**Working with AWS Config Custom Rules**

### Introduction

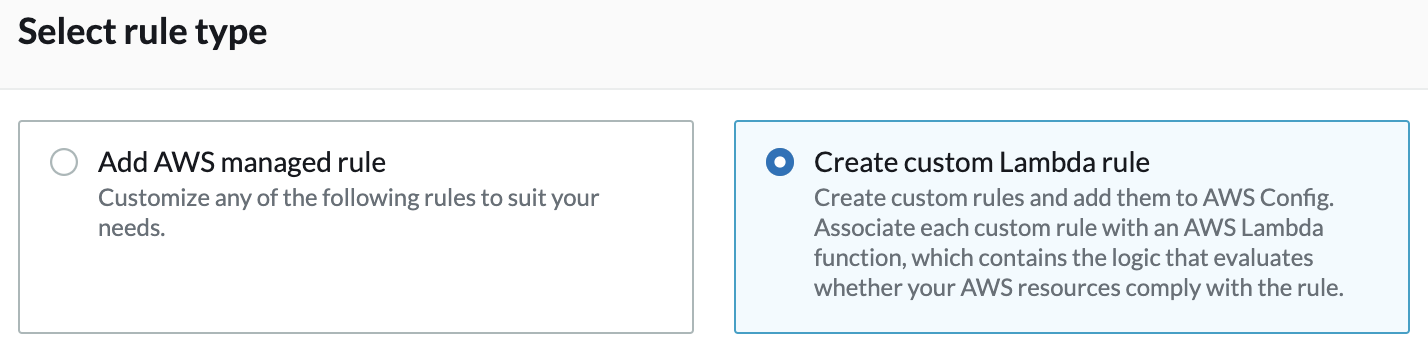
AWS Config custom rules can be used for a wide variety of scenarios when AWS managed rules don't fit your use case. Custom rules rely on an AWS Lambda function to evaluate resource configuration. You will be exposed to how to create a custom rule using pre-written code for a Lambda function. For demonstration purposes, the code simply evaluates whether an RDP tcp port is open to incoming traffic. You will check and correct noncompliance using the custom rule in this Lab Step.

Note: There is a managed rule that can achieve a similar result. The rule's name is restricted-common-ports. You should use that managed rule if you need to block the RDP port. The custom rule in the Lab Step is for demonstration purposes only, allowing you to get familiar with the process of working with custom rules.

### Instructions

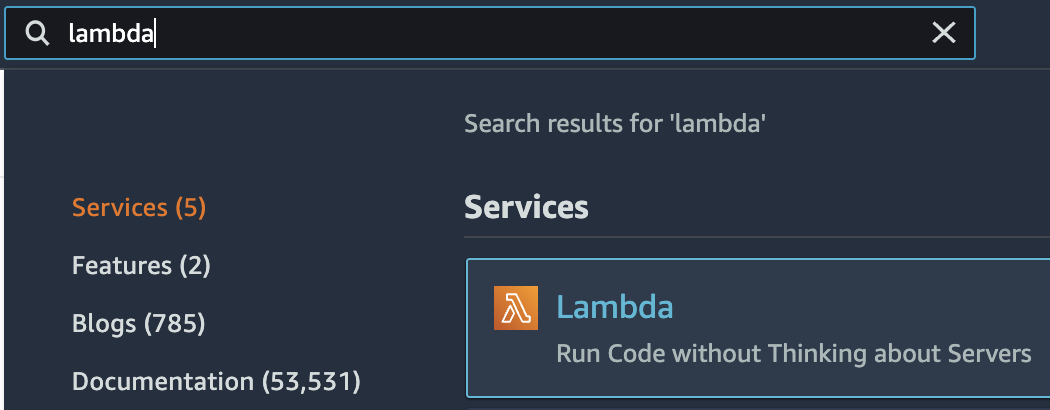
1. From the AWS Config **Rules** page, click **Add rule**.

2. Select **Create custom Lambda rule**, and click on **Next**:



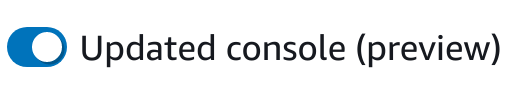
3. Move to the [AWS Lambda console](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions), and click on **Create function**.

