WORKSHOP INTRODUCTION

This workshop provides an introduction to chaos engineering using AWS tooling, with a focus on AWS Fault Injection Simulator (AWS FIS). It introduces the core elements of chaos engineering: stress, observe, and improve. You will learn how to use AWS FIS and other AWS tools to inject faults in you infrastructure to validate your system's resilience as well as verifying your alarms, observability, and monitoring practices.

Target audience

This is a technical workshop introducing chaos engineering concepts for dev, QA and Ops teams. For best results, the participants should have familiarity with the AWS console as well as some proficiency with command-line tooling.

Additionally, chaos engineering is about proving or disproving an hypothesis of how a particular fault might affect the overall system behavior (steady-state) so an understanding of the systems being disrupted is helpful but not required to do the workshop.

Duration

When run in a prepared AWS account the core sections (EC2, RDS, SSM) of the workshop will take between 1-2h. The whole workshop about 2-4h. Using the Amazon DevOps Guru section will require an additional 2-24h of wait time after the infrastructure has been configured.

When run in a customer account, deploying the workshop's core infrastructure will require an additional 45min.

Cost

When run in a private customer account, this workshop will incur costs on the order of USD1/h for the infrastructure created. Please ensure you clean up all infrastructure after finishing the workshop to prevent continuing expenses. You can find instructions in the **Cleanup** section.

START THE WORKSHOP

To start the workshop, follow one of the following depending on whether you are…

- …running the workshop on your own (in your own account), or
- …attending an AWS hosted event (using AWS provided hashes)

Once you have completed with either setup, continue with **Region Selection**

...ON YOUR OWN

Running the workshop on your own

Warning

Only complete this section if you are running the workshop on your own. If you are at an AWS hosted event (such as re:Invent, Kubecon, Immersion Day, etc), go to Start the workshop at an AWS event.

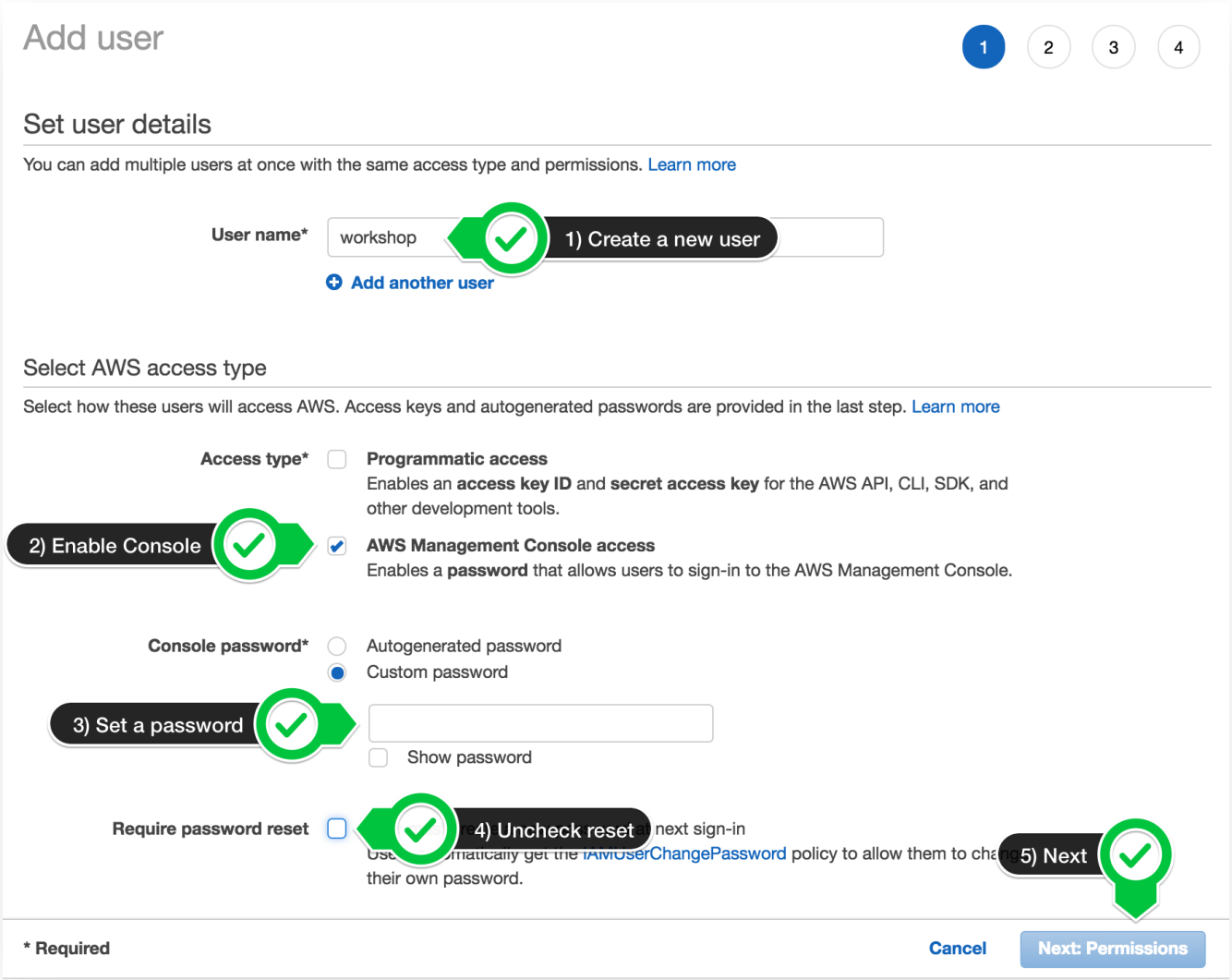
- Create an AWS account

CREATE AN AWS ACCOUNT

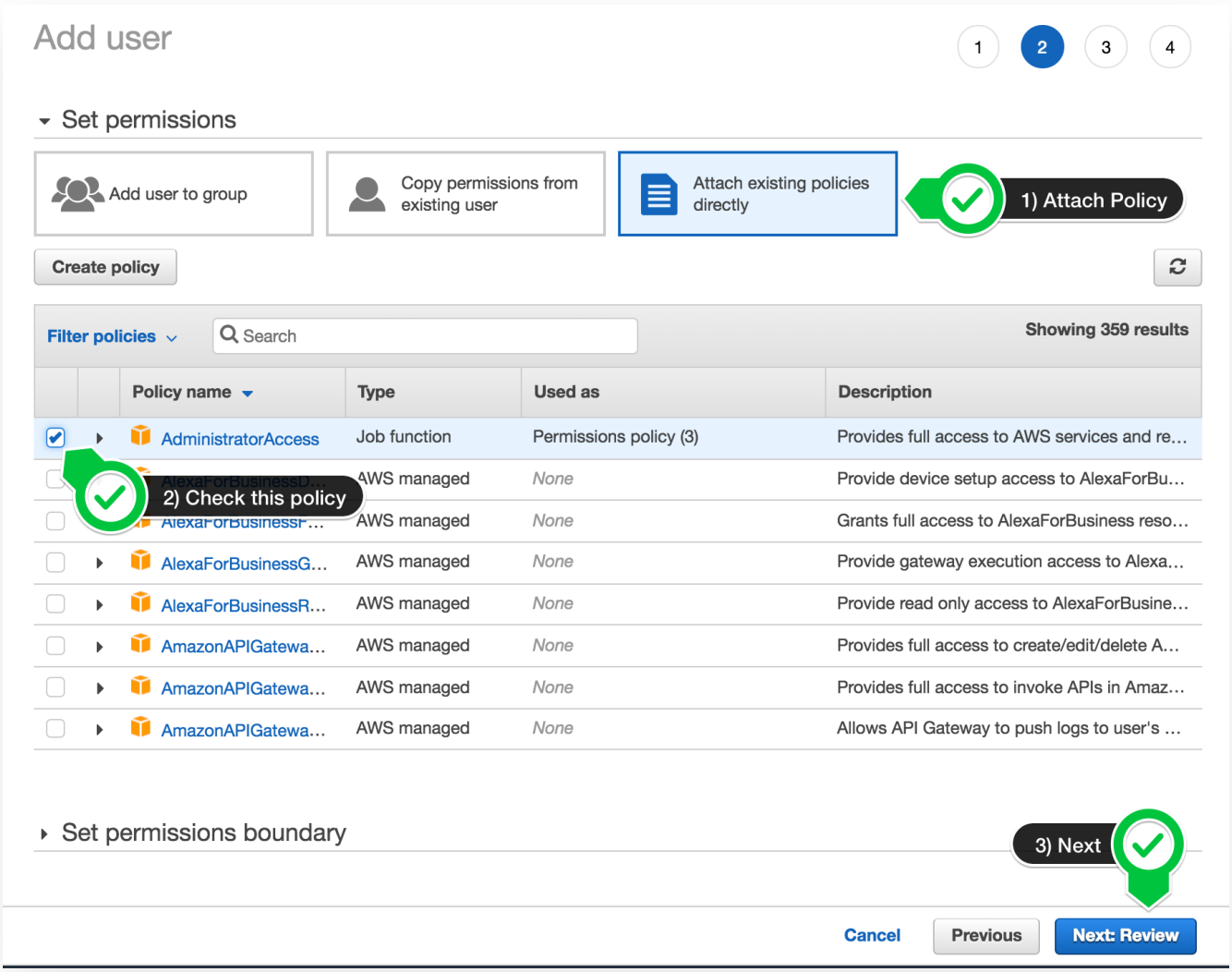
Warning

Your account must have the ability to create new IAM roles and scope other IAM permissions.

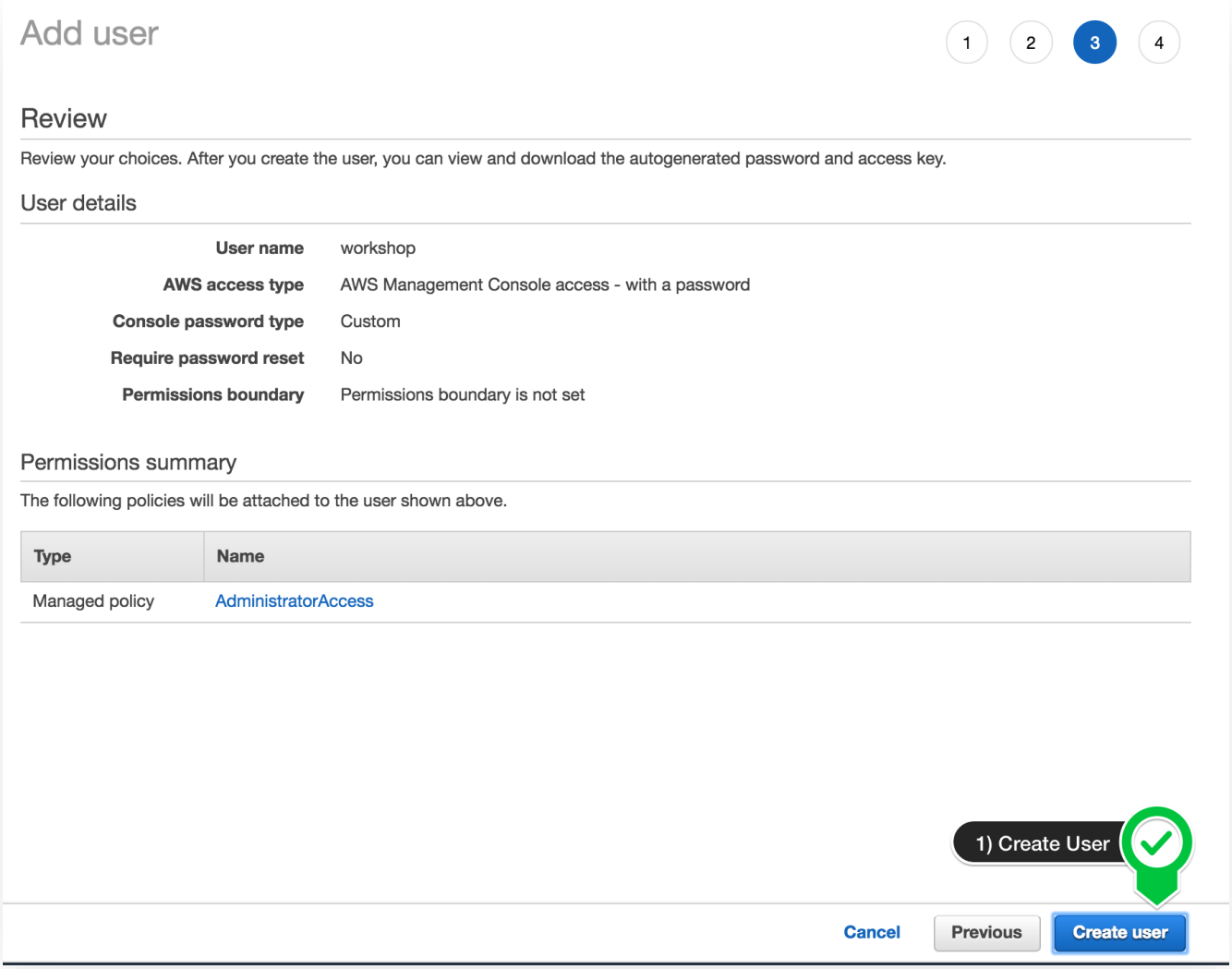
1. If you don't already have an AWS account with Administrator access: create one now by clicking here
2. Once you have an AWS account, ensure you are following the remaining workshop steps as an IAM user with administrator access to the AWS account: Create a new IAM user to use for the workshop
3. Enter the user details:



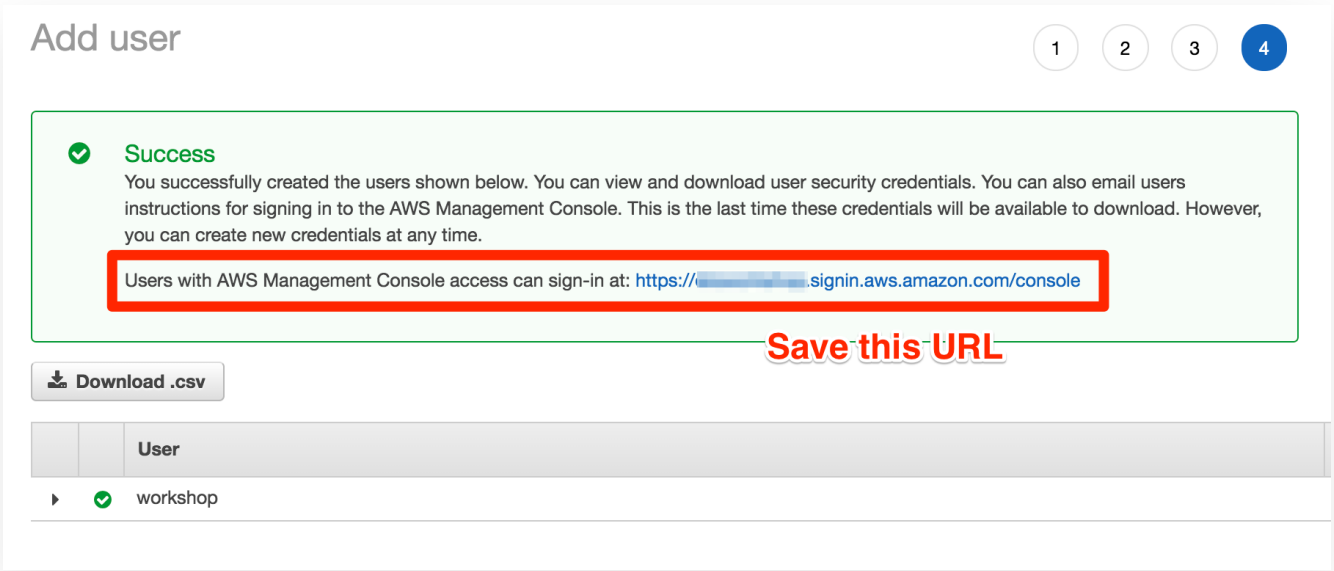
4. Attach the AdministratorAccess IAM Policy:



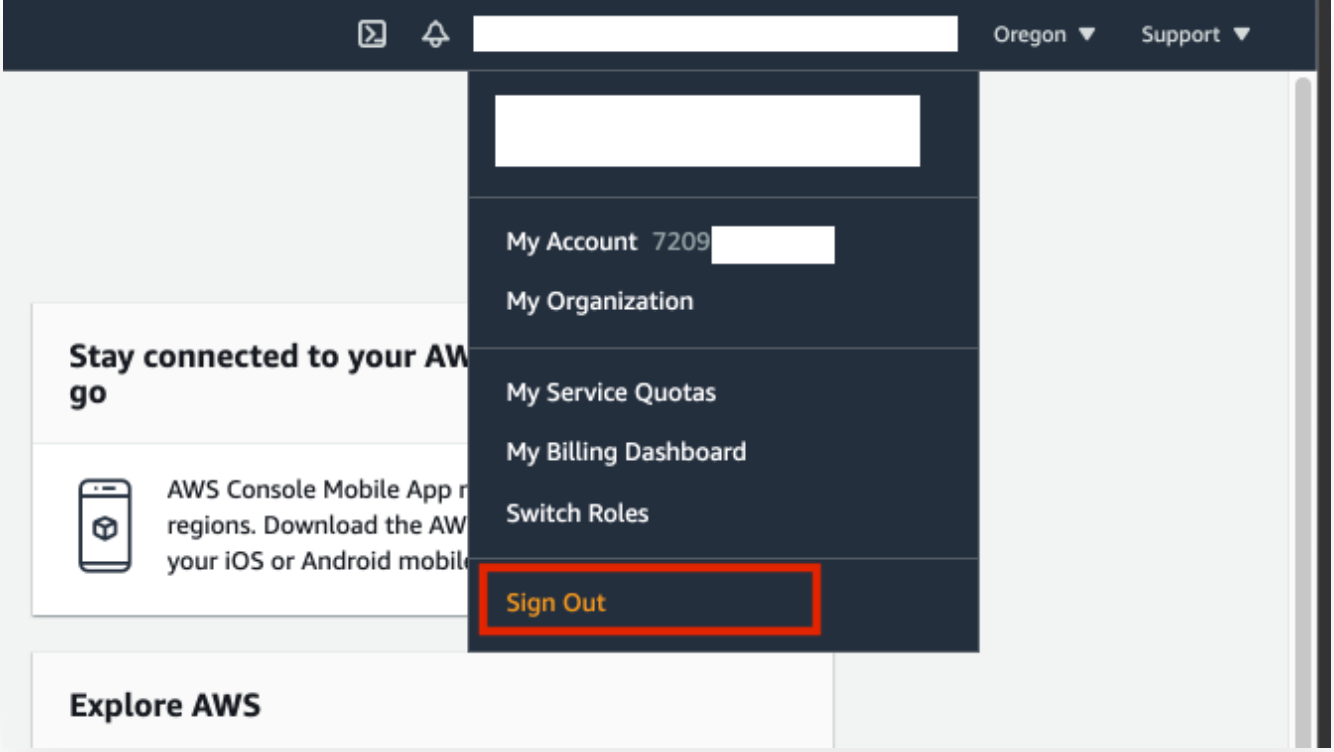
5. Click to create the new user:



6. Take note of the sign-in URL and save:



7. Sign out of your current AWS Console session: on the top menu, click on your login and select “Sign out”



8. Sign in to a new AWS Console session by using the sign-in URL saved and the newly created user credentials.

9. Once you have completed the steps above, you can head straight to the **Region Selection**.

...AT AN AWS EVENT

Running the workshop at an AWS Event

Warning

Only complete this section if you are at an AWS hosted event (such as re:Invent, AWS Summit, Immersion Day, or any other event hosted by an AWS employee). If you are running the workshop on your own, go to: Start the workshop on your own.

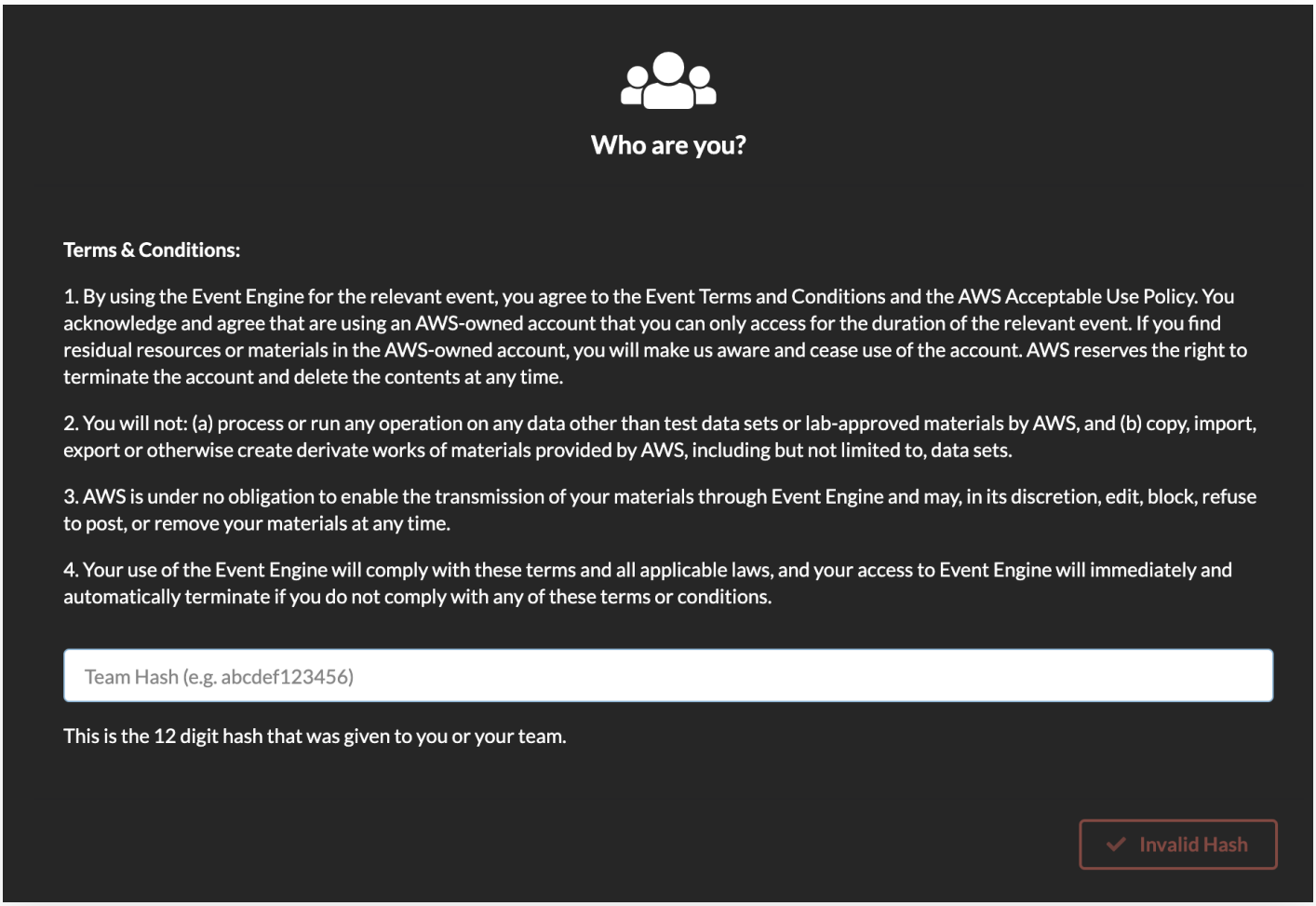
- AWS Workshop Portal

AWS WORKSHOP PORTAL

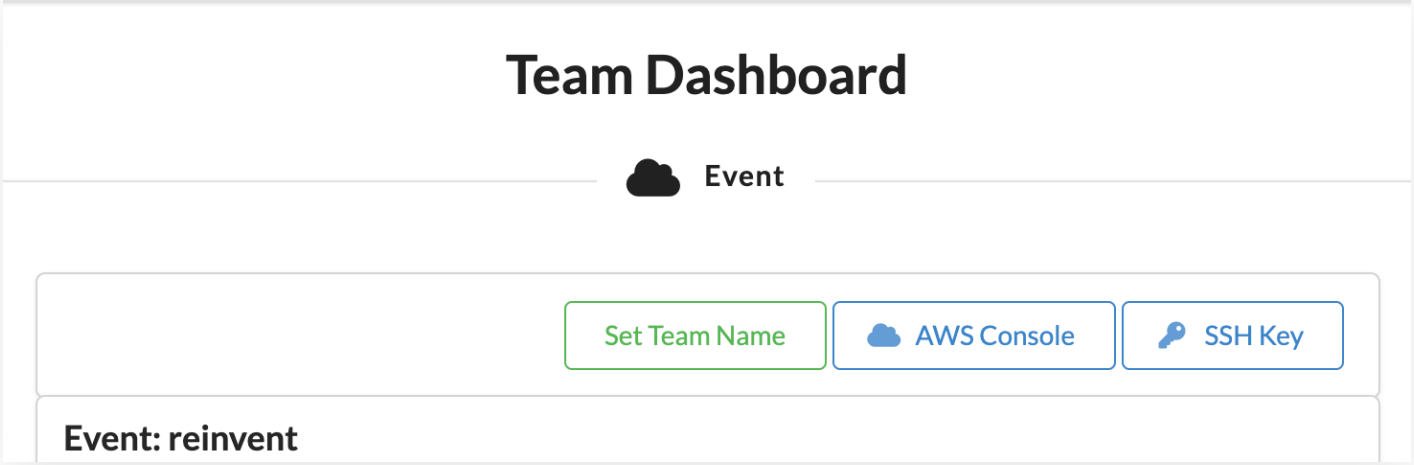
Login to AWS Workshop Portal

This workshop creates an AWS account and a Cloud9 environment. You will need the **Participant Hash** provided upon entry, and your email address to track your unique session.

