AWS Database Blog

Amazon DocumentDB (with MongoDB compatibility) read autoscaling

by Randy DeFauw | on 16 DEC 2020 | in Amazon DocumentDB | Permalink | Comments | Share

Amazon Document DB (with MongoDB compatibility) is a fast, scalable, highly available, and fully managed document database service that supports MongoDB workloads. Its architecture supports up to 15 read replicas, so applications that connect as a replica set can use driver read preference settings to direct reads to replicas for horizontal read scaling. Moreover, as read replicas are added or removed, the drivers adjust to automatically spread the load over the current read replicas, allowing for seamless scaling.

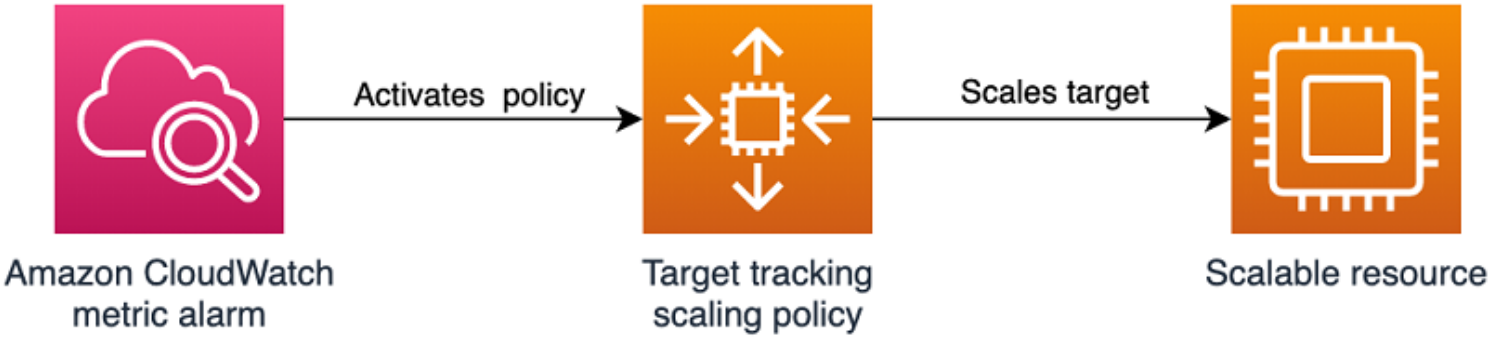
Amazon DocumentDB separates storage and compute, so adding and removing read replicas is fast and easy regardless of how much data is stored in the cluster. Unlike other distributed databases, you don't need to copy data to new read replicas. Although you can use the Amazon DocumentDB console, API, or AWS Command Line Interface (AWS CLI) to add and remove read replicas manually, it's possible to automatically change the number of read replicas to adapt to changing workloads.

In this post, I describe how to use Application Auto Scaling to automatically add or remove read replicas based on cluster load. I also demonstrate how this system works by modifying the load on a cluster and observing how the number of read replicas change. The process includes three steps:

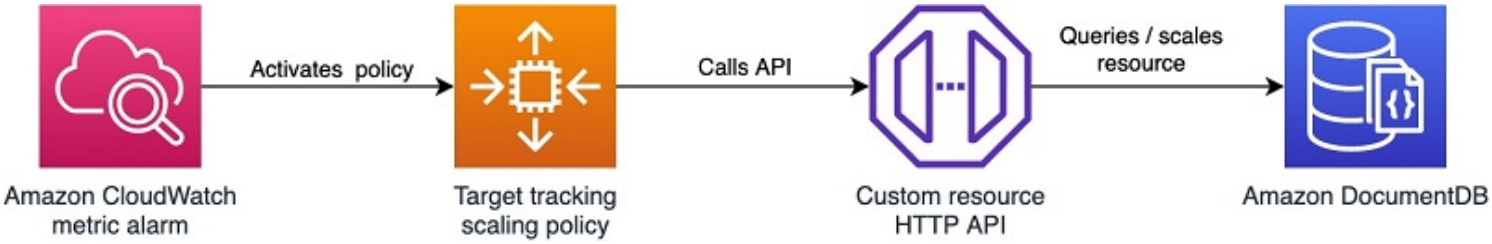
1. Deploy an Amazon DocumentDB cluster and required autoscaling resources.
2. Generate load on the Amazon DocumentDB cluster to trigger a scaling event.
3. Monitor cluster performance as read scaling occurs.

