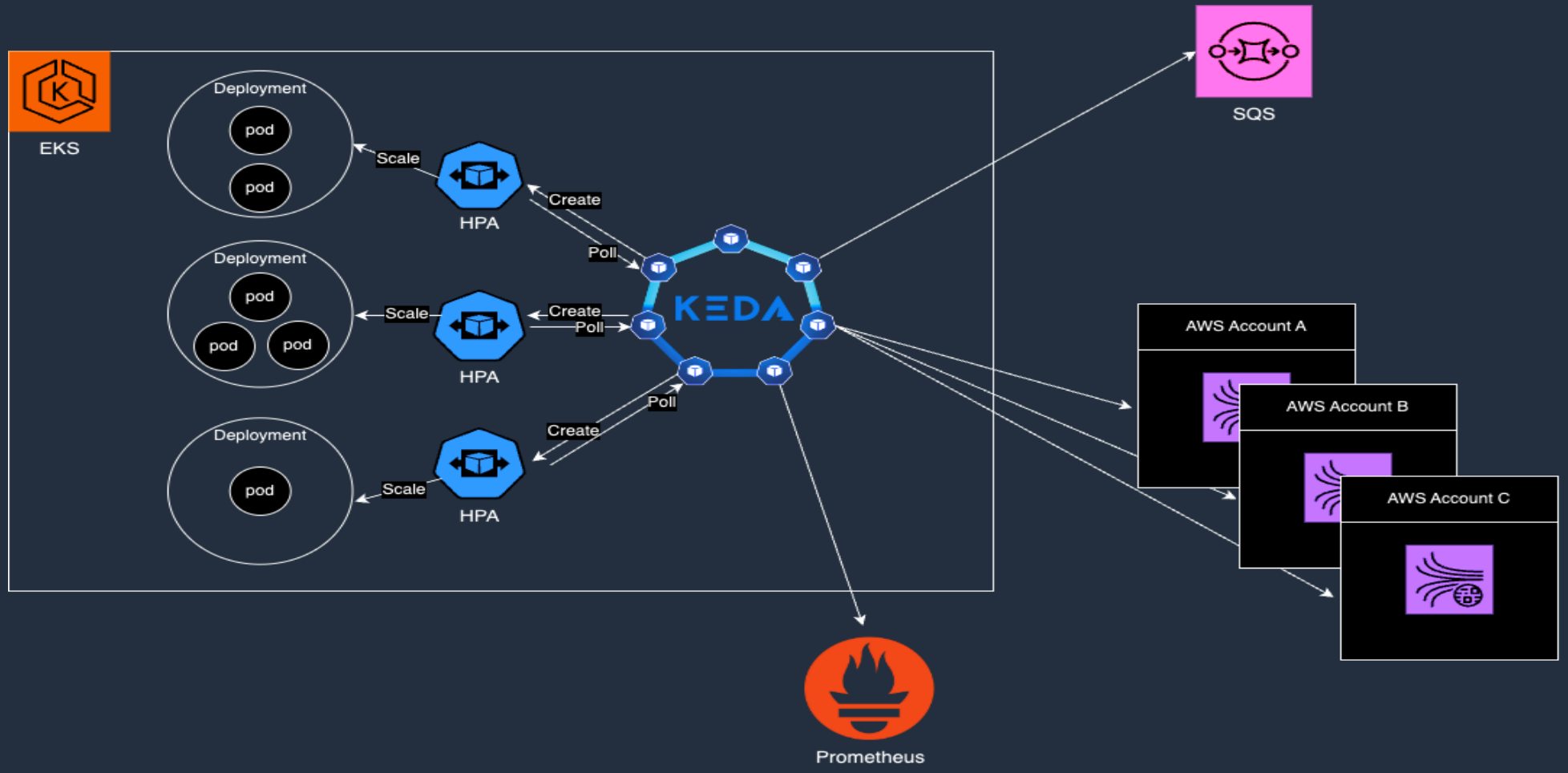
Why Choose KEDA Over HPA?

KEDA has the ability to scale down to zero, allowing for efficient resource utilization.

HPA's custom metrics often require publishing metrics to CloudWatch and deploying additional components in the cluster, adding complexity and effort, AWS deprecated cloudwatch adapter.

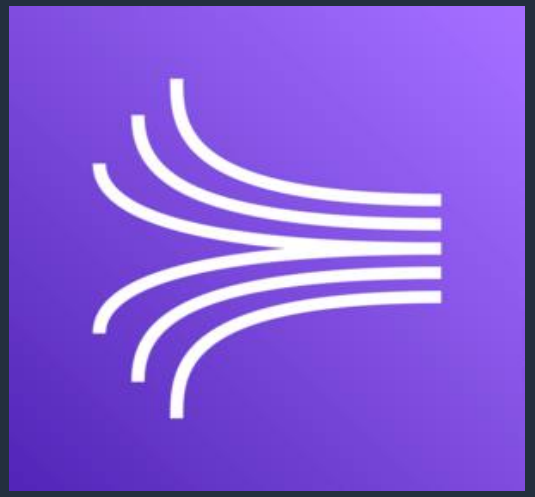
KEDA offers easy fallback mechanisms and intuitive configurations, making it more developer-friendly.

How Careem Uses KEDA ?



