Beanstalk Worker Auto-Scaling

March 2, 2020

Michael Heyd, Solutions Architect

In this workshop, you will be creating a simple ElasticBeanstalk Worker that will pull messages from a pre-configured SQS Queue. The number of workers will initially scale based on the Number of messages in the Queue. We will then update the application so that scaling is based on a custom metric that we create. We will load the messages using a Lambda function that writes messages directly to the SQS Queue.

# Provision Elastic Beanstalk environment

1. Open a new console window and select Elastic Beanstalk
2. Click on Create New Application in the upper right corner
   1. Application Name: QueueDepthScaling
   2. Description: QueueDepthScaling
   3. Click Create

You should see the ‘No Environments currently exist for this application. Create one now’. We will now create a default Ruby environment. After it has been provisioned we will upload our custom code.

1. Select ‘Worker environment’ and click ‘Select’.
   1. Application name: queueDepthScaling
   2. Environment name: QueueDepthScaling-dev
   3. Description: environment for testing queuedepthscaling
   4. Platform: Ruby
   5. Application Code: Sample Application
   6. Create environment.

# Create Lambda for Populating SQS

It will take about 5 minutes to create the environment. While it is provisioning, clone the following github repo:

git clone https://github.com/heydaws/ruby-scaling.git

Next we need to create a lambda function that will post test messages to the Beanstalk’s SQS queue. We will start by creating the lambda and then updating it to communicate with our newly created SQS queue.

1. Select Lambda Console
2. Click ‘Create Lambda’
   1. Author from scratch
   2. Function Name: rew-populate-sqs
   3. Runtime: python 3.6
   4. Create!

The code for the function is in the git repo that you downloaded. Copy and paste the following function into the Lambda code window:

ruby-scaling \ generate-transactions \ lambda-populate-sqs.py

## Update with Environment Details

The code that you just copied needs to be updated with the Queue details from your Beanstalk environment.

1. Return to the Beanstalk console,
2. select the application, and the newly created environment. It should be called something like ‘Queuedepthscaling-env’.
3. Click on Configuration
4. Find the ‘Worker’ section and copy the Worker queue URL. It will be something like ‘https://sqs.us-west-2.amazonaws.com/997023692343/awseb-e-m2t8tupwsp-stack-AWSEBWorkerQueue-ARGFQCHK1RN3’



You will need to update the following variables. They should be around line 32.

Duration: 5 (this is number of minutes the Lambda function will post messages to the SQS queue

QueueURL: <this is the Worker queue URL that you just copied>

## Update Lambda Permissions

The default lambda role does not have access to write to SQS queue, so update the role.

1. Scroll down until you find the role and click on the link that says something like ‘View the rew-populate-sqs-role-xi7dwdwr role’. This will open an IAM console window where you will add the permissions to access SQS.
2. Click ‘Attach Policies’
   1. Type in ‘SQS’ and select ‘AmazonSQSFullAccess’. Remember this is a test account.
   2. Click AttachPolicy button at bottom of screen
3. In the upper-right, click ‘Save’

## Test the Lambda

We will now test the function and confirm that messages are being processed by our worker. In the upper-right corner of the Lambda console, there is a ‘test’ button. When you click it for the first time, you will need to define a test message. Accept the default.

You will need click ‘test’ again, the function will run for <Duration> minutes, and after completion you will get a message such as

|  |
| --- |
| Response:  {  "statusCode": 200,  "body": "Sent 108 messages"  } |

We can confirm that the worker is processing the messages. Return to the ElasticBeanstalk console and select your environment (Queuedepthscaling-env).

