Containers

Scalable and Cost-Effective Event-Driven Workloads with KEDA and Karpenter on Amazon EKS

by Sanjeev Ganjihal and Asif Khan | on 16 NOV 2023 | in Amazon Elastic Kubernetes Service, Amazon Simple Queue Service (SQS), Compute, Containers, Messaging, Technical How-to | Permalink | Share

In today's cloud-native landscape, efficient management of event-driven workloads is essential for real-time data processing. Traditional autoscaling often falls short amidst unpredictable event volumes, leading to inefficiencies and increased costs. Amazon Elastic Kubernetes Service (EKS), which is a managed container orchestration platform and is well-suited for deploying container-based applications. By integrating Kubernetes Event-Driven Autoscaling (KEDA) and Karpenter with Amazon EKS, we can overcome these challenges. KEDA enables detailed autoscaling based on event metrics, and Karpenter ensures timely node provisioning, addressing the inefficiencies of manual scaling. This post aims to provide a simplified guide to integrating KEDA and Karpenter with Amazon EKS for a more efficient, cost-effective, and high-performing setup.

Through a step-by-step approach, we'll simplify the configuration process to help teams better manage event-driven workloads on Amazon EKS. This integration showcases a scalable, cost-effective, and resilient setup for handling event-driven workloads in a Kubernetes environment, making the most of the capabilities of Amazon EKS, KEDA, and Karpenter.

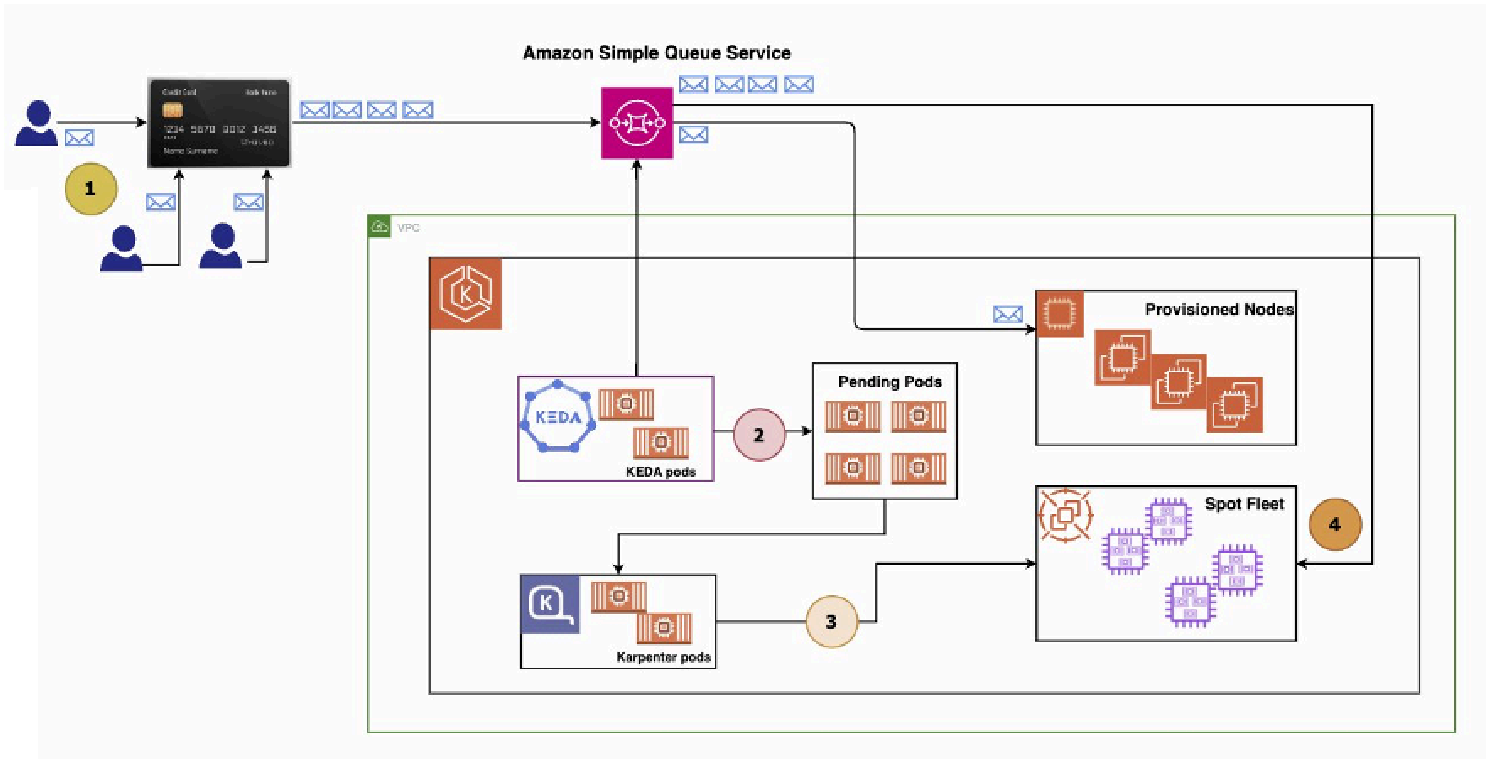
Kubernetes Event-Driven Autoscaling (KEDA)

Shifting our attention to the domain of autoscaling, KEDA stands out as a cornerstone for event-driven scaling. It equips your Kubernetes deployments with the ability to dynamically adapt in response to external events funneled from a variety of sources, such as message queues and event streams. Embrace an event-driven paradigm, that allows your applications to scale precisely in line with event influx, ensuring resource optimization and cost-efficiency. With KEDA, tap into a plethora of scalers catering to different event sources, fortifying the bridge between Kubernetes and the external world.

Karpenter

Karpenter is a flexible, high-performance Kubernetes cluster autoscaler offering dynamic, groupless provisioning of worker node capacity to cater to unscheduled pods. Thanks to Karpenter's groupless architecture, the restriction of using similarly specified instance types is eliminated. Karpenter perpetually assesses the collective resource demands of pending pods along with other scheduling constraints (for instance, node selectors, affinities, tolerations, and topology spread constraints), and orchestrates the ideal instance compute capacity as defined in the NodePool configuration. This augmented flexibility empowers different teams to tailor their own NodePool configurations to suit their application and scaling requisites. Moreover, Karpenter directly provisions nodes utilizing the Amazon EC2 fleet application programming interface (API), bypassing the need for nodes and Amazon EC2 auto scaling groups, which significantly accelerates provisioning and retrial times (shifting from minutes to mere milliseconds). This acceleration not only boosts performance but also enhances service level agreements (SLAs).

Solution architecture for running event driven workloads on Amazon EKS



Workflow outline

1. **Initiation of financial transaction:** Users initiate financial transactions which are captured and forwarded to an Amazon Simple Queue Service (SQS) For demonstration purposes, a script emulating the role of a Message Producer will be executed to populate the queue with messages representing transactions.
2. **Transaction message consumption:** Within an Amazon EKS cluster, a containerized Message Consumer application is actively monitoring and retrieving messages from the Amazon SQS queue for processing.
3. **Queue depth escalation:** As user engagement escalates, exemplified by running the Message Producer script in multiple terminals during the demonstration, the influx of transaction messages causes an increase in queue depth, indicating a backlog as the Message Consumer application struggles to keep pace with the incoming message rate.
4. **Automated scalability response:** Integrated within the Amazon EKS cluster are KEDA and Karpenter, which continuously monitor the queue depth. Upon detecting a message backlog, KEDA triggers the provisioning of additional Message Consumer pods, enhancing the system's processing capacity to effectively manage the higher message volume in a timely manner.

Solution overview

Prerequisites:

- An AWS account
- eksctl – v0.162.0
- AWS CLI – v2.10.0
- kubectl – v1.28.3
- helm – v3.13.1
- Docker – v24.0.6
- EKS – v1.28

- KEDA – v 2.12
- Karpenter – v0.32

Walkthrough

Sample application overview: Orchestrating event-driven messaging

We'll dive into the construction of a basic message processing application, showcasing the seamless orchestration of event-driven messaging. The application comprises the following integral components:

- **Amazon SQS:** Serving as the backbone of our messaging architecture, Amazon SQS is a fully-managed message queuing service that graciously handles incoming messages and avails them to consumers upon request.
- **Message producer:** Encapsulated within a container, this Python-based application emulates real-world transaction scenarios by publishing messages to the Amazon SQS queue.
- **Message consumer:** Residing in a separate container, the Message Consumer stands vigil, incessantly scrutinizing the SQS queue to promptly seize the earliest message awaiting processing.
- **Amazon DynamoDB:** Acting as the terminal point of our message journey, the Message Consumer retrieves messages from the queue and dutifully deposits them into a DynamoDB table, earmarking the conclusion of the message processing cycle.

Deploying Amazon EKS Cluster, KEDA, Karpenter and sample application

Follow the steps to set up an Amazon EKS cluster, and install KEDA and Karpenter on your Amazon EKS cluster.

Clone the repository to your local machine or download it as a ZIP file using the following command:

```
git clone https://github.com/aws-samples/amazon-eks-scaling-with-keda-and-karpenter.git
```

Navigate to the repository's directory:

```
cd amazon-eks-scaling-with-keda-and-karpenter
```

Modify the environmentVariables.sh file located in the deployment directory as per your requirements.