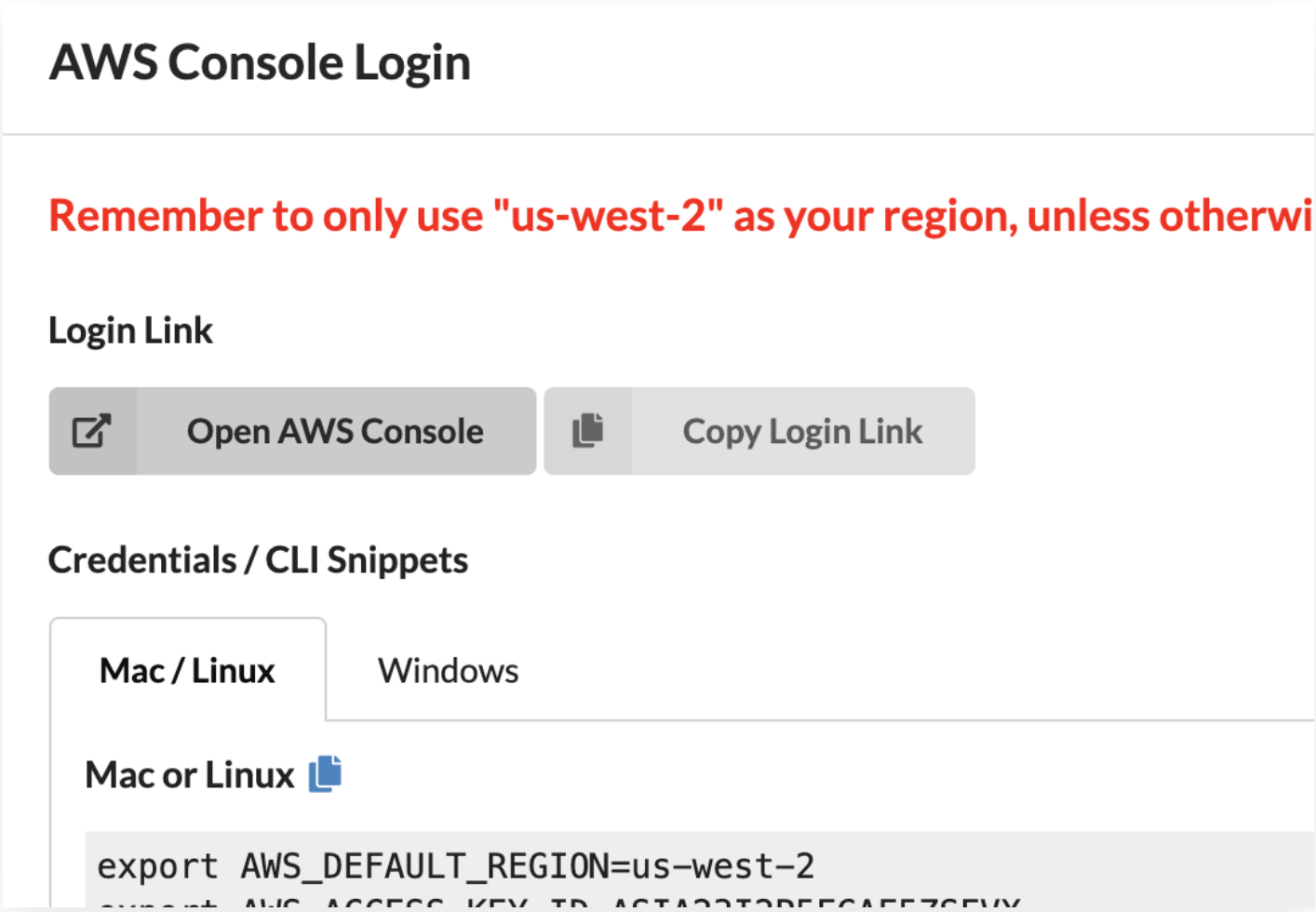
Connect to the portal by clicking the button or browsing to https://dashboard.eventengine.run/. The following screen shows up.



Enter the provided hash in the text box. The button on the bottom right corner changes to **Accept Terms & Login**. Click on that button to continue.



Click on **AWS Console** on dashboard.



Take the defaults and click on **Open AWS Console**. This will open AWS Console in a new browser tab.

Once you have completed the steps above, you can head straight to the **Region Selection**.

REGION SELECTION

This workshop relies heavily on AWS Fault Injection Simulator (AWS FIS) and assumes you will be running all your experiments in the same region. AWS FIS is currently available in the regions listed here.

If not otherwise instructed, please *pick an appropriate region* from the list and perform all the operations in the workshop in that region.

CREATE A WORKSPACE

Info

A list of supported browsers for AWS Cloud9 is found here.

Tip

Ad blockers, javascript disablers, and tracking blockers should be disabled for the cloud9 domain, or connecting to the workspace might be impacted. Cloud9 requires third-party-cookies. You can whitelist the specific domains.

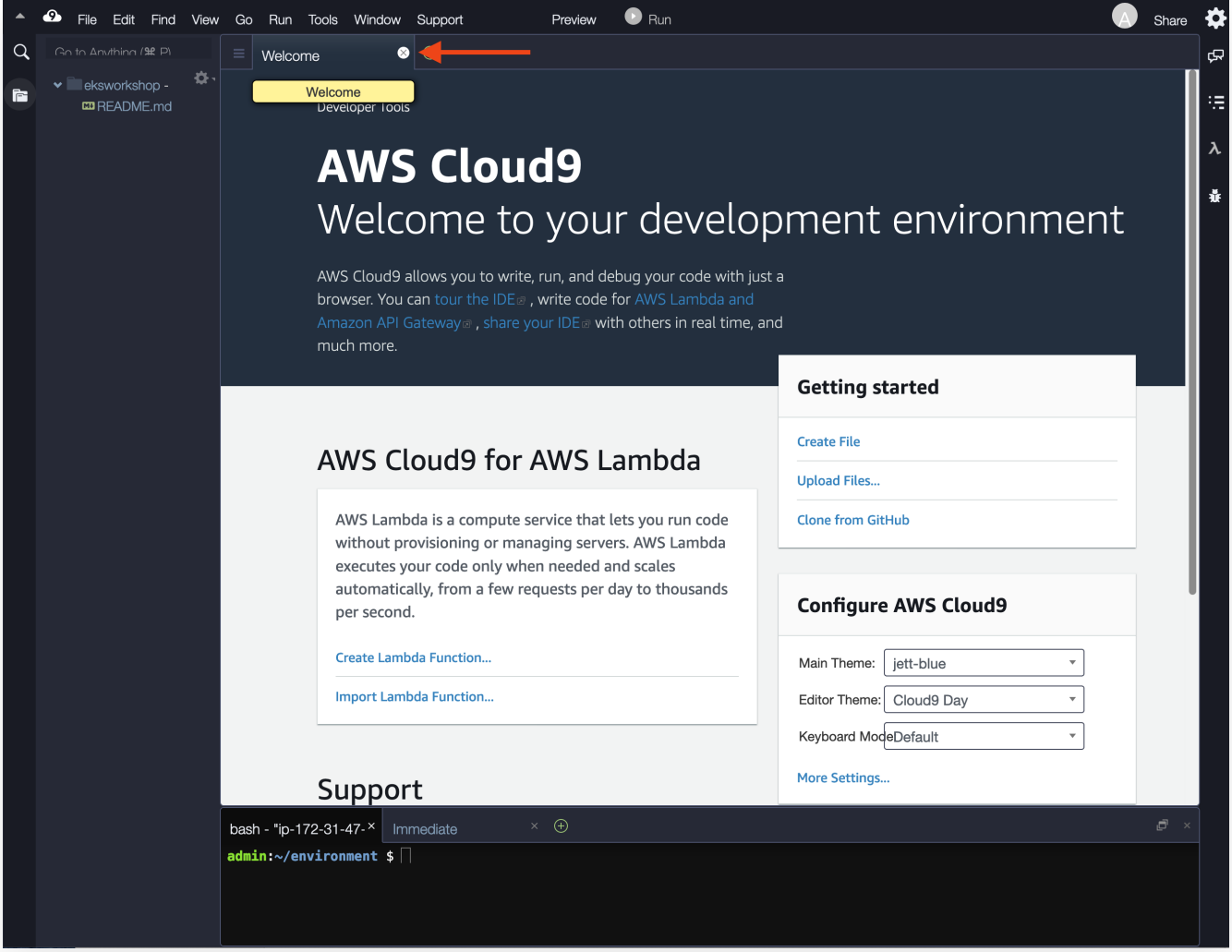
Launch Cloud9 in the region selected previously

Using the region selected in **Region Selection**, navigate to the Cloud9 console.

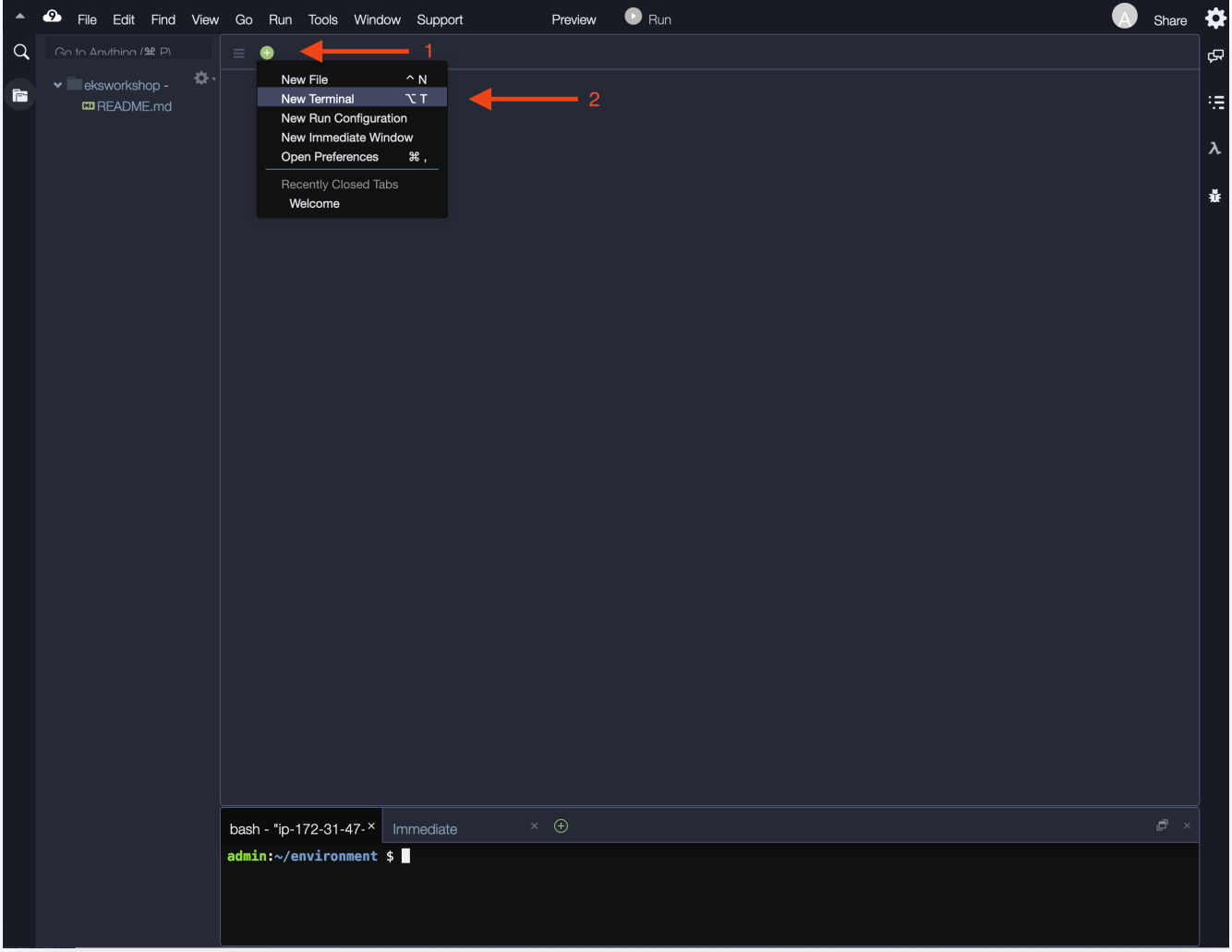
- Select **Create environment**
- Name it **fisworkshop**, click **Next step**.
- Since we only need to access our Cloud9 environment via web browser, please select the **Create a new no-ingress EC2 instance for environment (access via Systems Manager)** under the Environment Type.
- Choose **t3.small** for instance type, go through the wizard with the default values and click **Create environment**

When it comes up, customize the environment by:

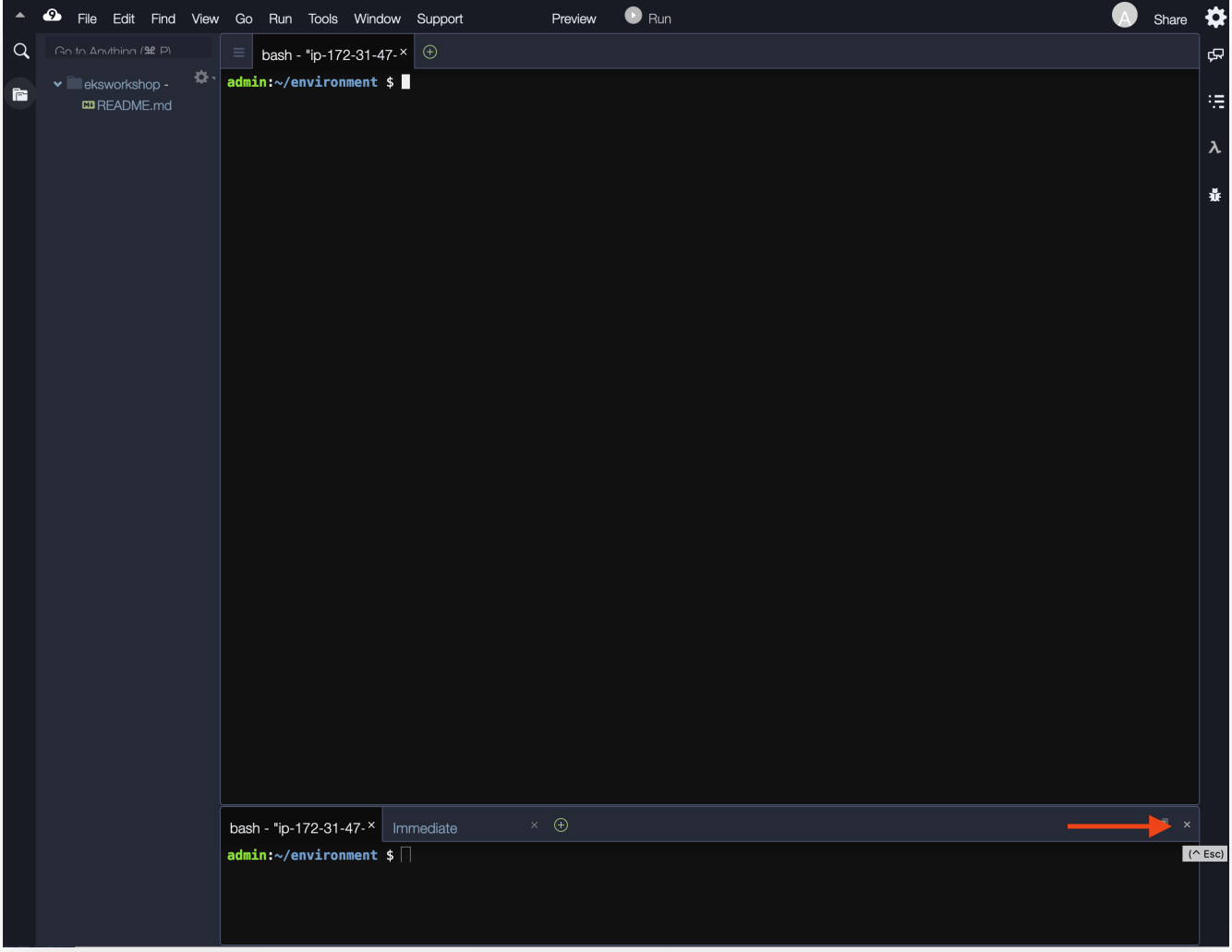
- Closing the **Welcome tab**



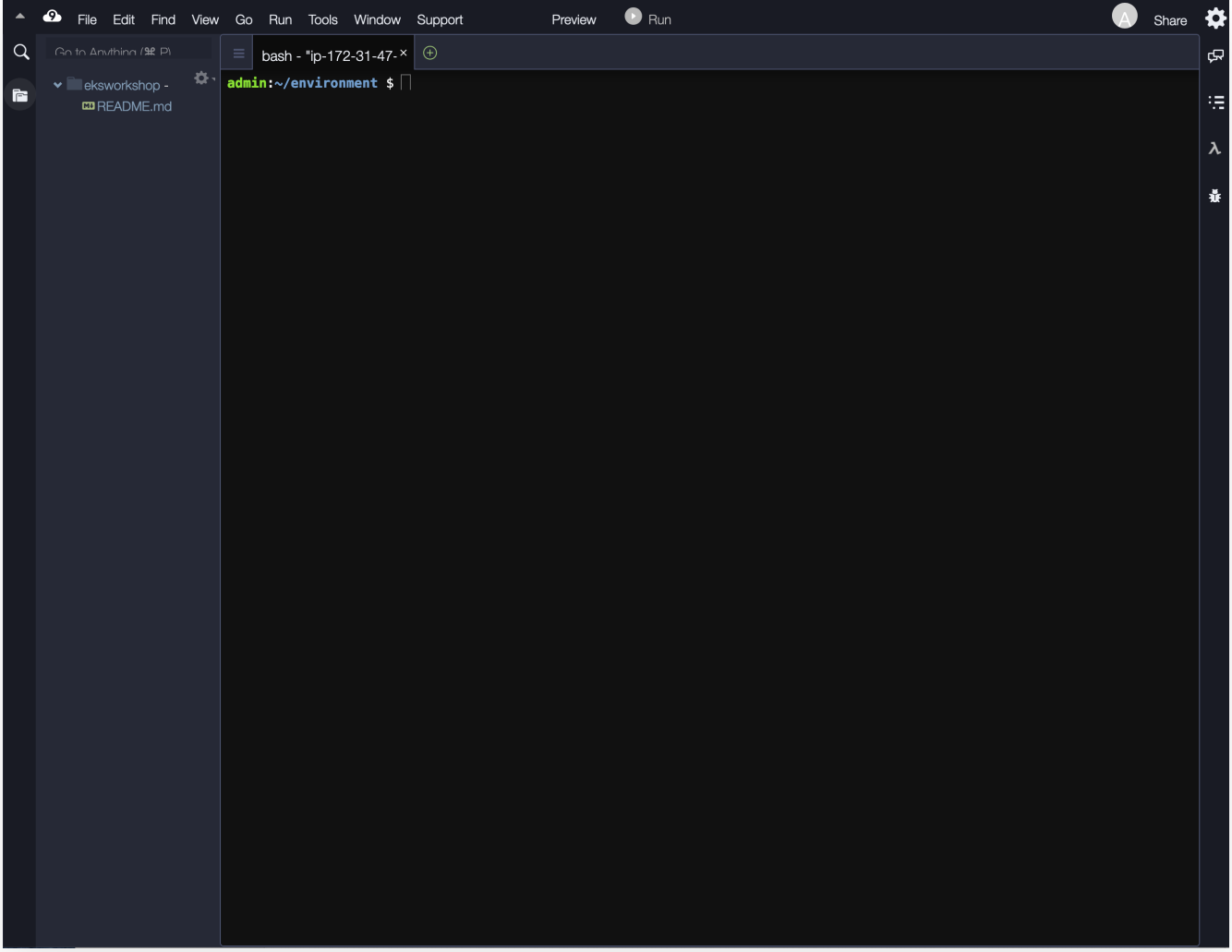
- Opening a new **terminal** tab in the main work area



- Closing the lower work area



- Your workspace should now look like this



Increase the disk size on the Cloud9 instance

Info

Some commands we will run require more than the default disk allocation on a cloud9 workspace. The following command adds more disk space to the root volume of the EC2 instance that Cloud9 runs on.

Copy/Paste the following code in your cloud9 terminal (you can paste the whole block at once).
Once the command completes, we reboot the instance and it could take a minute or two for the IDE to come back online.

```
pip3 install --user --upgrade boto3
export instance_id=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
python -c "import boto3
import os
from botocore.exceptions import ClientError
ec2 = boto3.client('ec2')
volume_info = ec2.describe_volumes(
    Filters=[
```

```
        {
            'Name': 'attachment.instance-id',
            'Values': [
                os.getenv('instance_id')
            ]
        }
    ]
)
volume_id = volume_info['Volumes'][0]['VolumeId']
try:
    resize = ec2.modify_volume(
            VolumeId=volume_id,
            Size=30
    )
    print(resize)
except ClientError as e:
    if e.response['Error']['Code'] == 'InvalidParameterValue':
        print('ERROR MESSAGE: {}'.format(e))"
if [ $? -eq 0 ]; then
    sudo reboot
fi
```

Update tools and dependencies

Info

The instructions in this workshop assume you are using a bash shell in a linux-like environment. They also rely on a number of tools. Follow these instructions to install the required tools in a cloud9 workspace:

Copy/Paste the following code in your cloud9 terminal (you can paste the whole block at once).

```
# Update to the latest stable release of npm and nodejs.
nvm install stable

# Install typescript
npm install -g typescript

# Install CDK
npm install -g aws-cdk

# Install the jq tool
sudo yum install jq -y
```

PROVISION AWS RESOURCES

Warning

Only complete this section if you are running the workshop on your own. If you are at an AWS hosted event (such as re:Invent, Kubecon, Immersion Day, etc), these steps have already been executed for you.

Before we start running fault injection experiments we need to provision our resources in the cloud. The rest of the workshop uses these resources.

Clone the repository

```
cd ~/environment
git clone https://github.com/aws-samples/aws-fault-injection-simulator-
workshop.git
```

Deploy the resources

```
cd aws-fault-injection-simulator-workshop
cd resources/templates
./deploy-parallel.sh
```

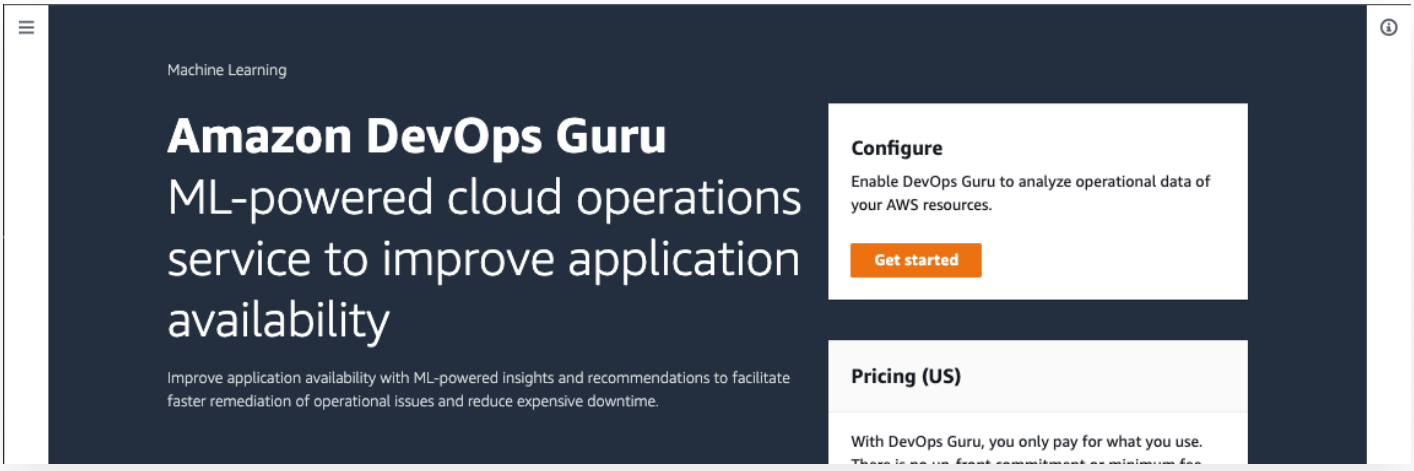
It can take up to 30 minutes to complete.

OPTIONAL: SETUP FOR DEVOPS GURU

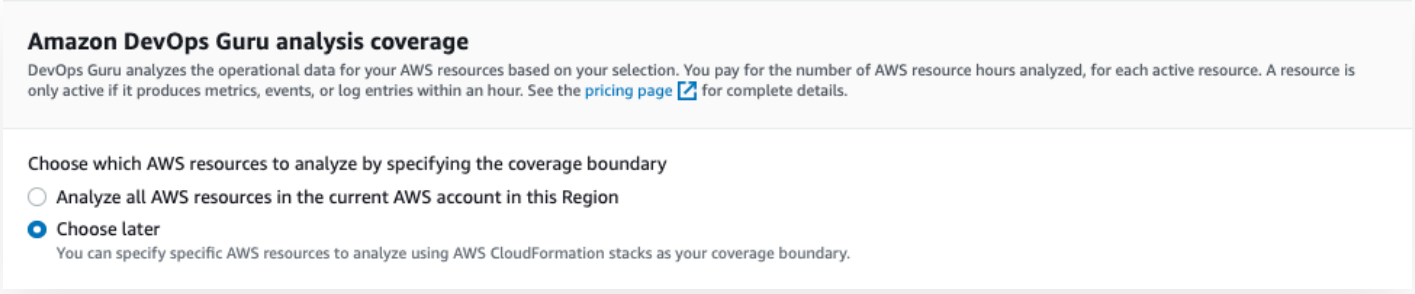
Warning

Only complete this section if you are planning to explore the DevOps Guru section at the end of the workshop. If you are planning to explore DevOps Guru in this way please allow sufficient time for DevOps Guru to perform initial resource discovery and baselining. Depending on the number of resources in the account/region you select this may take from 2-24h.

Navigate to the DevOps Guru console and click select the "Get Started" button:



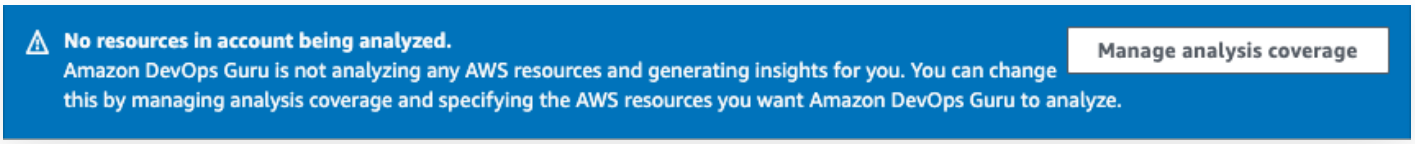
For "Amazon DevOps Guru analysis coverage" select "Choose later" if you will only be exploring as part of this workshop. Otherwise you can choose "Analyze all AWS resources in the current AWS account in this Region" but it may take more time and incur more cost to get started.



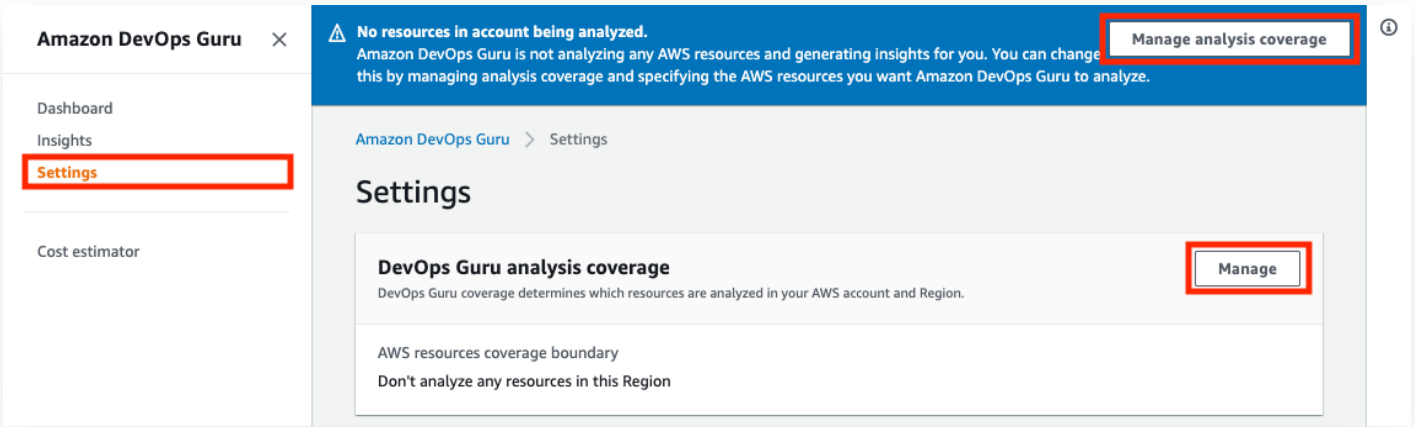
During this workshop we will not be exploring SNS notifications and thus don't need to specify an SNS topic.