1. On the left-hand menu, select ‘Logs’
   1. Request Logs > Last 100 Lines
   2. In the table, click on ‘Download’ and the log file will open.
   3. Scroll down, pausing to read the deployment details, and you will get a list of text that looks like:

|  |
| --- |
| I, [2020-02-28T17:12:57.119631 #3483] INFO -- : Received message: "message 29"  I, [2020-02-28T17:12:57.611630 #3483] INFO -- : Received message: "message 30"  I, [2020-02-28T17:12:58.033897 #3483] INFO -- : Received message: "message 31"  I, [2020-02-28T17:12:58.501268 #3483] INFO -- : Received message: "message 32" |

# What have we accomplished?

To summarize, we have provisioned an elastic beanstalk environment that will process messages from an SQS queue. We have a lambda function that will populate the SQS queue with test messages and the Worker will pull the messages off the queue. This is a good start, but what happens when the worker can’t keep up with the messages being added to the Queue?

You could manually add them, but that is so 2019. The real answer is Automation. We will create an automated mechanism for adding new workers to process the backlog of messages.

The good news is that Elastic Beanstalk has a mechanism for scaling the workers using an Auto-Scale Group. The bad news is that the default metrics for scaling the workers does not contain a ‘number of messages in sqs queue’ metric. It contains metrics such as Network Out, Network In, CPU, …

However, we can extend the base functionality and use a CloudWatch metric for scaling the number of workers. This allows us to use a metric emitted by SQS Queues to scale the number of workers. Specifically, we will be using the SQS – ApproximateNumberofMessagesVisible metric. As you expect, this gives you the number of messages that are sitting in the queue waiting to be processed.

To achieve this goal, we will…

1. Use the console to enable worker scaling
2. Use .ebextensions to add scaling based on our CloudWatch metrics
3. Test and confirm that workers will scale up and scale down

# Update Beanstalk to scale workers

We will start by returning to our Beanstalk environment: Queuedepthscaling-env, and selecting ‘Configuration’ from the menu.

1. Find ‘Capacity’ and click ‘Modify’
   1. Environment Type: load balanced
   2. Accept defaults and scroll down until you see the ‘Apply’ button.

You will get a warning. ‘Confirm’ that you understand the details of what you are changing.

## CloudWatch Alarms

Beanstalk will update the environment by updating the autoscaling group and creating cloudwatch alarms for scaling based on Network Out. We can confirm the creation of these resources by

1. Opening a new tab and navigate to CloudWatch console.
2. Select Alarms, and filter by ‘awseb’. You will see something like…



You can see the two alerts that will monitor the network out from the beanstalk workers. You can click on the name and view a graph and other details of the alert. Remember that while you could manually update (and even delete) these resources, they will be auto-created the next time that you update your Beanstalk.