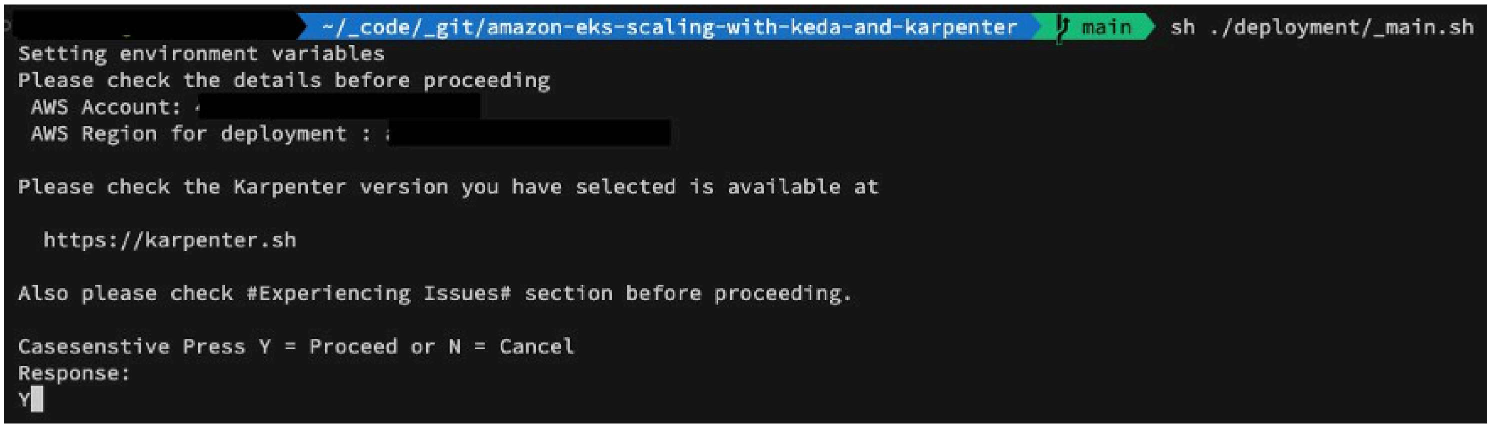
	A	B
1	**Variable Name**	**Description**
2	**AWS_REGION**	**The AWS region.**
3	**ACCOUNT_ID**	**The AWS account ID.**
4	**TEMPOUT**	**Temporary output file. This used to temp. store CFN for karpenter**

5	**DYNAMODB_TABLE**	**The name of the Amazon DynamoDB table.**
6	**CLUSTER_NAME**	**The name of the Amazon EKS cluster.**
7	**KARPENTER_VERSION**	**The version of Karpenter.**
8	**NAMESPACE**	**The Kubernetes namespace for KEDA.**
9	**SERVICE_ACCOUNT**	**The Kubernetes service account for KEDA.**
10	**IAM_KEDA_ROLE**	**The AWS IAM role for KEDA.**
11	**IAM_KEDA_SQS_POLICY**	**The AWS IAM policy for KEDA to access SQS.**
12	**IAM_KEDA_DYNAMO_POLICY**	**The AWS IAMpolicy for KEDA to access DynamoDB.**
13	**SQS_QUEUE_NAME**	**The name of the Amazon SQS queue**
14	**SQS_QUEUE_URL**	**The URL of the Amazon SQS queue.**
15	**SQS_TARGET_DEPLOYMENT**	**The target deployment for KEDA to scale based on Amazon SQS messages.**
16	**SQS_TARGET_NAMESPACE**	**The target namespace for the deployment that KEDA scales based on Amazon SQS messages.**

To deploy the application, run this command

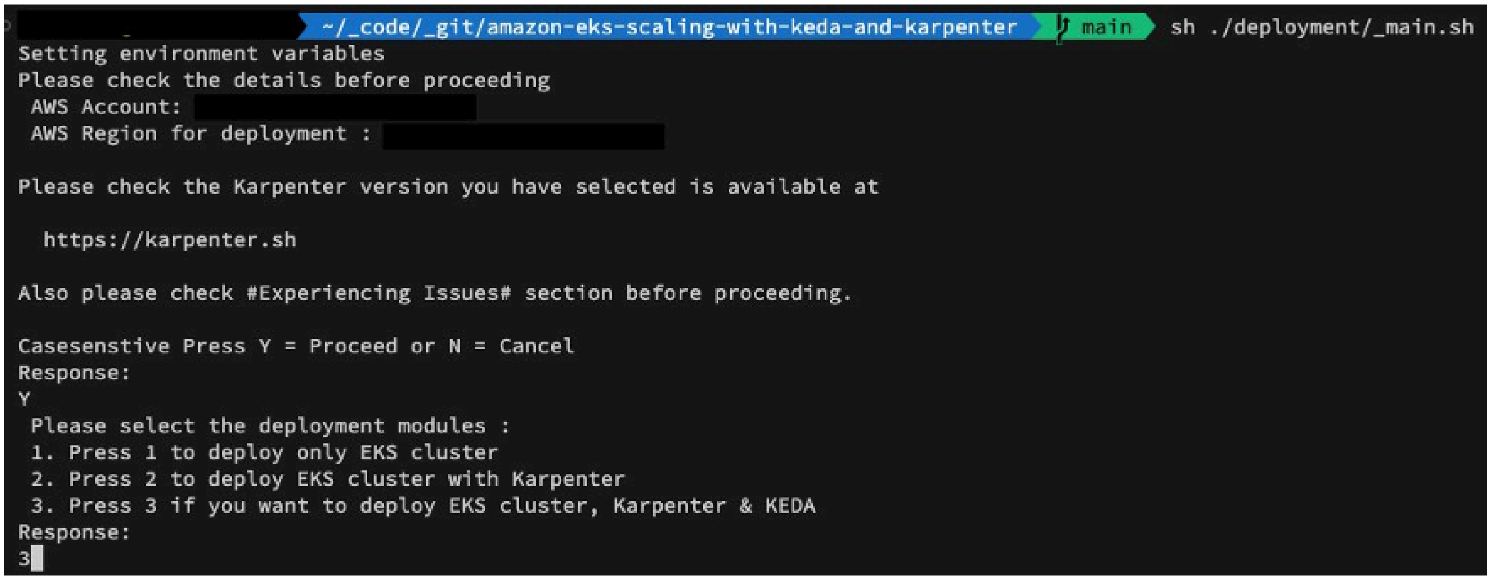
```
sh ./deployment/_main.sh
```

You'll be asked to verify the account in the context:



```
[redacted] ~/_code/_git/amazon-eks-scaling-with-keda-and-karpenter  main  sh ./deployment/_main.sh
Setting environment variables
Please check the details before proceeding
 AWS Account: [redacted]
 AWS Region for deployment : [redacted]

Please check the Karpenter version you have selected is available at

  https://karpenter.sh

Also please check #Experiencing Issues# section before proceeding.

Casesenstive Press Y = Proceed or N = Cancel
Response:
Y
```

Next, select your deployment option:



```
[redacted] ~/_code/_git/amazon-eks-scaling-with-keda-and-karpenter  main  sh ./deployment/_main.sh
Setting environment variables
Please check the details before proceeding
 AWS Account: [redacted]
 AWS Region for deployment : [redacted]

Please check the Karpenter version you have selected is available at

  https://karpenter.sh

Also please check #Experiencing Issues# section before proceeding.

Casesenstive Press Y = Proceed or N = Cancel
Response:
Y
 Please select the deployment modules :
 1. Press 1 to deploy only EKS cluster
 2. Press 2 to deploy EKS cluster with Karpenter
 3. Press 3 if you want to deploy EKS cluster, Karpenter & KEDA
Response:
3
```

Under the hood

Upon selecting options 1, 2, or 3, the corresponding components are deployed. Specifically, option three triggers the deployment of all components – the Amazon EKS cluster, Karpenter, and KEDA.

Each of these components comes with its own independent deployment scripts. Thus, if you wish to bypass the aforementioned deployment process or desire to modify features for each component, you can alter the relevant files associated with each component:

- `createCluster.sh`
- `createKarpenter.sh`
- `createKeda.sh`

This setup provides a flexible approach to deploying and customizing the components as per your requirements.

The GitHub repository houses the essential configuration files required for deploying KEDA and Karpenter. These files can be tailored to meet your particular needs.