Select "Enable"

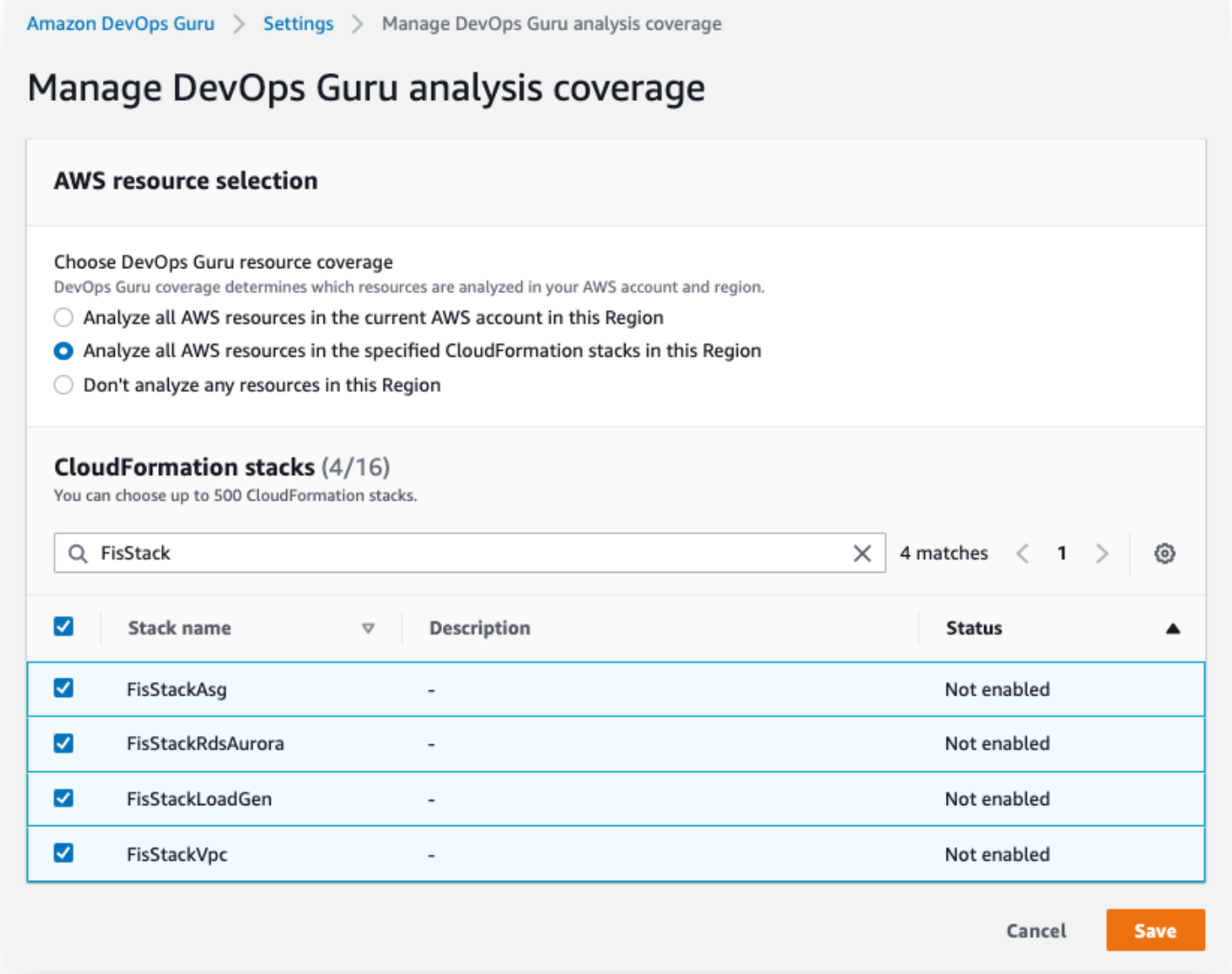
If you set coverage to "Choose later" you should now see an information banner notifying you that you have not yet selected resources:



Select the "Manage analysis coverage" option in the banner or navigate to the DevOps Guru console, choose "Settings" and select "Manage" option under "DevOps Guru analysis coverage":



Select all the Fis stacks:


Amazon DevOps Guru
Settings
Manage DevOps Guru analysis coverage
Manage DevOps Guru analysis coverage
AWS resource selection
Choose DevOps Guru resource coverage
DevOps Guru coverage determines which resources are analyzed in your AWS account and region.
Analyze all AWS resources in the current AWS account in this Region
Analyze all AWS resources in the specified CloudFormation stacks in this Region
Don't analyze any resources in this Region
CloudFormation stacks (4/16)
You can choose up to 500 CloudFormation stacks.
FisStack
4 matches
1
Stack name
Description
Status
FisStackAsg
-
Not enabled
FisStackRdsAurora
-
Not enabled
FisStackLoadGen
-
Not enabled
FisStackVpc
-
Not enabled
Cancel
Save

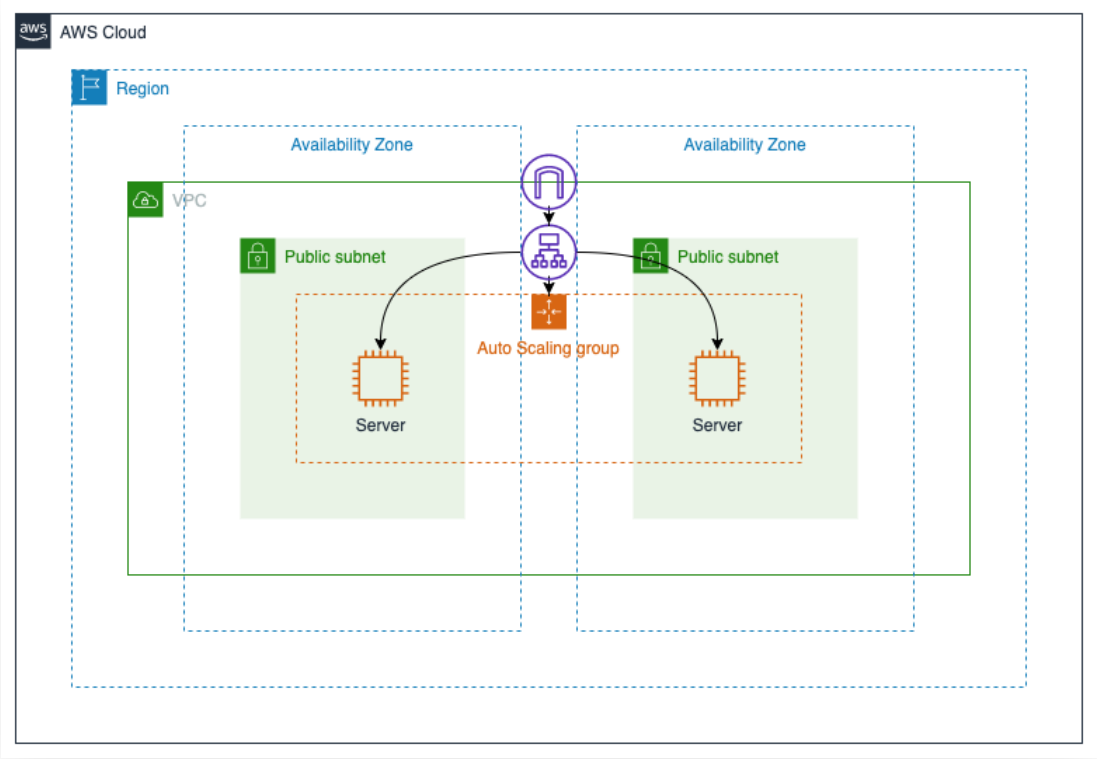
Select “Save”.

This workshop is broken into multiple chapters. The chapters are designed to be done in sequence. At the end of each chapter we include a “cheat” that should allow you to easily implement all the work required and allow you to move forward to the next chapter if you are already familiar with the material covered.

Chapters:

Architecture Diagrams

This workshop is focused on how to inject fault into an existing infrastructure. For this purpose the template in the **Getting Started** section sets up a variety of components. Throughout this workshop we will be showing you architecture diagrams focusing on only the components relevant to the section, e.g.:



You can click on these images to enlarge them.

› Click to expand if you are running a demo

BASELINING AND MONITORING

Before we start down injecting faults into our system we should consider the following thought experiment:

"If a tree falls in a forest and no one is around to hear it, does it make a sound?"

For the purpose of our fault injection experiments we can rephrase this in two ways:

"If part of our system is disrupted and we do not receive any irate calls from users, did anything break?"

"If part of our system is disrupted and sysops isn't alerted, did anything break?"

Think about this for a second. There is a distinct difference between those two statements because users and ops have very different experiences.

What the users see

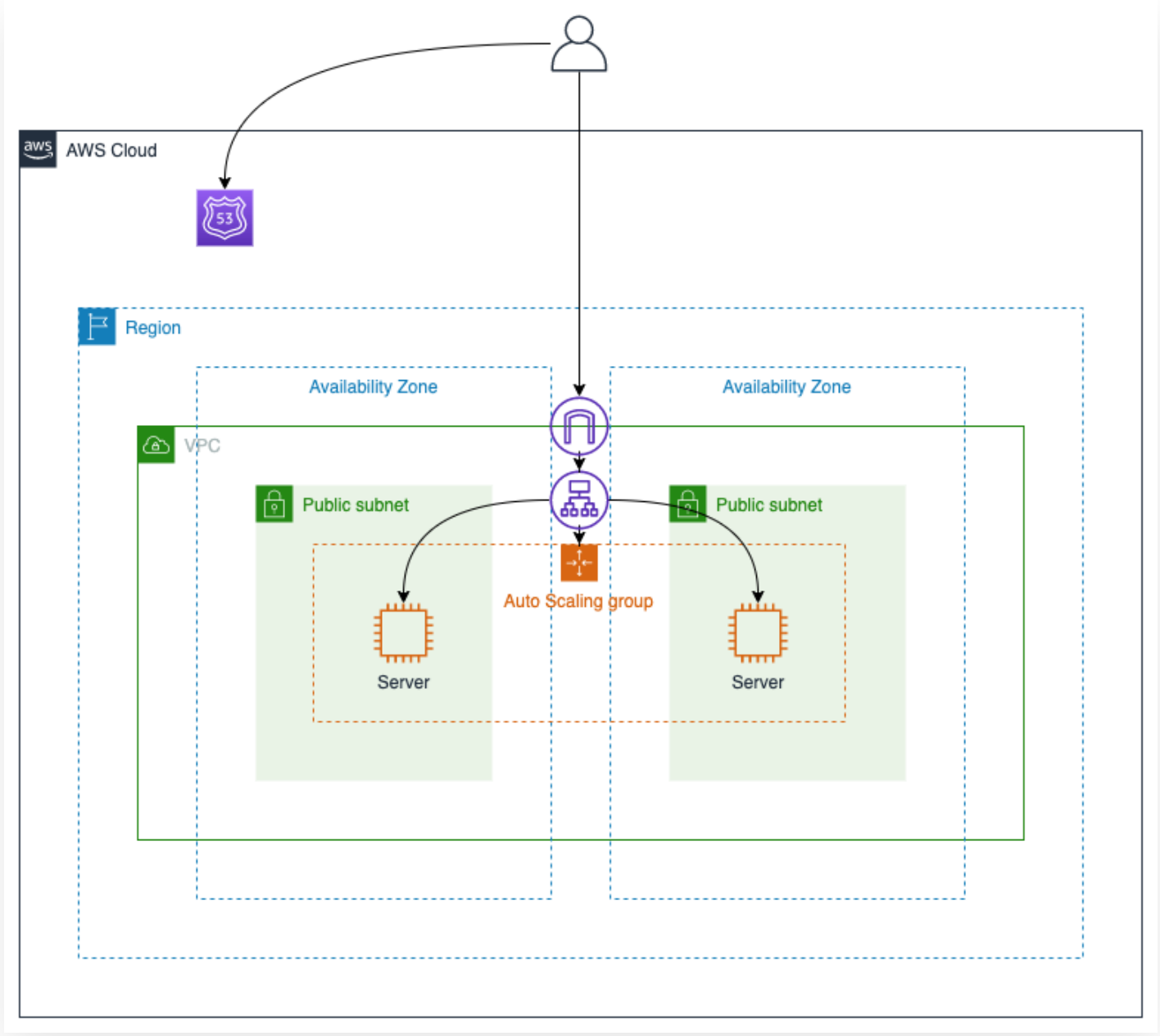
What the users see is immediate, e.g. the website not loading or loading slowly. What the users see is also an end-to-end test of all system components, and not all components of the system are in your purview, e.g. you cannot see the speed of the users' network connection or the state of their DNS caches. Finally an individual user can have an experience entirely different from all other users. For this workshop, this is particularly important for a particular edge case: developers and ops typically have better system configurations and better experiences than the average user but tend to rely on the anecdotal evidence of "it worked for me".

What sysops sees

Typically, what sysops see is a wealth of individual health and performance indicators. These often grow organically over time and especially after outages. Even where dashboards have been built with overall system health in mind, the metrics are delayed against the user experience and aggregate over the experience of many users, requiring extra effort to notice poor experiences specific to a subset of users.

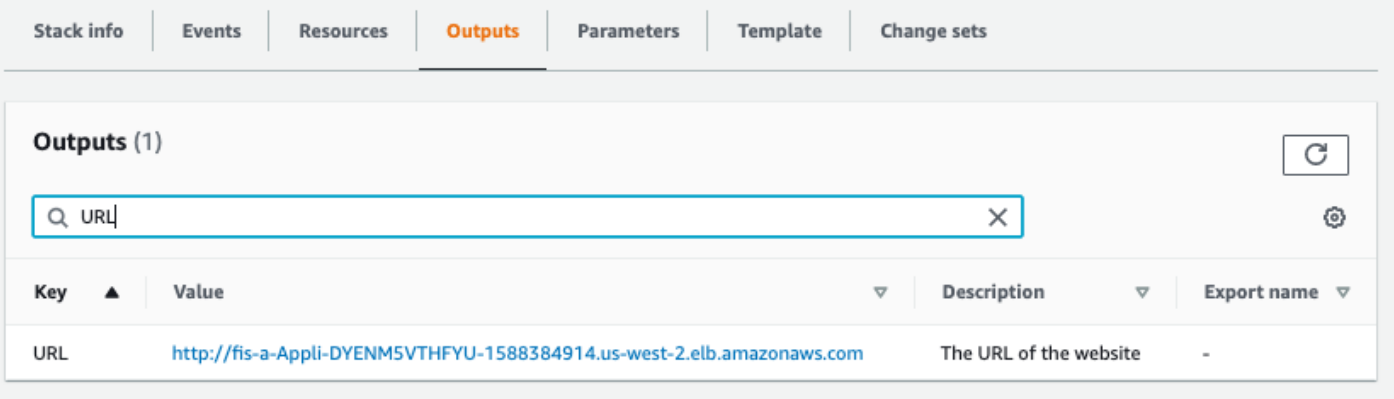
Setting up for fault injection

Before starting our first fault injection experiment, let's take a look at our most basic infrastructure:

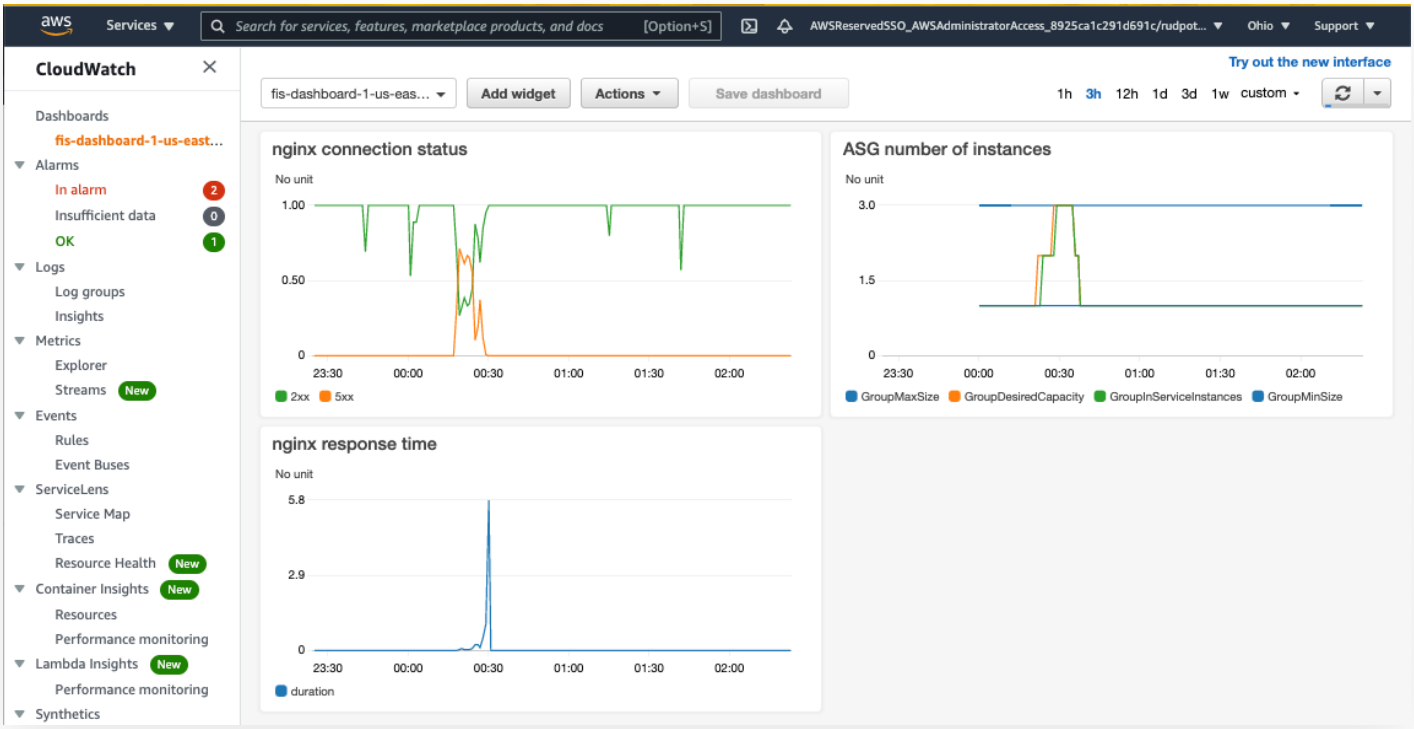


We have a user trying to access a website running on AWS. We have designed it for high availability. We used EC2 instances with an auto scaling group and a load balancer to ensure that users can always reach our website even under heavy load or if an instance suddenly fails.

Once you've started the template as described in **Getting Started** you can navigate to CloudFormation, select the "FisStackAsg" stack and click on the "Outputs" tab which will show you the server URL:



To gain visibility into the user experience from the sysops side we've used the cloudwatch agent to export our web server logs to cloudwatch logs and we created CloudWatch Logs metrics filters to track server response codes and speeds on a dashboard. Note that the dashboard's name is based on the region. If you chose another region the dashboard's name will be different. The dashboard also shows the number of instances in our Auto Scaling Group (ASG).



› Accessing the dashboard from the console

In the next section we will cover how to measure the user experience.

SYNTHETIC USER EXPERIENCE

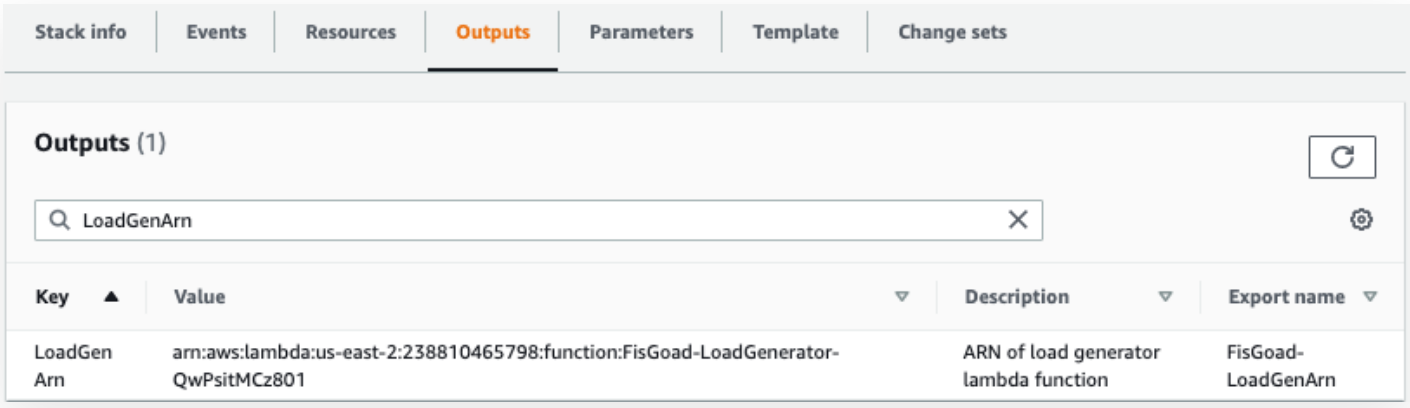
To gain insights from our fault injection experiments, we want to correlate user-experience with the sysops view from the previous section. In production, we could instrument the clients to send telemetry back to us, but in non-production we don't usually have sufficient load to do this. *You* also probably have better things to do than sit there clicking reload on a browser page while your experiment is running.

Generating load against our website

In the previous section, you navigated to the basic website setup as well as the sysops performance dashboard. Open a linux terminal and save the URL from the previous page in an environment variable:

```
export URL_HOME=[PASTE URL HERE]
```

Next, we need to generate load. There are many load testing tools available to generate a variety of load patterns. However, for the purpose of this workshop we have included a Lambda function that will make HTTP GET calls to our website and log performance data to CloudWatch. To find the Lambda function, navigate to CloudFormation, select the "**FisStackLoadGen**" stack, and click on the "**Outputs**" tab. It will show you the Lambda function ARN:



Save the Lambda function ARN in another environment variable:

```
export LAMBDA_ARN=[PASTE ARN HERE]
```

Finally, invoke the Lambda function using the AWS CLI:

```
aws lambda invoke \
  --function-name ${LAMBDA_ARN} \
  --payload "{
        \"ConnectionTargetUrl\": \"${URL_HOME}\",
        \"ExperimentDurationSeconds\": 180,
        \"ConnectionsPerSecond\": 1000,
        \"ReportingMilliseconds\": 1000,
        \"ConnectionTimeoutMilliseconds\": 2000,
        \"TlsTimeoutMilliseconds\": 2000,
        \"TotalTimeoutMilliseconds\": 2000
    }" \
  --invocation-type Event \
  invoke.txt
```

Warning

If you are running AWS CLI v2, you need to pass the parameter `--cli-binary-format raw-in-base64-out` or you'll get the error "Invalid base64" when sending the payload.

Now, let's generate some load. The invocation above will generate 1000 connections per second for 3 minutes. We expect our website's performance to degrade and for Auto Scaling to kick in.

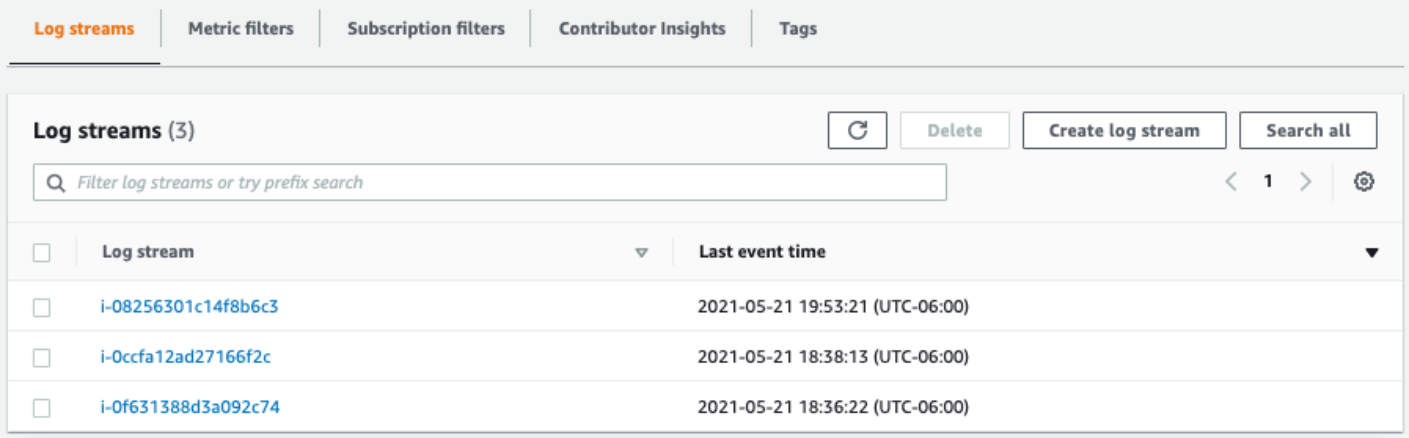
Explore impact of load

While our load is running let's explore the setup a little more.

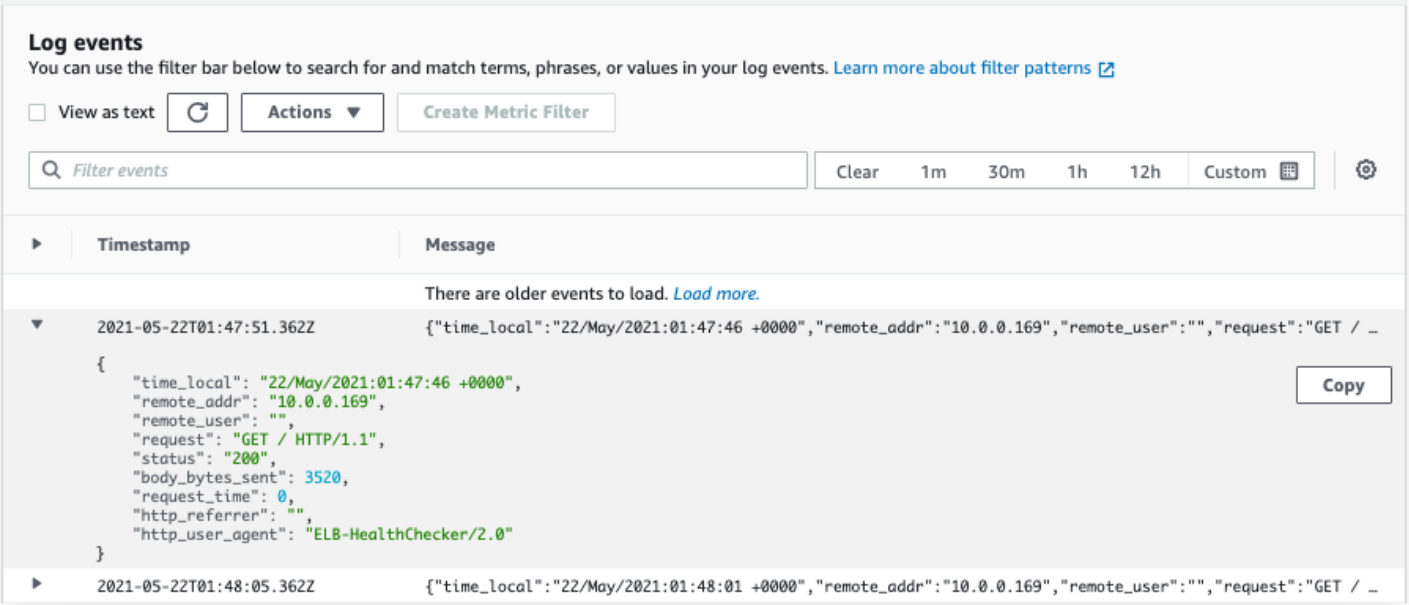
Webserver logs and metrics

The first thing we want to look at is our webserver logs. Because we are using AWS Auto Scaling, virtual machines can be terminated and recycled which means logs written locally on the EC2 instance won't be accessible anymore. Therefor, we have installed the Unified CloudWatch Agent on configured our webserver to write logs to a CloudWatch Log Group.

