**­**

**Lambda Automation Basics Activity Guide**

We’ll be using AWS remote environments for our labs. Every student MUST HAVE AN AWS ACCOUNT CREATED IF YOU’D LIKE TO FOLLOW ALONG IN ACTIVITIES DURING CLASS.

*NOTE: The code you’ll use in this class is located at .*[*https://github.com/rich-morrow/autom8d-foundations*](https://github.com/rich-morrow/autom8d-foundations)

*You can install with:*

git clone https://github.com/rich-morrow/autom8d-foundations.git

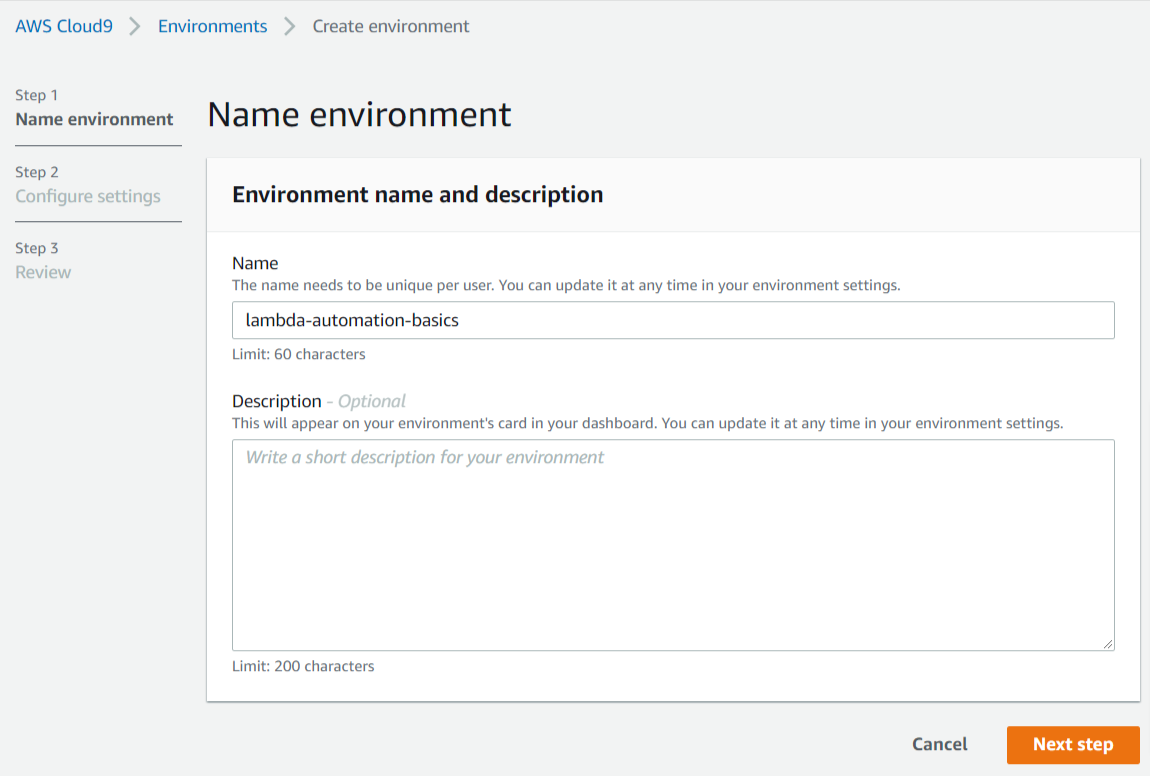
*Also helpful to know:*

* *For internet browser, we strongly recommend using Google Chrome or Firefox. The AWS web gui which we’ll use extensively doesn’t play Microsoft’s game of “write a completely different version of your website to deal with MS’ inability to abide by internet standards like Javascript”, so avoid IE, Edge, or whatever Microsoft is calling their crappy browser today.*
* *In Linux (which we’ll use exclusively), you can “Tab Complete” long filenames. For example, if I had a directory named “i-am-a-super-long-hard-to-type-directory”, I can cd into it by typing “cd i-am-a” then just press tab. If there are files or directories that match those first few characters, linux will list them out for you, and if there’s only a single match, it will automatically fill in the remaining characters for you. Carpal Tunnel BEGONE!!!*
* *Commands you’re asked to enter on the terminal are formatted as so in this document:*

cd ~/Desktop

**Activity #1 (Real world Lambda function in Cloud9 IDE)**

Log into your AWS console and browse to the [Cloud9 area](https://console.aws.amazon.com/cloud9/home/create) (we use us-east-1, but you can switch Regions if you like):



NOTE: As of 2 May, 2022, these are all the default settings.

Environment Name: lambda-automation-basics

Environment type: “Create a new EC2 instance for environment (direct access)

Instance type: “t2.micro”

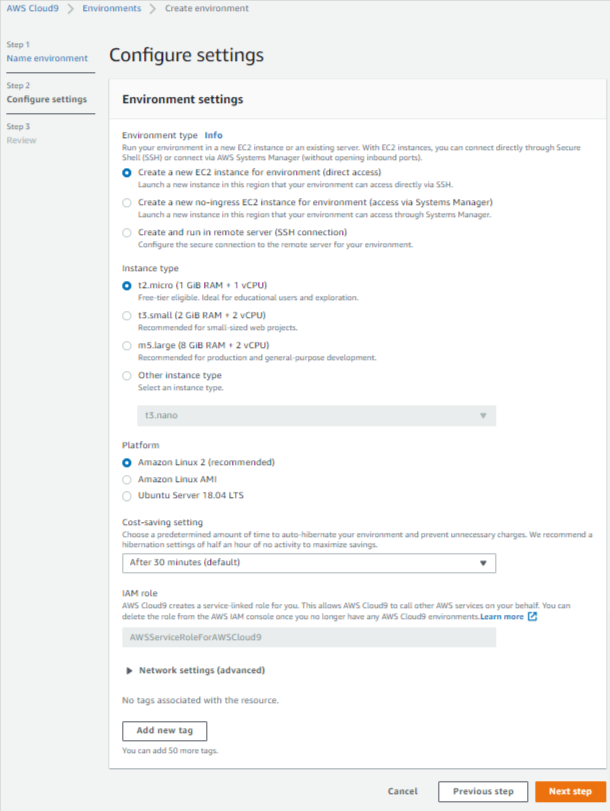
Platform: “Amazon Linux 2”

Cost-saving setting: “After 30 minutes”

IAM Role: “AWSServiceRoleForAWSCloud9”

Click “Next Steps” until you get to “Review” page,

then click “Create Environment”. You will see messages like “We are creating your AWS Cloud9 environment. This can take a few minutes”, then “Connecting”. When your environment is complete, you’ll see a page that says “Welcome to your development environment”.



**Git clone the code locally**

At the bottom of the window, you’ll see a terminal. In the terminal, enter:

git clone <https://github.com/rich-morrow/autom8d-foundations.git>

#then cd into the codebase, activity1

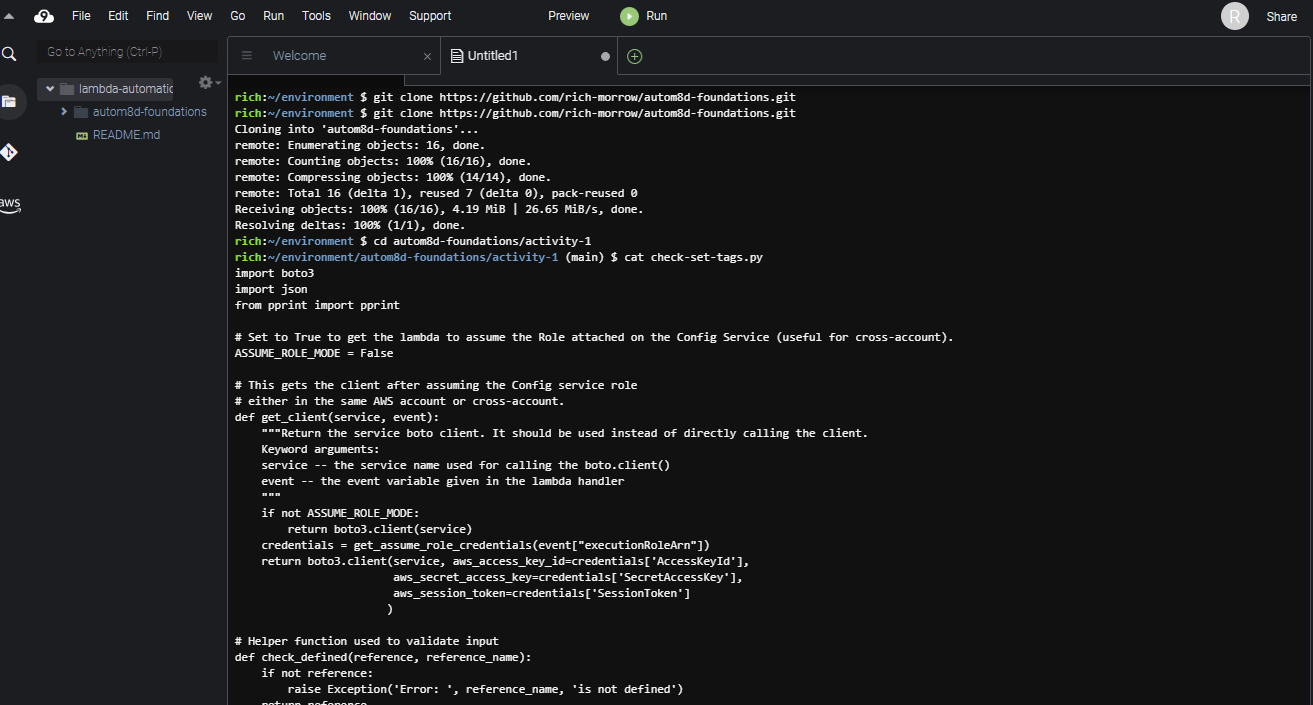
cd autom8d-foundations/activity-1

#finally, cat the code out:

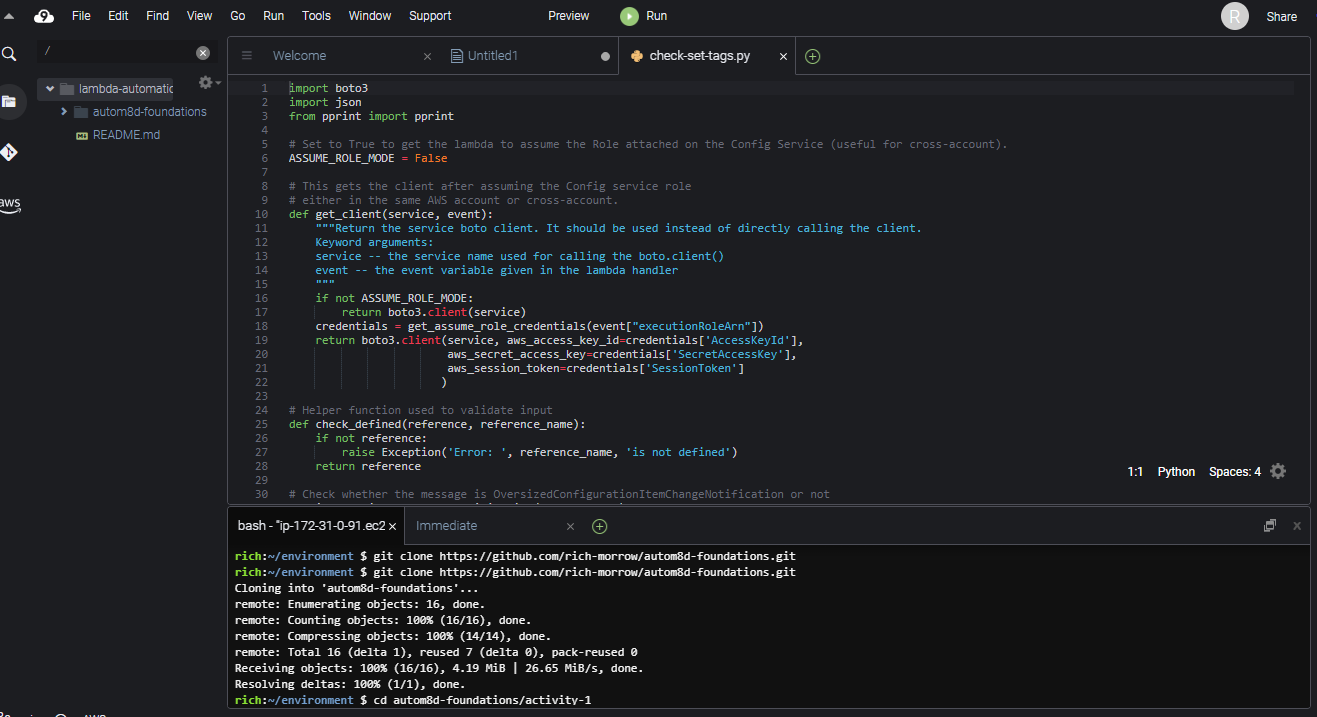
cat check-set-tags.py

**Browse the code**

Grab the terminal divider (double lines just above the terminal) and drag it to a larger view so it looks something like this:

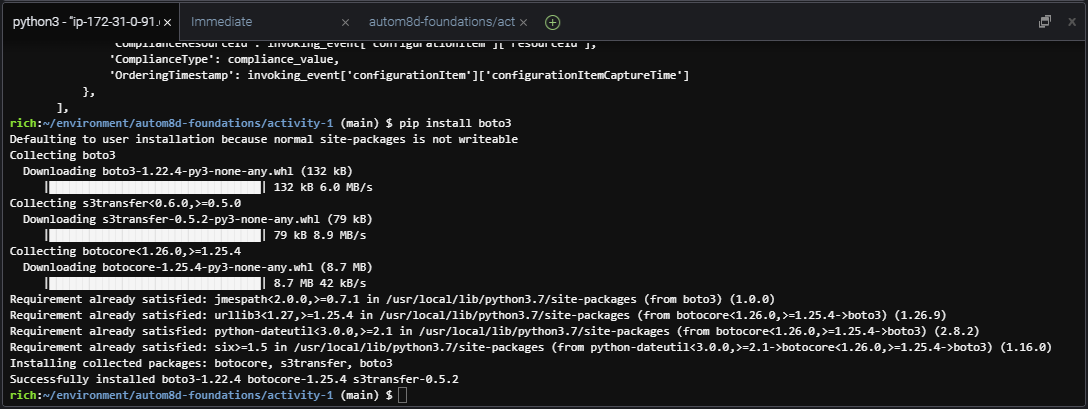


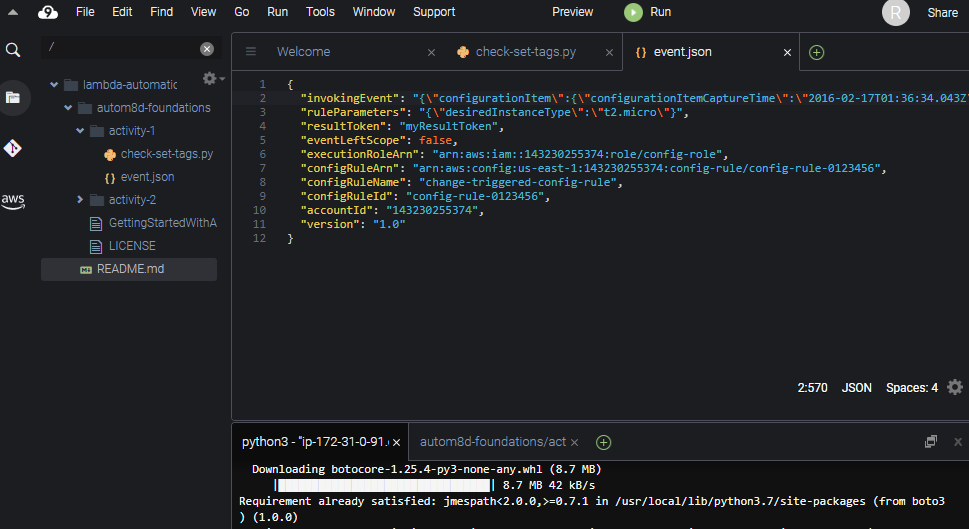
This is basically the code we’ll use for activity #2. For now, don’t worry too much about what it does… let’s open it in Cloud9 by choosing “File→Open”, then selecting “activity-1/check-set-tags.py”. Drag your terminal view down smaller so you can better see the code. As shown below:



You can attempt to run the code directly by pushing the green “play” button named “Run” at page top, but it will fail as boto3 has not been loaded into our Cloud9 environment. Let’s fix that by doing a pip install in the terminal at page bottom:

pip install boto3

Your install should look something like this:



Expand autom8d-foundations→activity-1 in the left hand navigation. Double click the “event.json” object to open it in a new tab. This event object is a mock of what would be passed to the Lambda function in a real, live execution. We can alter it later on if we choose

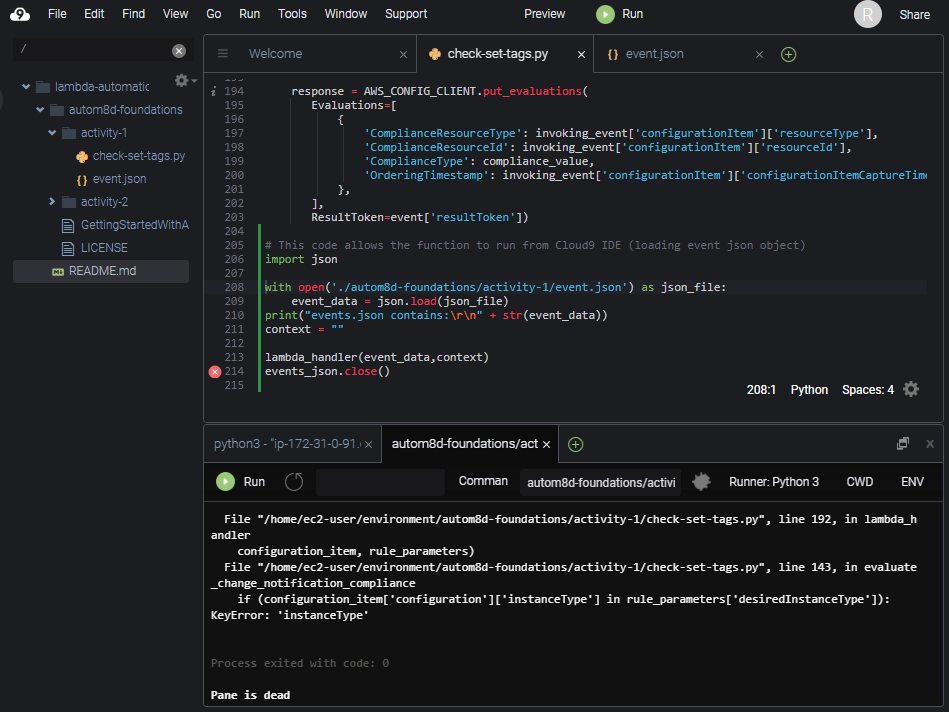
select the “check-set-tags.py” tab to look at our code again. Notice how there are several “def” functions defined. Your instructor will walk through each, describing at a high level what’s going on.

Note, also that since we’re running this code in the IDE, we have to:

1. Manually load our events.json code (lines 208,209)
2. Directly invoke our “lambda\_handler” method (line 213). When this function runs natively in Lambda (as you’ll see in Activity #2), you’ll see that “lambda\_handler” is what is called automatically for us by the Lambda service.

Also, we should point out that if you’re doing a lot of local development and/or dealing with event objects frequently, there is a more elegant way to do this – with the SAM CLI. Although what we’ve done here will work just fine for our demo purposes (and it also nicely highlights some python operations like file open, read, etc), if you want to learn the “right way” to deal with Lambda in Cloud9, check out this AWS user guide: [Working with AWS serverless applications using the AWS Toolkit](https://docs.aws.amazon.com/cloud9/latest/user-guide/serverless-apps-toolkit.html).

Now that boto is installed, we should be all ready to run our code. With our focus on the “check-set-tags.py” tab, let’s hit “Run” again at the page top!



Uh oh. We see errors again: “KeyError: ‘instanceType’”. This time, however, the error is expected (our “Configuration” object doesn’t have an expected “instanceType” for the mock. We can correct this, but then we’ll see other errors for “tags”, etc. For our purposes, this is just fine. We’ve validated that our code works… it’s just a bad “events” object being passed in. In Activity 2, we’ll see proper operation of this code when we come back and watch it run in a live scenario with real “Configuration” data passed into it from AWS Config.

\*\*\*\*\*\*\*\*\* END OF ACTIVITY #1 STOP HERE!!! \*\*\*\*\*\*\*\*\*

You now know basic operation (env create, terminal usage, code loading/running/debugging) of Cloud9, and you’ve gotten some experience with a fairly advanced Lambda function.

**Activity #2: Automating tag enforcement with AWS Lambda and AWS Config**

In this lab, we’re going to set up our ‘check-set-tags’ Lambda function to automatically be invoked by AWS Config when it detects any configuration change on our EC2 instances. The pseudocode for this function looks like:

foreach ec2 instance {

if(instance is approved instanceType) {

if(instance is tagged with “env=prod”) {

if(instance is not tagged with “owner”) {

tag instance with “[owner=rich@quicloud.com](mailto:owner%3Drich@quicloud.com)”

}

mark COMPLIANT and EXIT

} else {

stop instance

}

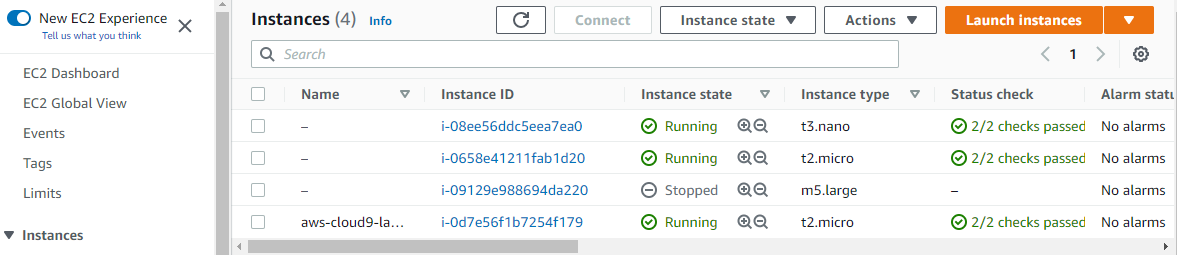
mark NON\_COMPLIANT

}

COMPLIANT OR NON\_COMPLIANT will appear in the AWS Config Dashboard, as well

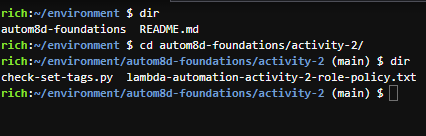
**Start up two (or more) EC2 instances**

Before we start our function running, we’ll need some test instances for the script to check. In the AWS Console, browse to EC2, and start up two instances. One of size t2.micro, and one of size t3.nano. You may still have a t2.micro running from activity 1 where we played with Cloud9. If desired, you may also start instances of other sizes as well (just be careful… larger instance types can incur significant costs in AWS). Once your instances are started, you should see something like this below (we still have our Cloud9 instance running).