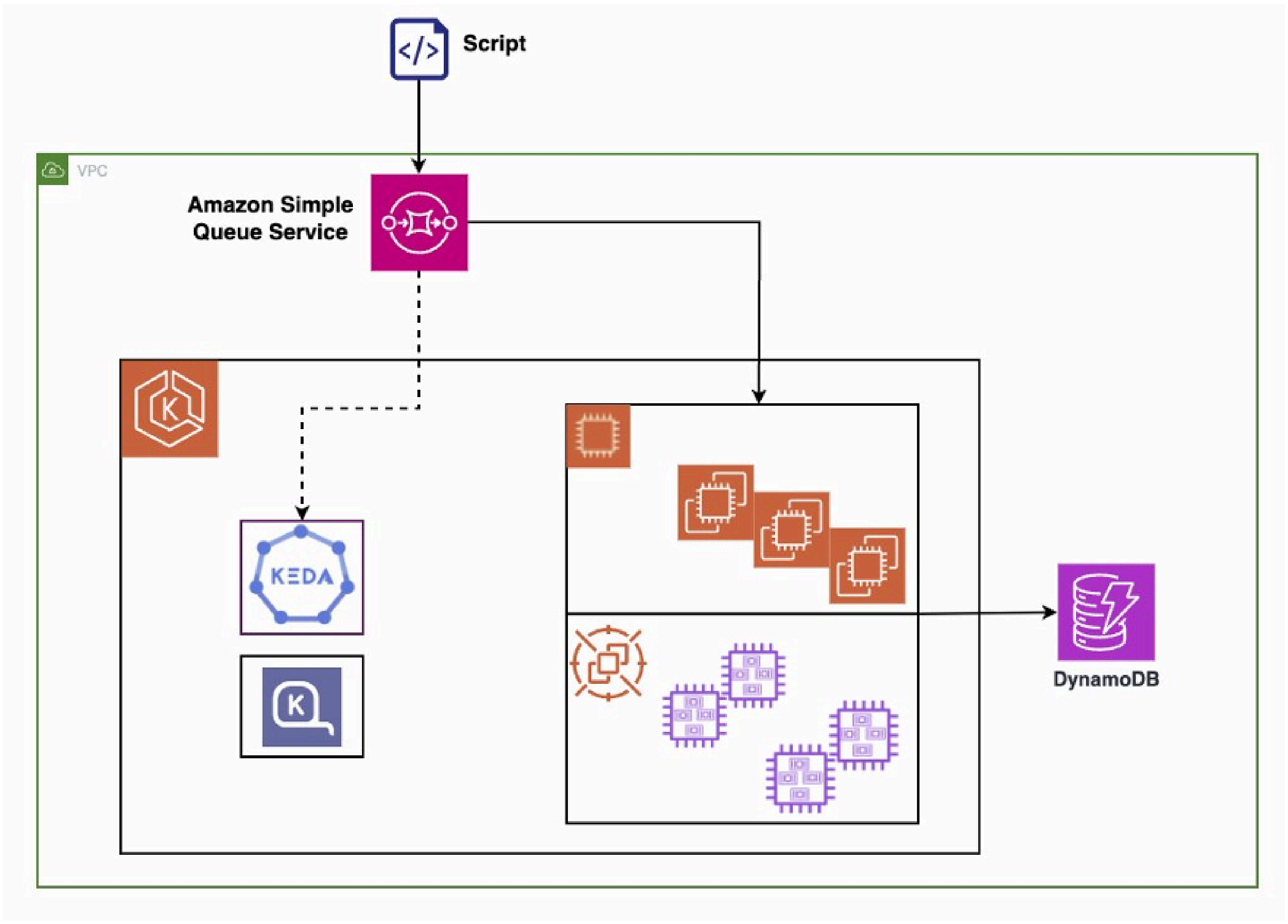
Below are some notable files to consider:

- deployment/keda: Contains the deployment files for KEDA components.
- deployment/karpenter: Contains the deployment files for Karpenter components.

Testing: KEDA and Karpenter scaling in action

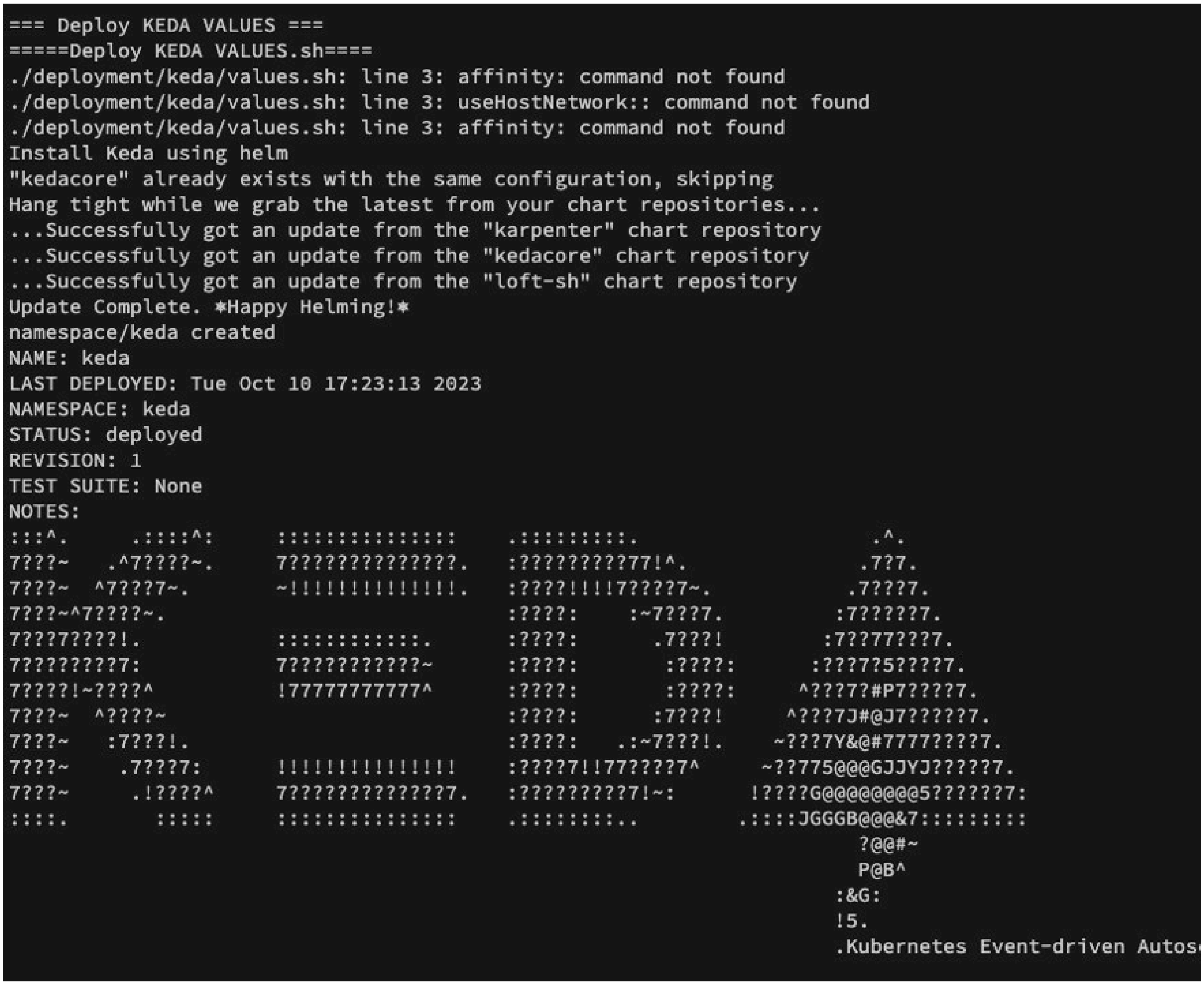
In this section, we'll evaluate the scaling efficacy through a simulation script. Rather than real-world multiple requests being funneled into Amazon SQS from human transactions, we'll execute a script that injects messages into Amazon SQS, which mimics a real-world scenario.

The script keda-mock-sqs-post.py has a straightforward function that dispatches messages to Amazon SQS at specified recurring intervals.

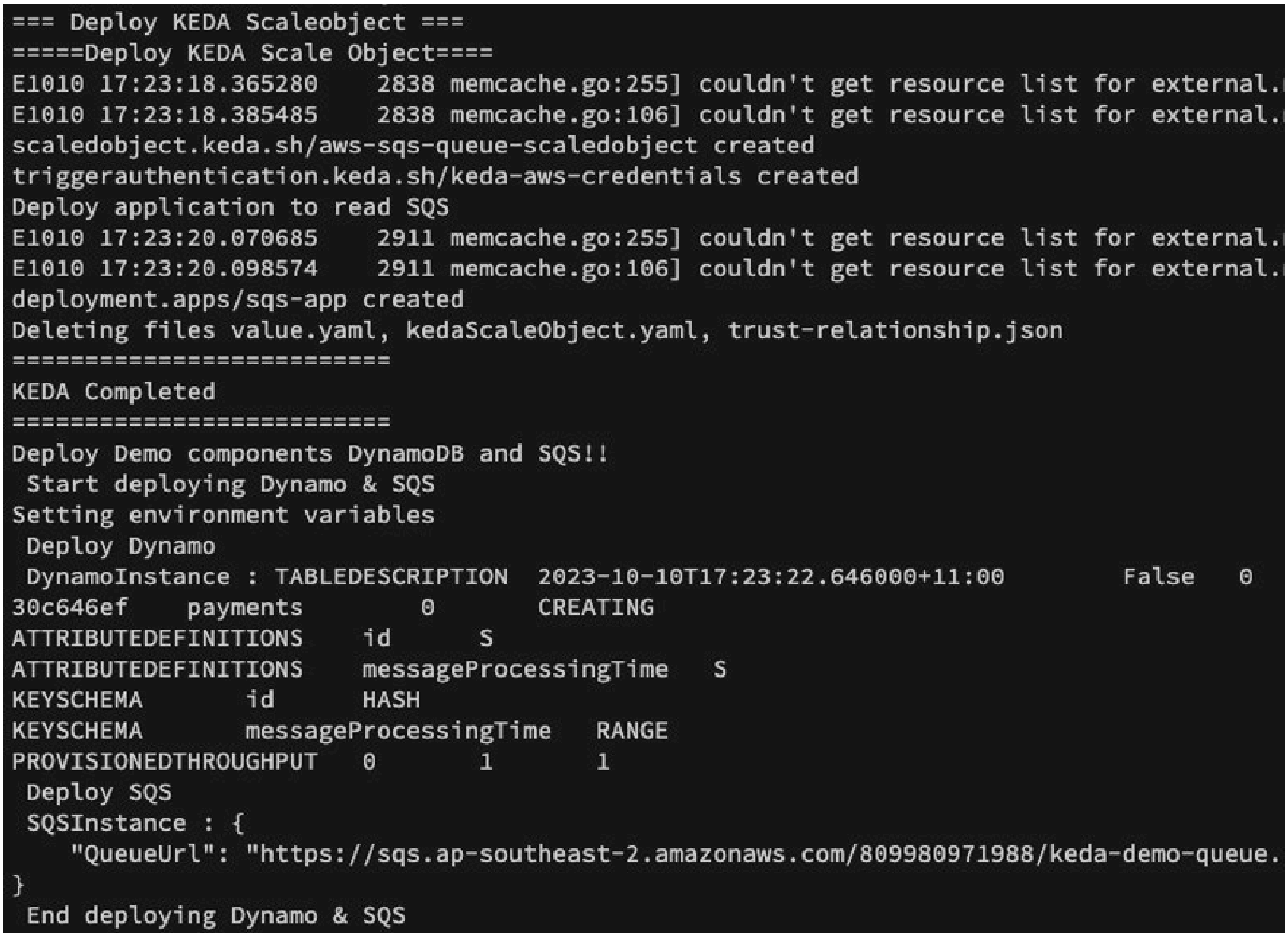


As illustrated in the previous diagram, the architecture of our environment deployment remains unchanged; however, we have introduced a message pump script. Multiple instances of this script will be run to simulate a realistic load, thereby triggering KEDA and Karpenter to scale pods and nodes accordingly.