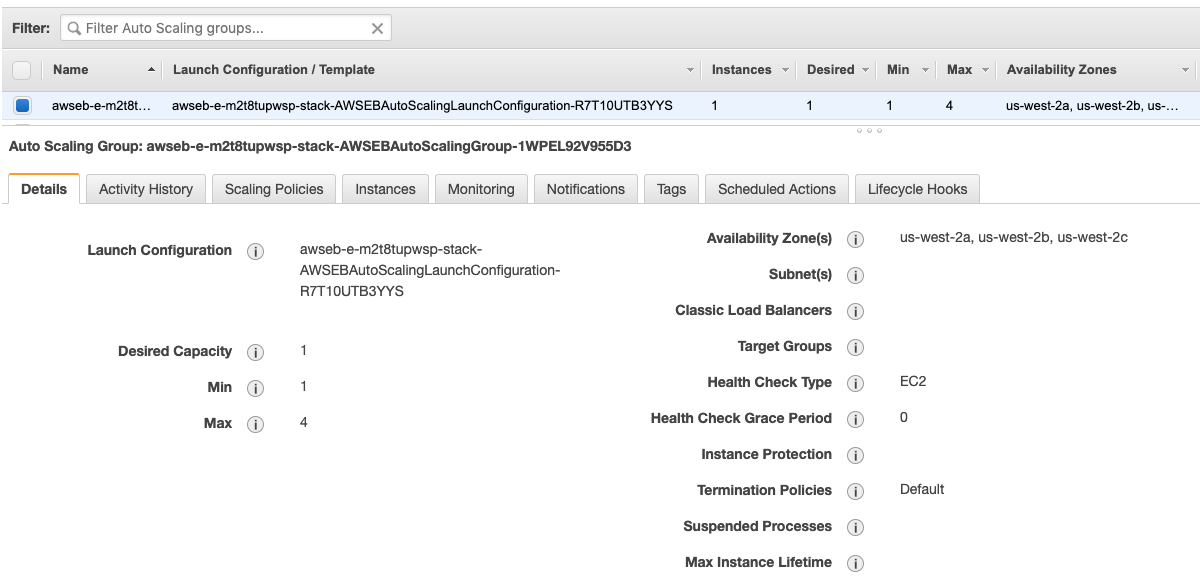
## AutoScaling Groups

In addition to the CloudWatch alerts, there was an Autoscaling group created.

Open a new tab and navigate to EC2 console.

Scroll down until you see Auto Scaling > Auto Scaling Groups

You will see something like



A bit of explanation for the different aspects of the AutoScaling Group (ASG):

|  |  |
| --- | --- |
| **Details**: | shows you the capacity settings and other attributes that were set by Beanstalk |
| **Activity Hist**: | Will show you the scaling acrtivity. There will be a single message to start. |
| **Scaling Policy**: | There will be two policies. One to scale up when network out is high and a second to scale down when network is low |
| **Instances**: | Just one for now, but as you scale, you will see other instances being provisioned here |
| **Monitoring** | Shows metrics associated with the scale group |
| **Notifications** | shows where notifications are sent and picked up by beanstalk for display on the beanstalk console |
| **Tags** | Shows the tags set by beanstalk |
| **Sched Action** | You can schedule based on a time schedule. You set the schedule in the beanstalk console, but this is where you see the values. |
| **Lifecycle hook** | We don’t use lifecycle hooks with elastic beanstalk |

# Scaling based on SQS Metrics

We have created a Beanstalk cluster that will scale based on the network activities of the workers. However, we want to scale based on the queue depth of the SQS Queue. We will now create a new deployment package and upload it to the beanstalk. Within that package, will be a series of configuration files that create the custom Cloud Watch alarms and associate them with the auto-scale group. It will also remove the default Alarms from auto-scaling.

The code we want is in the git repo, under the **ruby-puma-v3** directory. This is the sample application that we started from, but it has .ebextension files for configuration and it has added some computation to the worker to slow the processing of messages.

## ebextensions

A short word on .ebextensions. These allow you to specify the configuration of your beanstalk environment without using the AWS CLI or the Console. You can specify a number of default settings that are specific to Elastic Beanstalk. For additional functionality, you can use CloudFormation templates to provision resources and associate then with your Beanstalk environment. We use these CloudFormation templates to create our custom scaling alarms.

Find the code the you downloaded, and we will be looking at the .**ebextensions** folder. There are three files in this folder:

**00log\_file\_install.config** This specifies details on the log files. We will ignore this.

**Options.config**  This allows us to set values on the AutoScaling Group.

**Worker-scaling.config** This allows us to create the CloudWatch Alarms and associate them with the scaling policies on the AutoScaling Group

Taking a deeper dive on the options.config file:

**Cooldown**: Is the amount of time, in seconds, between scaling events

**MinSize**: The minimum number of EC2 instances. You can have this set to 0 which would reduce your cost when no messages are in the queue, but then you have a warm-up time before messages will be processed.

**MaxSize**: The maximum number of EC2 instances. You cap this to manage the trade-off between cost and performance.

And looking at the worker-scaling.config file, there are some Resources that are automatically provisioned by Elastic Beanstalk. This file references the following resources:

**AWSEBCloudwatchAlarmHigh** – This is the CloudWatch alarm auto-created by the system to monitor the NetworkOut. By setting the AlarmActions to ‘null’ we are ensuring it won’t scale our worker nodes.