› Navigating to CloudWatch Log Groups

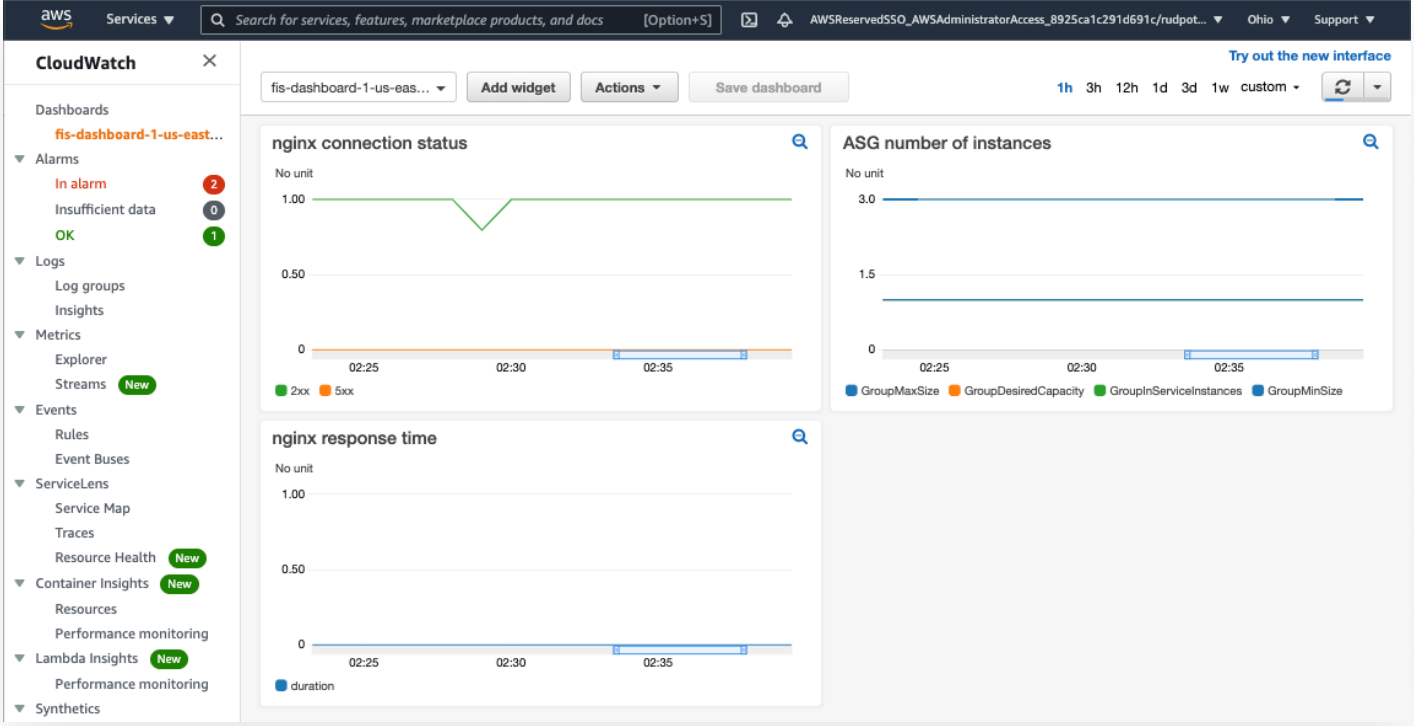


Click through on the topmost entry and expand any of the log lines. You may notice that we've modified the Nginx output format to use JSON instead of the default format:

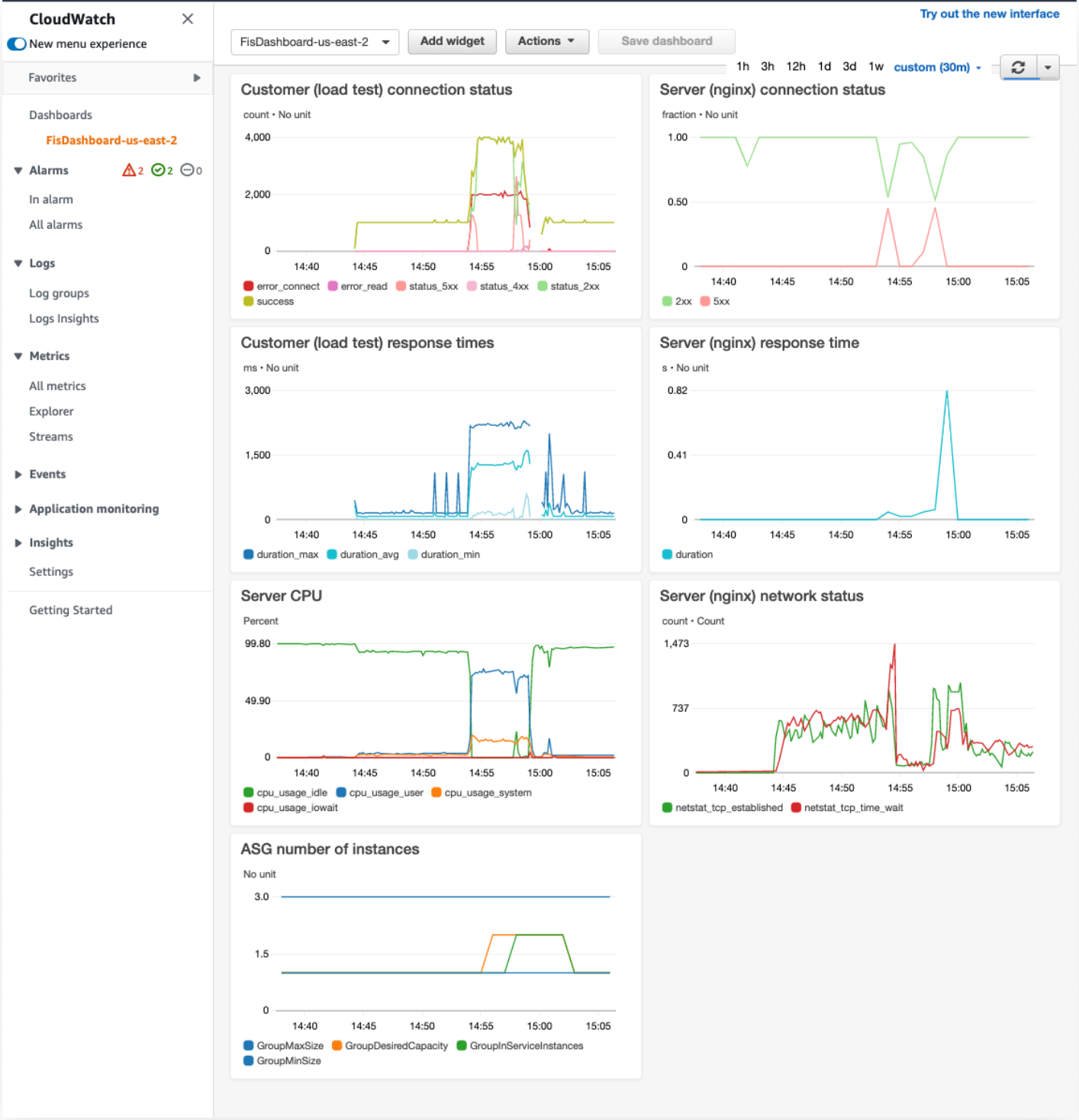


While not necessary, this makes it easy to create Metric Filters. Navigate back to the `/fis-workshop/asg-access-log` log group and select the "**Metric filters**" tab. You will see that we have created filters to extract the count of responses with HTTP `status` codes in the `2xx` (good responses) and `5xx` (bad responses) ranges. We also created a filter to select all entries that have a `request_time` set. The resulting metrics can be found under **Metrics** / **All metrics** / **Custom Namespaces** / **fisworkshop**. These are also the metrics for `nginx connection status` and `nginx response time` you saw on the dashboard in the previous section.

Let's look at our dashboard:



That's odd, did anything happen? According to nginx, it looks like nothing happened. Remember the falling tree in the forest and no one is around to hear it? We need to look at what the server CPU and the load runner. For this, we have added a more detailed dashboard:



Now, it's clearer what happened. We were requesting a small static page and Nginx is really efficient. In the `Server CPU` graph, we can see minimal CPU utilization correlating with the load data in the `Customer (load test)` graphs.

Increasing the load

Clearly, hitting a static page isn't a good test to validate our Auto Scaling configuration works as intended. Fortunately, the server also exposes a `phpinfo.php` page. Let's try loading that instead. Define another

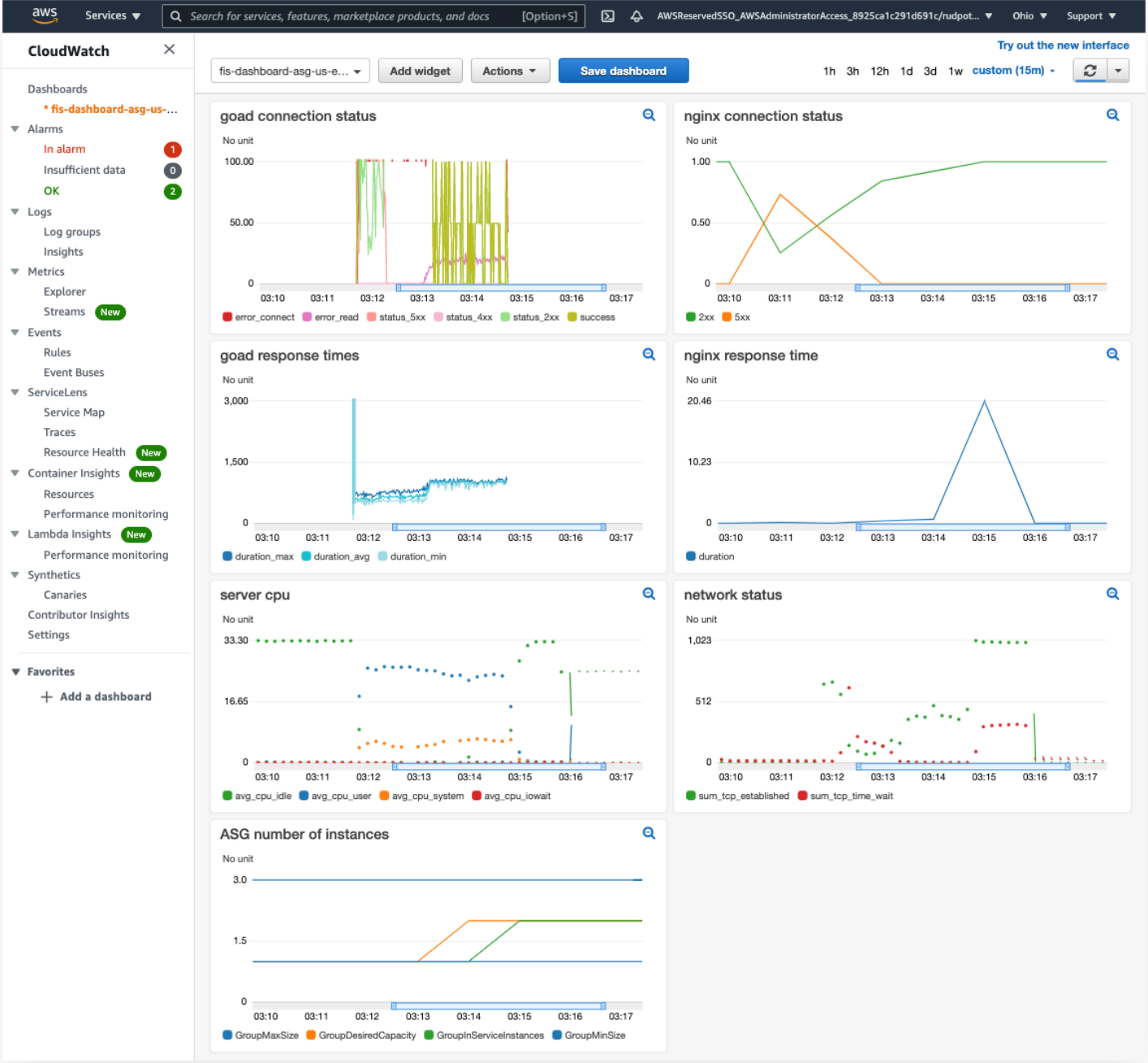
environment variable and run the load test against the new URL. Since we want to see Auto Scaling happen, let's run more than one copy:

```
export URL_PHP=${URL_HOME}/phpinfo.php

for ii in 1 2 3; do
  aws lambda invoke \
    --function-name ${LAMBDA_ARN} \
    --payload "{
          \"ConnectionTargetUrl\": \"${URL_PHP}\",
          \"ExperimentDurationSeconds\": 300,
          \"ConnectionsPerSecond\": 1000,
          \"ReportingMilliseconds\": 1000,
          \"ConnectionTimeoutMilliseconds\": 2000,
          \"TlsTimeoutMilliseconds\": 2000,
          \"TotalTimeoutMilliseconds\": 2000
      }" \
    --invocation-type Event \
    invoke-${ii}.txt
done
```

Warning

If you are running AWS CLI v2, you need to pass the parameter `--cli-binary-format raw-in-base64-out` or you'll get the error "Invalid base64" when sending the payload.

While this is executing, we encourage you to explore CloudWatch logs and create some dashboard views of your own.



Finally, we generated enough load to force a Auto Scaling event. We can also see how different the user experience is from the Nginx report. Requests timeout after 2s, substantially affecting user experiences, and rendering the website unavailable. Nginx, in contrast, doesn't report this as an error because the connection was terminated before being served. We will leave it as an exercise to the reader to figure out more details and will move on to fault injection experiments.

FIRST EXPERIMENT

In this section, we will cover the setup required for using AWS FIS to run our first fault injection experiment

Experiment idea

In the previous section, we ensured that we can measure the user experience. We also have configured an Auto Scaling group that should make sure we can “always” provide a good experience to the customer. Let’s validate this:

- **Given**: we have an Auto Scaling group with multiple instances
- **Hypothesis**: Failure of a single EC2 instance may lead to slower response times but should not affect service availability for our customers.

CONFIGURING PERMISSIONS

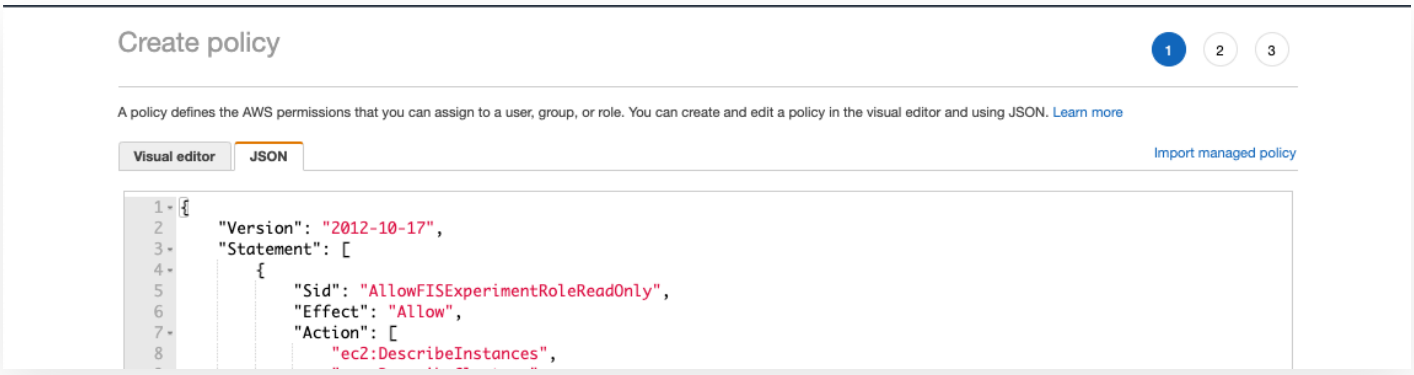
The AWS FIS security model uses two IAM roles. The first IAM role, the one you used to log into the console, controls access to AWS FIS service. It governs whether you are able to see, modify, and run AWS FIS experiments.

The second role governs what resources an AWS FIS experiment can affect during a fault injection experiment. For the purposes of this workshop, we will create one generic role. However, you can create fine grained IAM roles for each fault injection experiment.

Create FIS service role

We need to create an IAM role for the AWS FIS service to grant it permissions to inject faults into the system. While we could have pre-created this IAM role for you, we think it is important to review its scope with you.

Navigate to the IAM console and create a new policy called `FisWorkshopServicePolicy`. On the *Create Policy* page select the JSON tab



and paste the following policy - take the time to look at how broad these permissions are:

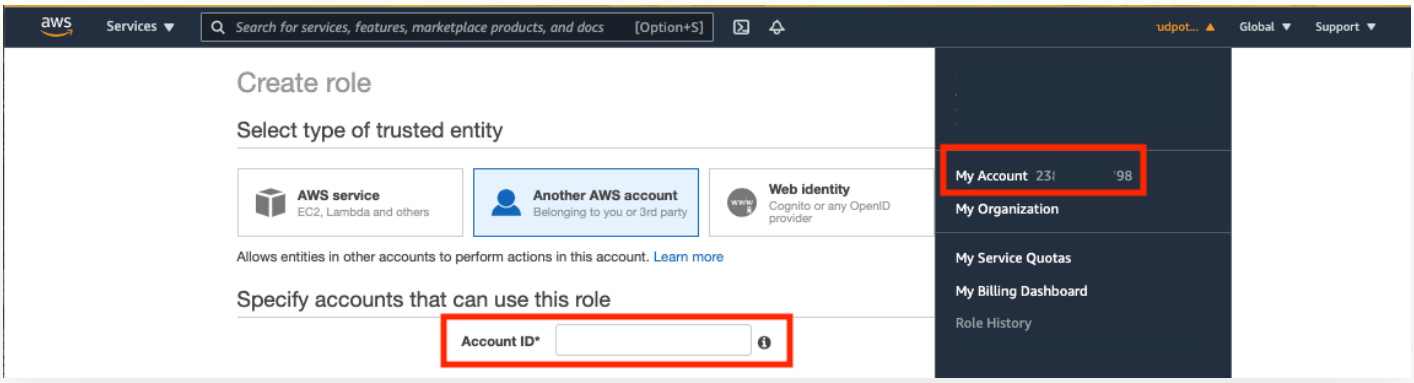
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowFISExperimentRoleReadOnly",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ecs:DescribeClusters",
```

```
            "ecs:ListContainerInstances",
            "eks:DescribeNodegroup",
            "iam:ListRoles",
            "rds:DescribeDBInstances",
            "rds:DescribeDbClusters",
            "ssm:ListCommands"
        ],
        "Resource": "*"
    },
    {
        "Sid": "AllowFISExperimentRoleEC2Actions",
        "Effect": "Allow",
        "Action": [
            "ec2:RebootInstances",
            "ec2:StopInstances",
            "ec2:StartInstances",
            "ec2:TerminateInstances"
        ],
        "Resource": "arn:aws:ec2:*:*:instance/*"
    },
    {
        "Sid": "AllowFISExperimentRoleECSActions",
        "Effect": "Allow",
        "Action": [
            "ecs:UpdateContainerInstancesState",
            "ecs:ListContainerInstances"
        ],
        "Resource": "arn:aws:ecs:*:*:container-instance/*"
    },
    {
        "Sid": "AllowFISExperimentRoleEKSActions",
        "Effect": "Allow",
        "Action": [
            "ec2:TerminateInstances"
        ],
        "Resource": "arn:aws:ec2:*:*:instance/*"
    },
    {
        "Sid": "AllowFISExperimentRoleFISActions",
        "Effect": "Allow",
        "Action": [
            "fis:InjectApiInternalError",
            "fis:InjectApiThrottleError",
            "fis:InjectApiUnavailableError"
        ],
        "Resource": "arn:*:fis:*:*:experiment/*"
    },
    {
        "Sid": "AllowFISExperimentRoleRDSReboot",
        "Effect": "Allow",
        "Action": [
            "rds:RebootDBInstance"
        ],
        "Resource": "arn:aws:rds:*:*:db:*"
    },
```

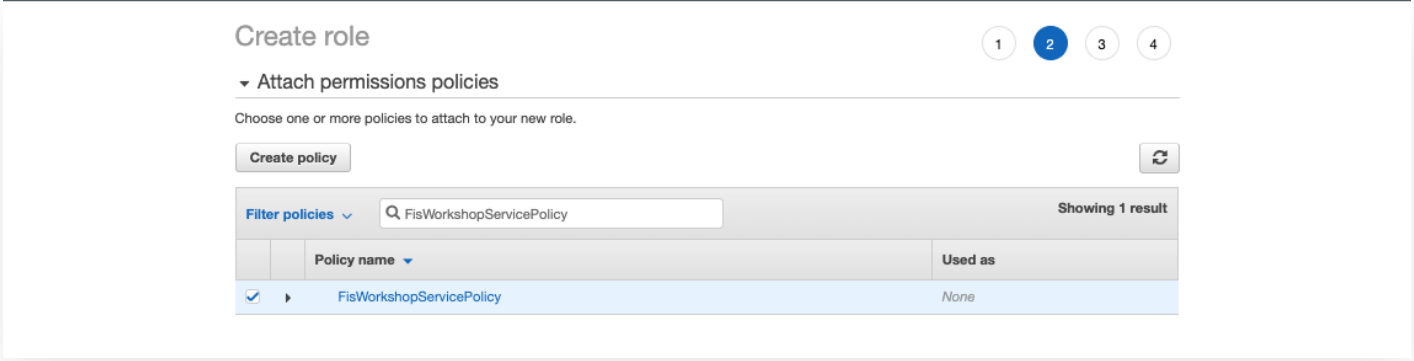
```
        {
            "Sid": "AllowFISExperimentRoleRDSFailOver",
            "Effect": "Allow",
            "Action": [
                "rds:FailoverDBCluster"
            ],
            "Resource": "arn:aws:rds:*:*:cluster:*"
        },
        {
            "Sid": "AllowFISExperimentRoleSSMSendCommand",
            "Effect": "Allow",
            "Action": [
                "ssm:SendCommand"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:instance/*",
                "arn:aws:ssm:*:*:document/*"
            ]
        },
        {
            "Sid": "AllowFISExperimentRoleSSMCancelCommand",
            "Effect": "Allow",
            "Action": [
                "ssm:CancelCommand"
            ],
            "Resource": "*"
        }
    ]
}
```

Navigate to the IAM console and create a new role called `FisWorkshopServiceRole`.

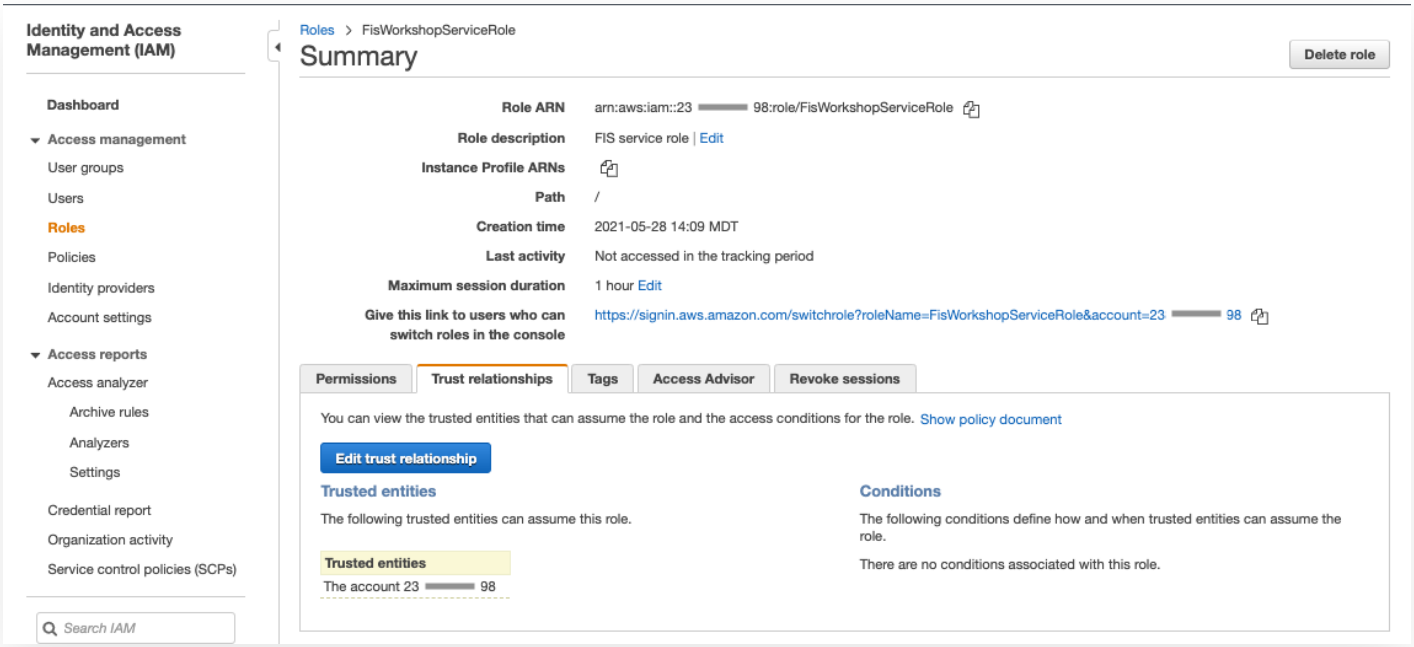
On the **Select type of trusted entity** page AWS FIS does not exist as a trusted service yet. Select "Another AWS Account" and add the current account number. You can find the account number in the drop-down menu as shown:



On the **Attach permissions** page search for the `FisWorkshopServicePolicy` we just created and check the box beside it to attach it to the role.



Back in the **IAM Roles** page, find and edit the `FisWorkshopServiceRole`. Select *Trust relationships* and the *Edit trust relationship* button.



Replace the policy document with the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                  "fis.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {}
        }
```

```
    ]
}
```

EXPERIMENT (CONSOLE)

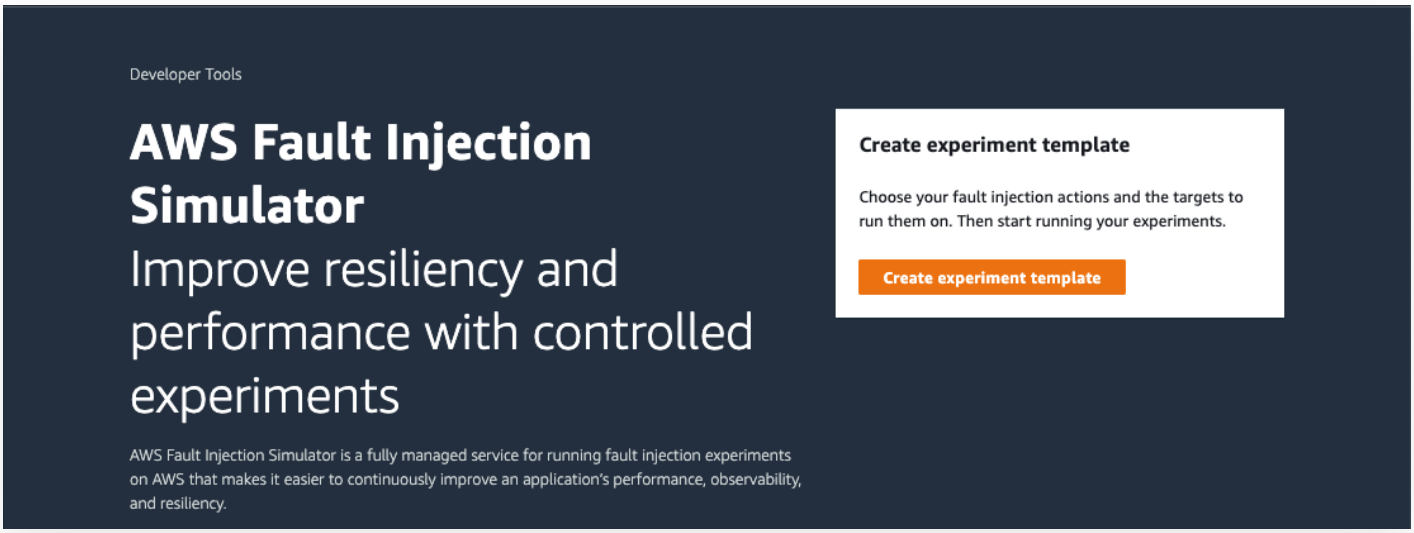
In this section, we will learn over how to create an AWS FIS experiment template using the AWS Console.