Upon the deployment of the Cluster, Karpenter, and KEDA, your shell interface will exhibit a similar appearance.



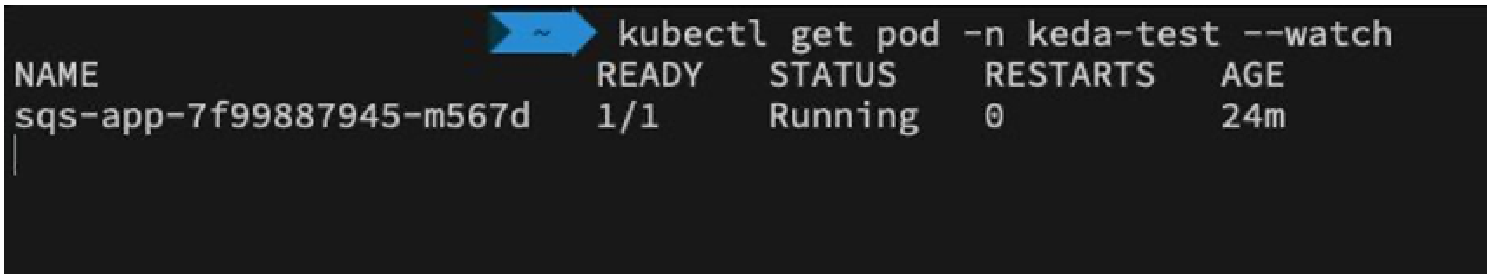
```
=== Deploy KEDA VALUES ===
=====Deploy KEDA VALUES.sh====
./deployment/keda/values.sh: line 3: affinity: command not found
./deployment/keda/values.sh: line 3: useHostNetwork:: command not found
./deployment/keda/values.sh: line 3: affinity: command not found
Install Keda using helm
"kedacore" already exists with the same configuration, skipping
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "karpenter" chart repository
...Successfully got an update from the "kedacore" chart repository
...Successfully got an update from the "loft-sh" chart repository
Update Complete. ⎈Happy Helming!⎈
namespace/keda created
NAME: keda
LAST DEPLOYED: Tue Oct 10 17:23:13 2023
NAMESPACE: keda
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
:::^.     .::::^:     :::::::::::::::    .:::::::::.                   .^.
7???~   .^7?????~.    7??????????????.   :??????????77!^.             .7?7.
7???~  ^7???7~.       ~!!!!!!!!!!!!!!.   :????!!!!7????7~.           .7???7.
7???~^7????~.                            :????:    :~7????7.        :7?????7.
7???7????!.           ::::::::::::.      :????:      .7???!        :7??7????7.
7????????7:           7???????????~      :????:       :????:      :???7?5????7.
7????!~????^          !77777777777^      :????:       :????:     ^????7?#P7????7.
7???~  ^????~                            :????:      :7???!     ^???7J#@J7?????7.
7???~   :7???!.                          :????:   .:~7???!.    ~???7Y&@#777?????7.
7???~    .7???7:      !!!!!!!!!!!!!!!    :????7!!7?????7^     ~??775@@@GJJYJ?????7.
7???~     .!????^     7?????????????7.   :?????????7!~:      !????G@@@@@@@@5??????7:
::::.       :::::     :::::::::::::::    .::::::::..        .::::JGGGB@@@&7:::::::::
                                                                      ?@@#~
                                                                      P@B^
                                                                    :&G:
                                                                    !5.
                                                                    .Kubernetes Event-driven Autos
```



```
=== Deploy KEDA Scaleobject ===
=====Deploy KEDA Scale Object====
E1010 17:23:18.365280    2838 memcache.go:255] couldn't get resource list for external.
E1010 17:23:18.385485    2838 memcache.go:106] couldn't get resource list for external.
scaledobject.keda.sh/aws-sqs-queue-scaledobject created
triggerauthentication.keda.sh/keda-aws-credentials created
Deploy application to read SQS
E1010 17:23:20.070685    2911 memcache.go:255] couldn't get resource list for external.
E1010 17:23:20.098574    2911 memcache.go:106] couldn't get resource list for external.
deployment.apps/sqs-app created
Deleting files value.yaml, kedaScaleObject.yaml, trust-relationship.json
==========================
KEDA Completed
==========================
Deploy Demo components DynamoDB and SQS!!
 Start deploying Dynamo & SQS
Setting environment variables
 Deploy Dynamo
 DynamoInstance : TABLEDESCRIPTION  2023-10-10T17:23:22.646000+11:00        False   0
30c646ef    payments        0       CREATING
ATTRIBUTEDEFINITIONS    id      S
ATTRIBUTEDEFINITIONS    messageProcessingTime   S
KEYSCHEMA       id      HASH
KEYSCHEMA       messageProcessingTime   RANGE
PROVISIONEDTHROUGHPUT   0       1       1
 Deploy SQS
 SQSInstance : {
    "QueueUrl": "https://sqs.ap-southeast-2.amazonaws.com/809980971988/keda-demo-queue.
}
 End deploying Dynamo & SQS
```

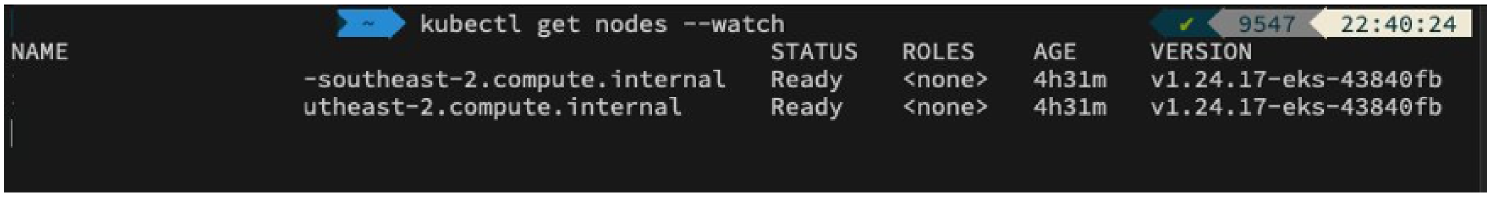
Open two additional terminals and establish a connection to the Amazon EKS cluster.

In the first terminal, navigate to the **keda-test** namespace.



```
                             ~   kubectl get pod -n keda-test --watch
NAME                          READY   STATUS    RESTARTS   AGE
sqs-app-7f99887945-m567d      1/1     Running   0          24m
```

In the second terminal, navigate to the **Nodes** section (your terminal should appear as shown below).