Solution overview

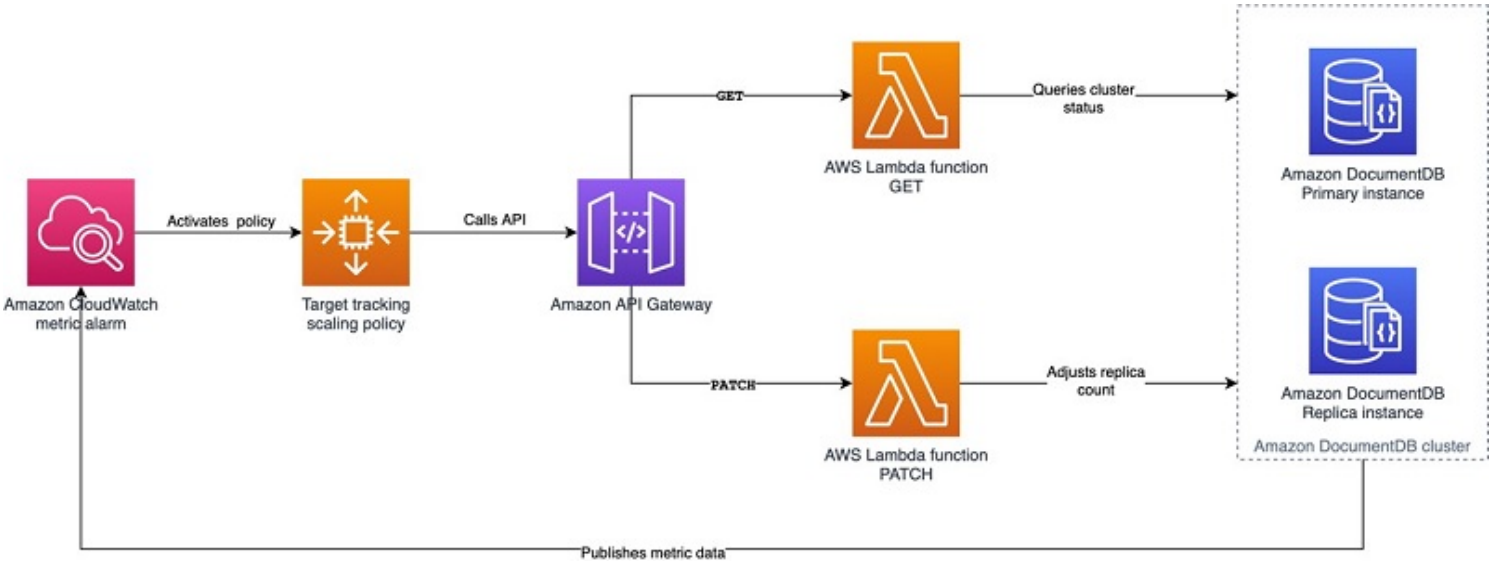
Application Auto Scaling allows you to automatically scale AWS resources based on the value of an Amazon CloudWatch metric, using an approach called *target tracking scaling*. Target tracking scaling uses a scaling policy to define which CloudWatch metric to track, and the AWS resource to scale, called the *scalable target*. When you register a target tracking scaling policy, Application Auto Scaling automatically creates the required CloudWatch metric alarms and manages the scalable target according to the policy definition. The following diagram illustrates this architecture.



Application Auto Scaling manages many different AWS services natively, but as of this writing, Amazon DocumentDB is not included among these. However, you can still define an Amazon DocumentDB cluster as a scalable target by creating an Auto Scaling custom resource, which allows our target tracking policy to manage an Amazon DocumentDB cluster's configuration through a custom HTTP API. This API enables the Application Auto Scaling service to query and modify a resource. The following diagram illustrates this architecture.



We create the custom HTTP API with two AWS services: Amazon API Gateway and AWS Lambda. API Gateway provides the HTTP endpoint, and two Lambda functions enable Application Auto Scaling to discover the current number of read replicas, and increase or decrease the number of read replicas. One Lambda function handles the status query (a GET operation), and the other handles adjusting the number of replicas (a PATCH operation). Our complete architecture looks like the following diagram.