Note: You can click AWS Lambda console link above or type lambda in the search box and click **Lambda**:





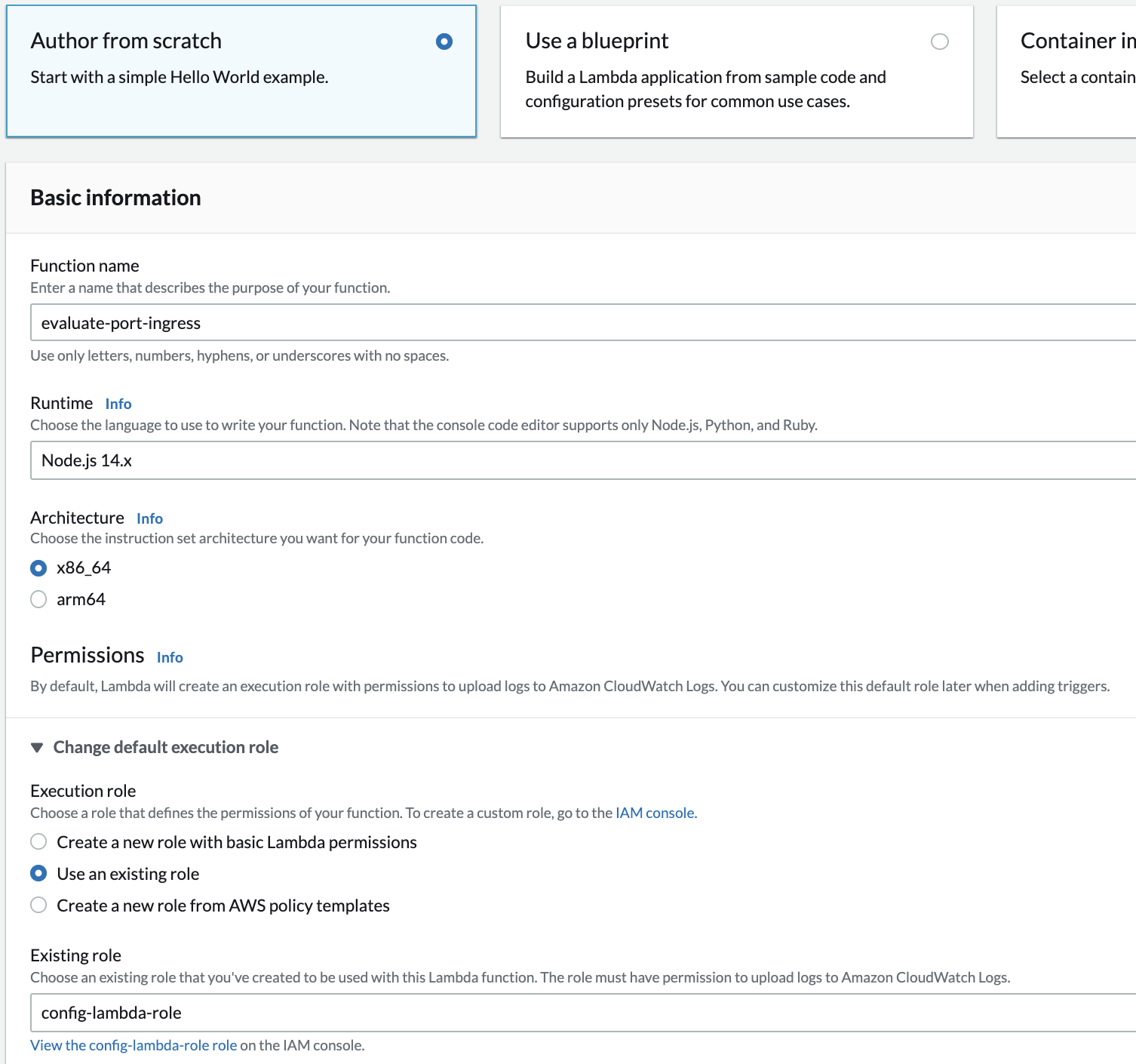
Note: If the **Updated console** toggle in the top-left corner is disabled, click it to enable the latest AWS Lambda user interface:



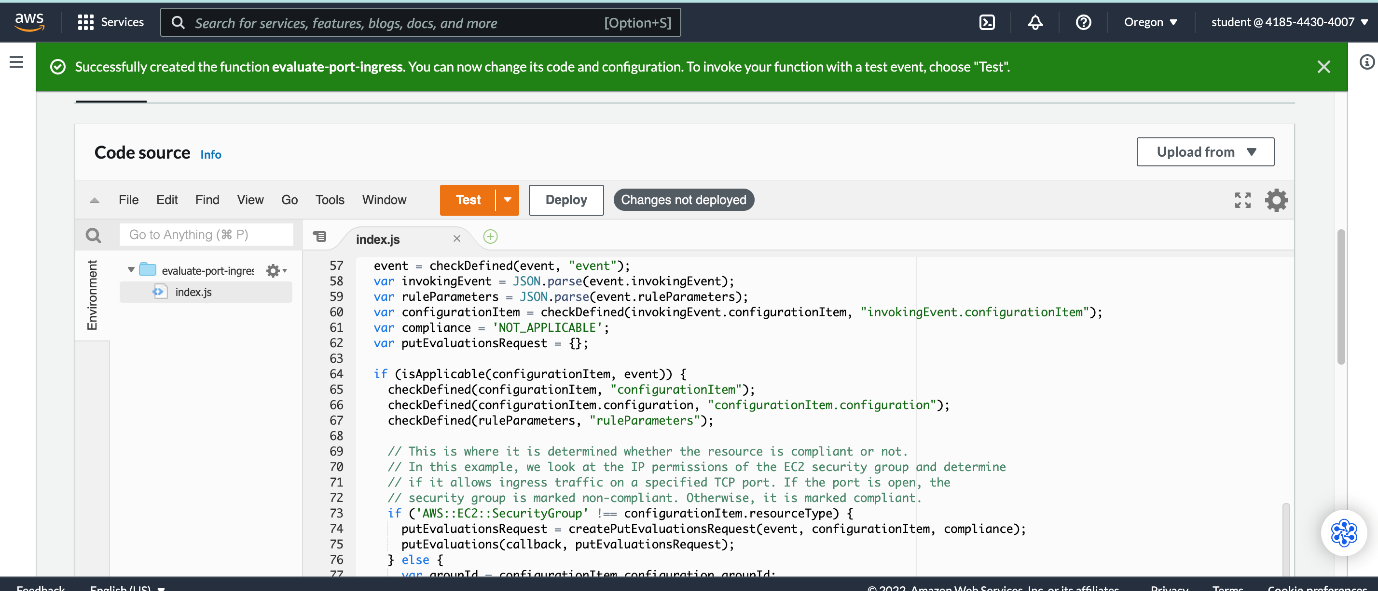
4. In the **Author from scratch** section, enter the following before clicking **Create Function**:

* **Name**: Enter evaluate-port-ingress (Important to enter exactly as written or the Lambda function will fail to create)
* **Runtime**: Select **Node.js** (use the latest version available)
* **Role**: Use an existing role
* **Existing role**: Select **config-lambda-role**

The config-lambda-role was created for you during the Lab startup. It authorizes Lambda to describe security groups (ec2.describeSecurityGroups) and put evaluations into AWS Config (config.putEvaluations).



5. Move to the **Code source** section. Double-click **index.js** in the code editor, and overwrite the existing contents with the following JavaScript code block that targets Node.js:



[**Copy code**](https://cloudacademy.com/lab/compliance-check-using-aws-config-rules-managed-custom/working-with-aws-config-custom-rules/?context_id=201&context_resource=lp)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

*//*

*// Ensure a security group does not allow tcp ingress on a specified port*

*//*

**var** aws = require('aws-sdk');

**var** config = **new** aws.ConfigService();

*// Helper function used to validate input*

**function** **checkDefined**(reference, referenceName) {

**if** (!reference) {

console.log("Error: " + referenceName + " is not defined");

**throw** referenceName;

}

**return** reference;

}

*// Check whether the resource has been deleted. If it has, then the evaluation is unnecessary.*

**function** **isApplicable**(configurationItem, event) {

checkDefined(configurationItem, "configurationItem");

checkDefined(event, "event");

**var** status = configurationItem.configurationItemStatus;

**var** eventLeftScope = event.eventLeftScope;

**return** ('OK' === status || 'ResourceDiscovered' === status) && false === eventLeftScope;

}

**function** **createPutEvaluationsRequest**(event, configurationItem, compliance) {

**var** putEvaluationsRequest = {};

*// Put together the request that reports the evaluation status*

putEvaluationsRequest.Evaluations = [

{

ComplianceResourceType: configurationItem.resourceType,

ComplianceResourceId: configurationItem.resourceId,

ComplianceType: compliance,

OrderingTimestamp: configurationItem.configurationItemCaptureTime

}

];

putEvaluationsRequest.ResultToken = event.resultToken;

putEvaluationsRequest.TestMode = false;

**return** putEvaluationsRequest;

}

**function** **putEvaluations**(callback, putEvaluationsRequest) {

*// Invoke the Config API to report the result of the evaluation*

config.putEvaluations(putEvaluationsRequest, **function** (err, data) {

**if** (err) {

callback(err);

} **else** {

callback(null, "success");

}

});

}

*// This is the handler that's invoked by Lambda*

exports.handler = **function** (event, context, callback) {

event = checkDefined(event, "event");

**var** invokingEvent = JSON.parse(event.invokingEvent);

**var** ruleParameters = JSON.parse(event.ruleParameters);

**var** configurationItem = checkDefined(invokingEvent.configurationItem, "invokingEvent.configurationItem");

**var** compliance = 'NOT\_APPLICABLE';

**var** putEvaluationsRequest = {};

**if** (isApplicable(configurationItem, event)) {

checkDefined(configurationItem, "configurationItem");

checkDefined(configurationItem.configuration, "configurationItem.configuration");

checkDefined(ruleParameters, "ruleParameters");