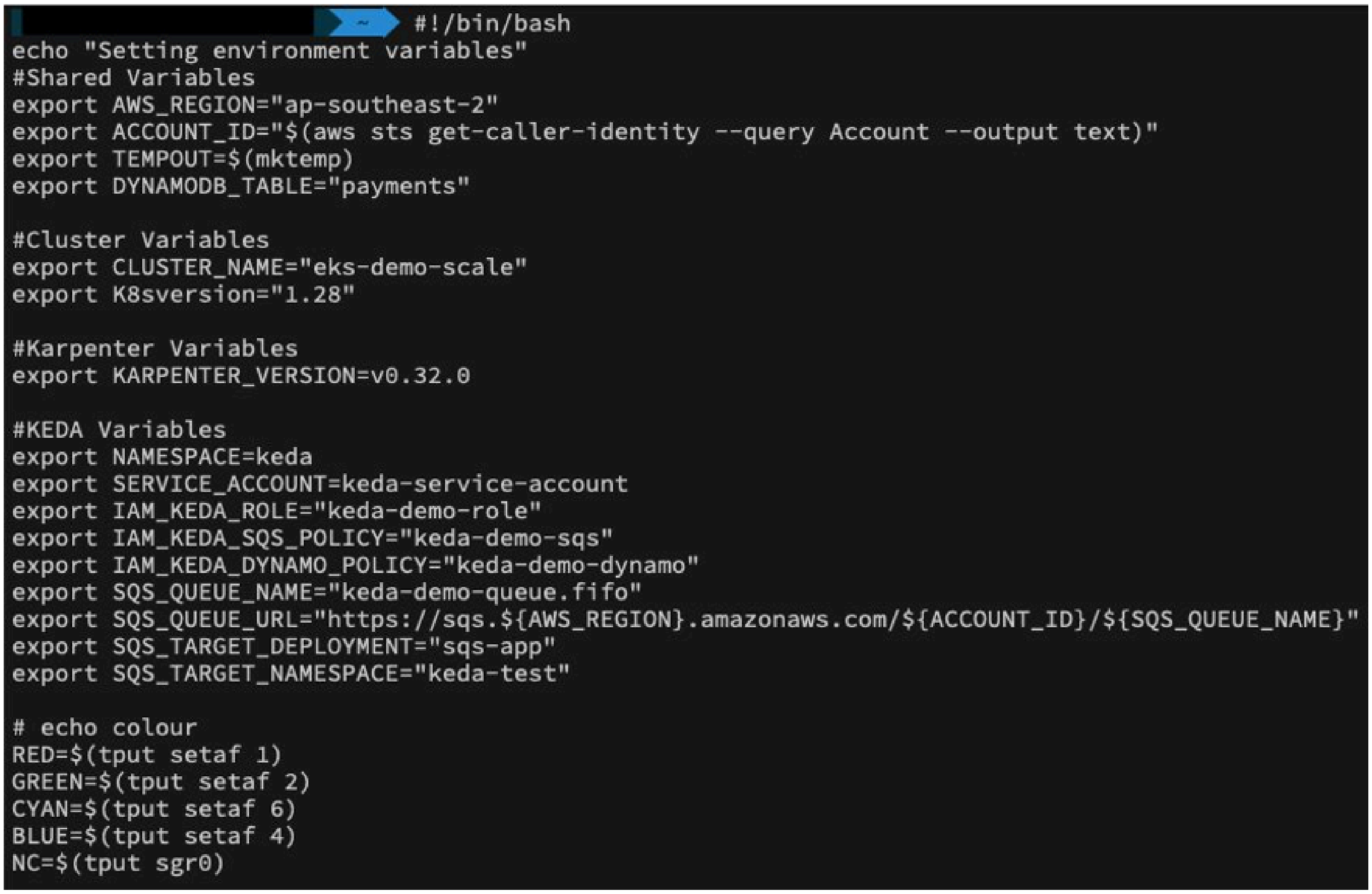


```
                      ~   kubectl get nodes --watch                                  ✓  9547   22:40:24
NAME                                                      STATUS   ROLES    AGE     VERSION
                 -southeast-2.compute.internal            Ready    <none>   4h31m   v1.24.17-eks-43840fb
                 utheast-2.compute.internal               Ready    <none>   4h31m   v1.24.17-eks-43840fb
```

Open three or more terminals, copy the content from deployment/environmentVariables.sh and execute it on all three terminals.

This action configures the terminal context with all the required environment variables needed for the script execution.

Run the keda-mock-sqs-post.py script on all four terminals.

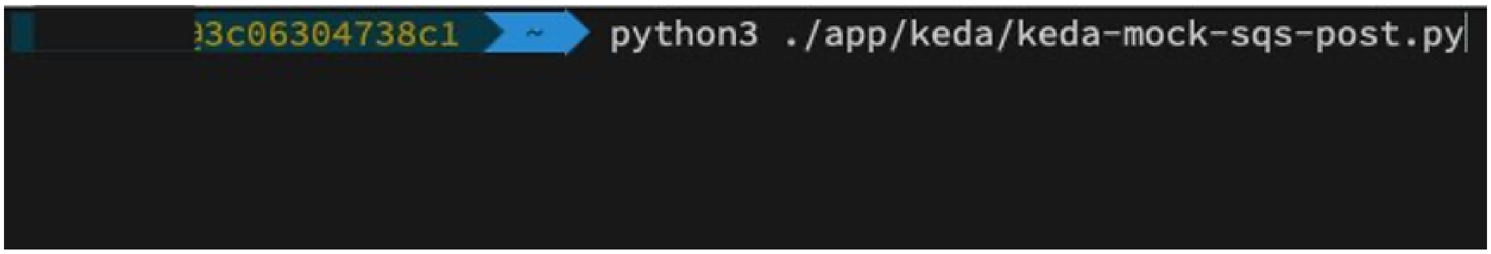


```
[redacted] ~ #!/bin/bash
echo "Setting environment variables"
#Shared Variables
export AWS_REGION="ap-southeast-2"
export ACCOUNT_ID="$(aws sts get-caller-identity --query Account --output text)"
export TEMPOUT=$(mktemp)
export DYNAMODB_TABLE="payments"

#Cluster Variables
export CLUSTER_NAME="eks-demo-scale"
export K8sversion="1.28"

#Karpenter Variables
export KARPENTER_VERSION=v0.32.0

#KEDA Variables
export NAMESPACE=keda
export SERVICE_ACCOUNT=keda-service-account
export IAM_KEDA_ROLE="keda-demo-role"
export IAM_KEDA_SQS_POLICY="keda-demo-sqs"
export IAM_KEDA_DYNAMO_POLICY="keda-demo-dynamo"
export SQS_QUEUE_NAME="keda-demo-queue.fifo"
export SQS_QUEUE_URL="https://sqs.${AWS_REGION}.amazonaws.com/${ACCOUNT_ID}/${SQS_QUEUE_NAME}"
export SQS_TARGET_DEPLOYMENT="sqs-app"
export SQS_TARGET_NAMESPACE="keda-test"

# echo colour
RED=$(tput setaf 1)
GREEN=$(tput setaf 2)
CYAN=$(tput setaf 6)
BLUE=$(tput setaf 4)
NC=$(tput sgr0)
```

Navigate to the **app** → **keda** directory and create a Python virtual environment.

```
cd app/keda
python3 -m venv env
source env/bin/activate
pip install boto3
cd  {your path}/amazon-eks-scaling-with-keda-and-karpenter
python3 ./app/keda/keda-mock-sqs-post.py
```

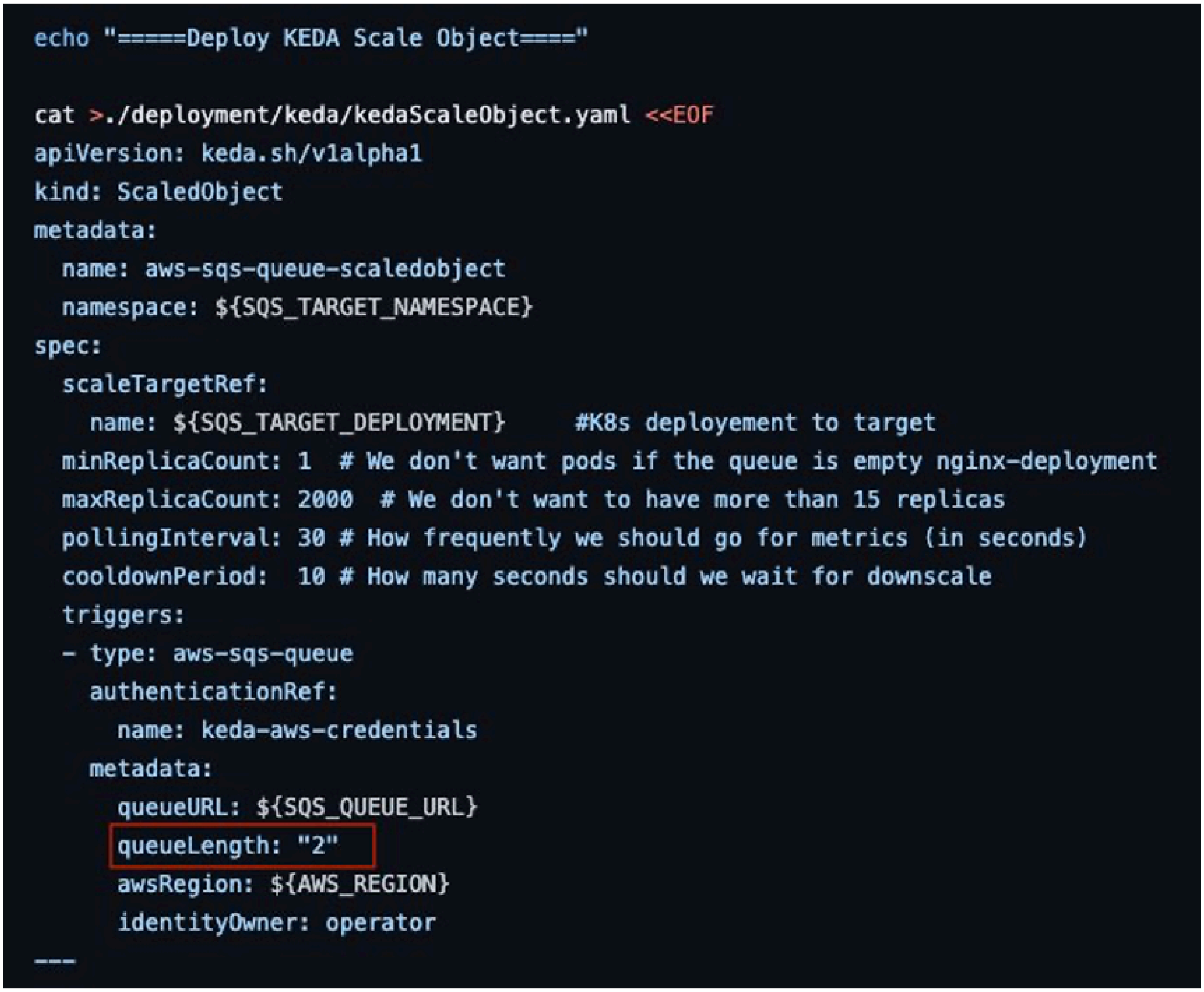
Open three or four terminals and run the script (i.e., multiple terminals are needed so that we can pump more messages in queue and increase the queue depth)



```
[redacted] 3c06304738c1 ~ python3 ./app/keda/keda-mock-sqs-post.py
```

Upon activation, the scripts will begin injecting messages into Amazon SQS. As previously mentioned, there is a single pod operating within the keda-test namespace, which continuously scans Amazon SQS for messages to process.