Experiment setup

Create an AWS FIS experiment template

To create a fault injection experiment, we first need to create an AWS FIS template defining the Actions, Targets, and optionally (but strongly recommended) the Stop Conditions.

Navigate to the FIS console and select "Create experiment template".



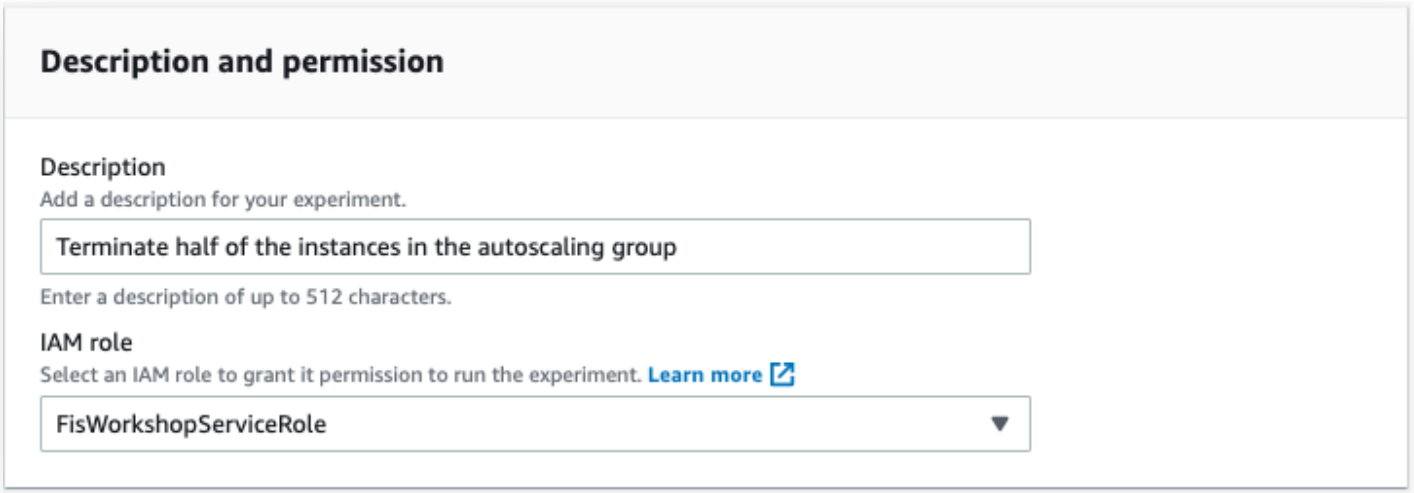
Note

Note: if you've used AWS FIS before you may not see the splash screen. In that case select "Experiment templates" in the menu on the left and access "Create experiment template" from there.

Template description and permissions

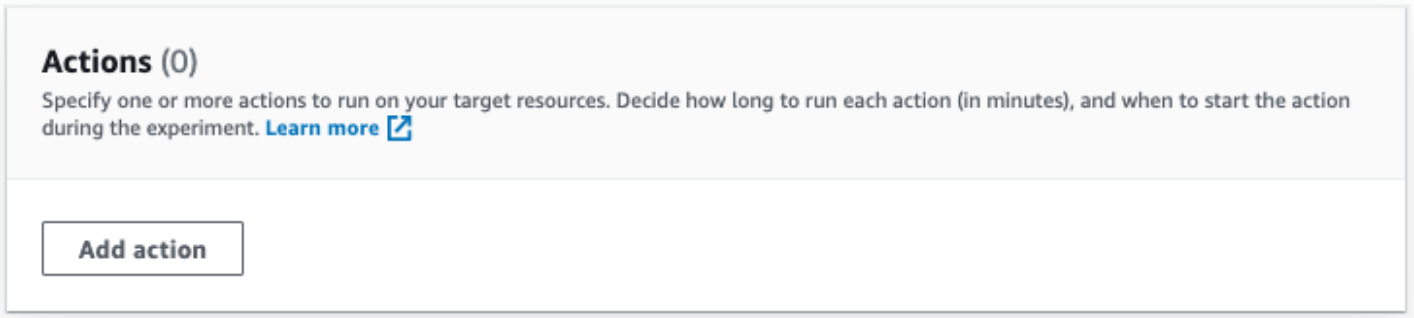
First, let's give a description of our experiment and an IAM role for the experiment. Go to the "Description and permission" section at the top. For "Description" enter

`Terminate half of the instances in the auto scaling group` and for “Role” select the `FisWorkshopServiceRole` role you created above.



Action definition

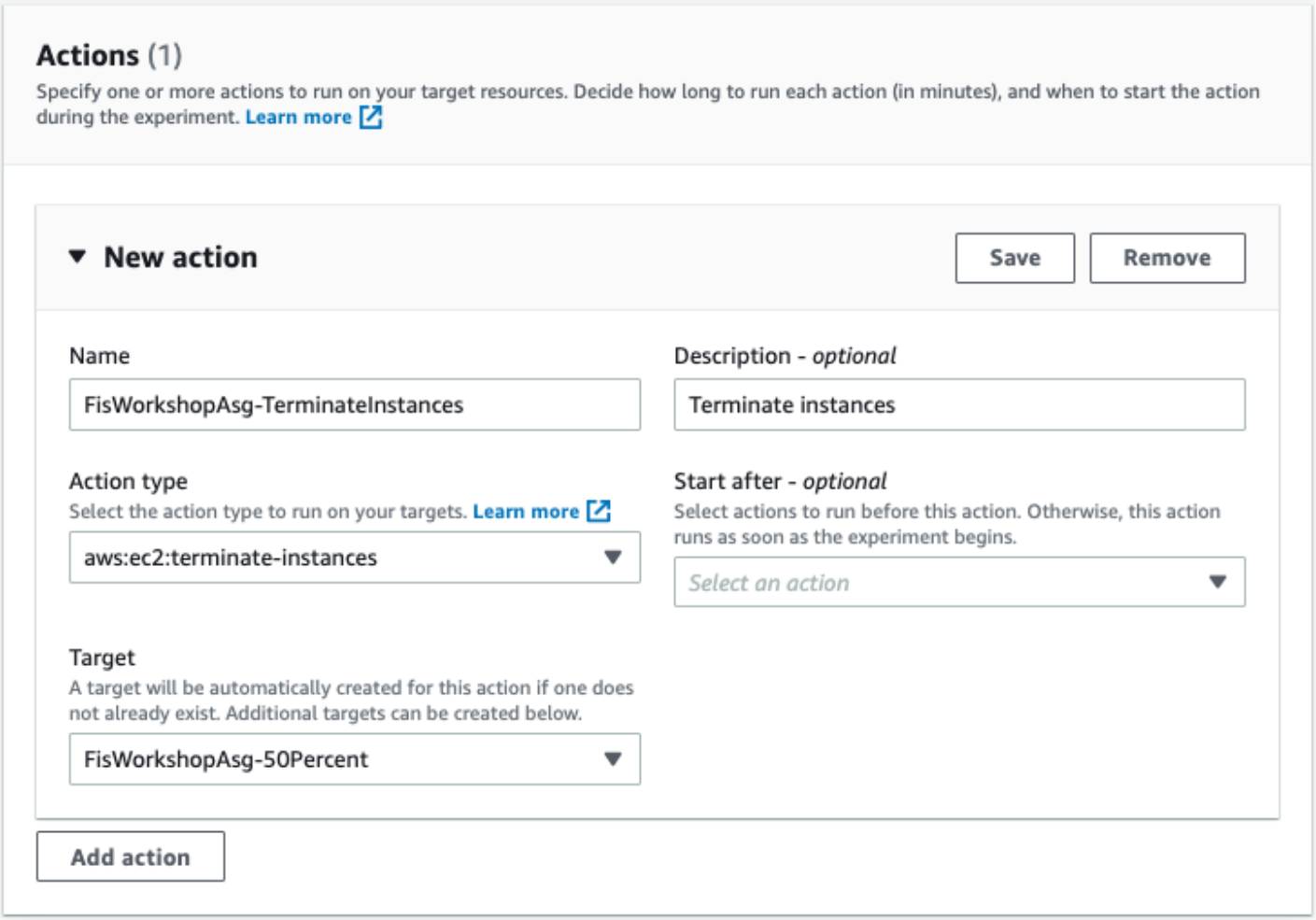
We first need to select which action to take. To test the hypothesis that we can safely impact half the instances in our Auto Scaling Group, we will terminate half of those instances. Go to the “Actions” section" and select “Add Action”



For “Name” enter `FisWorkshopAsg-TerminateInstances` and add a “Description” like `Terminate instances`. For “Action type” select `aws:ec2:terminate-instances`.

We will leave the “Start after” section blank since the instances we are terminating are part of an Auto Scaling Group and we can let it create new instances to replace the terminated ones.

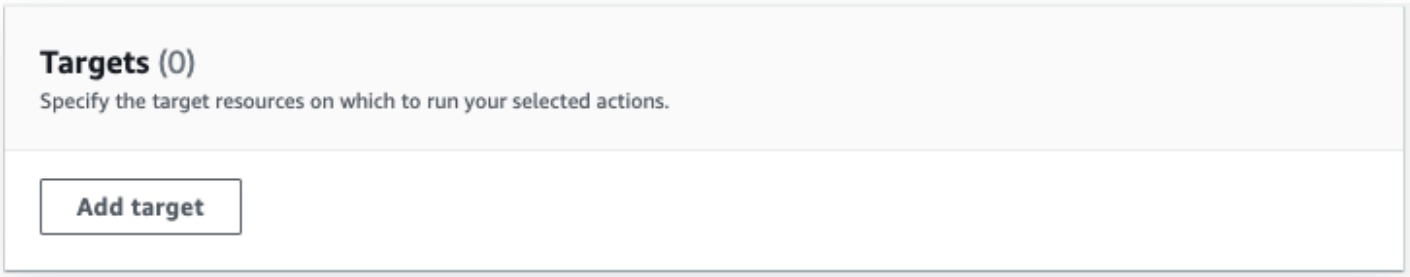
Leace the default “Target” `Instances-Target-1` and click “Save”.



#TO-UPDATE

Target selection

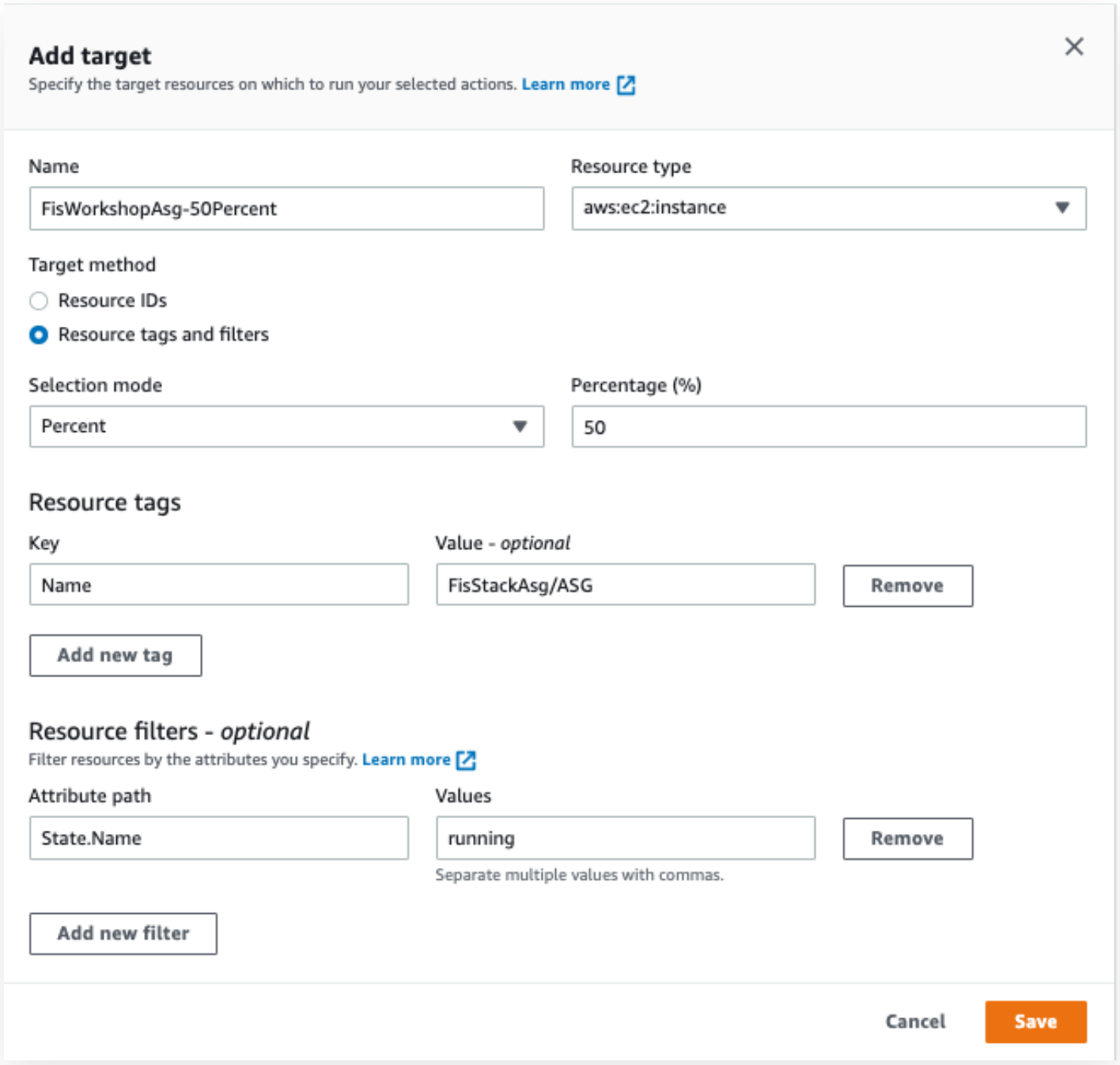
Now, let's define the EC2 instances to terminate. For this first experiment, we want to prove the following hypothesis: Since we use Auto Scaling, we can safely impact half the instances in our Auto Scaling Group. Scroll to the predefined "Instances-Target-1" section and click "Edit".



#TO-UPDATE

Leave the default name and make sure the Resource type is `aws:ec2:instances`. For "Target method" we will dynamically select resources based on an associated tag. Select the `Resource tags and filters` checkbox. Pick `Percent` from "Selection mode" and enter `50`. Under "Resource tags" enter `Name` in the "Key" field and

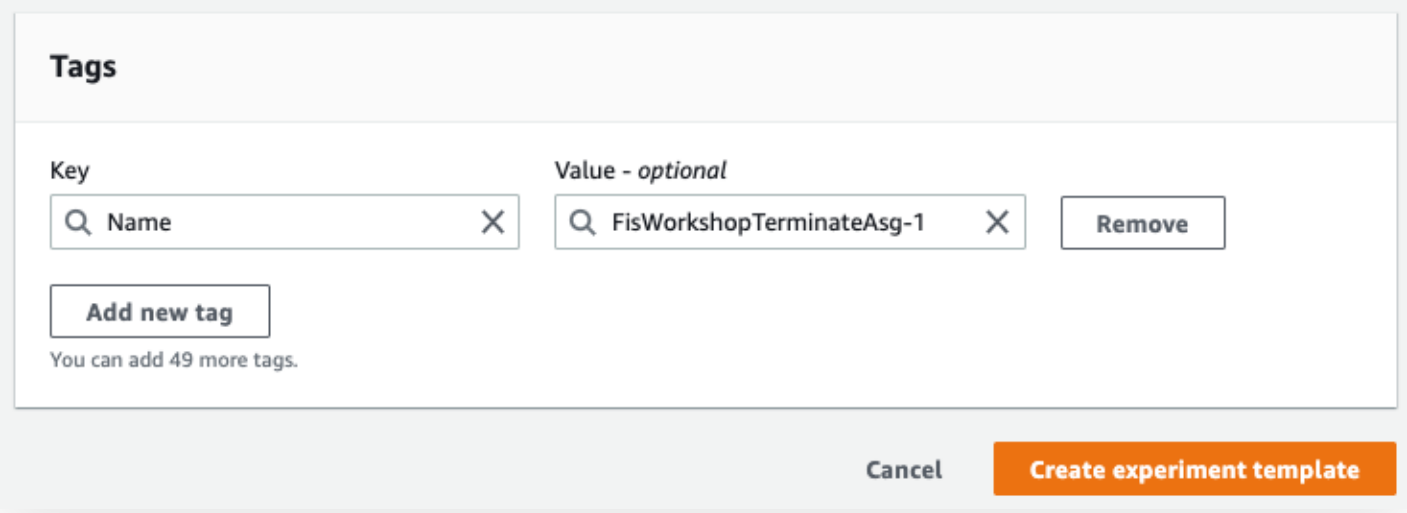
`FisStackAsg/ASG` for “Value”. Under filters enter `State.Name` in the “Attribute path” field and `running` under “Values”. We wil cover filters in more detail in the next section. Select “Save”.



#TO-UPDATE

Creating template without stop conditions

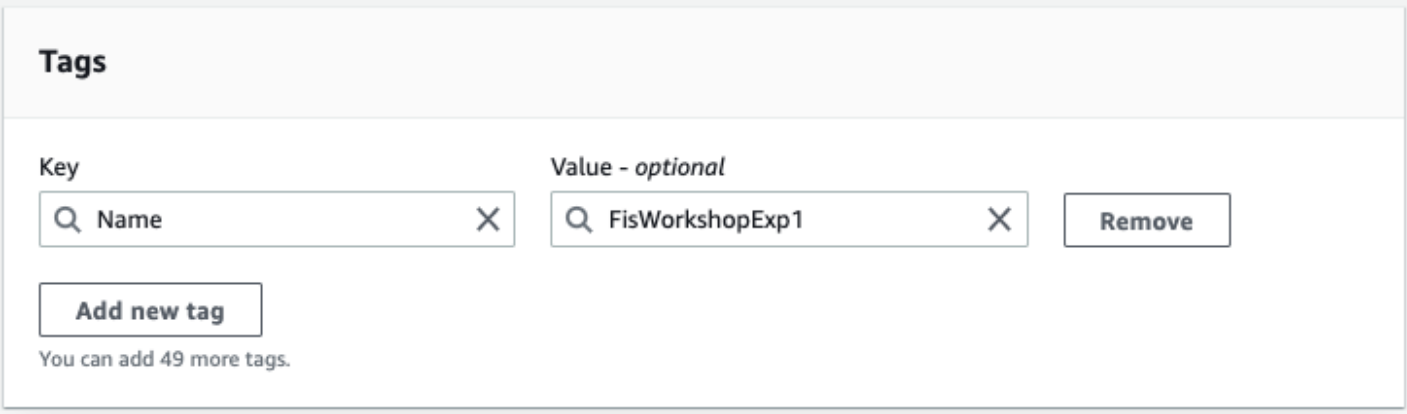
Scroll to the bottom of the template definition page and select “Create experiment template”.



#TO-UPDATE

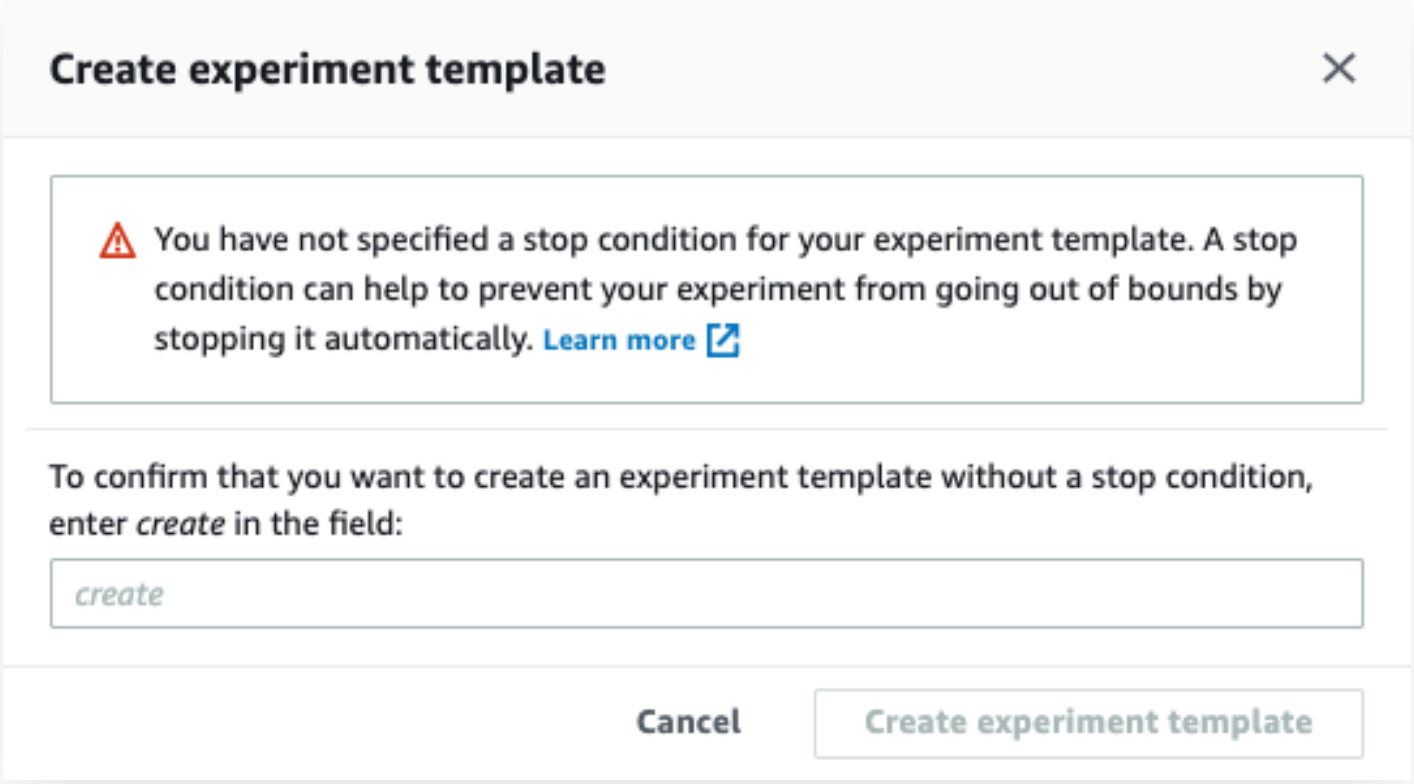
Template name

Finally, let's give our template a short name to be used on the list page. To do this scroll to the "Tags" section at the bottom, select "Add new tag", then enter `Name` in the "Key" field and `FisWorkshopExp1` for "Value"



#TO-UPDATE

Since we didn't specify a stop condition we receive a warning. This is ok, for this experiment we won't use a stop condition. Type `create` in the text box as indicated and select "Create experiment template".



#TO-UPDATE

Validation procedure

We will be using the CloudWatch dashboard from the previous sections for validation, no additional setup required.

Run FIS experiment

As previously discussed, we should collect both customer and ops metrics. In future sections, we will show you how you can add the load generator into your experiments.

However, for this experiment we will manually generate load on the system before starting the experiment, similarly to what we did in the previous section. Here we have increased the run time to 5 minutes by setting `ExperimentDurationSeconds` to 300:

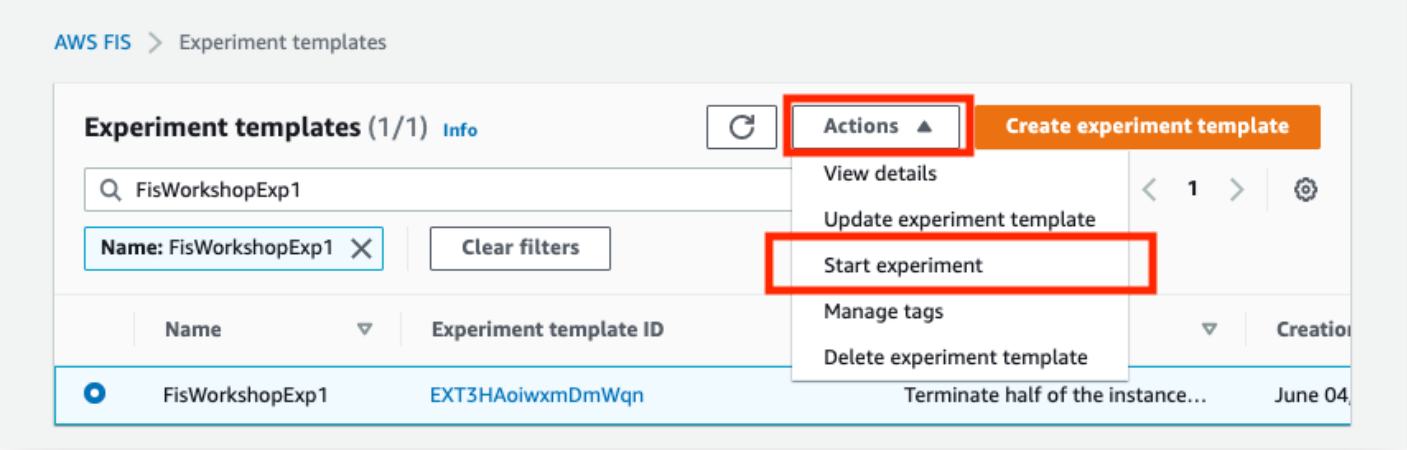
```
# Please ensure that LAMBDA_ARN and URL_HOME are still set from previous section
aws lambda invoke \
  --function-name ${LAMBDA_ARN} \
  --payload "{
        \"ConnectionTargetUrl\": \"${URL_HOME}\",
        \"ExperimentDurationSeconds\": 300,
```

```
        \"ConnectionsPerSecond\": 1000,
        \"ReportingMilliseconds\": 1000,
        \"ConnectionTimeoutMilliseconds\": 2000,
        \"TlsTimeoutMilliseconds\": 2000,
        \"TotalTimeoutMilliseconds\": 2000
    }" \
  --invocation-type Event \
  invoke.txt
```

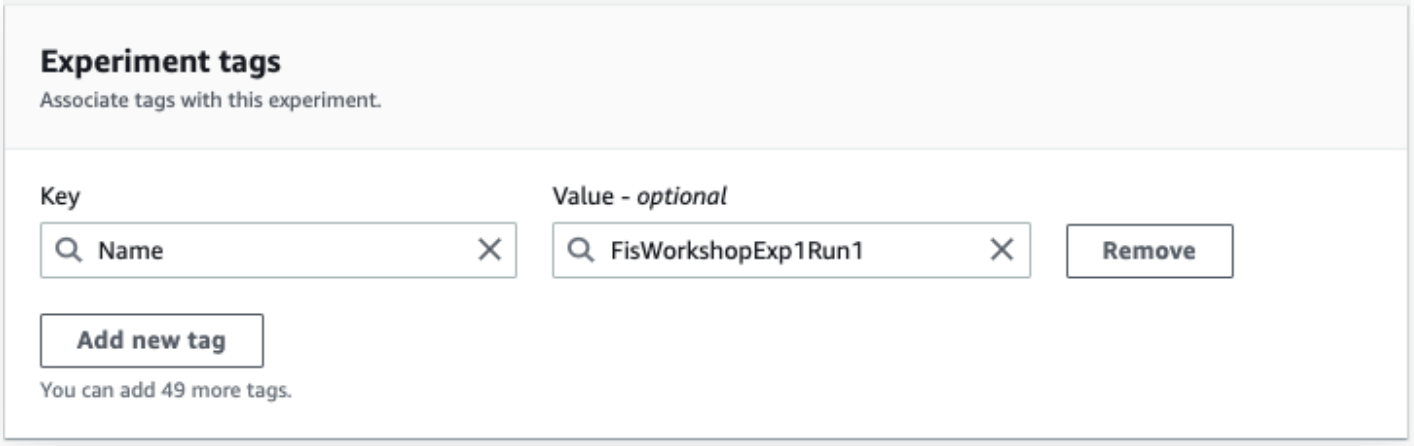
Warning

If you are running AWS CLI v2, you need to pass the parameter `--cli-binary-format raw-in-base64-out` or you'll get the error "Invalid base64" when sending the payload.