Required infrastructure

Before we try out Amazon DocumentDB read autoscaling, we create an AWS CloudFormation stack that deploys the following infrastructure:

- An Amazon Virtual Private Cloud (VPC) with two public and two private subnets to host our Amazon DocumentDB cluster and other resources.
- An Amazon DocumentDB cluster consisting of one write and two read replicas, all of size db.r5.large.
- A jump host (Amazon Elastic Compute Cloud (Amazon EC2)) that we use to run the load test. It lives in a private subnet and we access it via AWS Systems Manager Session Manager, so we don't need to manage SSH keys or security groups to connect.

- The autoscaler, which consists of a REST API backed by two Lambda functions.
- A preconfigured CloudWatch dashboard with a set of useful charts for monitoring the Amazon DocumentDB write and read replicas.

1. Start by cloning the autoscaler code from its Git repository.
2. Navigate to that directory.

Although you can create the stack on the AWS CloudFormation console, I've provided a script in the repository to make the creation process easier.

3. Create an Amazon Simple Storage Service (Amazon S3) bucket to hold the CloudFormation templates:

```
aws s3 mb s3://<bucket>
```

4. On the Amazon S3 console, enable versioning for the bucket.

We use versions to help distinguish new versions of the Lambda deployment packages.

5. Run a script to create deployment packages for our Lambda functions:

```
./scripts/zip-lambda.sh <bucket>
```

6. Invoke the create.sh script, passing in several parameters. The template prefix is the folder in the S3 bucket where we store the Cloud Formation templates.

```
./scripts/create.sh <bucket> <template prefix> <database password> <stack name> <region>
```

For example, see the following code:

```
./scripts/create.sh <bucket> cfn PrimaryPassword docdbautoscale us-east-1
```

The Region should be the same Region in which the S3 bucket was created. If you need to update the stack, pass in –update as the last argument.

Now you wait for the stack to create.

7. When the stack is complete, on the AWS CloudFormation console, note the following values on the stack **Outputs** tab:
    a. DBClusterIdentifier
    b. DashboardName
    c. DbEndpoint
    d. DbUser
    e. JumpHost
    f. VpcId

g. ApiEndpoint

When we refer to these later on, they appear in brackets, like *<ApiEndpoint>*

8. Also note your AWS Region and account number.
9. Register the autoscaler:

```
cd scripts
python register.py <ApiEndpoint> <Region> <DbClusterIdentifier> <Account>
```

Autoscaler design

The autoscaler implements the custom resource scaling pattern from the Application Auto Scaling service. In this pattern, we have a REST API that offers a GET method to obtain the status of the resource we want to scale, and a PATCH method that updates the resource.

The GET method

The Lambda function that implements the GET method takes an Amazon DocumentDB cluster identifier as input and returns information about the desired and actual number of read replicas. The function first retrieves the current value of the desired replica count, which we store in the Systems Manager Parameter Store:

```
param_name = "DesiredSize-" + cluster_id
r = ssm.get_parameter( Name= param_name)
desired_count = int(r['Parameter']['Value'])
```

Next, the function queries Amazon DocumentDB for information about the read replicas in the cluster:

```
r = docdb.describe_db_clusters( DBClusterIdentifier=cluster_id)
                cluster_info = r['DBClusters'][0]
                readers = []
                for member in cluster_info['DBClusterMembers']:
                    member_id = member['DBInstanceIdentifier']
                    member_type = member['IsClusterWriter']

                    if member_type == False:
                        readers.append(member_id)
```

It interrogates Amazon DocumentDB for information about the status of each of the replicas. That lets us know if a scaling action is ongoing (a new read replica is creating). See the following code:

```
r = docdb.describe_db_instances(Filters=[{'Name':'db-cluster-id','Values': [cluster_id]
                instances = r['DBInstances']
                desired_count = len(instances) - 1
                num_available = 0
                num_pending = 0
                num_failed = 0
                for i in instances:
                    instance_id = i['DBInstanceIdentifier']
                    if instance_id in readers:
                        instance_status = i['DBInstanceStatus']
                        if instance_status == 'available':
                            num_available = num_available + 1
                        if instance_status in ['creating', 'deleting', 'starting', 'st
                            num_pending = num_pending + 1
                        if instance_status == 'failed':
                            num_failed = num_failed + 1
```

Finally, it returns information about the current and desired number of replicas:

```
responseBody = {
                    "actualCapacity": float(num_available),
                    "desiredCapacity": float(desired_count),
                    "dimensionName": cluster_id,
                    "resourceName": cluster_id,
                    "scalableTargetDimensionId": cluster_id,
                    "scalingStatus": scalingStatus,
                    "version": "1.0"
                }
                response = {
                    'statusCode': 200,
                    'body': json.dumps(responseBody)
                }
                return response
```