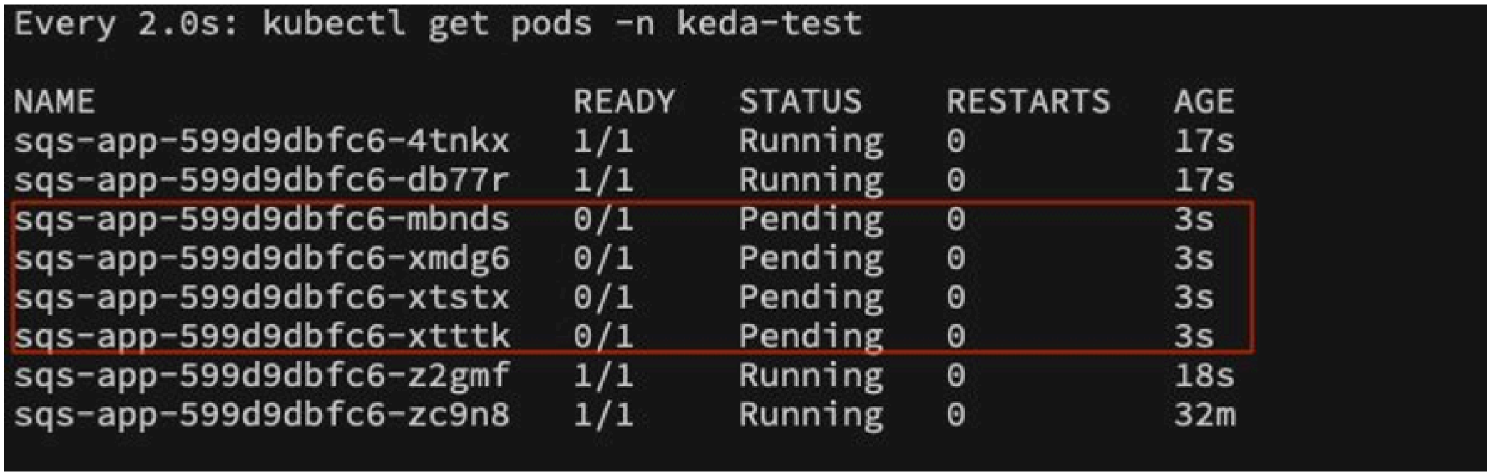
Given that the three script instances from earlier are dispatching messages at a high frequency, this pod struggles to maintain the requisite message processing scale to keep the queue length within the defined limits (i.e., 2) as specified in keda-scaledobject.sh.



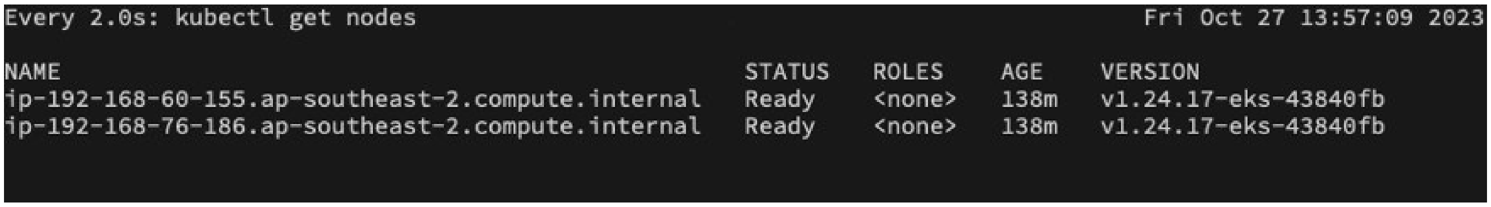
```
echo "=====Deploy KEDA Scale Object===="

cat >./deployment/keda/kedaScaleObject.yaml <<EOF
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: aws-sqs-queue-scaledobject
  namespace: ${SQS_TARGET_NAMESPACE}
spec:
  scaleTargetRef:
    name: ${SQS_TARGET_DEPLOYMENT}     #K8s deployement to target
  minReplicaCount: 1  # We don't want pods if the queue is empty nginx-deployment
  maxReplicaCount: 2000  # We don't want to have more than 15 replicas
  pollingInterval: 30 # How frequently we should go for metrics (in seconds)
  cooldownPeriod:  10 # How many seconds should we wait for downscale
  triggers:
  - type: aws-sqs-queue
    authenticationRef:
      name: keda-aws-credentials
    metadata:
      queueURL: ${SQS_QUEUE_URL}
      queueLength: "2"
      awsRegion: ${AWS_REGION}
      identityOwner: operator
---
```

Once the **scaleobject ==> queueLength** threshold is exceeded, KEDA will trigger the instantiation of additional pods to expedite the processing of Amazon SQS messages and reduce the queue length to the desired limits.

*Scaling to **eight** Pods and **two** Nodes:*



```
Every 2.0s: kubectl get pods -n keda-test

NAME                         READY   STATUS    RESTARTS   AGE
sqs-app-599d9dbfc6-4tnkx     1/1     Running   0          17s
sqs-app-599d9dbfc6-db77r     1/1     Running   0          17s
sqs-app-599d9dbfc6-mbnds     0/1     Pending   0          3s
sqs-app-599d9dbfc6-xmdg6     0/1     Pending   0          3s
sqs-app-599d9dbfc6-xtstx     0/1     Pending   0          3s
sqs-app-599d9dbfc6-xtttk     0/1     Pending   0          3s
sqs-app-599d9dbfc6-z2gmf     1/1     Running   0          18s
sqs-app-599d9dbfc6-zc9n8     1/1     Running   0          32m
```



```
Every 2.0s: kubectl get nodes                                              Fri Oct 27 13:57:09 2023

NAME                                              STATUS   ROLES    AGE    VERSION
ip-192-168-60-155.ap-southeast-2.compute.internal   Ready    <none>   138m   v1.24.17-eks-43840fb
ip-192-168-76-186.ap-southeast-2.compute.internal   Ready    <none>   138m   v1.24.17-eks-43840fb
```

Due to this rapid scaling action initiated by KEDA, we begin to observe several pods in a pending state, as the default number of nodes in the cluster (i.e., two) can only accommodate a certain number of pods.

At this point, Karpenter steps in and continuously monitors the queue of pending pods. Upon analyzing the situation, Karpenter initiates the provisioning of new nodes within the cluster.

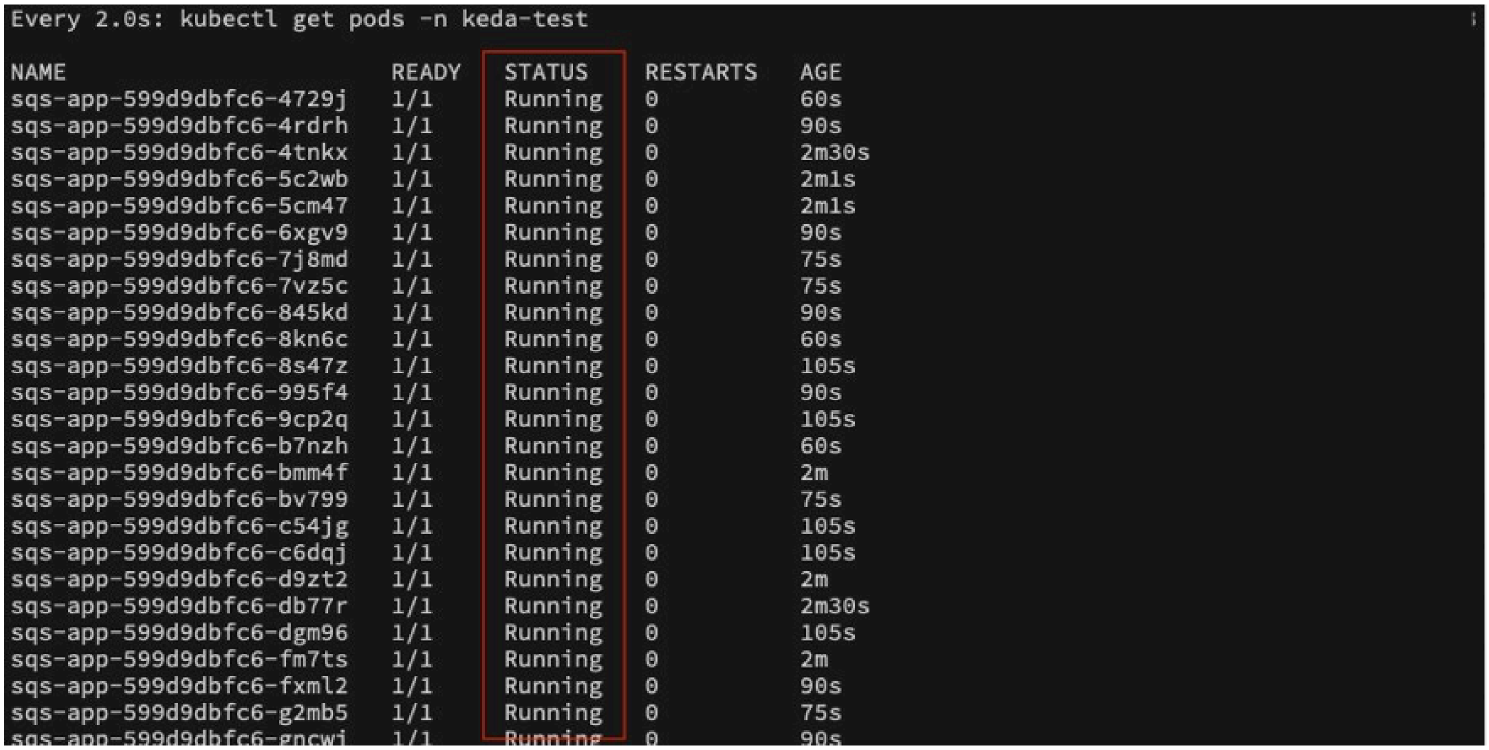
The horizontal scaling of pods and nodes orchestrated by KEDA and Karpenter will persist until either of the following conditions occurs:

- The target **queueLength** is attained (or)
- KEDA reaches the **maxReplicaCount** (or)
- Karpenter encounters the CPU and memory limits.