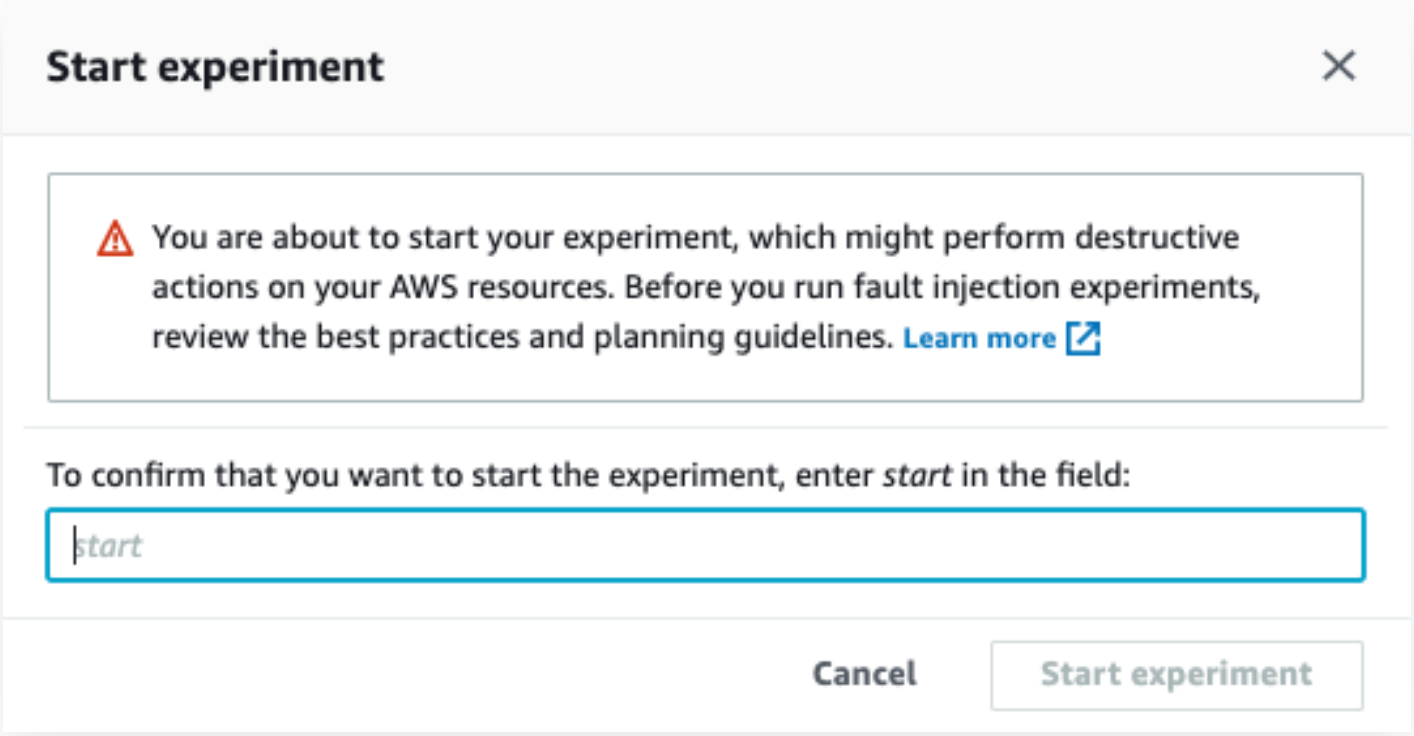
To start the experiment navigate to the FIS console, select the `FisWorkshopExp1` template we just created. Under "Actions" select "Start experiment".



Let's give the experiment run a friendly name. It will make it easier to find it from the list page. Under "Experiment tags" enter `Name` for "Key and `FisWorkshopExp1Run1` then select "Start experiment".



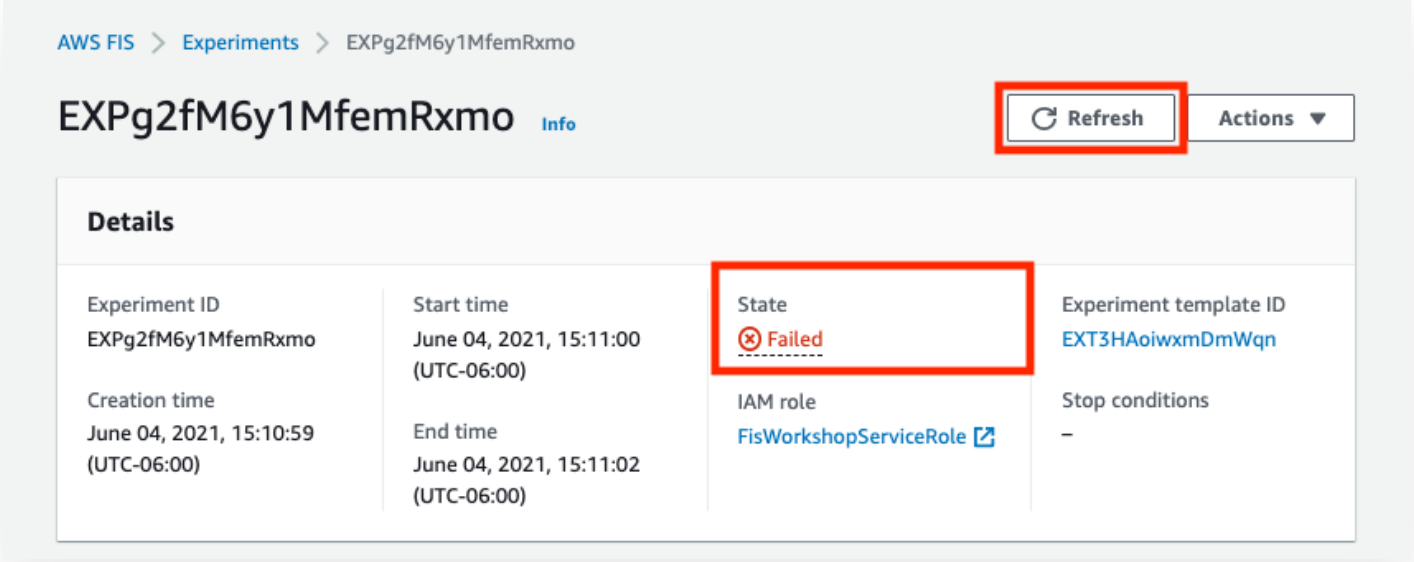
Because you are about to start a potentially destructive process, you will be asked to confirm that you really want to do this. Type `start` as directed and select "Start experiment".



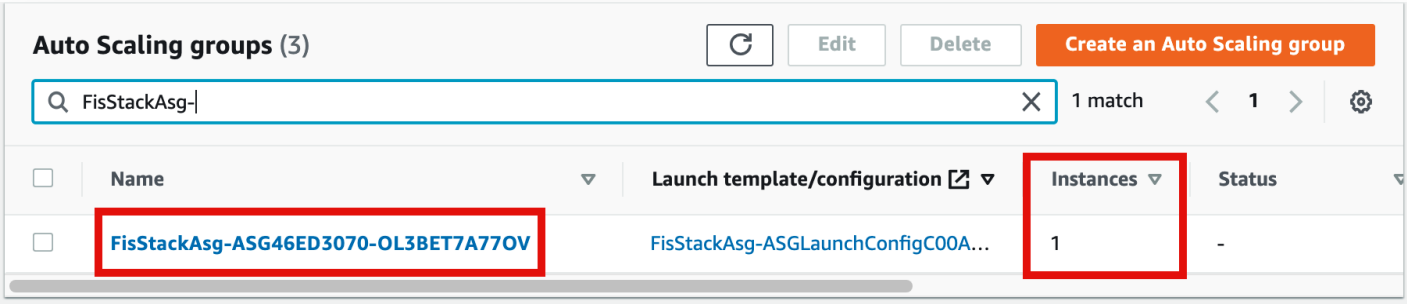
Review results

Navigate to the FIS console, select "Experiments", and select the experiment ID for the experiment you just started.

Look at the "State" entry. If this still shows pending, feel free to select the "Refresh" button a few times until you see a result. If you followed the above steps carefully there is a good chance that your experiment state will be "Failed".



Click on the failed result to get more information about why it failed. The message should say "Target resolution returned empty set". To see why this would happen, have a look at the auto scaling group from which we tried to select instances. Navigate to the EC2 console, select "Auto Scaling Groups" on the bottom of the left menu, and search for "FisStackAsg-":

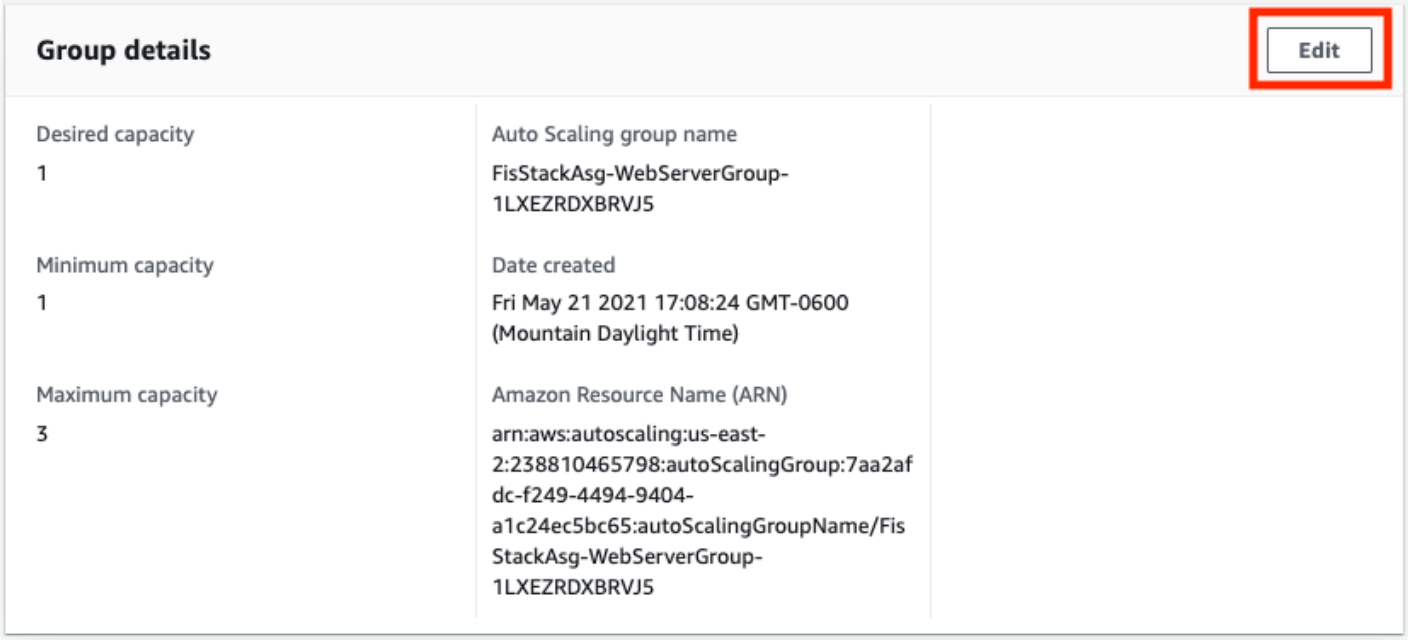


Learning and improving

It looks like our ASG was configured to scale down to just one instance while idle. Since we can't shut down half of one instance our 50% selector came up empty and the experiment failed.

Great! While this wasn't really what we expected, we just found a flaw in our configuration that would severely affect our system's resilience! Let's fix it and try again!

Click on the auto scaling group name and "Edit" the "Group Details" to raise both the "Desired capacity" and "Minimum capacity" to `2`.

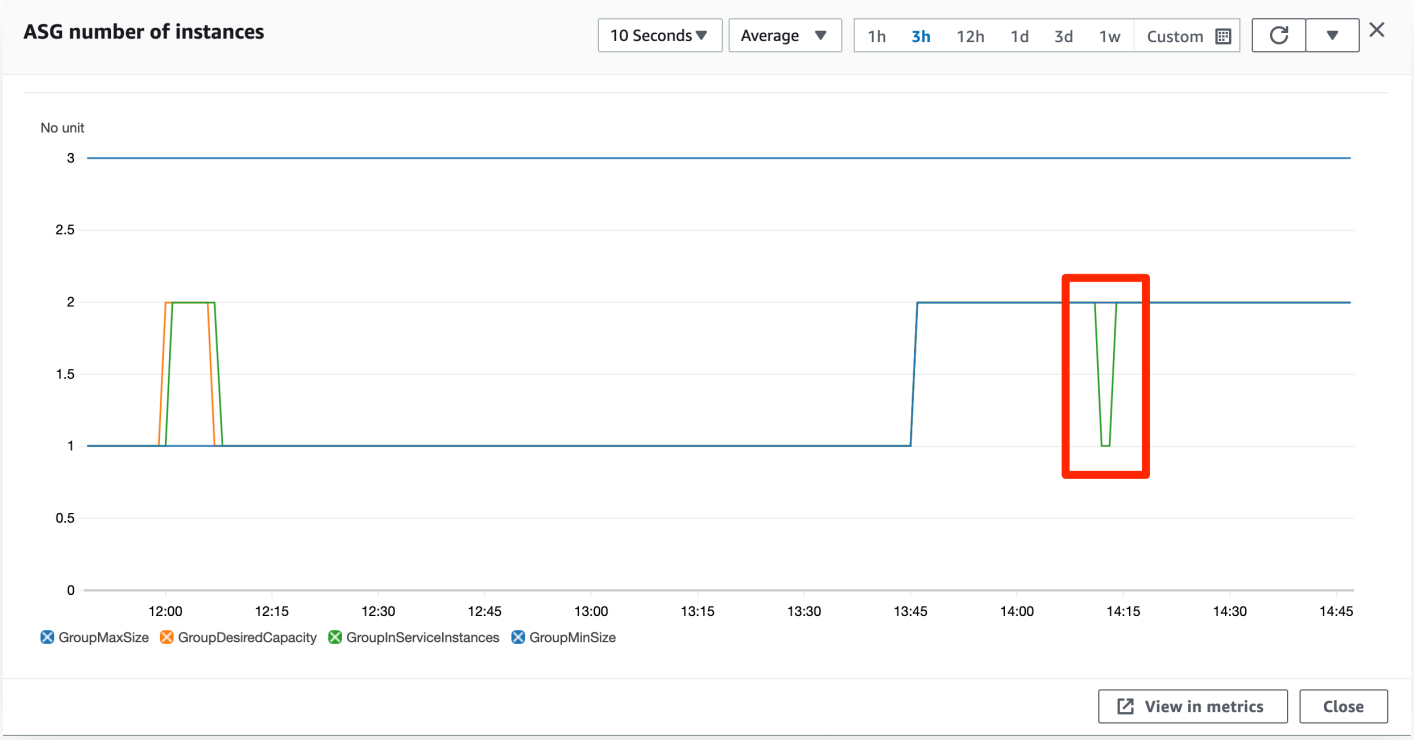


Check the ASG details or the CloudWatch Dashboard we explored in the previous section to make sure the active instances count has come up to 2.

To repeat the experiment, repeat the steps above:

- restart the load
- navigate back to the FIS Experiment Templates Console, start the experiment adding a `Name` tag of `FisWorkshopExp1Run2`
- check to make sure the experiment succeeded

Finally navigate to the CloudWatch Dashboard from the previous section. Review the number of instances in the ASG going down and then up again and review the error responses reported by the load test.



Findings and next steps

From this experiment we learned:

- Carefully choose the resource to affect and how to select them. If we had originally chosen to terminate a single instance (COUNT) rather than a fraction (PERCENT), we would have severely affected our service.
- Spinning up instances takes time. To achieve resilience, Auto Scaling groups should be set to have at least two instances running at all times

In the next section we will explore larger experiments.

EXPERIMENT (CLI)

In this section we will show you how to create an experiment using AWS FIS templates. For clarity, we will replicate the same experiment as we previously did via the AWS console.

Template overview

Experiment templates are JSON files containing Actions, Targets, an IAM role, and optional Stop Conditions, and Tags:

```
{
    "experimentTemplate": {
        "description": "...",
        "actions": {},
        "targets": {},
        "roleArn": "arn:aws:iam:...",
        "stopConditions": [],
        "tags": {}
    }
}
```

Actions

Actions specify an action name and `description`, an `actionId` and matching `parameters` picked from the AWS FIS Action reference, and a list of `targets` which references the target section in the same template:

```
"ActionName": {
    "description": "ActionDescription",
    "actionId": "aws:ec2:terminate-instances",
    "parameters": {},
    "targets": {}
}
```

Targets

Targets specify a name, a `resourceType` from which to select by `resourceArn`, `resourceTags` or `filters`, and `selectionMode` for sampling from the eligible resources by `COUNT()` or `PERCENT()`.

```
"TargetGroupName": {
    "resourceType": "aws:ec2:instance",
    "resourceArns": [],
    "resourceTags": {
        "TagName1": "TagValue1",
        "TagName2": "TagValue2",
        ...
    },
    "filters": [
        {
            "path": "State.Name",
            "values": [
                "running"
            ]
        }
    ],
    "selectionMode": "COUNT(1)"
}
```

A note on finding the `path` and `values` for `filters`: as described under "Resource filters", filter paths are based on API output. E.g.: if we want to only target running EC2 instances we could use the AWS CLI to list instances:

```
aws ec2 describe-instances
```

To find the relevant `path` and `values` start in the `Instances` block of the API output and identify entries you would like to filter on:

```
{
    "Reservations": [
        {
            "Groups": [],
            "Instances": [
                {
                    "ImageId": "ami-00c36fdebc0d948bd",
                    "InstanceType": "t2.micro",
                    "Placement": {
                        "AvailabilityZone": "us-east-2a",
                        "GroupName": "",
                        "Tenancy": "default"
```

```
                    },
                    "State": {
                        "Code": 16,
                        "Name": "running"
                    },
                    "SubnetId": "subnet-0e912694b51e205d6",
                    "VpcId": "vpc-0d4c31ce84606e7eb",
                    "Tags": [
                        {
                            "Key": "Name",
                            "Value": "FisStackAsg/ASG"
                        },
                        ...
                    ],
                    ...
                },
                ...
            ]
        }
    ]
}
```

E.g.: to select an instance that is `running` in `us-east-2a` we would add the following filters:

```
"filters": [
    {
        "path": "State.Name",
        "values": [
            "running"
        ]
    },
    {
        "path": "Placement.AvailabilityZone",
        "values": [
            "us-east-2a"
        ]
    }
],
```

Stop conditions

Stop conditions use a list of Amazon CloudWatch alarms to prematurely stop the experiment if it does not proceed along expected lines:

```
"stopConditions": [
    {
```

```
        "source": "aws:cloudwatch:alarm",
        "value": "arn:aws:cloudwatch:..."
    }
]
```

Finished template

Using the above, this would be the finished template.

Note

Before using this template, please ensure that you replace the ARN for the FIS execution role on the last line with the ARN of the role you you created earlier in this section.

```
{
    "description": "Terminate 50% of instances based on Name Tag",
    "tags": {
        "Name": "FisWorkshop-Exp1-CLI"
    },
    "actions": {
        "FisWorkshopTerminateAsg-1-CLI": {
            "actionId": "aws:ec2:terminate-instances",
            "description": "Terminate 50% of instances based on Name Tag",
            "parameters": {},
            "targets": {
                "Instances": "Instances-Target-1"
            }
        },
        "Wait": {
            "actionId": "aws:fis:wait",
            "parameters": {
                "duration": "PT3M"
            }
        }
    },
    "targets": {
        "Instances-Target-1": {
            "resourceType": "aws:ec2:instance",
            "resourceTags": {
                "Name": "FisStackAsg/ASG"
            },
            "selectionMode": "PERCENT(50)"
        }
    },
    "stopConditions": [
        {
            "source": "none"
        }
```

```
    ],
    "roleArn":
"arn:aws:iam::YOUR_ACCOUNT_NUMBER_HERE:role/FisWorkshopServiceRole"
}
```

Working with templates

The rest of this section uses the AWS CLI. If you are using AWS Cloud9 this should work out of the box. Otherwise please ensure you have configured AWS credentials for the CLI.

Creating template

To create an experiment template, copy the above JSON into a file named `fis.json` and ensure you have changed the `roleArn` entry to be the ARN of the role you created earlier. Then use the CLI to create the template in AWS:

```
aws fis create-experiment-template --cli-input-json file://fis.json
```

You should now be able to see the newly created experiment template in the AWS Console.

Listing templates

This command

```
aws fis list-experiment-templates
```

will list all the templates. If you happened to run the `create-experiment-template` command above multiple times you might notice that it is possible to have multiple copies of a template only differentiated by the `id` field.

While it is possible to update an existing experiment template via the `update-experiment-template` command, this will make it hard to establish what happened during an experiment.

Exporting / saving templates

Once you have established the `id` of an experiment template you can dump the template. This can be a good way of learning how to write templates as well:

```
export EXPERIMENT_TEMPLATE_ID=<YOUR_EXPERIMENT_TEMPLATE_ID_HERE>
aws fis get-experiment-template --id $EXPERIMENT_TEMPLATE_ID
```

You will note that the result is wrapped into an `experimentTemplate: {}` block. You may also notice that there are some additional fields that are not used during experiment template creation. You can generate reusable JSON like so:

```
aws fis get-experiment-template --id $EXPERIMENT_TEMPLATE_ID | jq
'.experimentTemplate | del( .id) | del(.creationTime) | del(.lastUpdateTime)'
```

Running the experiment

Finally we want to run the experiment:

```
aws fis start-experiment --experiment-template-id $EXPERIMENT_TEMPLATE_ID --tags
Name=FisWorkshopTerminateAsg-1-CLI | jq '.experiment.id'
```

Using the returned `id` field you can check on the outcome of the experiment:

```
aws fis get-experiment --id YOUR_EXPERIMENT_ID_HERE
```

Findings and next steps

The learnings here should be the same as for the console section:

- Carefully choose the resource to affect and how to select them. If we had originally chosen to terminate a single instance (COUNT) rather than a fraction (PERCENT), we would have severely affected our service.
- Spinning up instances takes time. To achieve resilience, ASGs should be set to have at least two instances running at all times

In the next section we will explore larger experiments.

EXPERIMENT (CLOUDFORMATION)

In this section we will cover how to define and update experiment templates using CloudFormation.

CFN template format

The CloudFormation template uses the same format as the API but capitalizes the first letter of section names. As such the FIS experiment template from the previous section would become:

```
{
"Type" : "AWS::FIS::ExperimentTemplate",
"Properties" : {
    "Description": "Terminate 50% of instances based on Name Tag",
    "Tags": {
        "Name": "FisWorkshop-Exp1-CFN-v1.0.0"
    },
    "Actions": {
        "FisWorkshopTerminateAsg-1-CFN": {
            "ActionId": "aws:ec2:terminate-instances",
            "Description": "Terminate 50% of instances based on Name Tag",
            "Parameters": {},
            "Targets": {
                "Instances": "Instances-Target-1"
            }
        },
        "Wait": {
            "ActionId": "aws:fis:wait",
            "Parameters": {
                "duration": "PT3M"
            }
        }
    },
    "Targets": {
        "Instances-Target-1": {
            "ResourceType": "aws:ec2:instance",
            "ResourceTags": {
                "Name": "FisStackAsg/ASG"
            },
```

```
            "SelectionMode": "PERCENT(50)"
        }
    },
    "StopConditions": [
        {
            "Source": "none"
        }
    ],
    "RoleArn": {
        "Fn::Sub": "arn:aws:iam::YOUR_ACCOUNT_ID:role/FisWorkshopServiceRole"
    }
}
```

We can wrap this into the `Resources` section of a CloudFormation template. Additionally CloudFormation allows us to use pseudo parameters which we can use to automatically insert the account number into the role definition using the `AWS::AccountId` parameter in conjunction with the `Fn::Sub` function. Thus, a simple CFN template would become:

```
{
    "Resources" : {
        "FisExperimentDemo" : {
            "Type" : "AWS::FIS::ExperimentTemplate",
            "Properties" : {
                "Description": "Terminate 50% of instances based on Name Tag",
                "Tags": {
                    "Name": "FisWorkshop-Exp1-CFN-v1.0.0"
                },
                "Actions": {
                    "FisWorkshopTerminateAsg-1-CFN": {
                        "ActionId": "aws:ec2:terminate-instances",
                        "Description": "Terminate 50% of instances based on Name
Tag",
                        "Parameters": {},
                        "Targets": {
                            "Instances": "Instances-Target-1"
                        }
                    },
                    "Wait": {
                        "ActionId": "aws:fis:wait",
                        "Parameters": {
                            "duration": "PT3M"
                        }
                    }
                },
                "Targets": {
                    "Instances-Target-1": {
                        "ResourceType": "aws:ec2:instance",
                        "ResourceTags": {
                            "Name": "FisStackAsg/ASG"
                        },
```

```
                    "SelectionMode": "PERCENT(50)"
                }
            },
            "StopConditions": [
                {
                    "Source": "none"
                }
            ],
            "RoleArn": {
                "Fn::Sub":
"arn:aws:iam::${AWS::AccountId}:role/FisWorkshopServiceRole"
            }
        }
    }
}
```

Using the CFN template

A deep dive into CloudFormation is beyond the scope of this workshop, so we will only cover how to create and update stacks via the CLI.

Create a new template / experiment

To create a stack, and thus the contained FIS experiment template, copy the above JSON into a file named `cfn-fis-experiment.json` then run this AWS CLI command:

```
aws cloudformation create-stack --stack-name FisWorkshopExperimentTemplate --
template-body file://cfn-fis-experiment.json
```

If you navigate to the CloudFormation console you should now see a new stack named `FisWorkshopExperimentTemplate` and navigating to the FIS console should show an experiment named `FisWorkshop-Exp1-CFN-v1.0.0`

Update template / experiment

To update the experiment template you will need to update the CFN template. Let's change the `Name` tag from `FisWorkshop-Exp1-CFN-v1.0.0` to `FisWorkshop-Exp1-CFN-v2.0.0` and save the file.

Then run the AWS CLI command:

```
aws cloudformation update-stack --stack-name FisWorkshopExperimentTemplate --
template-body file://cfn-fis-experiment.json
```

This should update the name of your experiment template in the FIS console. Obviously this is most useful if you make actual changes to the template itself too.

Findings and next steps

The learnings here should be the same as for the console section:

- Carefully choose the resource to affect and how to select them. If we had originally chosen to terminate a single instance (COUNT) rather than a fraction (PERCENT), we would have severely affected our service.
- Spinning up instances takes time. To achieve resilience, ASGs should be set to have at least two instances running at all times

In the next section we will explore larger experiments.