How Careem uses KEDA cont

Scaling based on SQS messages



Scaling based on Kinesis ShardCount

For Scaling HTTP services and Kafka consumer we rely on prometheus scaler

How Careem uses KEDA cont

Scaling based on SQS messages

- KEDA periodically polls the SQS queue to retrieve the ApproximateNumberOfMessagesVisible metric
- Based on the current number of messages in the queue and the thresholds defined (e.g., queueLength), KEDA decides whether to scale your workload up or down.
- fallback is usually equals the maxReplicas
- identityOwner = operator since most of our SQS's are in the same account as KEDA

```
spec:
  minReplicas: 2
  maxReplicas: 25
  pollingInterval: 6 # How frequently we should go for
metrics (in seconds)
  cooldownPeriod: 300
  fallback:
    failureThreshold: 5
    replicas: 25
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: <QUEUE_URL>
        queueLength: "10" # How many messages can a pod
handle in a specific time
        awsRegion: "eu-west-1"
        identityOwner: operator
```

How Careem uses KEDA cont

Scaling based on Kinesis Shards

- KEDA periodically checks the number of shards in the specified Kinesis stream.
- Based on the current number of shards and the shardCount defined (which represents the number of shards a single pod can handle), KEDA decides whether to scale your workload up or down.
- To scale Kinesis Shards we use a custom solution built on Lambdas.
- Since most of our Kinesis streams are in different accounts we use identityOwner = Pod with TriggerAuthentication to authenticate using AWS_ROLE

```
apiVersion: v1
kind: Secret
metadata:
 name: test-secrets
data:
 AWS_ROLE_ARN: <encoded-iam-role-arn>
---
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
 name: keda-trigger-auth-aws-credentials
 namespace: keda-test
spec:
 secretTargetRef:
 - parameter: awsRoleArn
   name: test-secrets
   key: AWS_ROLE_ARN
---
triggers:
 - type: aws-kinesis-stream
   authenticationRef:
     name: keda-trigger-auth-aws-credentials
      metadata:
        awsRegion: eu-west-1
        shardCount: '1'
        streamName: k8s-container-logs
    identityOwner: pod
```

How Careem uses KEDA cont

Scaling based on Custom Metric

- KEDA periodically query prometheus for the metric.
- KEDA will divide the threshold to the metric's result to specify the number of pods.
- overriding default scaleDown behaviour is crucial for stable scaling and to prevent fluctuating.
- we use it to scale HTTP services mostly

```
spec:
  minReplicas: 2
  maxReplicas: 25
  pollingInterval: 30
  advanced:
    horizontalPodAutoscalerConfig:
      scaleDown:
        stabilizationWindowSeconds: 300
        policies:
        - type: Percent
          value: 100
          periodSeconds: 60
    triggers:
    - type: prometheus
      metadata:
        serverAddress: http://prom.link:port
        metricName: http_server_requests_seconds_count
        threshold: "120"
        query: sum(rate(http_server_requests_seconds_count{}[5m]))
```

Results of using KEDA and moving to EKS

0 incidents

No incident contributed to Autoscaling

90%

Cost Savings - Scaling Down Min Replicas down to 2

100%

Cost Savings on staging - Scaling Down Non HTTP Services to 0

Challenges Encountered

01

Flagger

02

Conflicts with Kubernetes native rollout method

03

Fallback issue with Prometheus

Challenges: Progressive Delivery with KEDA

- Current Setup: Using Flagger for progressive delivery in our clusters.
- Challenge:
  - Flagger's limitation with consumer-based workloads due to the absence of HTTP traffic.
- Solution:
  - Adopted Kubernetes native rollout method.
  - Heavy reliance on probes for health checks.
- Issue with KEDA:
  - On new deployments, KEDA scales the new replicaSet to match the old one immediately.
  - Results in rapid promotion during rollout.
  - Risk: Faulty deployments get promoted quickly.

Combining Progressive Delivery & KEDA Autoscaling: Our Solution

- Objective: Merge the benefits of progressive delivery with metrics analysis, KEDA autoscaling, and granular control over new deployments.
- Decision: Adopted Argo Rollouts.
- Benefits:
  - Granular control over new deployment stages.
  - No conflicts with KEDA.
  - Harmonious integration of progressive delivery and autoscaling.

```
steps:
  - setWeight: 10
  - pause: { duration: 30s }
  - analysis:
      templates:
        - templateName: sqs-error-rate
  - setWeight: 100
```

Challenges: Prometheus Scaler & KEDA An Incident Avoided

- Issue:
  - In case of losing Prometheus target, KEDA should fallback to a predefined number of replicas. This is the behaviour in SQS and Kinesis scalers.
- Reality:
  - Lost Prometheus target in production.
  - Expected Behavior: KEDA should fallback to a defined number of replicas.
  - Actual Behavior: KEDA scaled down the deployment to minimum replicas.
- Root Cause:
  - Default behavior returns an empty list when Prometheus target is lost, This empty list is interpreted as a 0 value, leading to the scale-down.
- Solution:
  - Set "ignoreNullValues" to false to address the issue.

Best Practices & Falloffs

1. Override default scaleDown behaviour with stabilization window to prevent scaling fluctuation.

2. Handle Null Values: Be aware of the "ignoreNullValues" setting. If your metrics source might return null or empty values, configure this setting appropriately to prevent unintended scaling.

3. Fine-tune Poll Interval as it is only relevant when scaling 0<->1 .
   - 1<->N scaling is controlled by HPA –horizontal-pod-autoscaler-sync-period defaults to 15 seconds
   - To prevent excessive API calls and improve performance consider using “Metric Cache”

Thank you!

Sami Shabaneh
https://www.linkedin.com/in/thasami/

Talal Shobaita
https://www.linkedin.com/in/talalshobaita