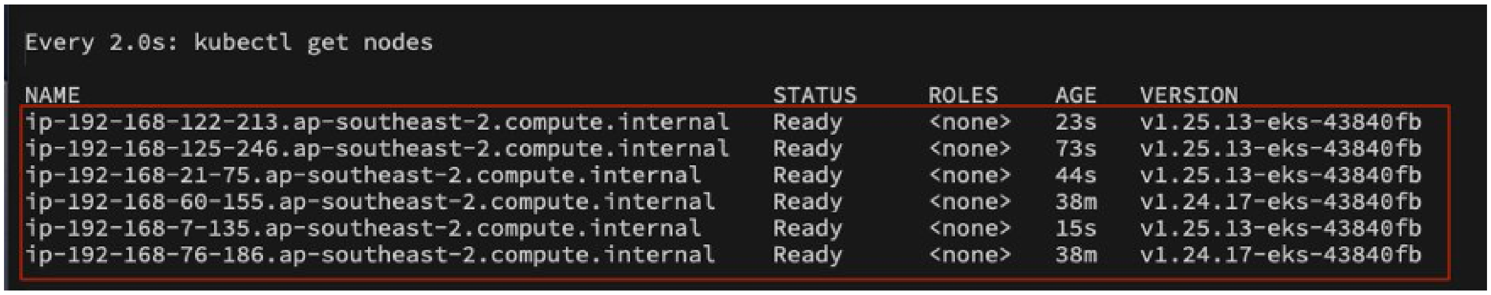
Upon clearing the backlog of messages in Amazon SQS, the strength of this architecture shines as it facilitates scaling in both directions:

- **Scale-out** – proficiently tackle pending activities or
- **Scale-in** – optimizes the infrastructure for enhanced utilization and cost benefits.

Scaling to ***50+*** *Pods and* ***six*** *Nodes (i.e., four Karpenter Nodes):*



```
Every 2.0s: kubectl get pods -n keda-test

NAME                         READY   STATUS    RESTARTS   AGE
sqs-app-599d9dbfc6-4729j     1/1     Running   0          60s
sqs-app-599d9dbfc6-4rdrh     1/1     Running   0          90s
sqs-app-599d9dbfc6-4tnkx     1/1     Running   0          2m30s
sqs-app-599d9dbfc6-5c2wb     1/1     Running   0          2m1s
sqs-app-599d9dbfc6-5cm47     1/1     Running   0          2m1s
sqs-app-599d9dbfc6-6xgv9     1/1     Running   0          90s
sqs-app-599d9dbfc6-7j8md     1/1     Running   0          75s
sqs-app-599d9dbfc6-7vz5c     1/1     Running   0          75s
sqs-app-599d9dbfc6-845kd     1/1     Running   0          90s
sqs-app-599d9dbfc6-8kn6c     1/1     Running   0          60s
sqs-app-599d9dbfc6-8s47z     1/1     Running   0          105s
sqs-app-599d9dbfc6-995f4     1/1     Running   0          90s
sqs-app-599d9dbfc6-9cp2q     1/1     Running   0          105s
sqs-app-599d9dbfc6-b7nzh     1/1     Running   0          60s
sqs-app-599d9dbfc6-bmm4f     1/1     Running   0          2m
sqs-app-599d9dbfc6-bv799     1/1     Running   0          75s
sqs-app-599d9dbfc6-c54jg     1/1     Running   0          105s
sqs-app-599d9dbfc6-c6dqj     1/1     Running   0          105s
sqs-app-599d9dbfc6-d9zt2     1/1     Running   0          2m
sqs-app-599d9dbfc6-db77r     1/1     Running   0          2m30s
sqs-app-599d9dbfc6-dgm96     1/1     Running   0          105s
sqs-app-599d9dbfc6-fm7ts     1/1     Running   0          2m
sqs-app-599d9dbfc6-fxml2     1/1     Running   0          90s
sqs-app-599d9dbfc6-g2mb5     1/1     Running   0          75s
sqs-app-599d9dbfc6-gncwj     1/1     Running   0          90s
```

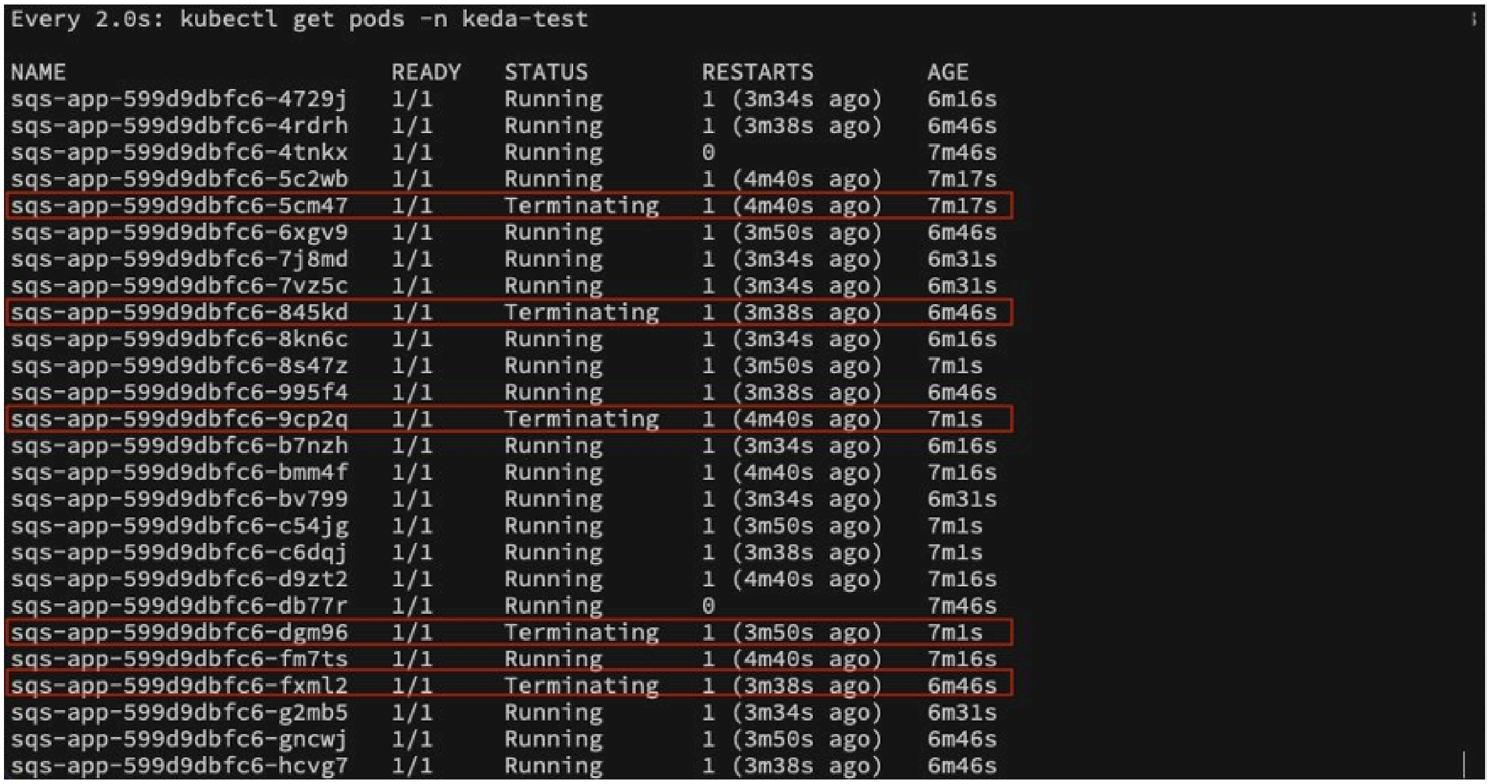


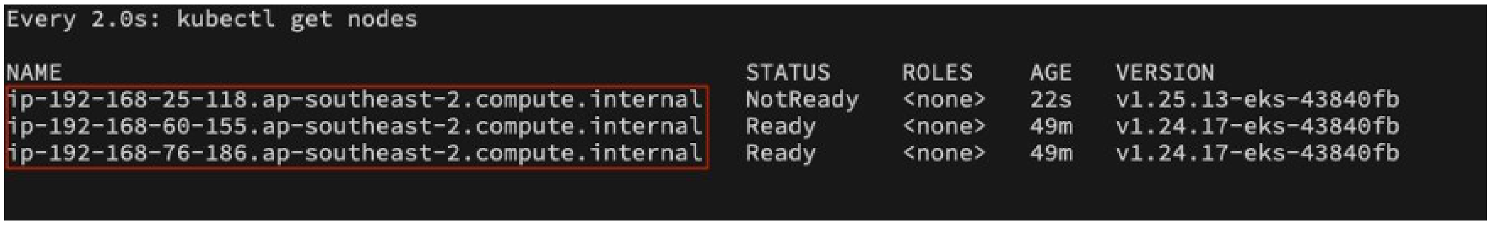
```
Every 2.0s: kubectl get nodes

NAME                                               STATUS   ROLES    AGE   VERSION
ip-192-168-122-213.ap-southeast-2.compute.internal   Ready    <none>   23s   v1.25.13-eks-43840fb
ip-192-168-125-246.ap-southeast-2.compute.internal   Ready    <none>   73s   v1.25.13-eks-43840fb
ip-192-168-21-75.ap-southeast-2.compute.internal     Ready    <none>   44s   v1.25.13-eks-43840fb
ip-192-168-60-155.ap-southeast-2.compute.internal    Ready    <none>   38m   v1.24.17-eks-43840fb
ip-192-168-7-135.ap-southeast-2.compute.internal     Ready    <none>   15s   v1.25.13-eks-43840fb
ip-192-168-76-186.ap-southeast-2.compute.internal    Ready    <none>   38m   v1.24.17-eks-43840fb
```