**Create appropriate Lambda role**

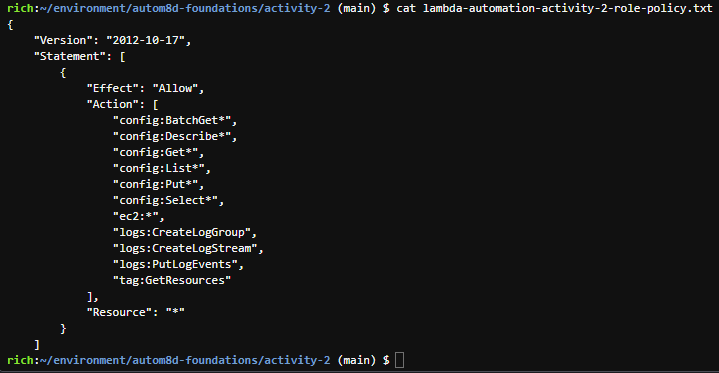
The Lambda function we’ll use needs proper ***execution permissions*** to perform all of it’s interactions with other AWS Services. We’ll use our pre-created policy document to create a role. From the git package you downloaded earlier (we just used the Cloud9 bash shell), browse into the ‘activity-2’ directory and run a directory listing.

cd autom8d-foundations/activity-2/

dir

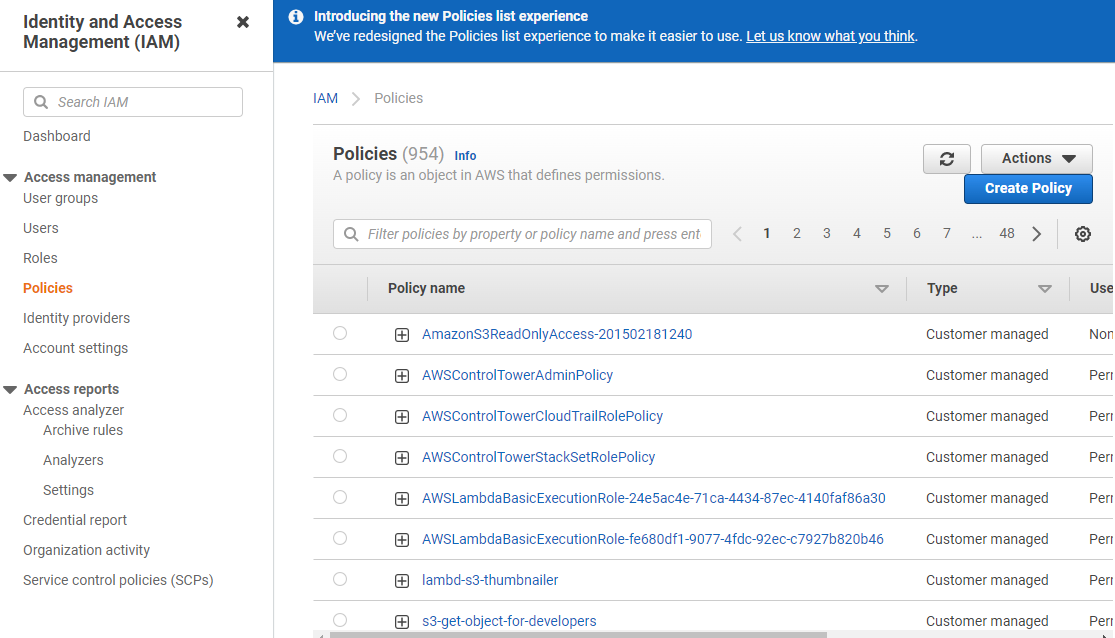
cat out the contents of the ‘lambda-automation-activity-2-rolepolicy.txt

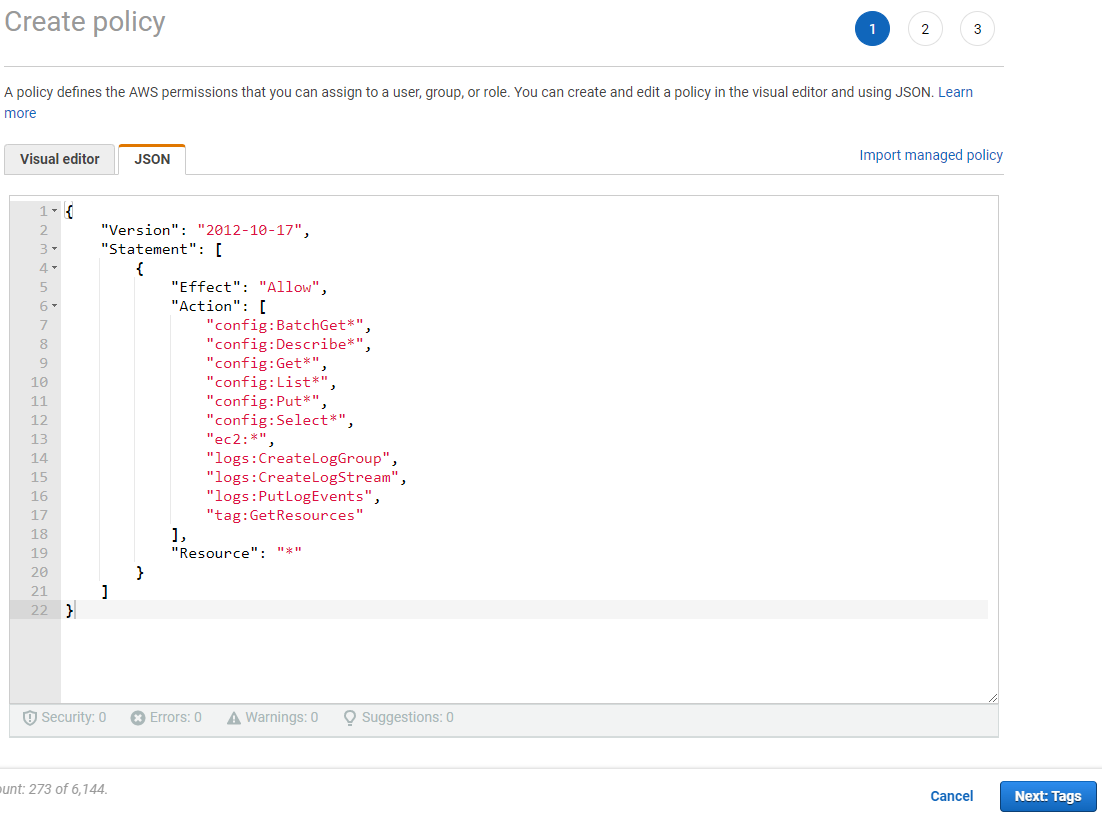
cat lambda-automation-activity-2-role-policy.txt

Copy the entire statement (from opening “{“ to closing “}”) into your copy buffer with CTRL-C. Your instructor will take a moment to explain the various ramifications of the statements inside of the document.

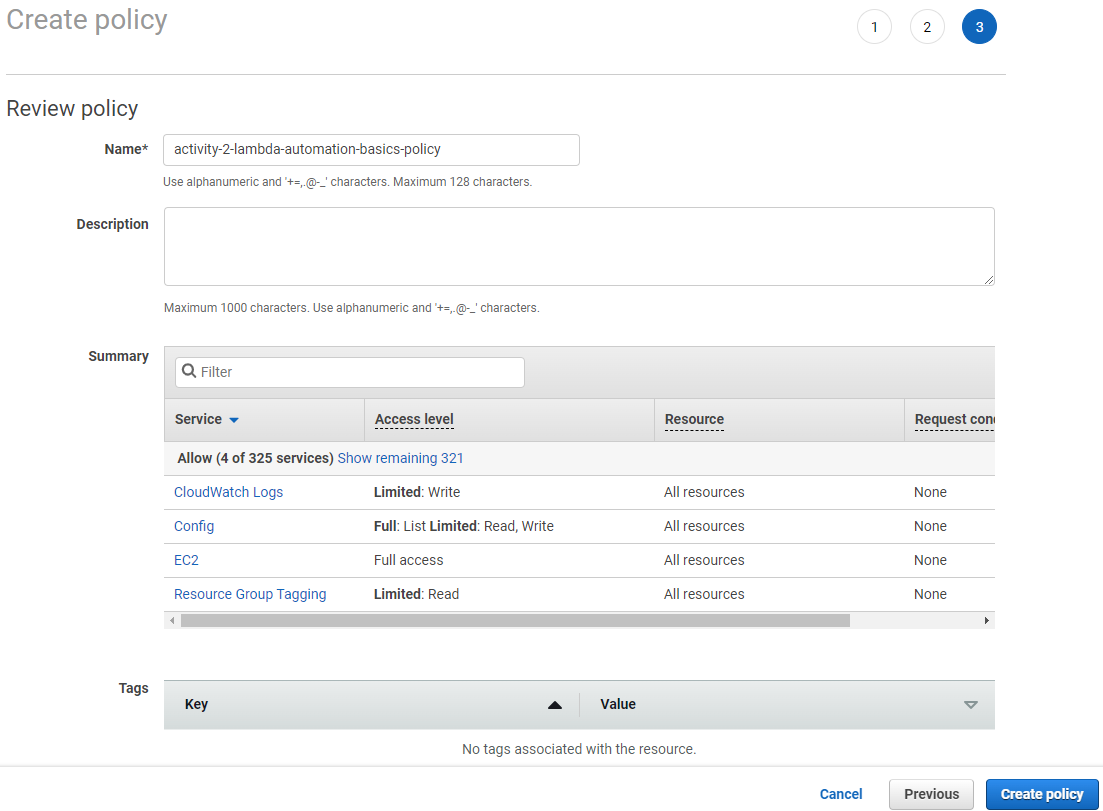
Back in your browser, open another tab to the [IAM service](http://console.aws.amazon.com/iam). Browse to “Policies” on the left hand navigation.