*// This is where it is determined whether the resource is compliant or not.*

*// In this example, we look at the IP permissions of the EC2 security group and determine*

*// if it allows ingress traffic on a specified TCP port. If the port is open, the*

*// security group is marked non-compliant. Otherwise, it is marked compliant.*

**if** ('AWS::EC2::SecurityGroup' !== configurationItem.resourceType) {

putEvaluationsRequest = createPutEvaluationsRequest(event, configurationItem, compliance);

putEvaluations(callback, putEvaluationsRequest);

} **else** {

**var** groupId = configurationItem.configuration.groupId;

**var** ec2 = **new** aws.EC2({ apiVersion: '2016-11-15' });

**var** params = {

DryRun: false,

GroupIds: [

groupId

]

};

ec2.describeSecurityGroups(params, **function** (err, data) {

**var** compliance = 'COMPLIANT';

**if** (err) {

compliance = 'NON\_COMPLIANT';

} **else** {

**var** ipPermissions = data.SecurityGroups[0].IpPermissions;

**for** (**var** i = 0; i < ipPermissions.length; i++) {

**var** ipPermission = ipPermissions[i];

*// The actual test condition (allows default allow all rule (IpProtocol === '-1') to be compliant for demonstration purposes)*

**if** (ipPermission.IpProtocol === 'tcp'

&& ipPermission.FromPort >= ruleParameters.port

&& ipPermission.ToPort <= ruleParameters.port) {

compliance = 'NON\_COMPLIANT';

**break**;

}

}

}

putEvaluationsRequest = createPutEvaluationsRequest(event, configurationItem, compliance);

putEvaluations(callback, putEvaluationsRequest);

});

}

} **else** {

putEvaluationsRequest = createPutEvaluationsRequest(event, configurationItem, compliance);

putEvaluations(callback, putEvaluationsRequest);

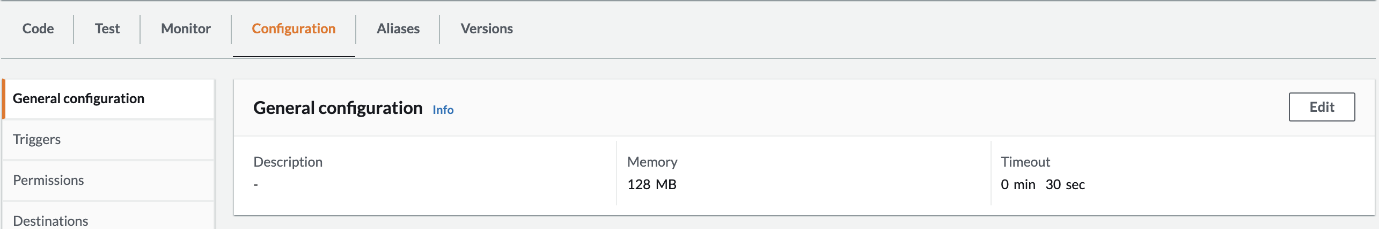
}

};

Note: It is not important to understand the code. All you need to know is that the code checks whether a security group allows inbound traffic on a specified TCP port. It makes use of the AWS SDK to access AWS Config and EC2. It is possible to modify the Lambda function so that it automatically enforces compliance by modifying the security group. This is left as an exercise for the student.

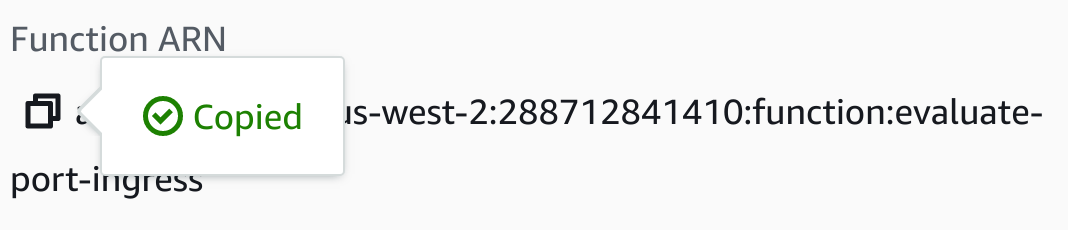
6. Click the **Configuration** tab, click **Edit**, set the following form values, and click **Save**:

* **Memory (MB)**: 128 (default value)
* **Timeout**: 0 min 30 sec



This allows the Lambda function 30 seconds and 128MB of peak memory to complete evaluation. This is more than enough for the simple check performed by the code.