Scale-out and scale-in

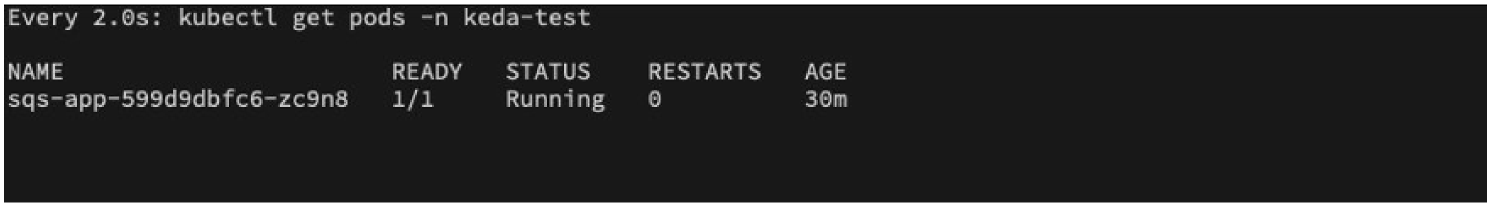
The scaling journey commences at the pod level (i.e., orchestrated by KEDA) based on the backlog of messages awaiting processing, and extends to Karpenter to ensure ample nodes are available to host the pods provisioned by KEDA. The scaling-in process mirrors this pattern. As KEDA notices the pending tasks (in this instance, Amazon SQS) nearing the desired queue depth, it begins a controlled termination of the provisioned pods. The elegance of this setup lies in KEDA's foresight—it doesn't delay until the exact queue depth is achieved, but rather proactively gauges the optimal timing for pod termination, rendering it highly resource-efficient.

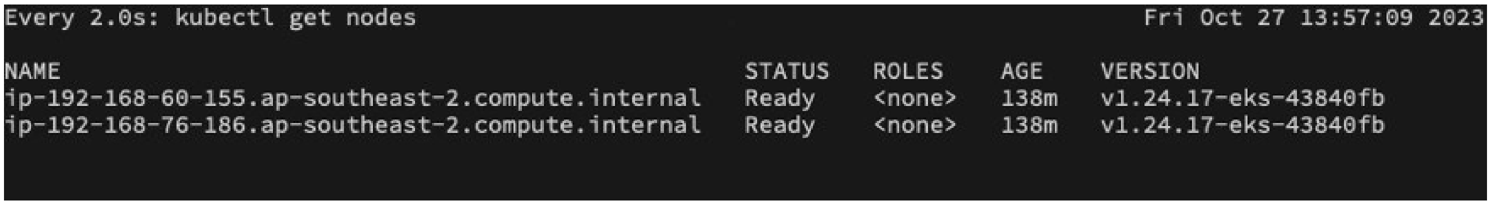
As Pods begin to wind down, Karpenter keeps a close eye on the resource utilization across nodes within the cluster. It can shut down underutilized nodes, minimizing resource squandering and trimming operational costs.


```
Every 2.0s: kubectl get pods -n keda-test

NAME                        READY   STATUS        RESTARTS        AGE
sqs-app-599d9dbfc6-4729j    1/1     Running       1 (3m34s ago)   6m16s
sqs-app-599d9dbfc6-4rdrh    1/1     Running       1 (3m38s ago)   6m46s
sqs-app-599d9dbfc6-4tnkx    1/1     Running       0               7m46s
sqs-app-599d9dbfc6-5c2wb    1/1     Running       1 (4m40s ago)   7m17s
sqs-app-599d9dbfc6-5cm47    1/1     Terminating   1 (4m40s ago)   7m17s
sqs-app-599d9dbfc6-6xgv9    1/1     Running       1 (3m50s ago)   6m46s
sqs-app-599d9dbfc6-7j8md    1/1     Running       1 (3m34s ago)   6m31s
sqs-app-599d9dbfc6-7vz5c    1/1     Running       1 (3m34s ago)   6m31s
sqs-app-599d9dbfc6-845kd    1/1     Terminating   1 (3m38s ago)   6m46s
sqs-app-599d9dbfc6-8kn6c    1/1     Running       1 (3m34s ago)   6m16s
sqs-app-599d9dbfc6-8s47z    1/1     Running       1 (3m50s ago)   7m1s
sqs-app-599d9dbfc6-995f4    1/1     Running       1 (3m38s ago)   6m46s
sqs-app-599d9dbfc6-9cp2q    1/1     Terminating   1 (4m40s ago)   7m1s
sqs-app-599d9dbfc6-b7nzh    1/1     Running       1 (3m34s ago)   6m16s
sqs-app-599d9dbfc6-bmm4f    1/1     Running       1 (4m40s ago)   7m16s
sqs-app-599d9dbfc6-bv799    1/1     Running       1 (3m34s ago)   6m31s
sqs-app-599d9dbfc6-c54jg    1/1     Running       1 (3m50s ago)   7m1s
sqs-app-599d9dbfc6-c6dqj    1/1     Running       1 (3m38s ago)   7m1s
sqs-app-599d9dbfc6-d9zt2    1/1     Running       1 (4m40s ago)   7m16s
sqs-app-599d9dbfc6-db77r    1/1     Running       0               7m46s
sqs-app-599d9dbfc6-dgm96    1/1     Terminating   1 (3m50s ago)   7m1s
sqs-app-599d9dbfc6-fm7ts    1/1     Running       1 (4m40s ago)   7m16s
sqs-app-599d9dbfc6-fxml2    1/1     Terminating   1 (3m38s ago)   6m46s
sqs-app-599d9dbfc6-g2mb5    1/1     Running       1 (3m34s ago)   6m31s
sqs-app-599d9dbfc6-gncwj    1/1     Running       1 (3m50s ago)   6m46s
sqs-app-599d9dbfc6-hcvg7    1/1     Running       1 (3m38s ago)   6m46s
```


```
Every 2.0s: kubectl get nodes

NAME                                               STATUS     ROLES    AGE   VERSION
ip-192-168-25-118.ap-southeast-2.compute.internal  NotReady   <none>   22s   v1.25.13-eks-43840fb
ip-192-168-60-155.ap-southeast-2.compute.internal  Ready      <none>   49m   v1.24.17-eks-43840fb
ip-192-168-76-186.ap-southeast-2.compute.internal  Ready      <none>   49m   v1.24.17-eks-43840fb
```

Back default state, all Karpenter instance are terminated post scale event


```
Every 2.0s: kubectl get pods -n keda-test

NAME                        READY   STATUS    RESTARTS   AGE
sqs-app-599d9dbfc6-zc9n8    1/1     Running   0          30m
```


```
Every 2.0s: kubectl get nodes                                   Fri Oct 27 13:57:09 2023

NAME                                               STATUS   ROLES    AGE    VERSION
ip-192-168-60-155.ap-southeast-2.compute.internal  Ready    <none>   138m   v1.24.17-eks-43840fb
ip-192-168-76-186.ap-southeast-2.compute.internal  Ready    <none>   138m   v1.24.17-eks-43840fb
```

Cleanup

Navigate to the root directory of the repository:

```
cd amazon-eks-scaling-with-keda-and-karpenter
```

To initiate cleanup:

```
sh ./cleanup.sh
```

Executing this command removes all AWS services and workloads established for this solution.

Conclusion

In this post, we showed you running event-driven workloads efficiently on Amazon EKS is substantially enhanced with the integration of KEDA and Karpenter. KEDA's meticulous autoscaling based on event metrics, coupled with Karpenter's timely node provisioning, creates a robust framework that excels in handling the dynamic nature of event-driven workloads. This synergy significantly bridges the scaling challenges, ensuring a responsive and cost-effective environment. The collaborative expertise of KEDA and Karpenter on Amazon EKS not only streamlines the management of event-driven workloads but also paves the way for a scalable, resource-efficient, and high-performing cloud-native ecosystem. Through this integration, businesses can effectively manage event-driven workloads, optimizing resource utilization while ensuring superior performance and cost effectiveness.

Additionally, stay tuned for upcoming posts focusing on various architecture patterns for running Event-Driven Architecture (EDA) workloads on EKS using KEDA. These forthcoming insights will further explore and illuminate the potential and versatility of this powerful integration in cloud-native environments.

Resources

For a deeper dive into KEDA and Karpenter, the following resources are highly recommended:

- Code repositoryshowcased in this post
- Watch this Youtube recording of the entire demo
- KEDA – Kubernetes-based Event Driven Autoscaling
- KEDA Scalers
- Karpenter – Kubernetes Node Autoscaler
- Karpenter Concepts

TAGS: Amazon EKS, Amazon SQS, Karpenter