You should see a view that looks similar to:



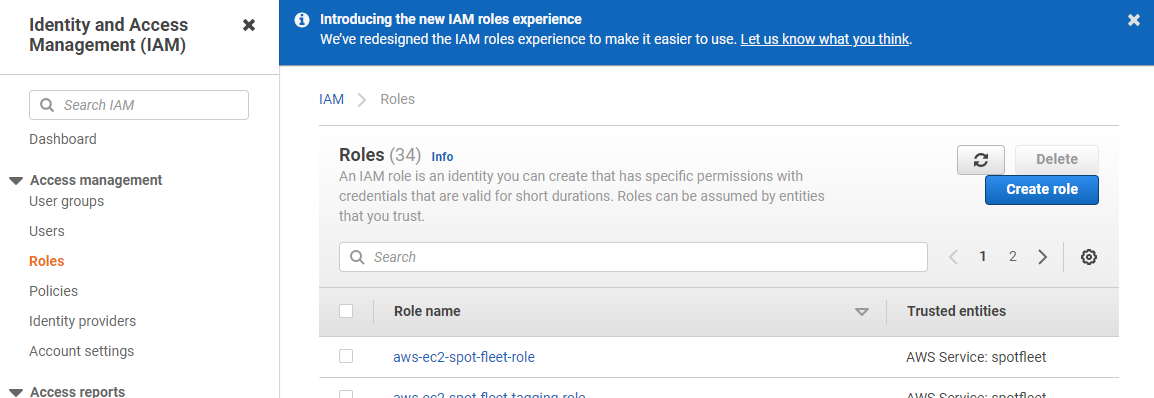
Click the “Create Policy” button on the upper right, choose the “JSON” tab, and paste in the policy document you copied above, overwriting the stub that was pre-entered.

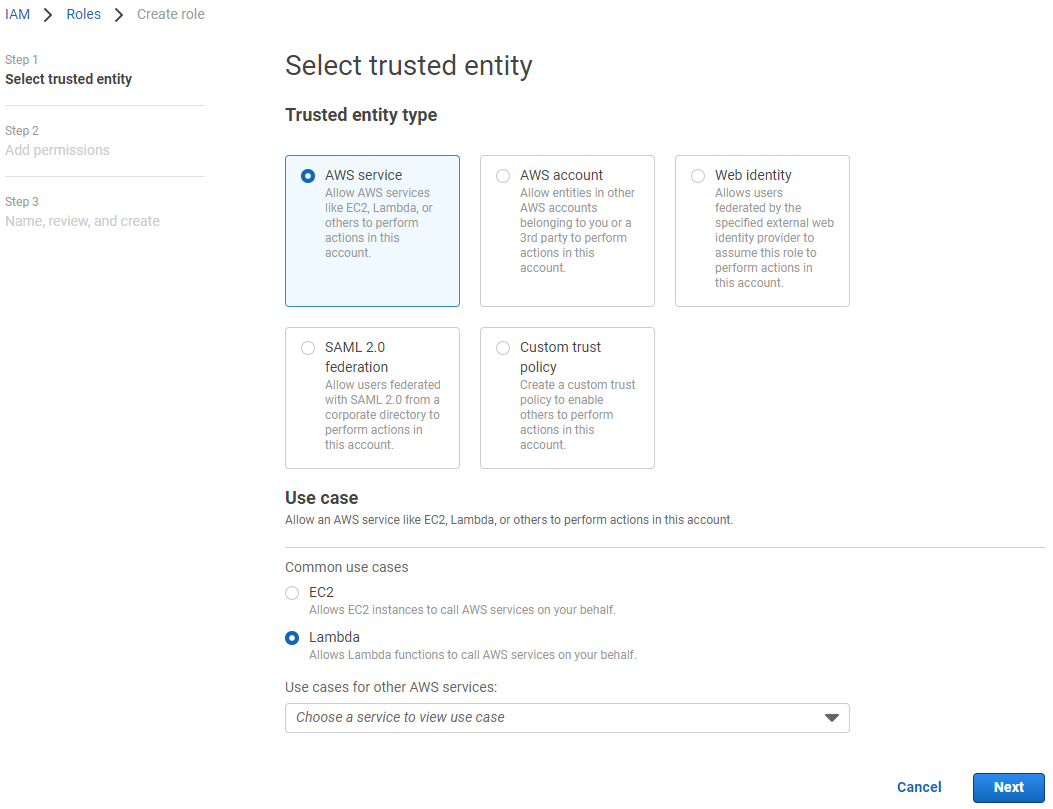
Click “Next: Tags”, then “Next: Review”. Give the policy a name of “activity-2-lambda-automation-basics-policy”. You can give a description if you like. Then push “Create policy” button



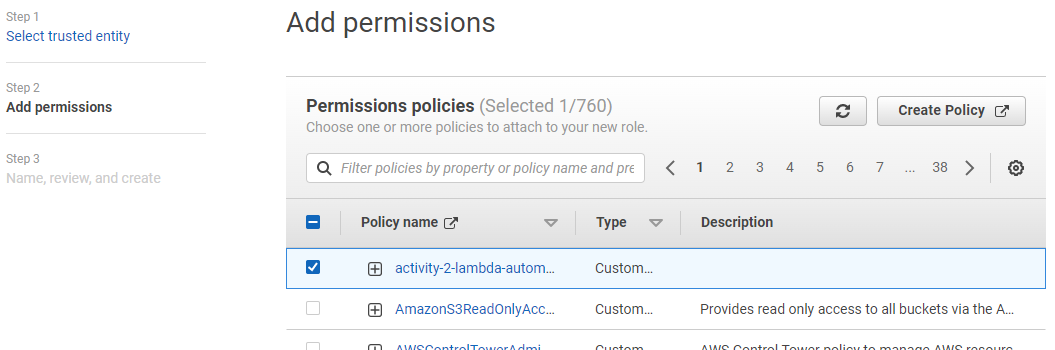
Now we’ll attach this policy to a role that Lambda will assume when it needs to.

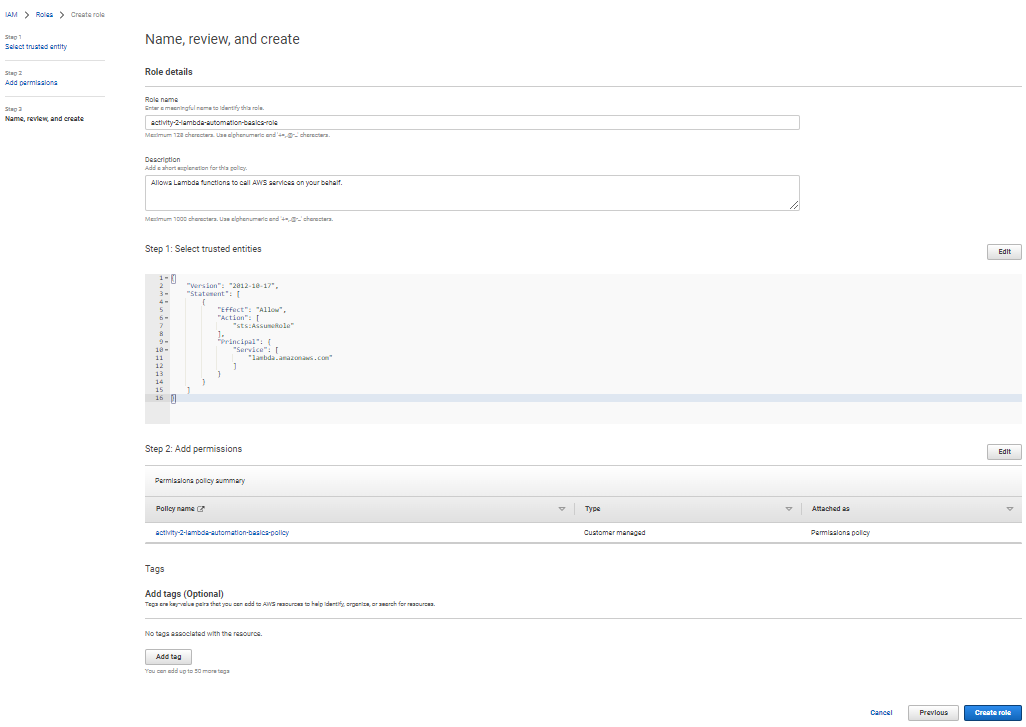
From the left hand nav in IAM, select “Roles”. And click “Create role” button in upper right.

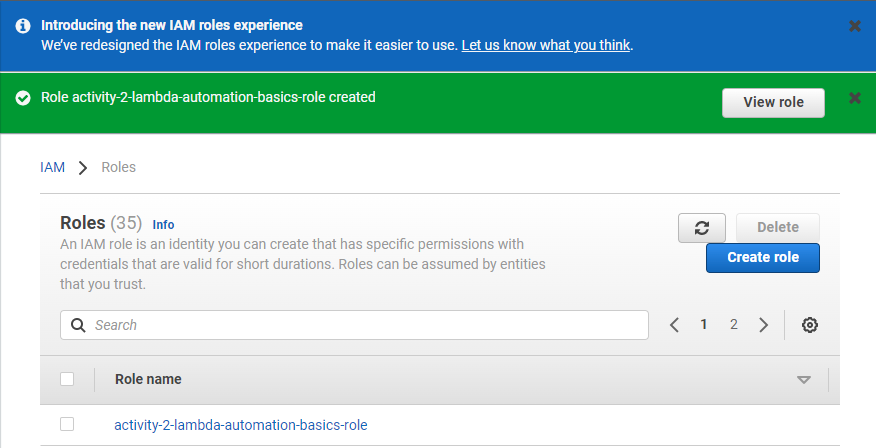




For “Trusted entity type”, choose “AWS service”, and for “Use case” choose “Lambda”, then push “Next”.

On the “Step 2: Add permissions” page, select the “activity-2-lambda-automation-basics-policy” that you entered before (it should appear at the top of the list). If you do not see this policy, you must repeat the steps above to enter the policy correctly. Push “Next” button.

On the “Step 3: Name, review, and create” page, name your role “ activity-2-lambda-automation-basics-role”, give it an optional description, and then press “Create role” button to finalize the creation.

You should finally see a screen that looks similar to the following, showing that your role has been successfully created.