AWS SYSTEMS MANAGER INTEGRATION

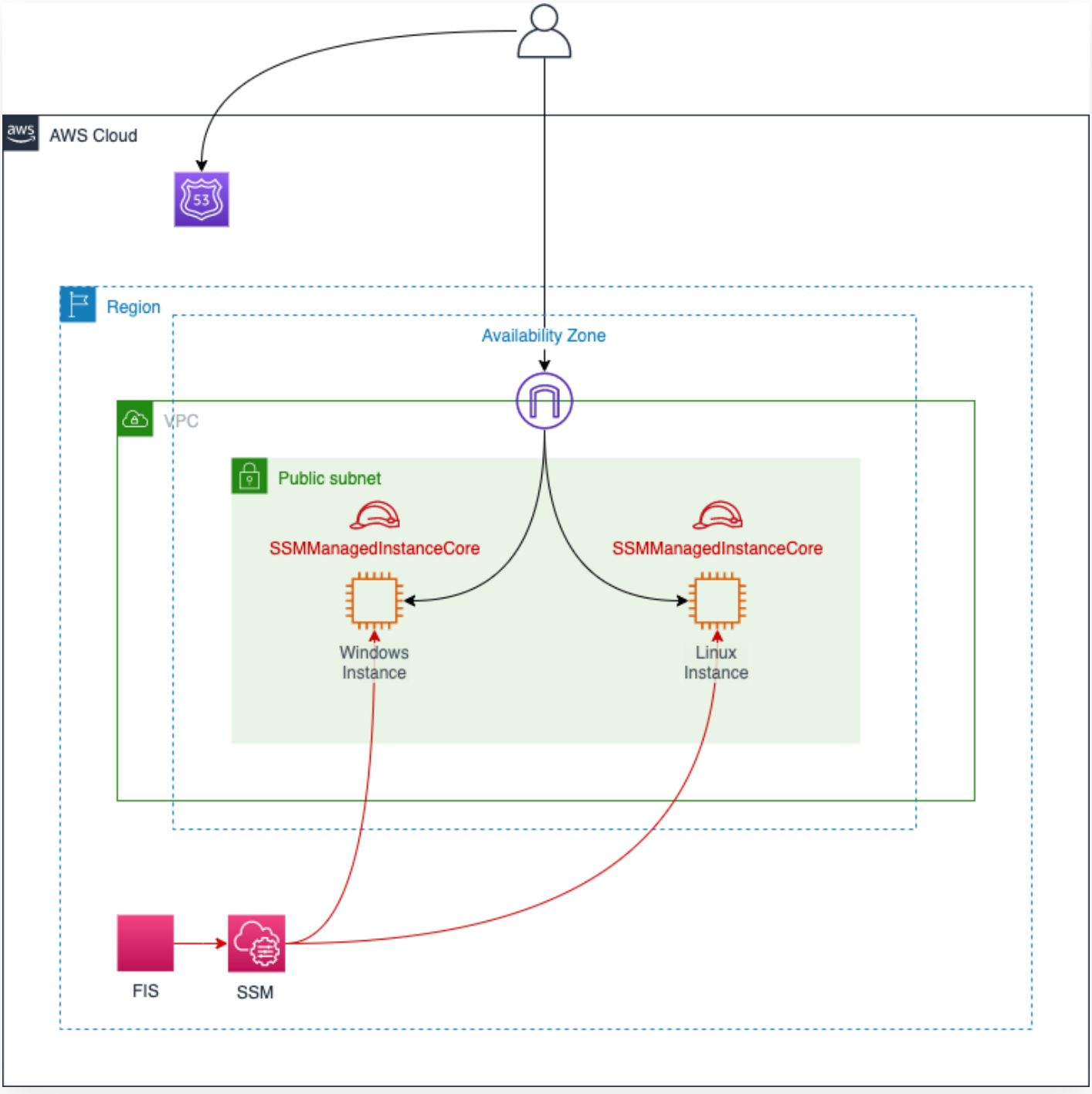
In this section, we will demonstrate how you can use AWS Systems Manager (SSM) along with AWS Fault Injection Simulator (FIS) to emulate faults within an EC2 Instance.

AWS FIS does not need an agent for actions affecting the AWS control plane like the ones we have worked with thus far, such as stop instances or failing over RDS Databases. However, there are actions that require us to initiate commands within the operating system of the EC2 Instance, such as affecting CPU or Memory consumption, or terminating processes. For these types of actions AWS FIS can use AWS Systems Manager (SSM) and the SSM Agent. This approach provides you with the access controls to grant FIS limited access to your instances under the shared responsibility model.

In the following sections we will show you how to use the built-in SSM actions and how to build your own SSM documents to create custom actions.

FIS SSM SEND COMMAND SETUP

For this section we will use Linux and Windows instances created specifically for the purpose of enabling FIS SSM commands. As shown in the diagram below, SSM access to the instance requires an instance role with the `AmazonSSMManagedInstanceCore` policy attached. Additionally FIS access to SSM is controlled via the execution policy as shown in the **First Experiment** section.



Info

The resources above have been created as part of the **Start the workshop** section. If you want to explore how the resources for this section have been created read on below. Otherwise continue to the next section.

CloudFormation deployment example

To demonstrate how to configure EC2 instances with SSM access we will use the AWS CLI with CloudFormation to provision our instances. You can inspect the template to see how it creates an instance role with the AWS Managed policy named *AmazonSSMManagedInstanceCore* attached, and how it attaches it to our instances via an Instance Profile.

If you want to try this yourself, using the Cloud9 instance you created in the **Start the workshop** section, clone the repository if you have not done so yet:

```
cd ~/environment
git clone https://github.com:aws-samples/aws-fault-injection-simulator-workshop.git
```

Next change directory into the templates folder.

```
cd aws-fault-injection-simulator-workshop/resources/templates/cpu-stress/
```

In this folder you can examine the template file named `CPUStressInstances.yaml`.

Finally, deploy the stack. By default, this template will deploy into the default VPC. In the context of this workshop, we want to ensure the instances are deployed into the public subnet created in the **Start the workshop** section. We will use the first public subnet created by the initial setup. You could do this manually by navigating to the CloudFormation console, selecting the `FisStackVpc` stack, selecting `Outputs` and picking the subnet ID associated with `FisPub1`. For your convenience we've added that as a CLI query in the code below:

```
# Query public subnet from VPC stack
SUBNET_ID=$( aws ec2 describe-subnets --query "Subnets[?Tags[?(Key=='aws-cdk:subnet-name') && (Value=='FisPub') ]] | [0].SubnetId" --output text )

# Launch CloudFormation stack
```

```
aws cloudformation create-stack \
  --stack-name CpuStress \
  --template-body file://CPUStressInstances.yaml  \
  --parameters \
    ParameterKey=SubnetId,ParameterValue=${SUBNET_ID} \
  --capabilities CAPABILITY_IAM
```

The stack will take a few minutes to complete. You can monitor the progress from the CloudFormation Console. Once this is finished you can continue to the next section.

LINUX CPU STRESS EXPERIMENT

In this section we will run a CPU Stress test using AWS Fault Injection Simulator against an Amazon Linux EC2 Instance. The Linux CPU stress test is an out of the box FIS action. We will do the following:

1. Create experiment template to stress CPU.
2. Connect to a Linux EC2 Instance and run the `top` command.
3. Start experiment and observe results.

Experiment Setup

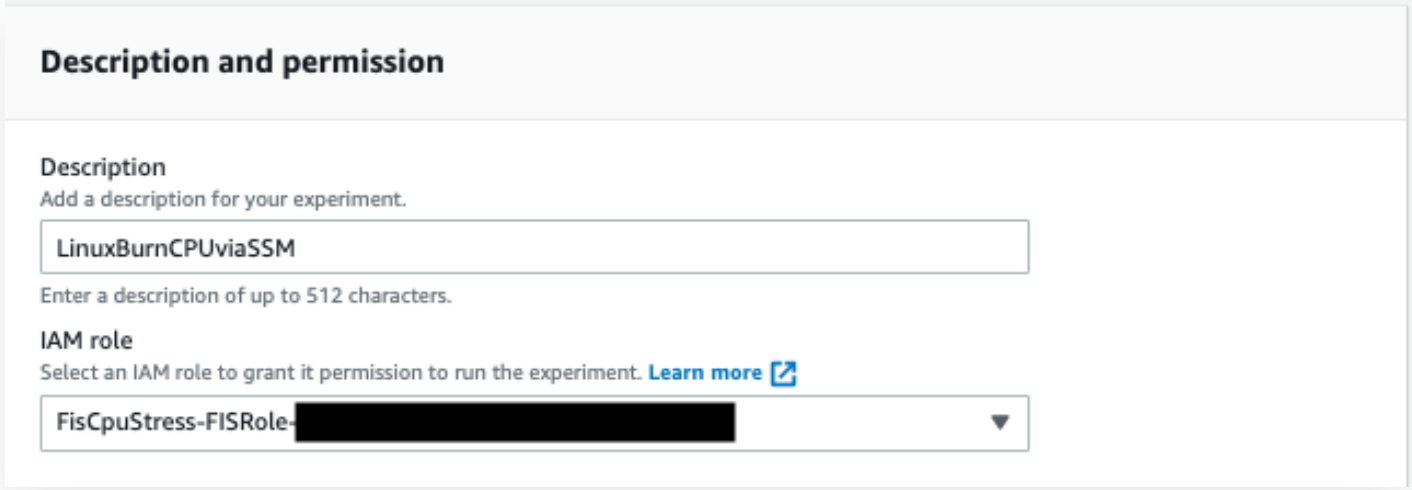
Create CPU Stress Experiment Template

First, lets create our stress experiment. We can do this programmatically but we will walk through this on the console.

1. Open the AWS Fault Injection Simulator Console. Once in the Fault Injection Simulator console, lets click on "Experiment templates" on the left side pane.

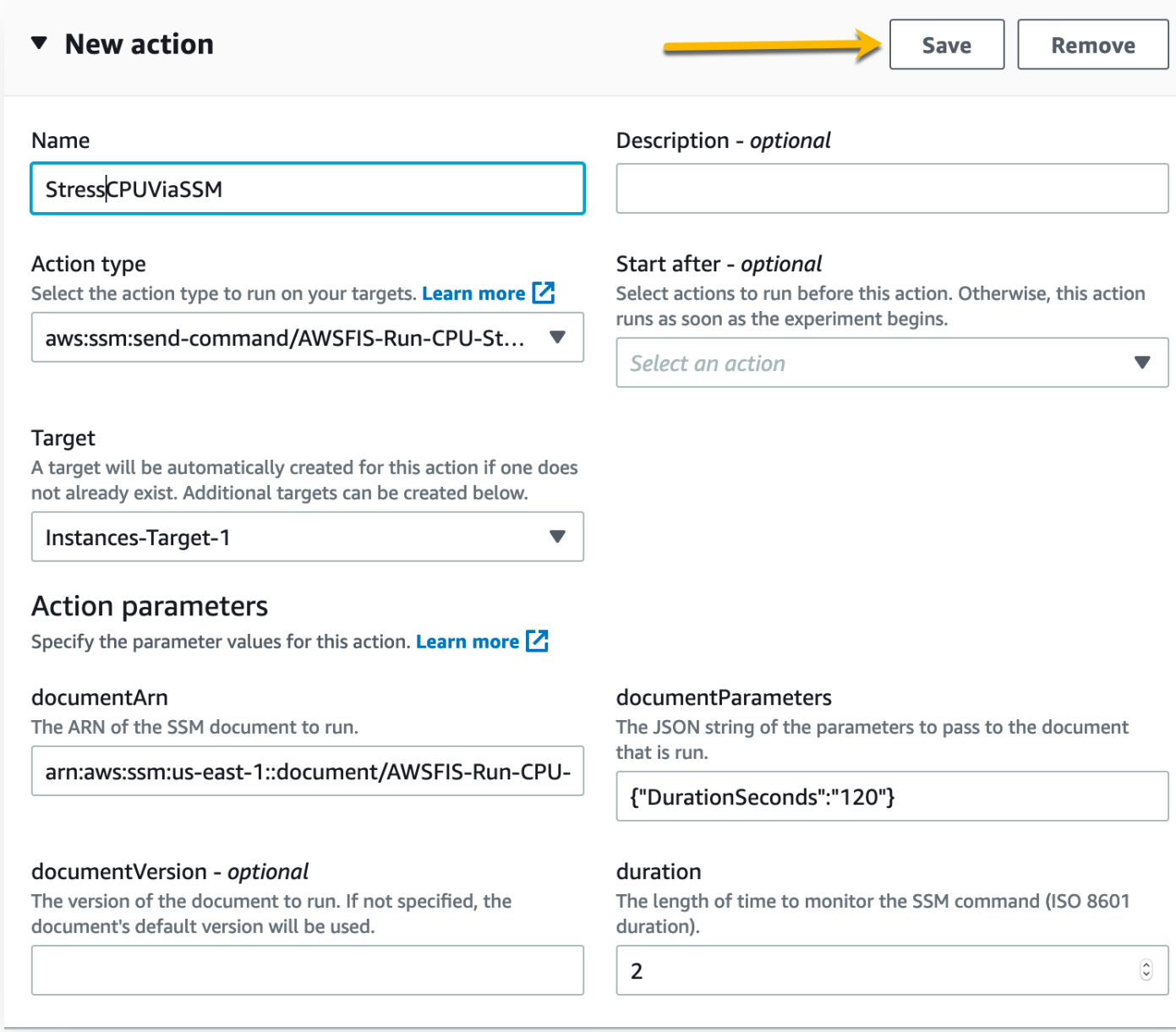
2. Click on "Create experiment template" on the upper right hand side of the console to start creating our experiment template.

3. Next we will enter the description of the experiment and choose the IAM Role. Let's put `LinuxBurnCPUviaSSM` for the description. The IAM role allows the FIS service permissions to execute actions on your behalf. As part of the CloudFormation stack a role was created for this experiment that starts with `FisCpuStress-FISRole`, select that role. Please examine the CloudFormation template or IAM Role for the policies in this role.

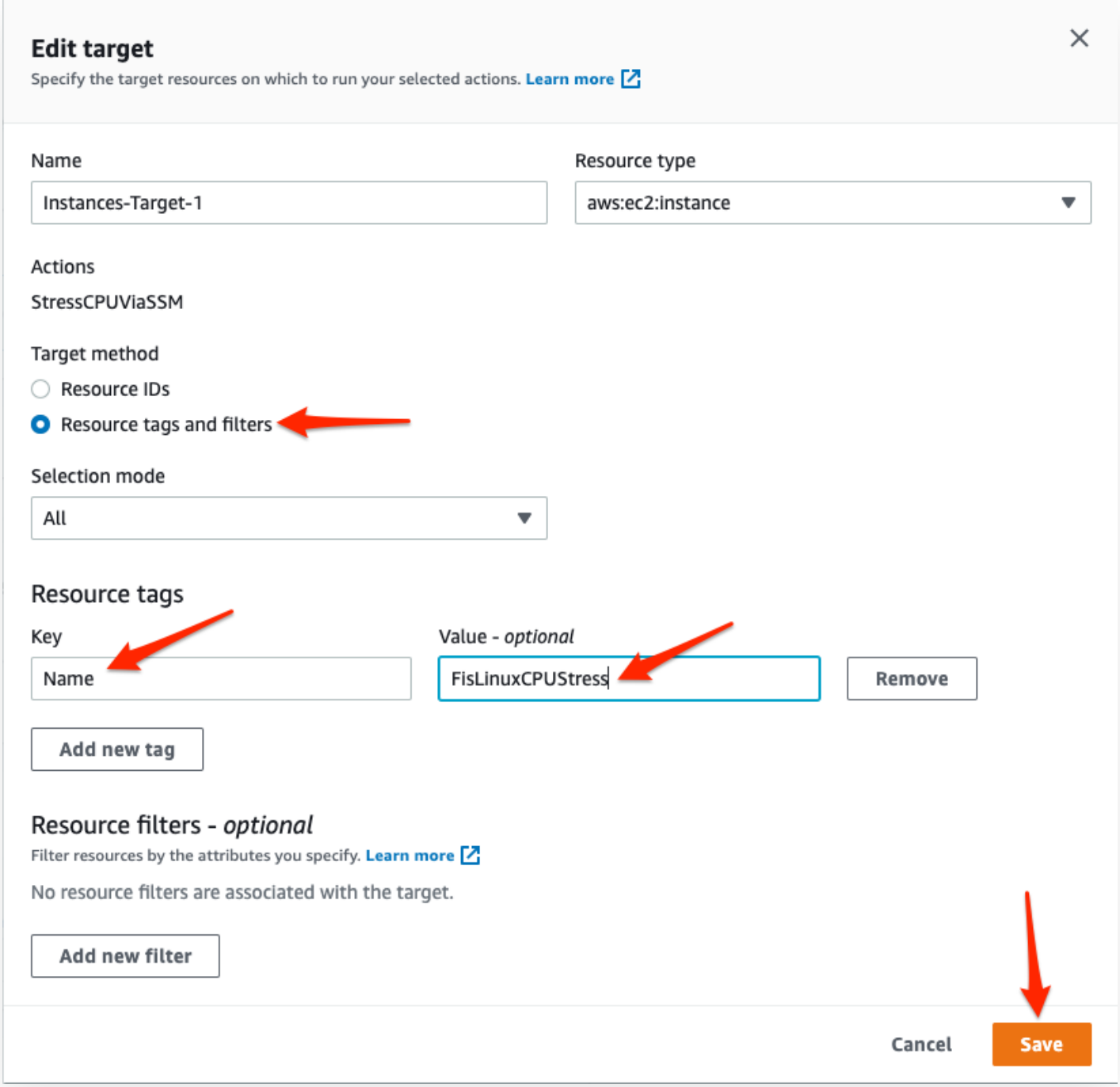


4. After we have entered a description and a role, we need to setup our actions. Click on the “Add Action” button in the Actions section.

Name the Action as `StressCPUViaSSM`, and under “Action Type” select `aws:ssm:send-command/AWSFIS-Run-Cpu-Stress`. This is an out of the box action to run stress test on Linux Instances using the stress-ng tool. Set the “documentParameters” field to `{"DurationSeconds":120}` which is passed to the script and the “duration” field to `2` which tells FIS how long to wait for a result. Finally click “Save”. This action will use AWS Systems Manager Run Command to run the AWSFIS-Run-Cpu-Stress command document against our targets for two minutes.

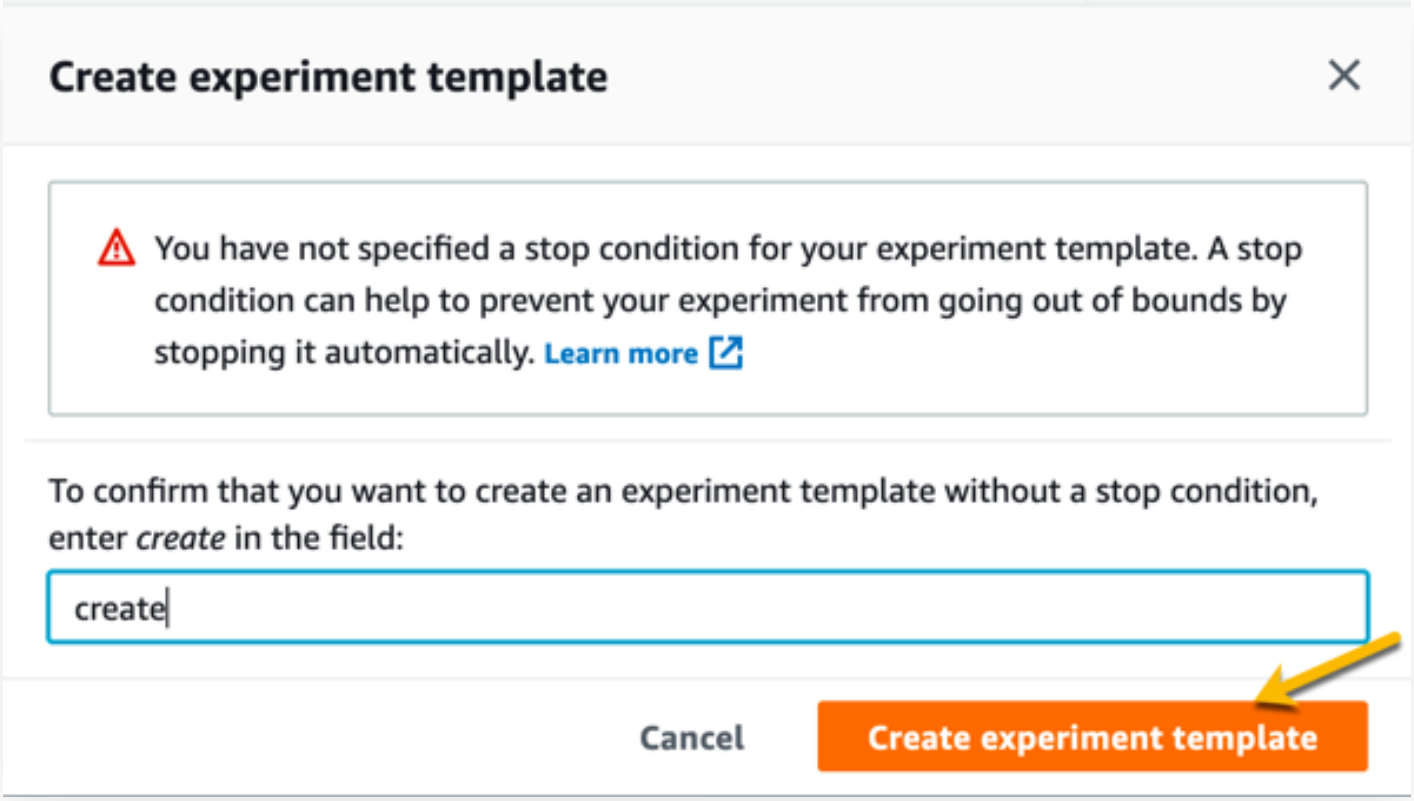


5. Once we have saved the action, let's edit our targets. Click on "Edit" in "Instaces-Target-1" card inside the "Targets" card. To select our target instances by tag select "Resource tags and filters" and keep selection mode `ALL`. Click "Add new tag" and enter a "Key" of `Name` and a "Value" of `FisLinuxCPUStress`. Finally click "Save".



6. Once we have actions and targets specified we can click on the "Create Experiment" button toward the bottom of the console to create our template.

Note: For this experiment we did not assign a stop condition, for a workshop or lab this is acceptable. However, it would be considered best practice to have stop conditions on your experiments so they don't go out of bounds. Because we do not have a stop condition we are being asked to confirm creation of this experiment. Type in `create` and then hit the "Create Experiment" button again to confirm.

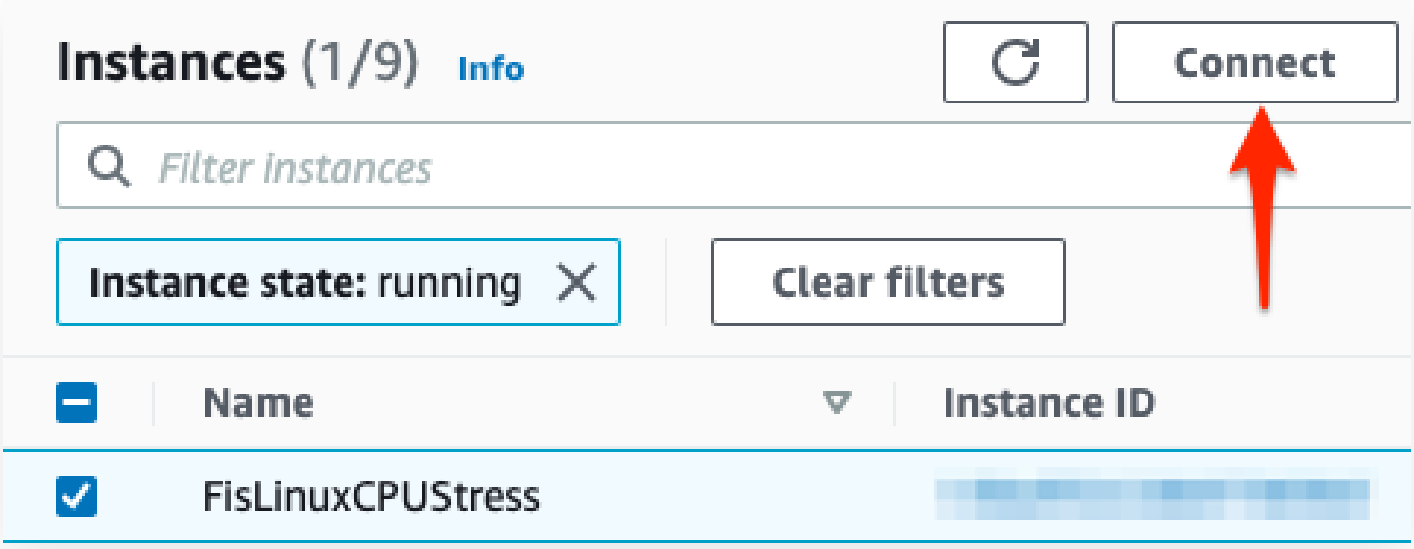


We have created our Linux CPU stress experiment template, now lets connect to our EC2 Instance.

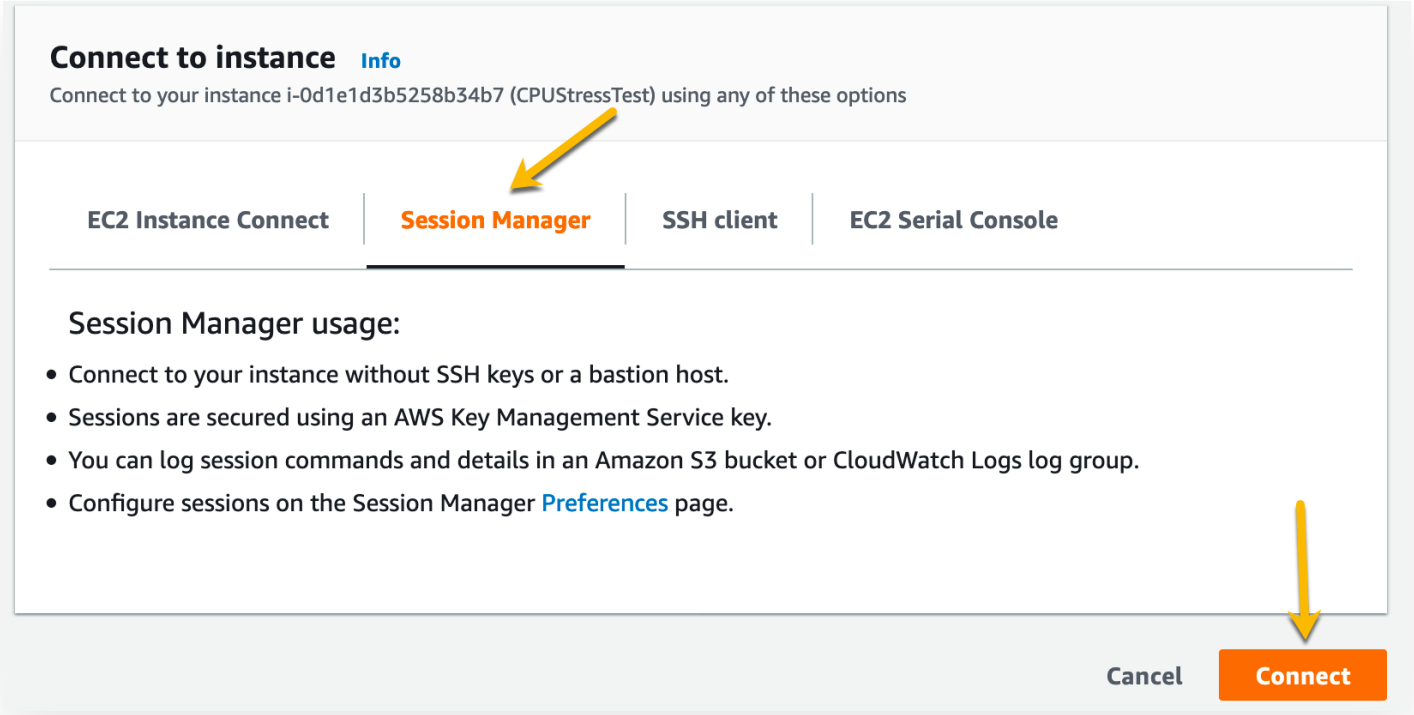
Validation procedure

We will use the linux `top` system command to observe the increased CPU load. To do this we now need to connect to our EC2 Instance so we can observe the CPU being stressed. Head over to the EC2 Console.