The PATCH method

The Lambda function that handles a PATCH request takes the desired number of read replicas as input:

```
{"desiredCapacity":2.0}
```

The function uses the Amazon DocumentDB Python API to gather information about the current state of the cluster, and if a scaling action is required, it adds or removes a replica. When adding a replica, it uses the same settings as the other replicas in the cluster and lets Amazon DocumentDB choose an Availability Zone automatically. When removing replicas, it chooses the Availability Zone that has the most replicas available. See the following code:

```
# readers variable was initialized earlier to a list of the read
# replicas.  reader_type and reader_engine were copied from
# another replica.  desired_count is essentially the same as
# desiredCapacity.
if scalingStatus == 'Pending':
    print("Initiating scaling actions on cluster {0} since actual count {1} does no
    if num_available < desired_count:
        num_to_create = desired_count - num_available
        for idx in range(num_to_create):
            docdb.create_db_instance(
                 DBInstanceIdentifier=readers[0] + '-' + str(idx) + '-' + str(int(t
                 DBInstanceClass=reader_type,
                 Engine=reader_engine,
                 DBClusterIdentifier=cluster_id
            )
else:
    num_to_remove = num_available - desired_count
    for idx in range(num_to_remove):
```

We also store the latest desired replica count in the Parameter Store:

```
r = ssm.put_parameter(
            Name=param_name,
            Value=str(desired_count),
            Type='String',
            Overwrite=True,
            AllowedPattern='^d+$'
        )
```

Defining the scaling target and scaling policy

We use the boto3 API to register the scaling target. The `MinCapacity` and `MaxCapacity` are set to 2 and 15 in the scaling target, because we always want at least two read replicas, and 15 is the maximum number of read replicas. The following is the relevant snippet from the registration script:

```
# client is the docdb boto3 client
response = client.register_scalable_target(
    ServiceNamespace='custom-resource',
    ResourceId='https://' + ApiEndpoint + '.execute-api.' + Region + '.amazonaws.com/p
    ScalableDimension='custom-resource:ResourceType:Property',
    MinCapacity=2,
    MaxCapacity=15,
    RoleARN='arn:aws:iam::' + Account + ':role/aws-service-role/custom-resource.applic
)
```

The script also creates the autoscaling policy. There are several important configuration parameters in this policy. I selected CPU utilization on the read replicas as the target metric (CPU utilization is not necessarily the best metric for your workload's scaling trigger; other options such as `BufferCacheHitRatio` may provide better behavior). I set the target value at an artificially low value of 5% to more easily trigger a scaling event (a more realistic value for a production workload is 70–80%). I also set a long cooldown period of 10 minutes for both scale-in and scale-out to avoid having replicas added or removed too frequently. You need to determine the cooldown periods that are most appropriate for your production workload. The following is the relevant snippet from the script:

```
response = client.put_scaling_policy(
    PolicyName='docdbscalingpolicy',
    ServiceNamespace='custom-resource',
    ResourceId='https://' + ApiEndpoint + '.execute-api.' + Region + '.amazonaws.co
    ScalableDimension='custom-resource:ResourceType:Property',
    PolicyType='TargetTrackingScaling',
    TargetTrackingScalingPolicyConfiguration={
        'TargetValue': 5.0,
        'CustomizedMetricSpecification': {
            'MetricName': 'CPUUtilization',
            'Namespace': 'AWS/DocDB',
            'Dimensions': [
                {
                    'Name': 'Role',
                    'Value': 'READER'
                },
                {
                    'Name': 'DBClusterIdentifier'
```

Generating load

I use the YCSB framework to generate load. Complete the following steps:

1. Connect to the jump host using Session Manager:

```
aws ssm start-session --target <JumpHost>
```

2. Install YCSB:

```
    sudo su - ec2-user
sudo yum -y install java
curl -O --location https://github.com/brianfrankcooper/YCSB/releases/download/0.17
tar xfvz ycsb-0.17.0.tar.gz
cd ycsb-0.17.0
```