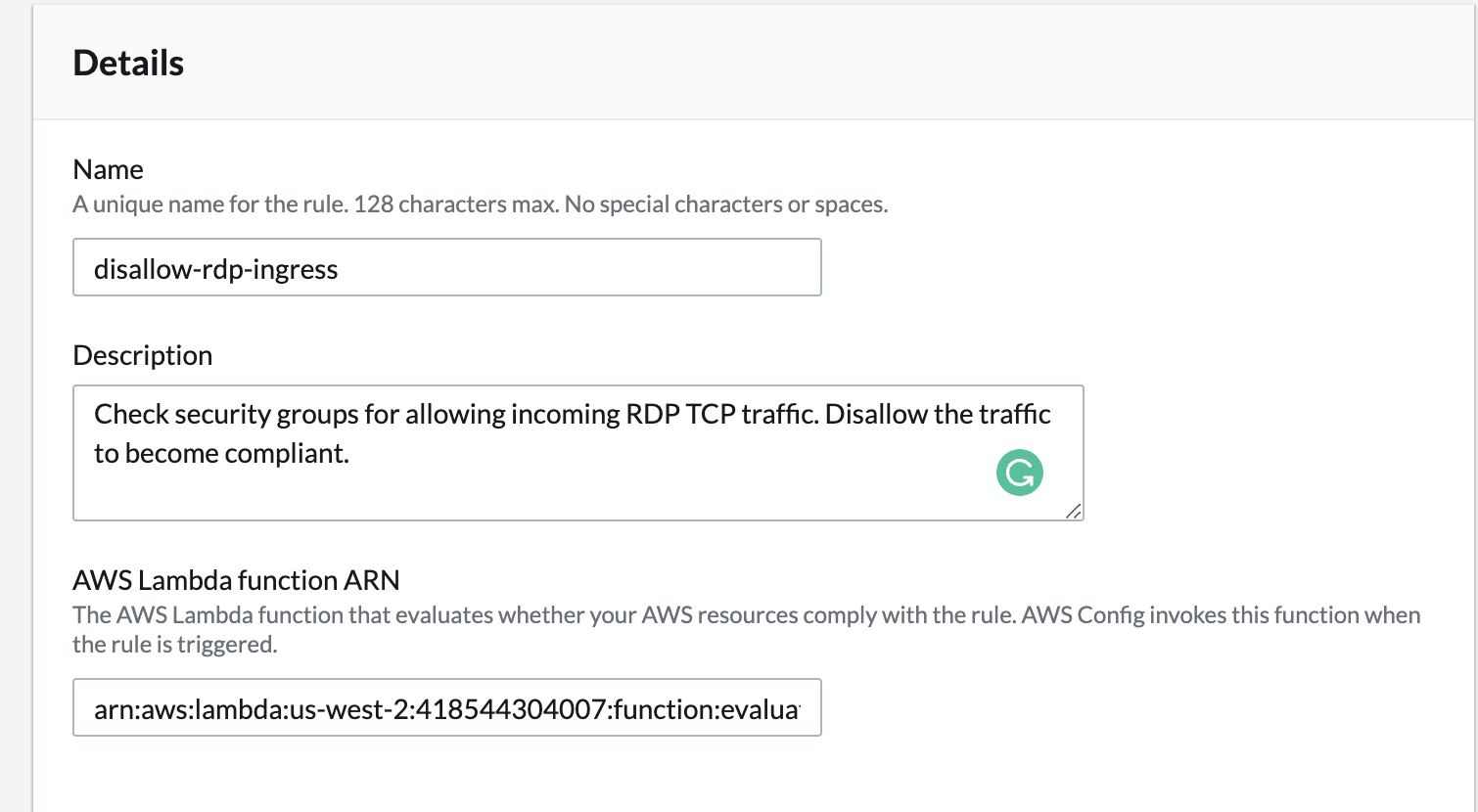
7. Return to the **Code** tab. Click **Deploy** at the top of the **Code source**panel and copy the Lambda's **Function ARN** in the upper-right of the Console (it looks similar to arn:aws:lambda:us-west-2:123456789012:function:evaluate-port-ingress):