**AWSEBCloudwatchAlarmLow** – CloudWatch alarm for scaling down based on NetworkOut. We are also setting the actions to ‘null’.

**QueueDepthAlarmHigh** – this is the new alarm for scaling up based on queue length. We are using a default SQS metric, but you can reference any other CloudWatch metric (including custom metrics).

**QueueDepthAlarmLow** – for scaling down the worker nodes

You can also take a look at the **worker\_sample.rb** to see how we are slowing the processing by calculating Fibonacci series for each message that is processed.

## Deploy to Beanstalk

Now that you are familiar with this app, it is time to package and deploy.

1. Zip the ‘ruby-puma-v3’ folder and all its contents. Take note of the location and name
2. In the beanstalk console, select your environment.
3. In the left-hand menu, select ‘Dashboard’
4. Click ‘Upload and Deploy’ in the middle of the screen
5. Select your zip file and add a version label
6. Beanstalk will upload the file and start re-configuring the environment.

Once you get the message ‘Environment Update completed successfully’, we can take a look at what was created.

**CloudWatch Console** – you can see the two new alarms, and note that the previous two alarms now have ‘no action’.

**AutoScaling Group** – you can see the max server =3 and default cooldown = 60

**AutoScaling Group** – you can see the scaling policies reference the two new cloudwatch alarms.

# Test Auto-Scaling

We are now ready to test the scaling. Remember the Lambda function that we created earlier? We you need to update the duration to 10 (minutes) and hit ‘test’. This will cause the lambda to start running and feeding messages into the SQS queue. As there are no workers running, the messages will sit in the queue and trigger the cloud watch alarm. This will in turn cause the autoscaling group to provision a server.

To watch this in action, you will want multiple browser tabs:

**Tab1** Elastic Beanstalk showing the messages when instances are added/removed

**Tab2** SQS console to watch the number of messages in the Queue

**Tab3** CloudWatch Alarms to view the details of the alarms and when they trigger

**Tab4** AutoScaling Group to view the History and Instances as they are scaled.

# Scale with Custom Metrics

Once the lambda has finished running, we can move onto the next task which is to create a custom metric in our lambda function and update our Beanstalk to scale based on that metric.

1. Select Lambda Console
2. Click ‘Create Lambda’
   1. Author from scratch
   2. Function Name: rew-create-custom-metric
   3. Runtime: python 3.6
   4. Create!

The code for the function is in the git repo that you downloaded. Copy and paste the following function into the Lambda code window:

ruby-scaling \ generate-transactions \ custom-metrics.py

The code that you just copied needs to be updated with the Queue details from your Beanstalk environment. Copy the values from your other Lambda function (or from Beanstalk).

You will notice that this code is pretty much the same as our previous Lambda function. However, we have added a new function that creates our custom metric:

|  |
| --- |
| MetricData = [  {  'MetricName': 'NumMessagesCreated',  'Dimensions': [  {  'Name': 'MessageProducer',  'Value': 'Lambda'  },  {  'Name': 'AppVersion',  'Value': '1.0'  },  ],  'Unit': 'None',  'Value': value  },  ],  Namespace='BeanstalkScaling' |

Some things to note here is the **namespace**. You want to use something that will easily identify your metrics and the second is the **MetricName**. Make sure you use understandable names for each of your metrics. You can provide further segmentation of your metrics with the use of the **Dimensions** attribute. All of these values will make much more sense once you have run the Lambda and can view the metric in Cloud Watch.

## Lambda Permissions

As you recall, we need to update the Lambda permissions. Scroll down until you find the role and click on the link that says something like ‘View the rew-populate-sqs-role-xi7dwdwr role’. This will open an IAM console window where you will add the permissions to access SQS.