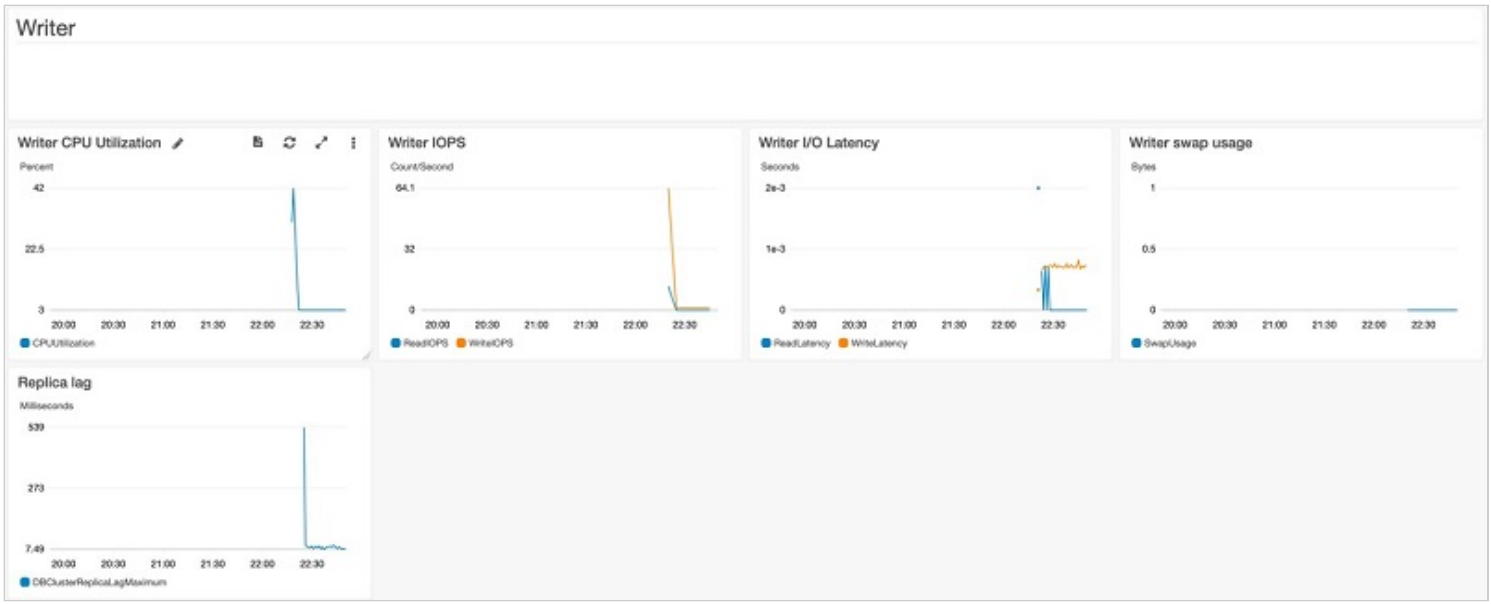
3. Run the load tester. We use workloadb, which is a read-heavy workload:

```
./bin/ycsb load mongodb -s -P workloads/workloadb -p recordcount=10000000 -p mor
./bin/ycsb run mongodb -threads 10 -target 100 -s -P workloads/workloadb -p recc
```

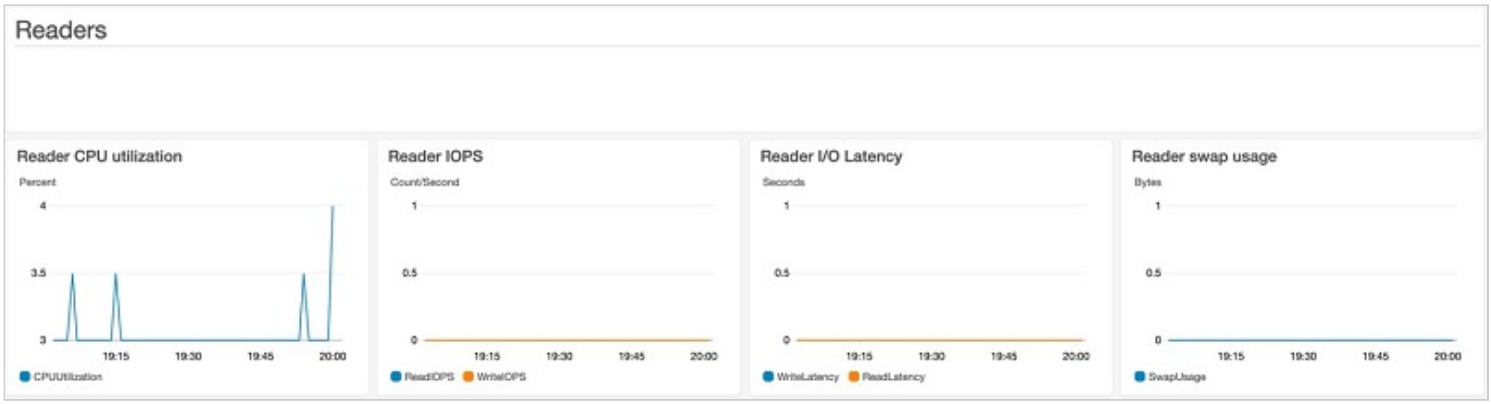
These two commands load data in the Amazon DocumentDB database and run a read-heavy workload using that data.