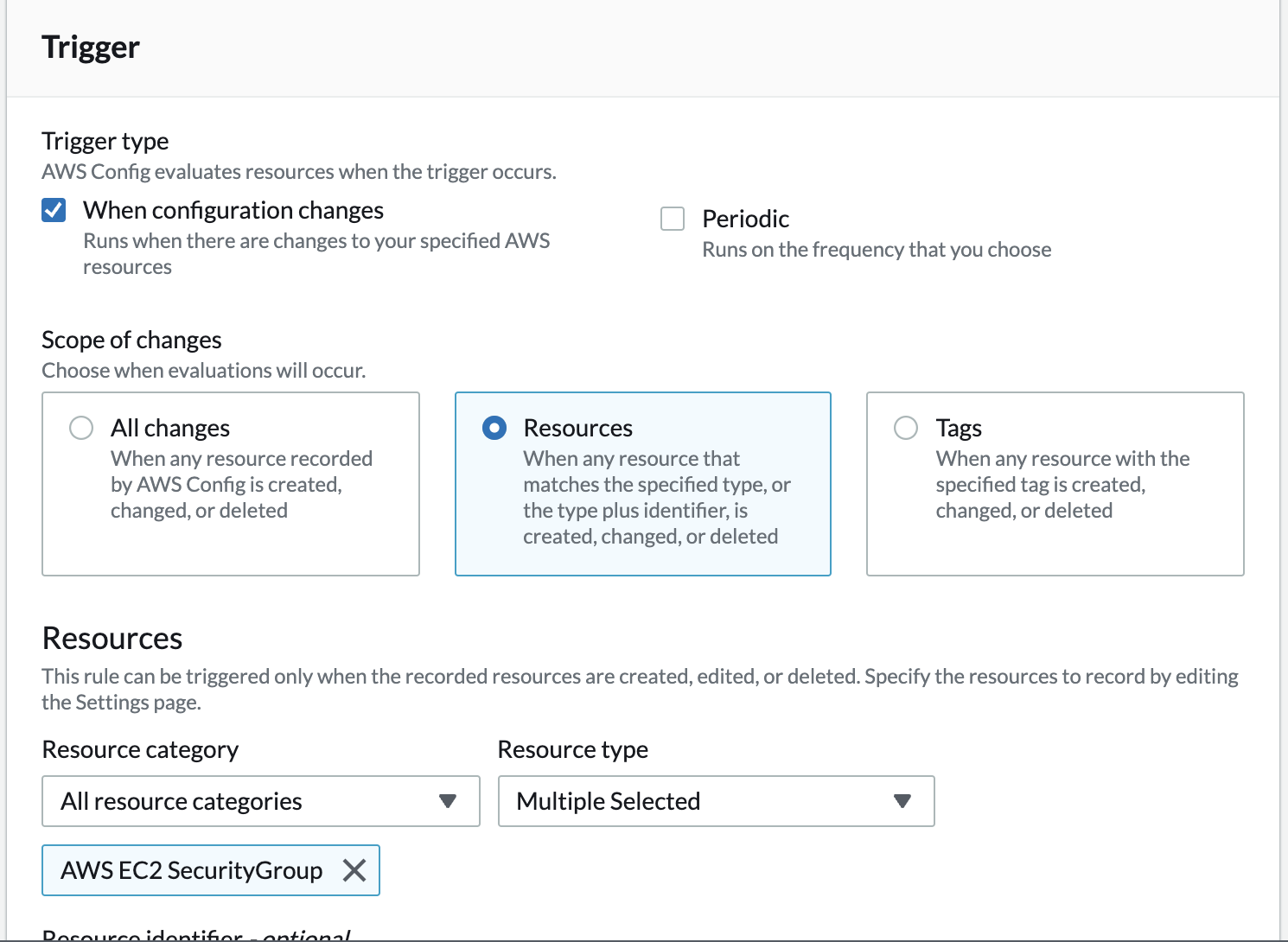


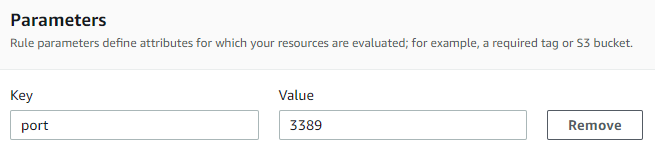
8. Return to the AWS Config **Add custom rule** browser tab and enter the following (paste the **AWS Lambda function ARN** first while it's in your clipboard):

* **Name**: disallow-rdp-ingress
* **Description**: Check security groups for allowing incoming RDP TCP traffic. Disallow the traffic to become compliant.
* **AWS Lambda function ARN**: Paste the ARN copied from the AWS Lambda function page
* **Trigger**
  + **Trigger type**: When configuration changes
  + **Scope of changes**: EC2
  + **Resources**: SecurityGroup
* **Rule parameters**
  + **Key**: port (lower-case)
  + **Value**: 3389

RDP traffic is over tcp port 3389 by default. There are ways to get RDP traffic around this rule, but it is only for demonstration purposes. **Rule parameters** are a way to make your custom rules generalizable. The same rule could be used to block any desired port using different settings of the **Rule parameters**. If you are curious about the code, the values set for **Rule parameters** correspond to ruleParameters in the code above.