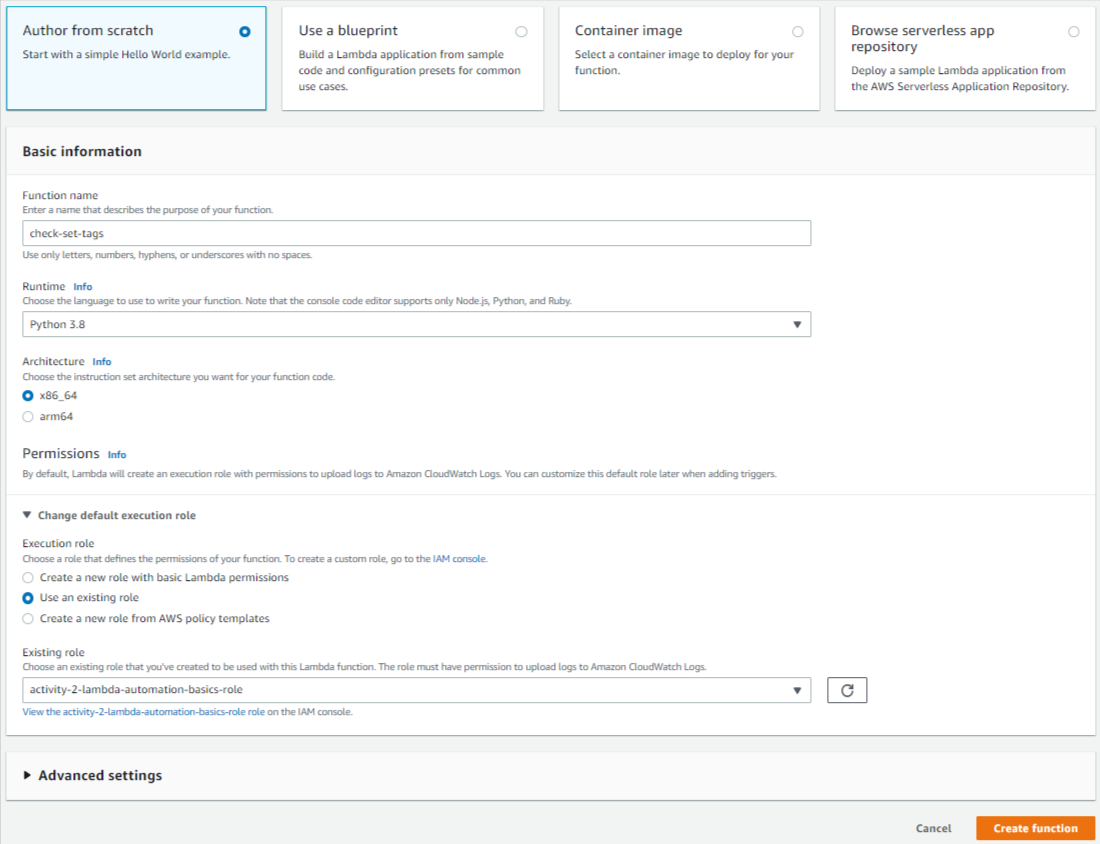
**Create our Lambda function**

Now that we have the appropriate permissions (the Lambda role) for our function to work, we can go ahead with the creation of the function itself. From wherever you installed the git package for this class, cat out the ‘’check-set-tags.py” python code.

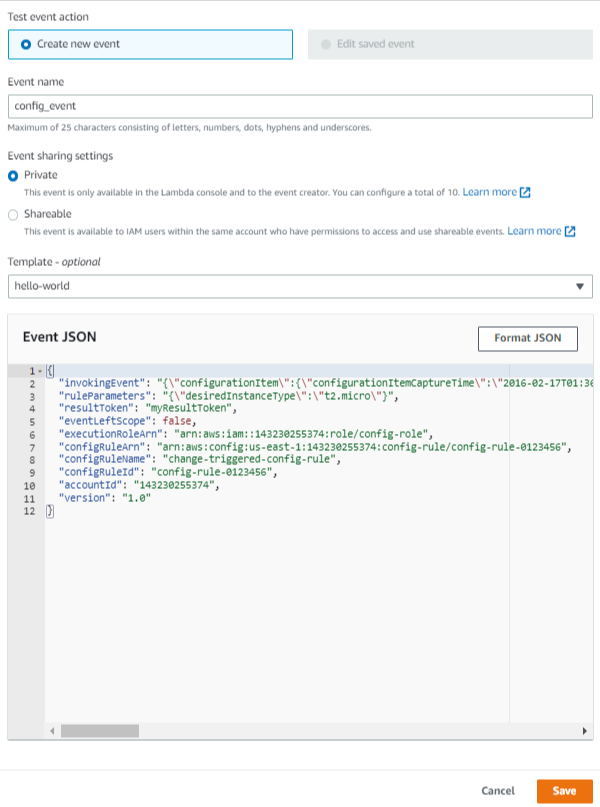
cat check-set-tags.py

Copy the entirety of this code (from “import boto3” at top to “resultToken’])” at bottom) by selecting it and pushing CTRL-C to copy it into your clipboard.

Open a new tab in your browser, and browse to the [Lambda section of the console](http://console.aws.amazon.com/lambda/). Press “Create function” button in the upper right.

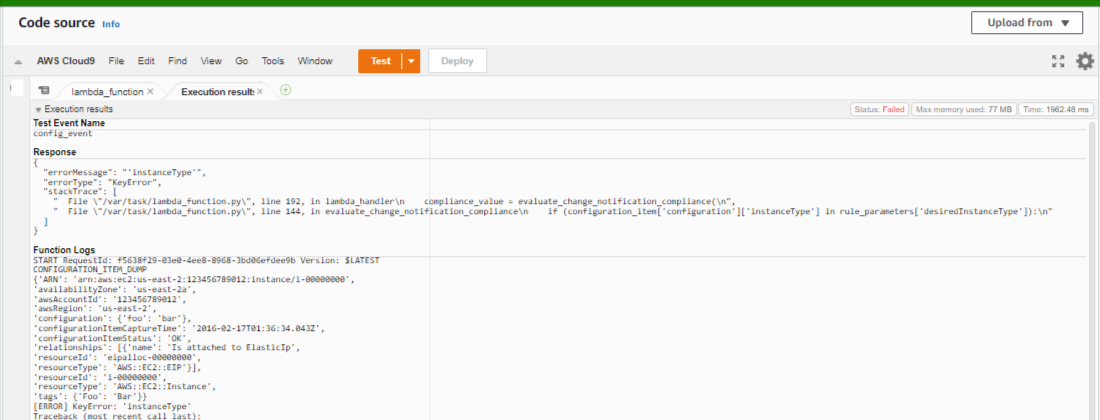
Choose “Author from scratch”, give your function the name “check-set-tags”, Choose the runtime of “Python 3.8”, and under “Permissions”, expand “Change the default execution role”, choose “Use an existing role”, then select your “activity-2-lambda-automation-basics-role” from the list. Then push the “Create function” button.

On the next page, under “lambda\_function” under “Code source”, paste your “check-set-tags” code in the area, overwriting the stub that was pre-entered. NOTE: Your changes ARE NOT LIVE until you push the “Deploy” button. Whenever you make changes to your code, make sure to always push the “Deploy” button to push it live. Go ahead and push the “Deploy” button as well.

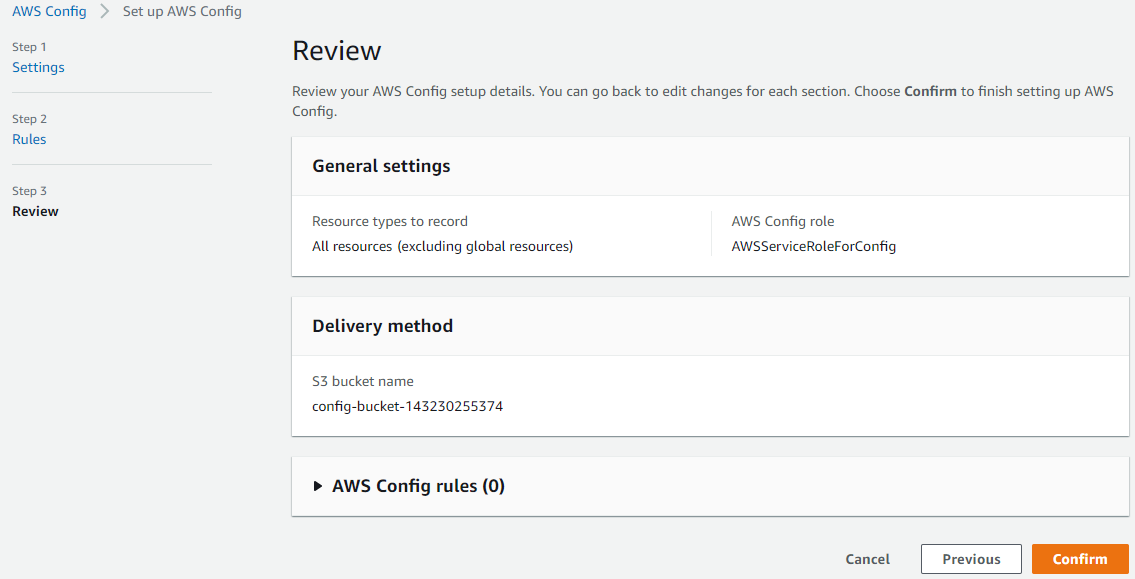


Select the dropdown to the right of the “Test” button, and choose “Configure test event”. Copy the contents of our “events.json” object from activity-1 into the “Event JSON”, and give the event the name “config\_event” and press “Save”.

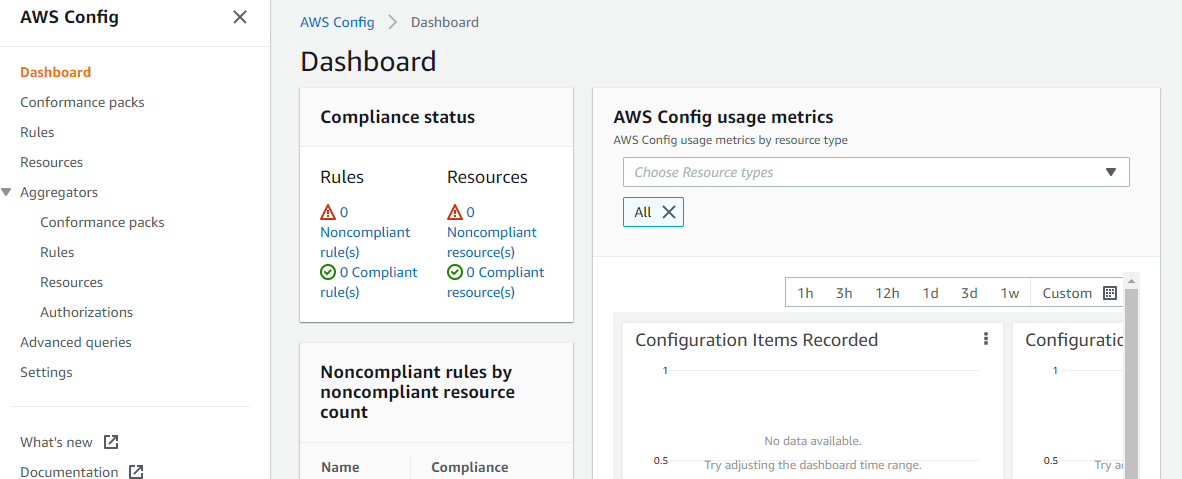
Now that we have a valid test object to test with, go ahead and press the “Test” button under Code source. You should see output that looks similar to the following. Notice that we see the same “instanceType” failure that we had from activity 1. This is to be expected, and at least validates that our function is set up correctly, the code is working properly, and that our function has the appropriate permissions to do what it needs to do.