1. Click ‘Attach Policies’
2. Type in ‘SQS’ and select ‘AmazonSQSFullAccess’. Remember this is a test account.
3. Click AttachPolicy button at bottom of screen
4. Type in ‘CloudWatch’ and select ‘CloudWatchFullAccess’.
5. Click AttachPolicy

In the Lambda window, in the upper-right, click ‘Save’

## Test the Lambda

Click ‘Test’. For the first time, you will need to define a test message. Just accept the default. After you click ‘test’, the function will run for <Duration> minutes, and after completion you will get a success such as

|  |
| --- |
| Response:  {  "statusCode": 200,  "body": "Sent 108 messages"  } |

# Custom Cloud Watch Metrics

After successful completion, you can look at the number of messages in the queue and also watch your workers scale up to process the messages. Alternately, you can open the CloudWatch console and take a look at your metric.

1. Click on ‘metrics’ in the left-hand menu
2. You should get a blank graph and a list of 1,608 metrics available in CloudWatch.
3. Search for ‘Beanstalkscaling’ and you should see a number of namespaces related to beanstalks.
4. Select ‘BeanstalkScaling > AppVersion, MessageProd…’ and you will get the metric that you just created.
5. Select the metric and it will be graphed.

# Update Beanstalk

Now that we have our custom metric, we will update our Beanstalk to scale based on the metric. Return to the Ruby code, open the ‘worker-scaling.config’ file, and replace with the following code:

|  |
| --- |
| Resources:  AWSEBCloudwatchAlarmHigh:  Type: AWS::CloudWatch::Alarm  Properties:  AlarmActions: []  AWSEBCloudwatchAlarmLow:  Type: AWS::CloudWatch::Alarm  Properties:  AlarmActions: []  QueueDepthAlarmHigh:  Type: AWS::CloudWatch::Alarm  Properties:  AlarmDescription: "Alarm if queue depth grows beyond 20 messages"  Namespace: "BeanstalkScaling"  MetricName: NumMessagesCreated  Dimensions:  - Name: MessageSource  Value: Lambda  Statistic: Sum  Period: 60  EvaluationPeriods: 1  Threshold: 20  ComparisonOperator: GreaterThanThreshold  TreatMissingData: notBreaching  AlarmActions:  - Ref: AWSEBAutoScalingScaleUpPolicy  QueueDepthAlarmLow:  Type: AWS::CloudWatch::Alarm  Properties:  AlarmDescription: "Alarm if queue depth is less than 5 messages"  Namespace: "BeanstalkScaling"  MetricName: NumMessagesCreated  Dimensions:  - Name: MessageSource  Value: Lambda  Statistic: Sum  Period: 60  EvaluationPeriods: 1  Threshold: 5  ComparisonOperator: LessThanThreshold  TreatMissingData: breaching  AlarmActions:  - Ref: AWSEBAutoScalingScaleDownPolicy |

You can see the changes for the following attributes:

**Namespace**: BeanstalkScaling

**MetricName**: NumMessagesCreated

**Dimensions**: MessageSource – Lambda

Save the file, and create a zip file for deployment

Find your beanstalk environment and deploy this newest file.

Once beanstalk has finished deployment, you can start the lambda function.

As the lambda is creating messages, you can look at the following moving pieces:

**SQS**: watch the number of messages in the queue

**CloudWatch**: take a look at the custom metric for NumMessagesCreated

**CloudWatch**: take a look at the alarms

**ASG**: Watch the instances and history

# Congratulations!

Congratulations, you have explored how to scale Beanstalk Workers using custom metrics. These metrics can be emitted by any application and then used by Beanstalk as primary scaling mechanism. Please also take note that the principles presented here apply outside Beanstalk. Any application can emit custom metrics and these metrics observed through Cloud Watch. In addition, AutoScale Groups are available for your EC2 instances and so you can scale them using CloudWatch Alarms. Beanstalk provides a wrapper and abstracts some of the complexity around you doing it yourself.