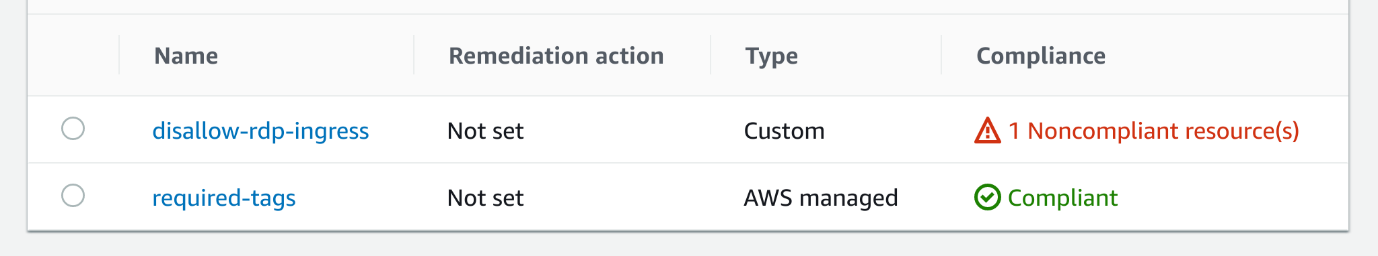




9. Click **Next**, and then **Add rule**.

Your custom rule automatically begins evaluating.

10. After a minute, refresh your browser tab to update the **Rules** page:



You will see the rule reports that there is **1 Noncompliant** **resource(s)**. You will now investigate and enforce compliance.

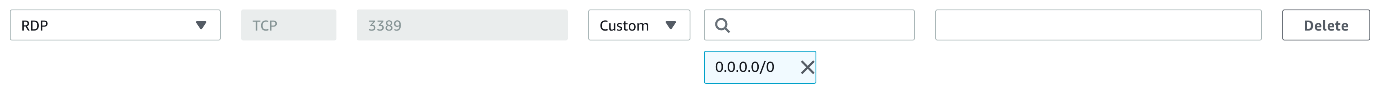
11. Click on **disallow-rdp-ingress**in the **Rules**table.

12. Click the **Noncompliant** security group and click **Manage resource**.

13. Click the **Inbound** **rules** tab.

Notice the security group is allowing incoming RDP traffic. That is causing the noncompliance.

14. Click **Edit inbound rules**.

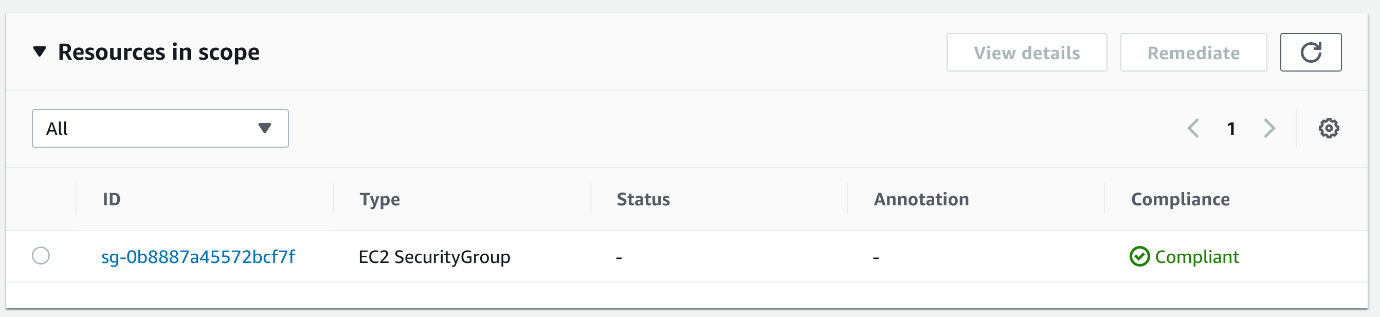
15. Click **Delete**on the right side of the **RDP**row:

16. Click **Save rules**.

17. Navigate back to **Service > Management & Governance > Config > Rules** and click on**disallow-rdp-ingress**.

18. In the **Resources in scope** section, select **All** in drop-down on the left-hand side.

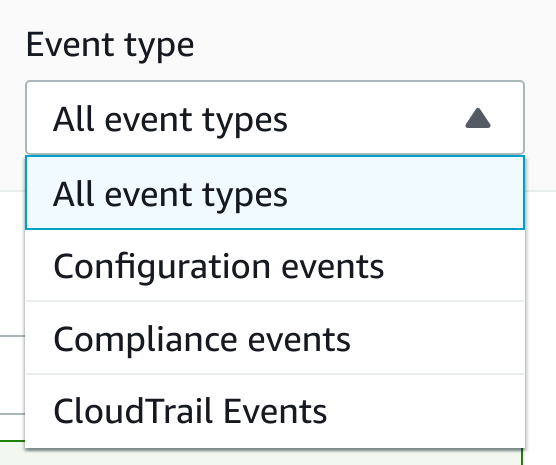
The rule is configured to evaluate on configuration changes. After a few minutes, the configuration change will be recorded and by refreshing the page, you will see the security group has changed to the **Compliant** state:



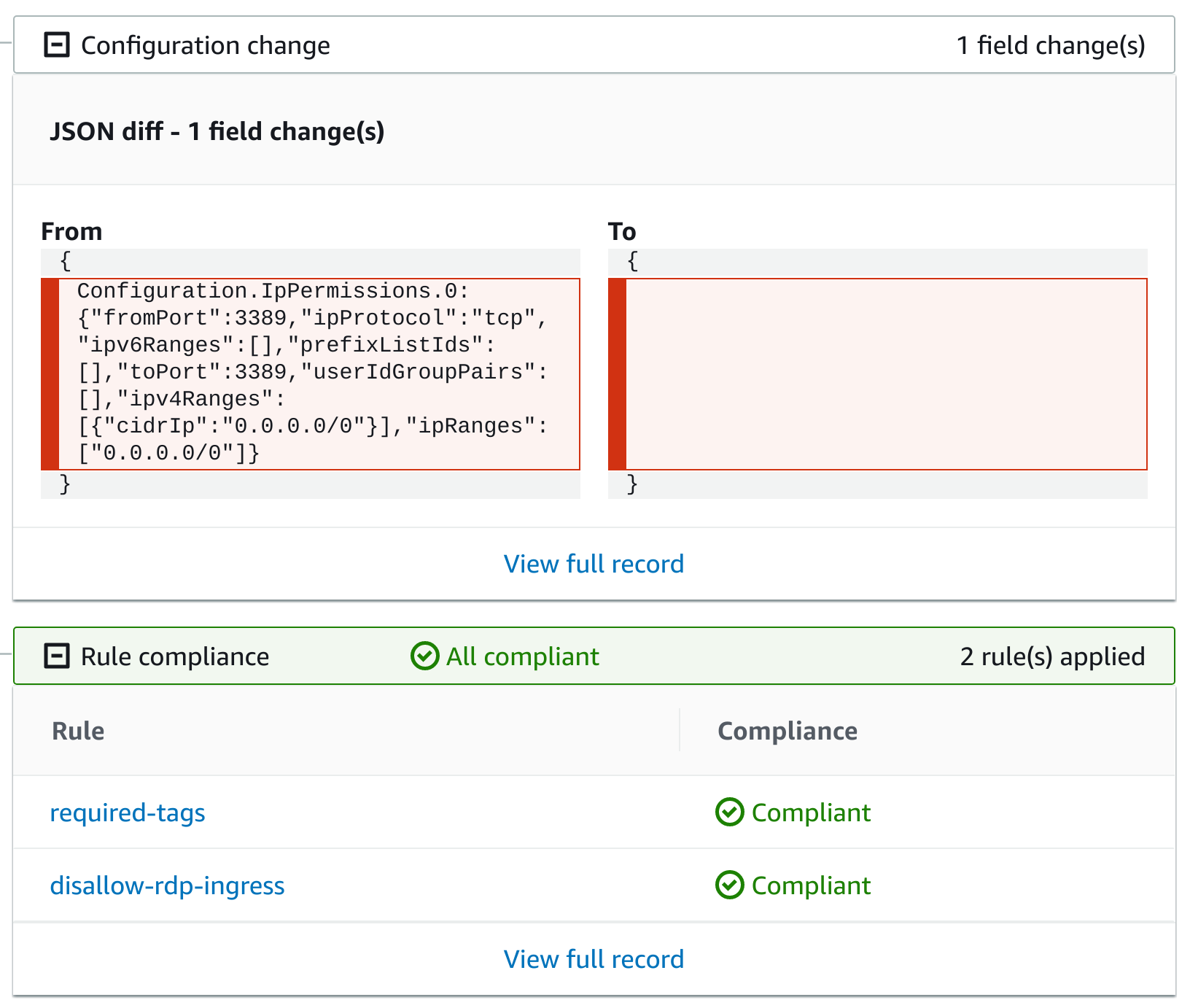
Note: You can click **Actions >** **Re-evaluate**button near the top of the page to check the rule immediately. This will allow the **Compliance**state to be updated, but it won't speed up the time it takes for configuration items to be recorded in the next instruction. This is because the code above directly queries the security group instead of depending on a configuration item.

19. Click on the security group name followed by **Resource timeline** and view the **Events** section.

20. In the **Event type** drop-down, ensure **All event types** is selected:



After a couple of minutes, you will see a new **Configuration change** event:



The latest **Configuration change** event will be a record of the RDP IpPermission ingress rule being removed from the security group.

Further down under **CloudTrail Events**, the **RevokeSecurityGroupIngress**event that triggered the change has been recorded in CloudTrail and linked to by Config.

Note: The CloudTrail event will take a few more minutes before showing up in the **Events** list. Refresh the browser tab once every minute or so.

### Summary

In this Lab Step, you created a custom rule for AWS Config. The rule used a Lambda function that detects if a security group is allowing incoming TCP traffic over the default RDP port. You also got more practice investigating non-compliant resources and modifying configurations to bring about compliance.