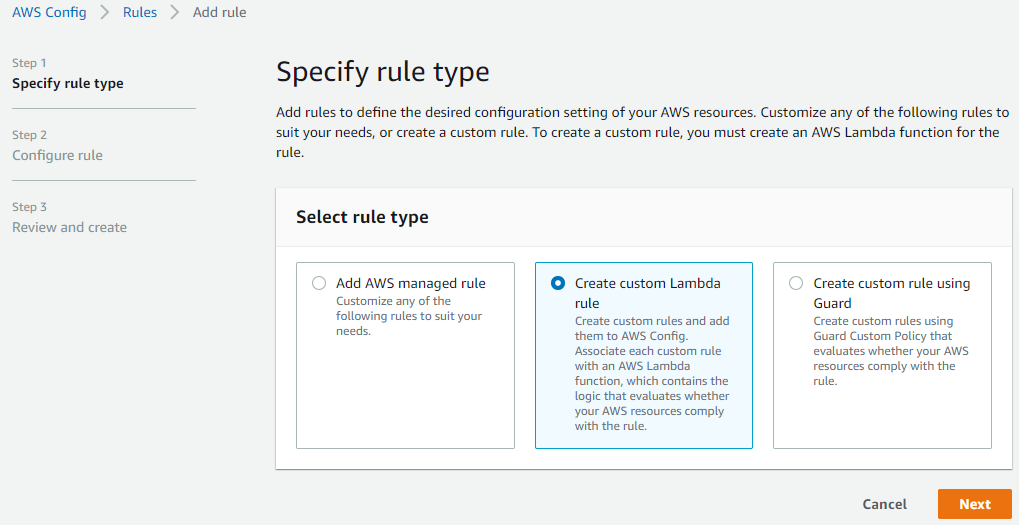
**Create custom AWS Config rule using our Lambda function**

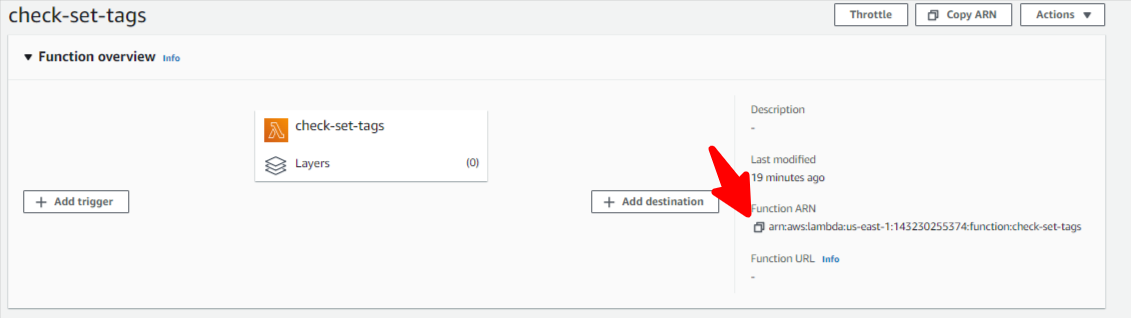
We’ll now set up AWS Config to monitor our EC2 instances, reporting in-or-out of compliance for the instances in the Config Dashboard. First, open a new tab in your browser and browse to [AWS Config](http://console.aws.amazon.com/config/) in that tab. Click “1-click setup” button, then push “Confim” (accepting all defaults) to enable AWS Config.

Now that config is enabled, you should see a dashboard that looks similar to the following.

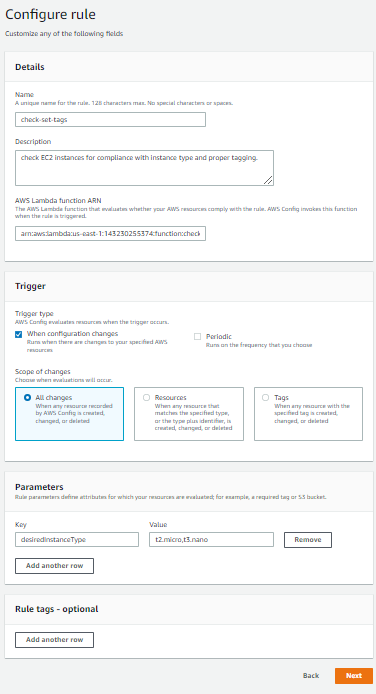


From the left-hand navigation, select “Rules”, then push the “Add rule” button. On the next screen, choose “Create custom Lambda rule” and press “Next”

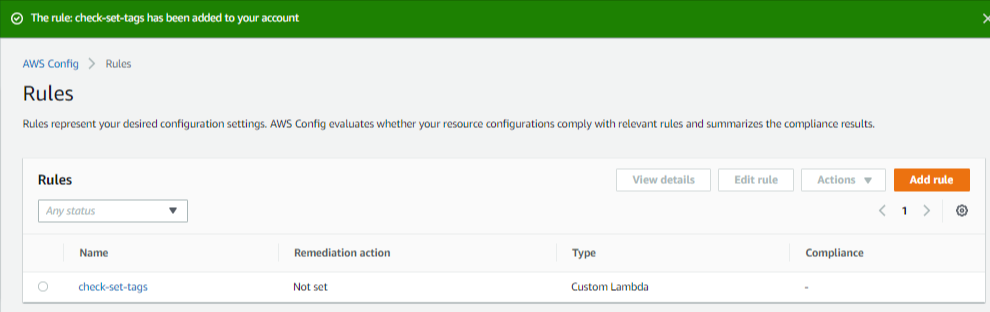


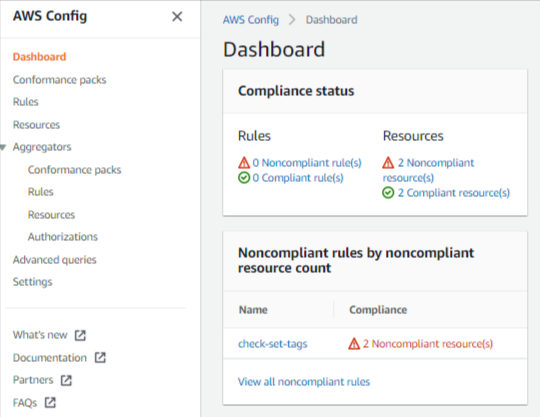


Back on your Lambda browser tab, copy the ARN of your “check-set-tags” lambda function by pressing the “copy” icon as shown in this screenshot:

Back in your AWS Config tab, give your rule the name “check-set-tags”, give it a description, paste your ARN under “AWS Lambda function ARN”, select “Trigger type” of “When configuration changes”, and then under parameters, set a “Key” of “desiredInstanceType” and a value of “t2.micro,t3.nano”. These Config parameters are hooked into our Lambda function to control the instance types we allow in our account, and you are welcome to change them later.

Press “Next” button, then on the Confirmation page, press “Add rule” button to finally add the rule.

You should now see the following:



And if you click back into Dashboard from the left hand nav, you should now start to see Compliance / Non-Compliance for your assets as so:

**Test our Config Rule by tweaking EC2 tags**

In your browser, open a new tab for CloudWatch Logs. We’ll revist this tab frequently as it’s where our “print” statements from our Lambda function will land.

Iteratively re-visit your EC2 tab in your browser, and try the following:

1. Set “env” tag to “prod” (then watch Config/Lambda tag your instances with “[owner=rich@quicloud.com](mailto:owner%3Drich@quicloud.com)”)
2. Set “env” tag to anything besides prod (then watch Config / Lambda stop your instances)
3. Start new EC2s with proper “env=prod” “[owner=your@email.com](mailto:owner%3Dyour@email.com)” and watch them go compliant in the Config Dashboard.

**Questions:**

1. What are some limitations of Config (can it check all resources, is it “real-time”)?
2. If we needed real-time, or other resource checking, what might be a better service?
3. How could we improve our Lambda function in a real-world environment?
4. How could we improve our permissioning of our Lambda function?

\*\*\*\*\*\*\*\*\* END OF ACTIVITY #2 STOP HERE!!! \*\*\*\*\*\*\*\*\*

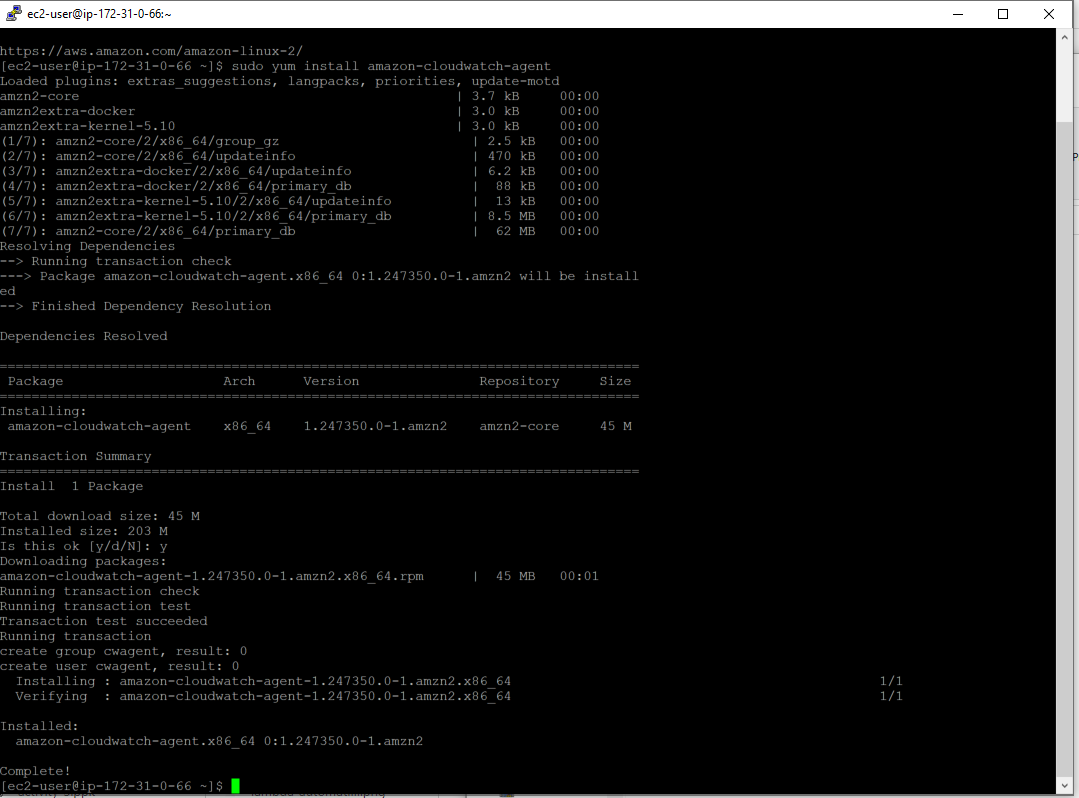
You now have a good understanding of everything involved with using AWS Config with custom Lambda functions to programatically and proactively enforce tagging policies.

**Activity #3: Automating EC2 log processing and alerting**

In this lab, we’re going to use a CloudWatch Logs agent to ship the /var/log/secure SSH logs into CloudWatch Logs. We’ll then route specific log entries to a Lambda to trigger with CloudWatch Subscription Filters. The Lambda function will log when unauthorized SSH attempts happen on our server.

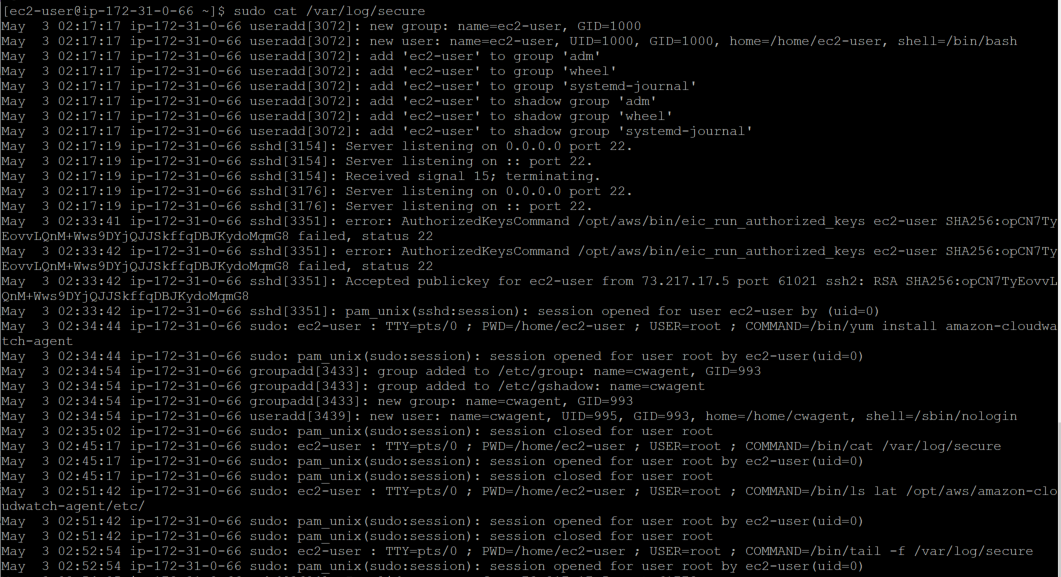
NOTE: Previous versions of our slides for this class referenced an “EventBridge” solution. This activity has been changed to simply use CloudWatch Subscription Filters as they are a much more elegant solution for this particular use case.