Monitoring scaling activity and cluster performance

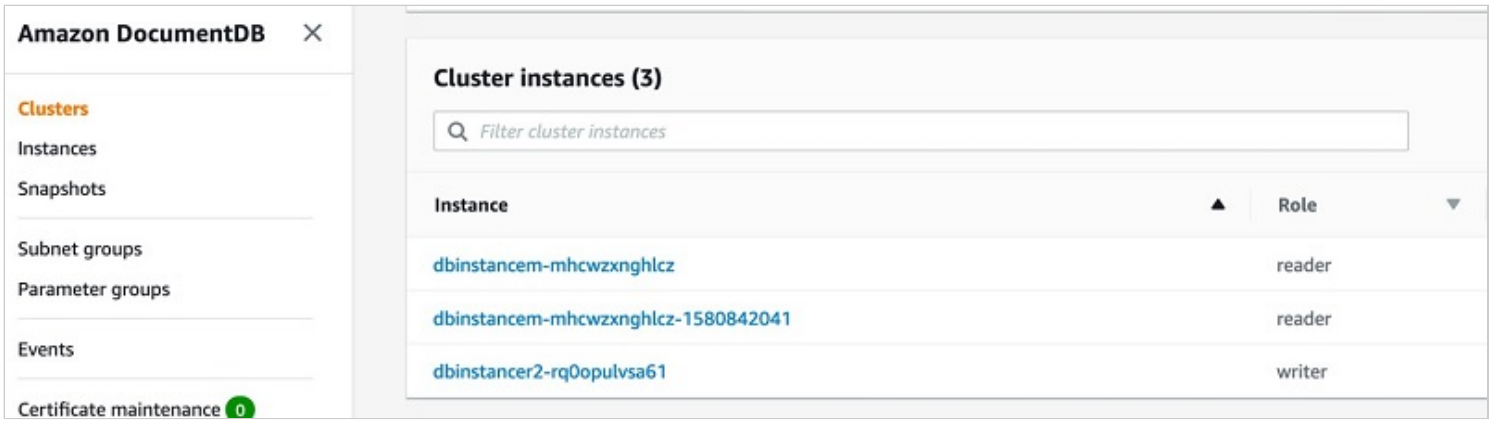
The CloudFormation stack created a CloudWatch dashboard that shows several metrics. The following screenshot shows the dashboard for the writer node.



The following screenshot shows the dashboard for the read replicas.



As YCSB runs, watch the dashboard to see the load increase. When the CPU load on the readers exceeds our 5% target, the autoscaler should add a read replica. We can verify that by checking the Amazon DocumentDB console and observing the number of instances in the cluster.



Cleaning up

If you deployed the CloudFormation templates used in this post, consider deleting the stack if you don't want to keep the resources.

Conclusion

In this post, I showed you how to use a custom Application Auto Scaling resource to automatically add or remove read replicas to an Amazon DocumentDB cluster, based on a specific performance metric and scaling policy.

Before using this approach in a production setting, you should decide which Amazon DocumentDB performance metric best reflects when your workload needs to scale in or scale out, determine the target value for that metric, and settle on a cooldown period that lets you respond to cluster load without adding or removing replicas too frequently. As a baseline, you could try a scaling policy that triggers a scale-up when `CPUUtilization` is over 70% or FreeableMemory is under 10%.

About the Author

Randy DeFauw is a principal solutions architect at Amazon Web Services. He works with the AWS customers to provide guidance and technical assistance on database projects, helping them improve the value of their solutions when using AWS.

TAGS: Amazon API Gateway, Amazon CloudWatch, Amazon DocumentDB (with MongoDB Compatibility), aws lambda

Comments