**Create CloudWatch Logs Role & launch EC2 Instance**

1. Follow the documented [AWS instructions](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/create-iam-roles-for-cloudwatch-agent.html) (Create IAM roles to use with the CloudWatch agent on Amazon EC2 instances) to create the role necessary for our EC2 instance to write it’s logs to CloudWatch Logs. At the end of the process, you will have created a role called “CloudWatchAgentAdminRole”
2. From the AWS EC2 console, launch a t2.micro instance with Amazon Linux (or, optionally re-use one of the ones from activity #2, if you still have it). Give the instance the name “activity-3”. Under “key pair name”, choose “Create a new key pair”, and give the keypair the name “activity-3”. If you use linux to SSH, leave the default “.pem”, or if you use Putty to SSH, select the “.ppk” option. Your keypair will download locally – note the location it saved to as you’ll need it later. Expand “Advanced details”, and under “IAM instance profile”, select the “CloudWatchAgentAdminRole” you created above. Accept all other defaults (including “Allow SSH traffic from Anywhere” – generally a BAD idea!). Note this instance ID.
3. SSH into your “activity-3” instance (user is “ec2-user”), and then install the cloudwatch agent (it’s a standard package

sudo yum install amazon-cloudwatch-agent

Once the command runs, you should see the output as shown on the right.

**Configure CloudWatch Logs Agent**

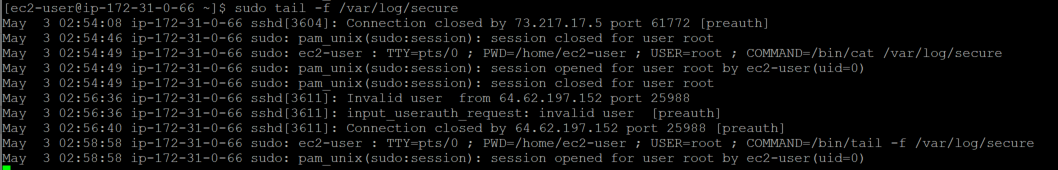
The logfile that we’re going to analyze is /var/log/secure on Amazon Linux 2. This log has all the entries whenever an SSH attempt is made against this server. Cat the file out to see our current logged in session.

sudo cat /var/log/secure

Now, let’s tail the log file to see what an invalid access would look like.

sudo tail -f /var/log/secure

Open a 2nd putty session to this server, and attempt to log in with another user than the authorized “ec2-user”. In the example below, we’re trying to lgin as the user “xyz”. Note the failed “invalid user” line.



Let’s now configure the CloudWatch Logs agent to ship these logs into CloudWatch. The CloudWatch Agent installs under /opt/aws/amazon-cloudwatch-agent/, so we’ll update our configuration there. The configuration files we’ll create will be stored under the “/etc” directory in that path, so let’s cd into there first and browse the contents.

cd /opt/aws/amazon-cloudwatch-agent/etc

ls -lat

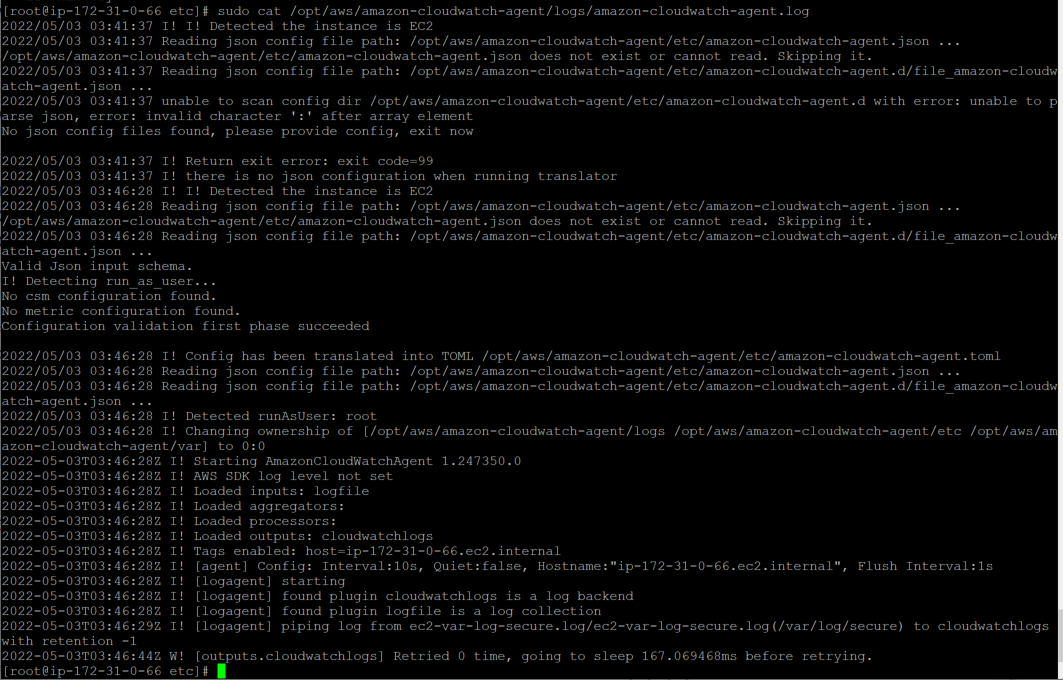
You’ll only see two one file in there currently, “common-config.toml”, and one directory “amazon-cloudwatch-agent.d”. Following AWS recommendations, we’ll install a config file here named “amazon-cloudwatch-agent.json”. Copy the contents of our git “/activity-3/amazon-cloudwatch-agent.json” file into here (using vi, nano, or whatever editor works for you. NOTE: It may be faster/easier to “git clone” our base code on this server and copy the file directly if you have problems with copy/paste in your SSH client). Your instructor will now pull up the contents of the file and walk you through the details.

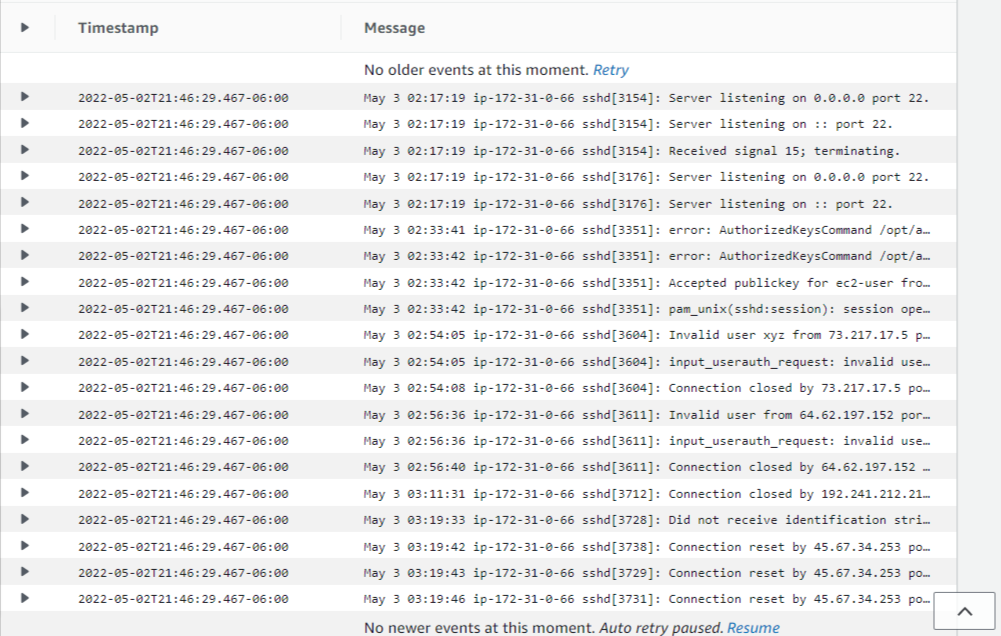
Once you’ve browsed the file contents and you’re aware of what they each do, we’ll go ahead and start our cloudwatch agent with the following command:

sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -s -c file:/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json

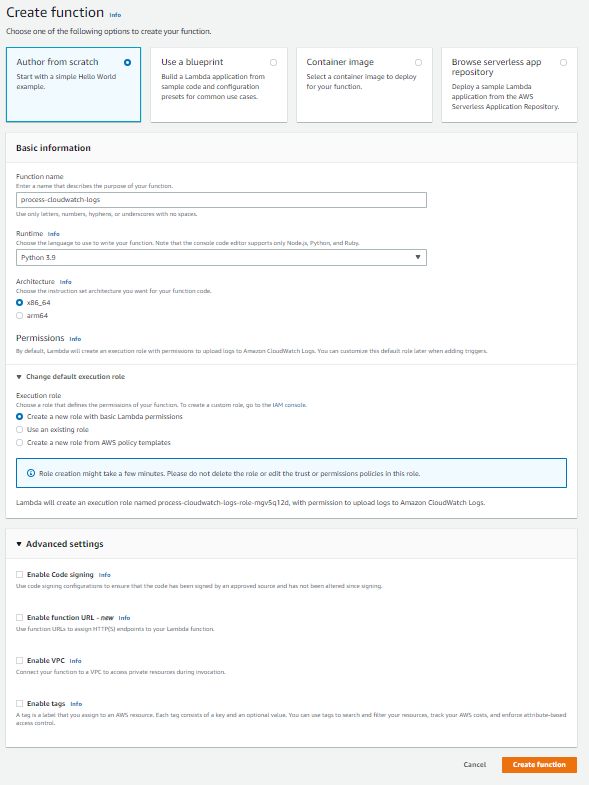
To ensure Cloudwatch started correctly, let’s first look at our log file…

sudo cat /opt/aws/amazon-cloudwatch-agent/logs/amazon-cloudwatch-agent.log

If the agent started successfully, we should see something similar to the output at the right.

The true test of whether or not everything has been set up correctly to date (including the EC2 role), is to actually browse the CloudWatch Logs consolerect operation. Browse to the [CloudWatch Console](http://console.aws.amazon.com/cloudwatch/), select “Log groups” under “Logs” on the right-hand navigation, and then click on our “ec2-var-log-secure.log” group. Once you see the group, click on the stream under “Log streams” (Note: We only have a single stream here, because we only have one agent running on one server. If we had several servers running an agent each, we’d see multiple log files). Click on the single “ec2-var-log-secure.log” stream to view it’s contents. It should look similar to what we see on the right (showing that we only captured entries which match ‘sshd’).

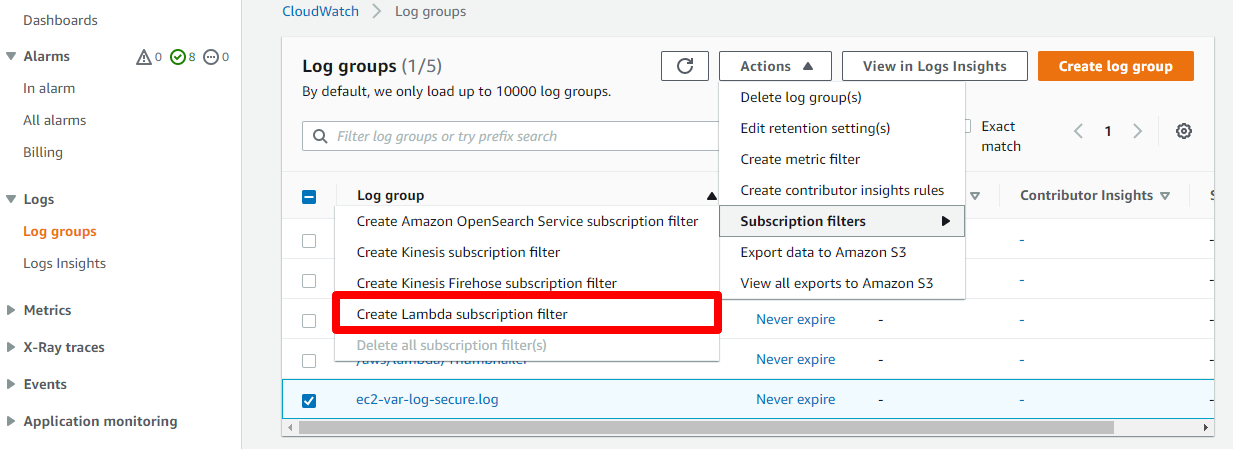
If you like, you can attempt another failed SSH connection to see another “Invalid user [user-name]” message appear in the logs.

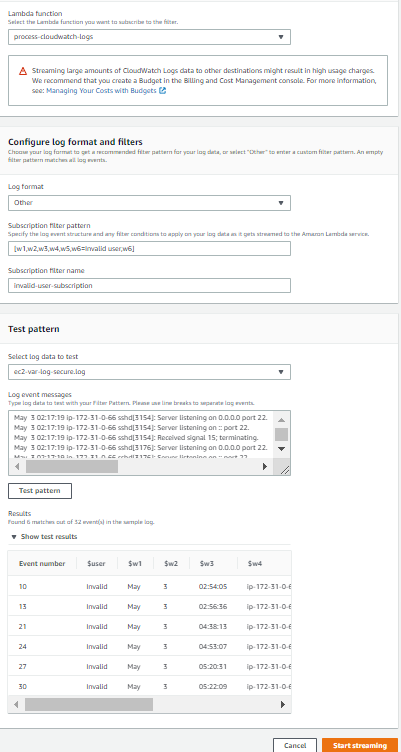
**Create Lambda function to process Invalid SSH attempts.**

Using the steps you went through in activity #2, go ahead & create a 2nd lambda function, using the screenshot at right as a guide. Once the function is created, go ahead & paste the code for “process-cloudwatch-logs.py” into the function & deploy it by pressing the “deploy” button. We could create an event object to test the code with, but we’re just going to go ahead & create a CloudWatch subscription filter which is actually probably faster than even testing.

**Create CloudWatch subscription filter to call Lambda log processor and respond to events in EC2 logs**

Back in the CloudWatch Console, click “Log groups” under Logs in the right hand nav, tick the checkbox next to “ec2-var-log-secure-log”, and then from the “Actions” dropdown, choose “Subscription filters→Create Lambda subscription filter”. If you’re confused, use the screenshot below as a guide.



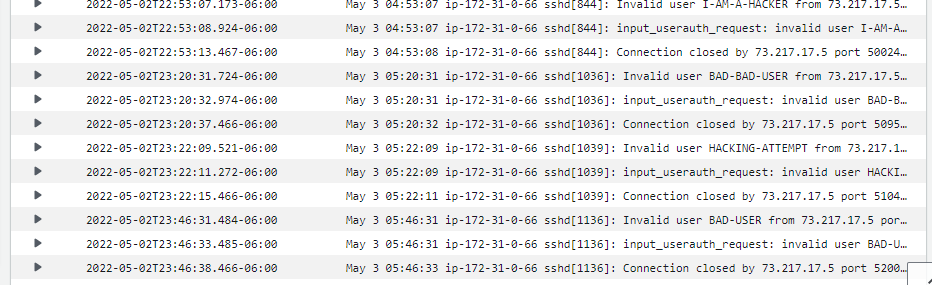
On the next page, point the subscription to our function (process-cloudwatch-logs), use “Other” log format, set the “Subscription filter pattern to” “[w1,w2,w3,w4,w5,w6=Invalid user,w7]” (your instructor will explain this pattern in detail), give the subscription a name (we used “invalid-user-subscription”), then under “Test pattern”, select our “ec2-var-log-secure.log” log, and push the “Test pattern” button to validate our Subscription filter pattern.

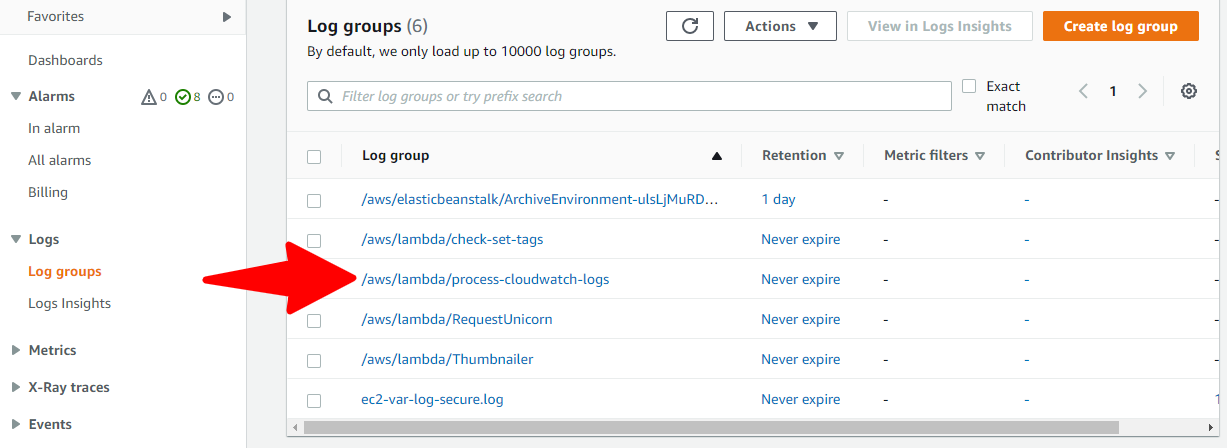
You should only see the results that match “Invalid user”, and you can see that we’ve tested our results (on the right) with several failed SSH attempts.

If all looks good, go ahead & press “Start streaming” to connect this final piece of our activity.

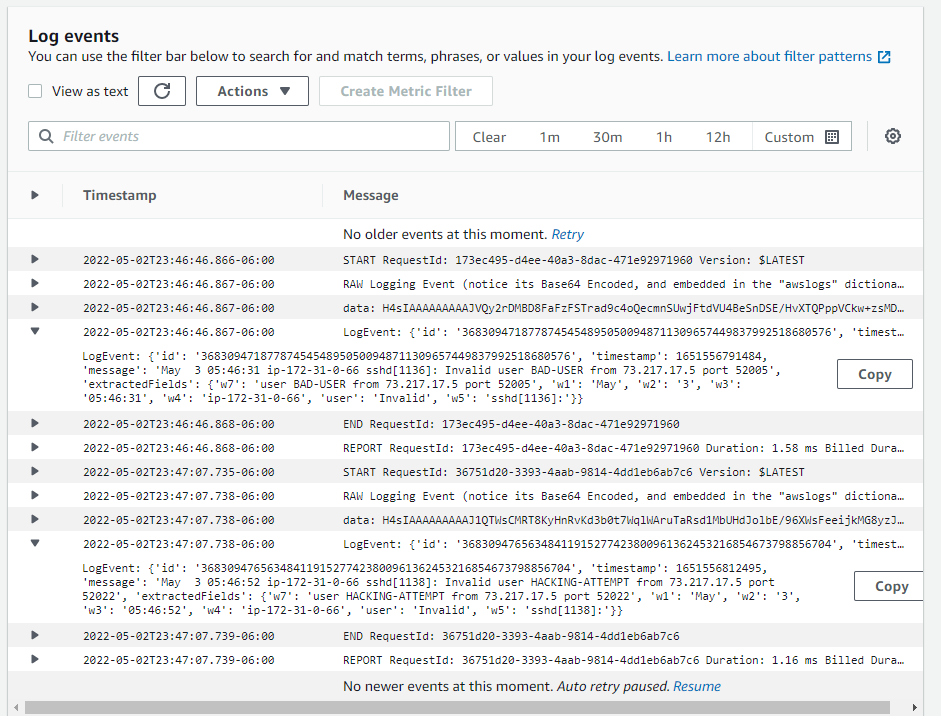
You’ll be redirected back to the “Log groups” page with a message saying “LLog events streamed to Amazon Lambda.” at page top.

**Validate operation of solution**

Attempt to SSH to this server again, but this time do not supply the valid name “ec2-user”. Try to log in with anything else like “BAD-USER” or “HACKING-ATTEMPT”. Immediately after failed logins, you should see the appropriate entries in “ec2-var-log-secure.log”. Here’s what ours looked like after a few attempts:



After 10-20 seconds, back in “Log groups”, you should see a new log group appear “process-cloudwatch-logs”.

Go ahead & click into that group, click into the stream to view the entries inside. You should see something similar to our results on the right.

You can expand out the “LogEvent” entries to see what matched our patterns (note these are the ONLY entries that matched) and what was placed into the w1,w2, etc variables.

This solution could easily be extended to allow Lambda to notify us (via SNS), update a dashboard (which we’ll do in the next lab), Add the remote IP to a WAF or SG rule block, etc…

The possibilities are endless!!!

\*\*\*\*\*\*\*\*\* END OF ACTIVITY #3 STOP HERE!!! \*\*\*\*\*\*\*\*\*

You can now set up an end-to-end log capture, ship, and analysis toolchain using CloudWatch Logs Agent, CloudWatch subscription filters, and Lambda.

**Activity #4: Automating Trusted Advisor and EventBridge alerts to update CloudWatch dashboards**

This lab is currently not working as of 3/3/2022 (Trusted Advisor is not behaving itself). Rich will fix the problems and post a link to a walkthrough video of the demo (along with lab instructions) no later than Friday, May 6th. Look for the link in the README of this git package.

\*\*\*\*\*\*\*\*\* END OF ACTIVITY #4 STOP HERE!!! \*\*\*\*\*\*\*\*\*

You are now able to respond to Trusted Advisor alerts with automated Lambda processors that update dashboards to provide real-time insight into operational issues.