1. Once at the EC2 Console lets select our instance named `FisLinuxCpuStress` and click on the "Connect" button.



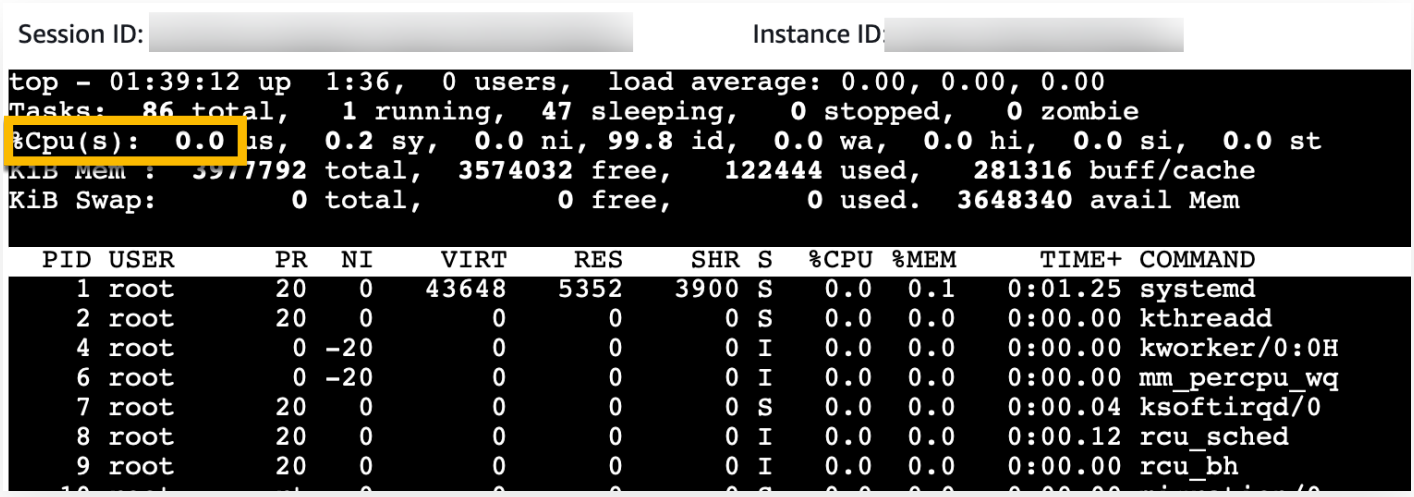
2. Select “Session Manager” and click on “Connect”.



This will open a session to the EC2 instance in another tab. In the new tab enter:

```
top
```

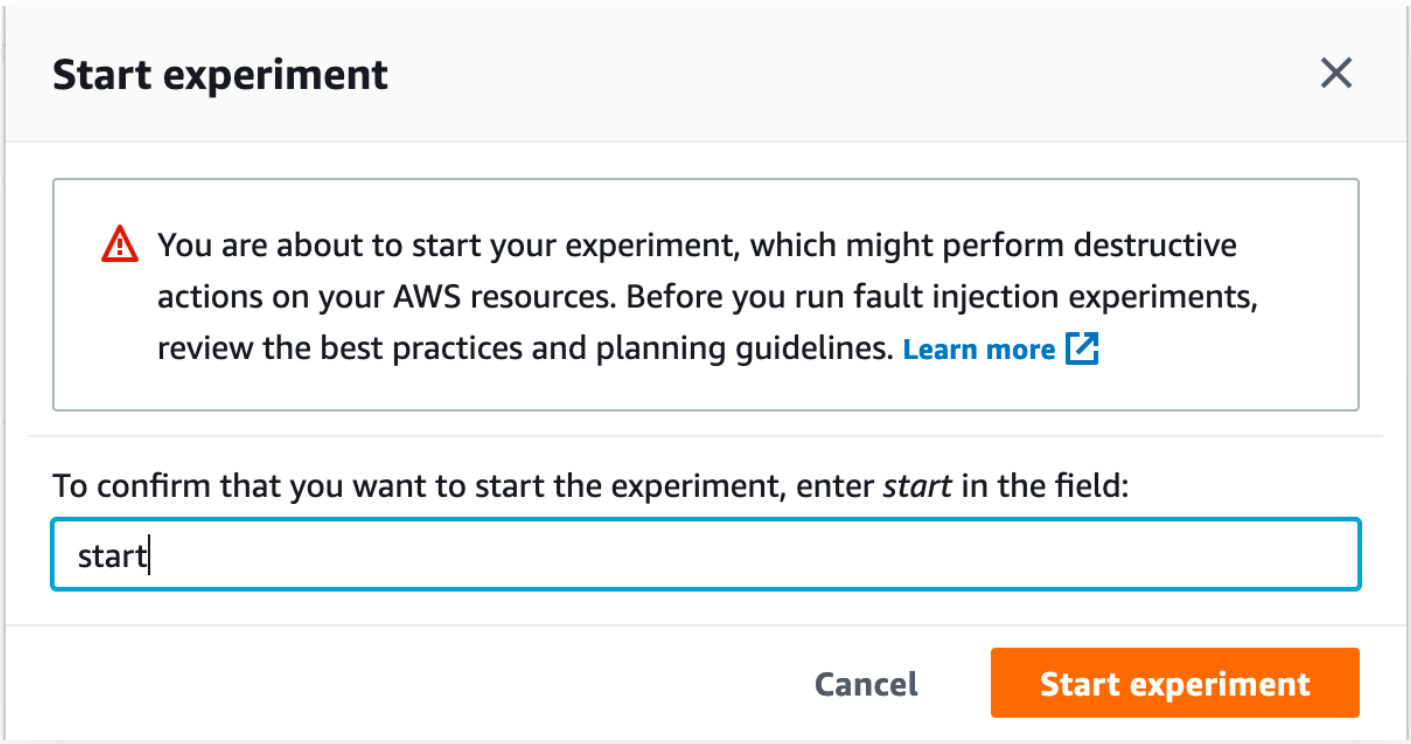
You should now see a continuously updating display similar to the next screenshot. Initially the CPU percentage should be at or close to zero as this instance is not doing anything. Keep this tab open, we will come back once we have started our experiment.



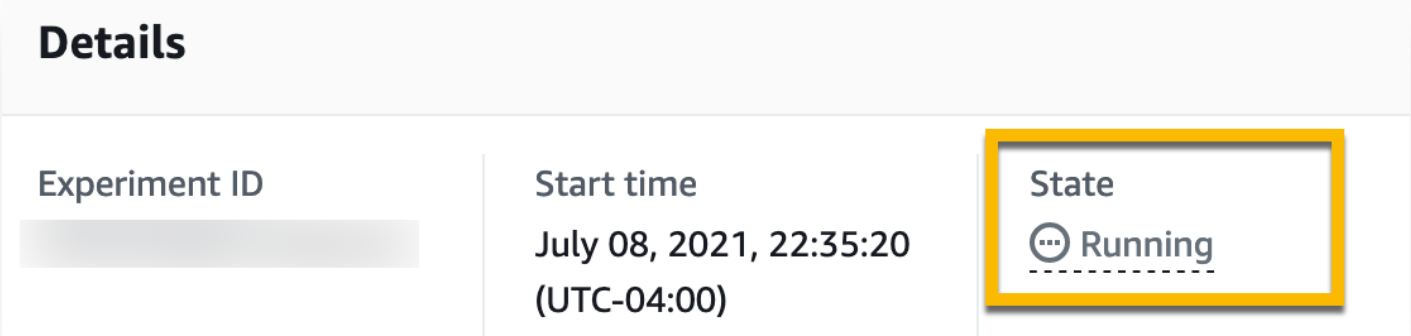
Run CPU Stress Experiment

Let's head back to the AWS Fault Injection Simulator Console.

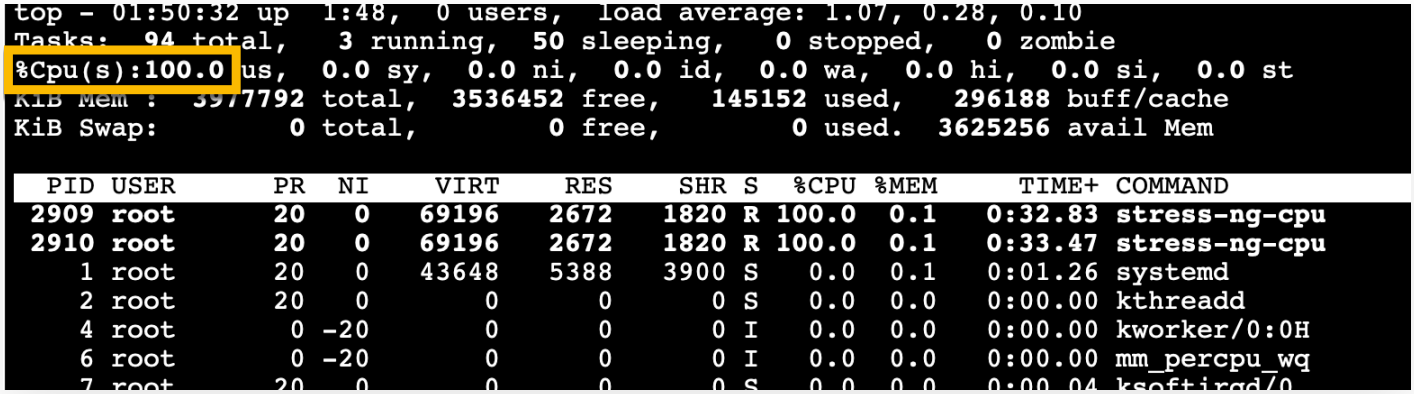
1. Once in the Fault Injection Simulator console, lets click on "Experiment templates" again on the left side pane.
2. Select the experiment template with the `LinuxBurnCPUviaSSM` description, then click on the "Actions" button and select "Start". This will allow us to enter additional tags before starting our experiment. Then click on the "Start Experiment" button.
3. Next type in `start` and click on "Start Experiment" again to confirm you want to start the experiment.



This will take you to the running experiment that is started from the template. In the detail section of the experiment check `State` and you should see the experiment is initializing. Once the experiment is running, lets head back to the open session on the EC2 Instance.



Watch the CPU percentage, it should hit 100% for a few minutes and then return back to 0%. Once we have observed the action we can click the `Terminate` button to terminate our Session Manager session.



Congrats for completing this lab! In this lab you walked through running an experiment that took action within a Linux EC2 Instance using AWS Systems Manager. Using the integration between Fault Injection Simulator and AWS Systems Manager you can run scripted actions within an EC2 Instance. Through this integration you can script events against your applications or run other chaos engineering tools and frameworks.

Learning and improving

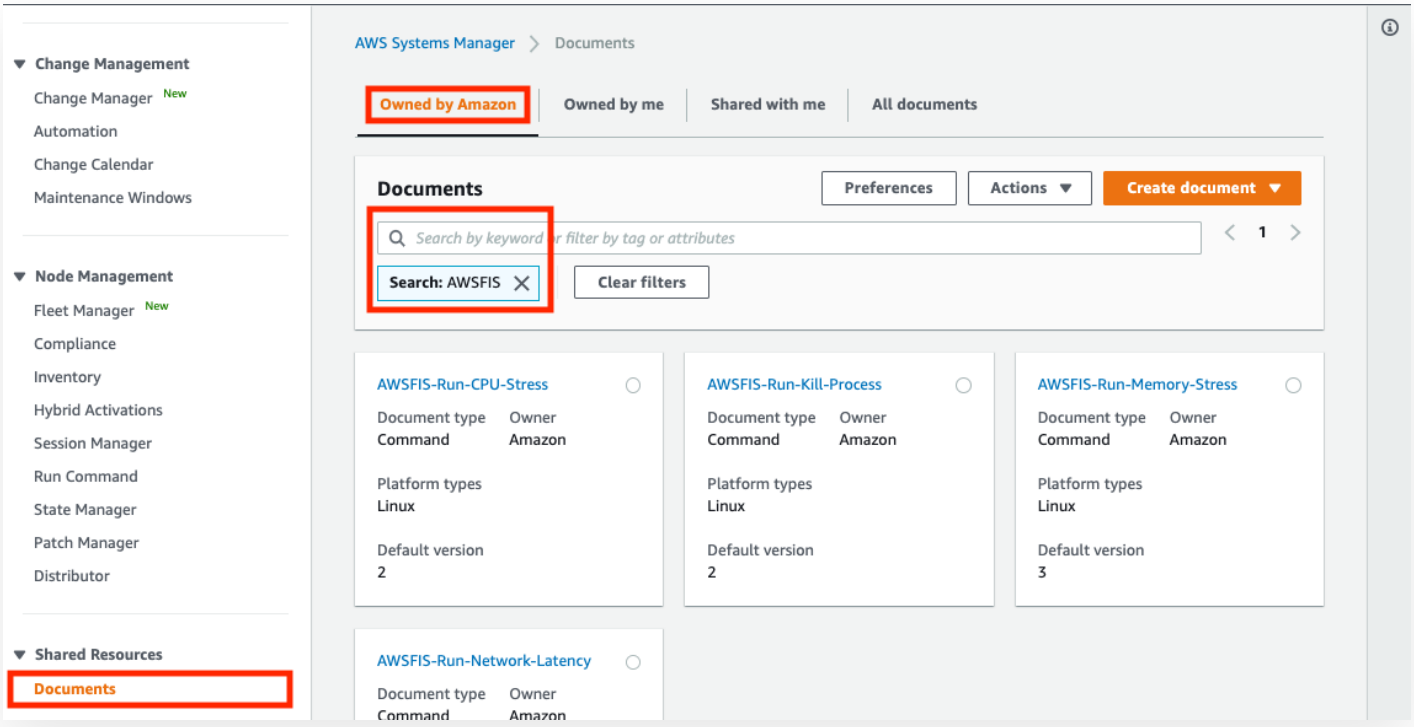
Since this instance wasn't doing anything, there aren't any actions. To think about how to use this to test a hypothesis and make an improvement, consider running the same experiment against the ASG instances from the **First Experiment** section. Maybe you could use this to tune the optimal CPU levels for scaling up or down?

WORKING WITH SSM DOCUMENTS

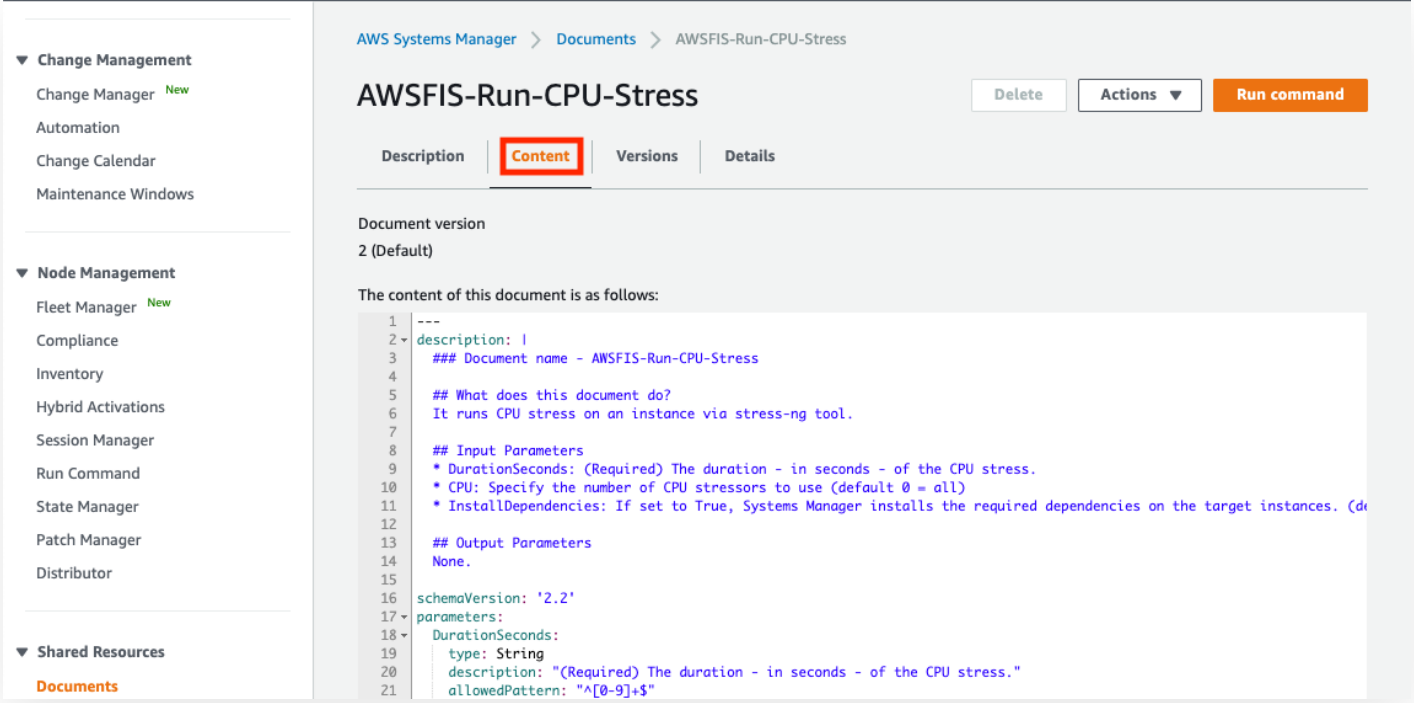
Pre-configured SSM documents

The linux CPU stress experiment we saw in the previous section used one of the pre-configured SSM documents to run a script on our Linux instance.

To find the script, navigate to the AWS Systems Manager console, scroll down in the left-hand menu all the way to the bottom to "Documents", select "Owned by Amazon", and search for `AWSFIS`. Note that this search may take a few seconds to display results.



To inspect the script, click on the script name, i.e. `AWSFIS-Run-CPU-Stress`, then select the "Content" tab.



The document is a YAML file defining two `aws:runShellScript` actions, `InstallDependencies` to install the `stress-ng` package, and `ExecuteStressNg` to inject CPU stress.

Custom SSM documents

Currently AWS does not provide a CPU stress document for Windows, but we can create our own. For more information on writing SSM documents please see these resources

- AWS Systems Manager documentation
- Writing your own SSM documents blog
- AWS SSM workshop

If you want to see an example how one might inject stress, you can have a look at the `WinStressDocument` resource in the CloudFormation template. Alternatively you can follow the same search procedure as for the AWS owned documents but search the “Owned by me” or “Shared by me” tabs instead of “Owned by AWS”.

For additional SSM sample documents relating to FIS see these resources

- https://github.com/adhorn/chaos-ssm-documents

Working with custom SSM documents in FIS

While writing custom SSM documents is outside the scope of this workshop, there are a few aspects of SSM documents you should be aware of:

- **Document ARN** - FIS requires the full SSM document ARN. However, the only time SSM will list the full document ARN is if the document is shared from another account. However, based on the document ID you can create the ARN based on this format string: `arn:${AWS::Partition}:ssm:${AWS::Region}:${AWS::AccountId}:document/${WinStressDocumen`
- **Exit status** - Shell script convention is to signal success with a return/exit value of `0` and a failure with any non-zero numeric value. If FIS detects a non-zero exit status on an SSM script it will mark the action as "Failed", terminate all running actions, cancel queued actions, invoke any outstanding roll-back actions, cancel experiment execution, and mark the overall experiment as "Failed".
- **Duration** - FIS actions have a "Duration" setting and will stop action execution if the action has not finished within this time period. For SSM actions this will "cancel" the command. If the command has a sequence of steps, this will result in only some of the steps being executed.
- **onCancel** / **onFailure** - SSM provides you with a means to ensure that automation can fail / clean-up safely by providing `onCancel` and `onFailure` properties on each step. These properties allow designating clean-up steps to perform.

WINDOWS CPU STRESS EXPERIMENT

Warning

This section requires that you have an RDP client on your local machine. This section cannot be performed from a cloud9 instance.

In this section we will run a CPU Stress test using AWS Fault Injection Simulator against an Amazon Windows EC2 Instance. The Windows CPU stress test will use a custom SSM command document. We will do the following:

1. Create experiment template to stress CPU.
2. Reset password on Windows Instance.
3. Connect to Windows EC2 Instance and run task manager.
4. Start experiment and observe results.

Experiment Setup

Create CPU Stress Experiment

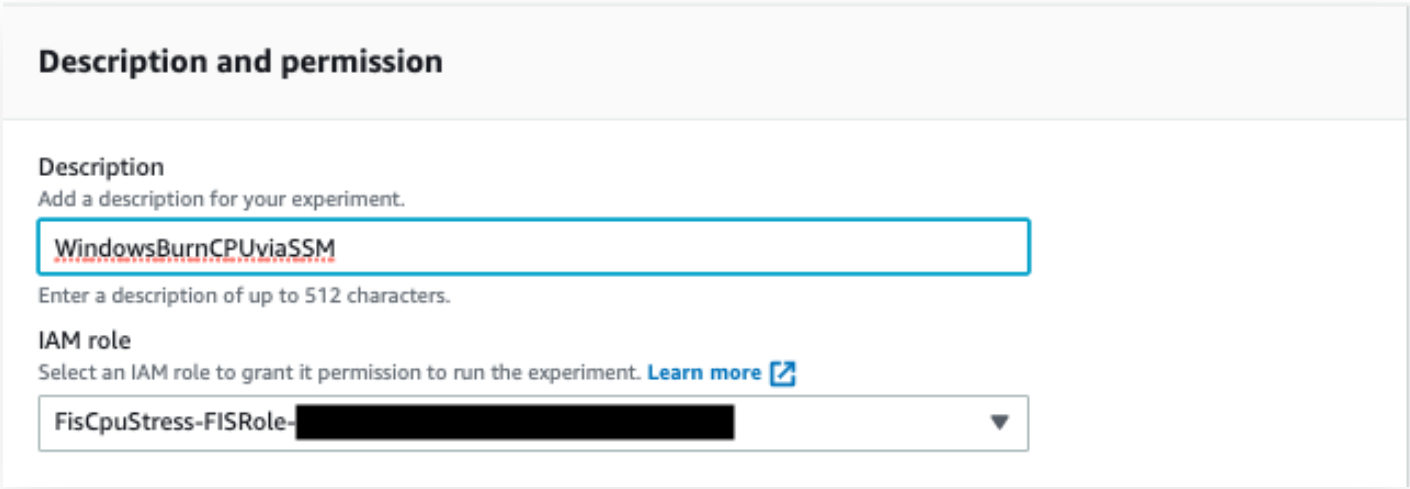
First, lets create our stress experiment. We can do this programmatically but we will walk through this on the console.

1. Open the AWS Fault Injection Simulator Console. Once in the Fault Injection Simulator console, lets click on “Experiment templates” on the left side pane.

2. Click on “Create experiment template” on the upper right hand side of the console to start creating our experiment template.

3. Next we will enter the description of the experiment and choose the IAM Role. Let’s put `WindowsBurnCPUviaSSM` for the description. The IAM role allows the FIS service permissions to execute

the actions on your behalf. As part of the CloudFormation stack a role was created for this experiment that starts with `FisCpuStress-FISRole`, select that role. Please examine the CloudFormation template or IAM Role for the policies in this role.

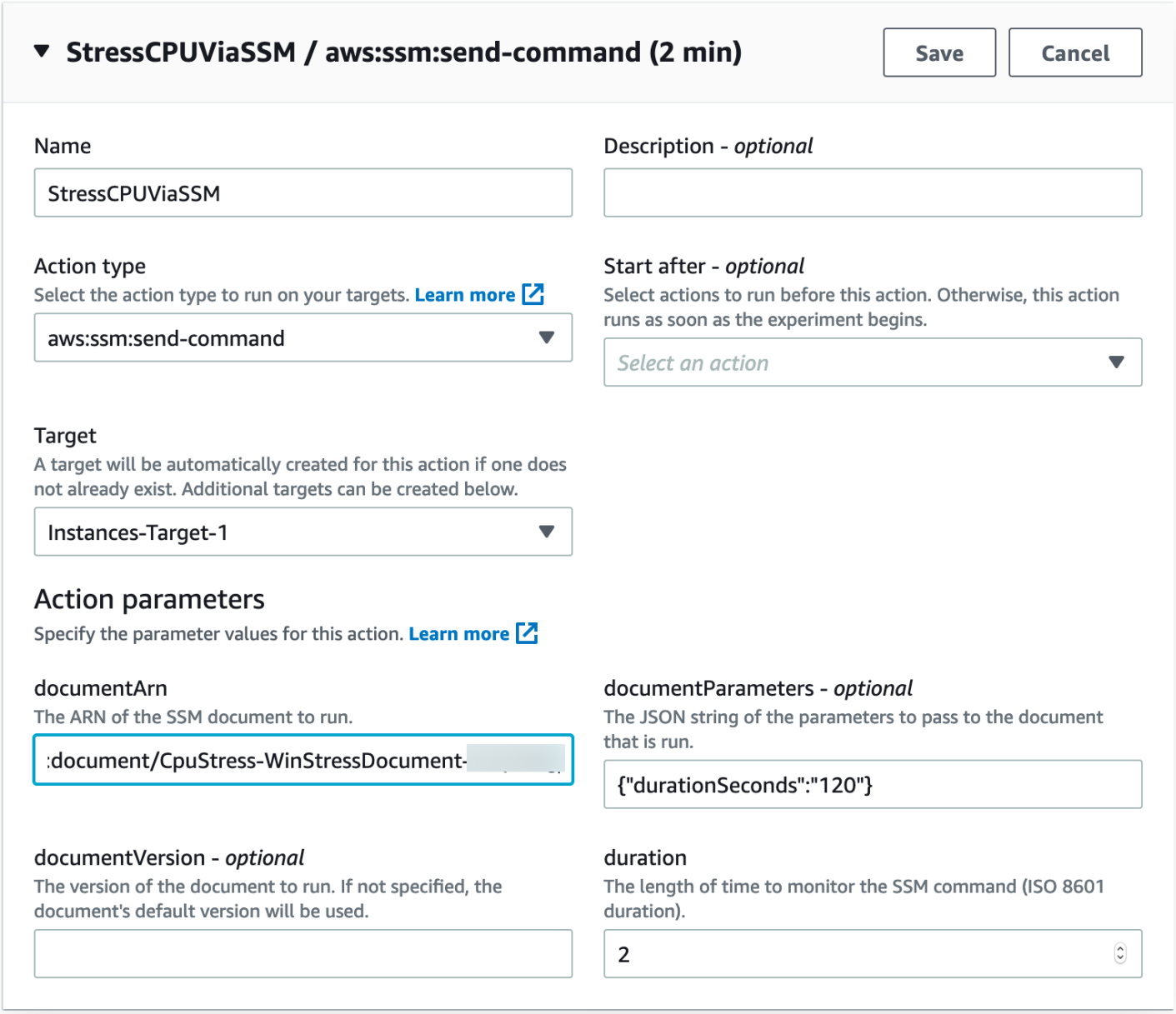


4. After we have entered a description and a role, we need to setup our actions. Click the "Add action" button in the Actions Section.

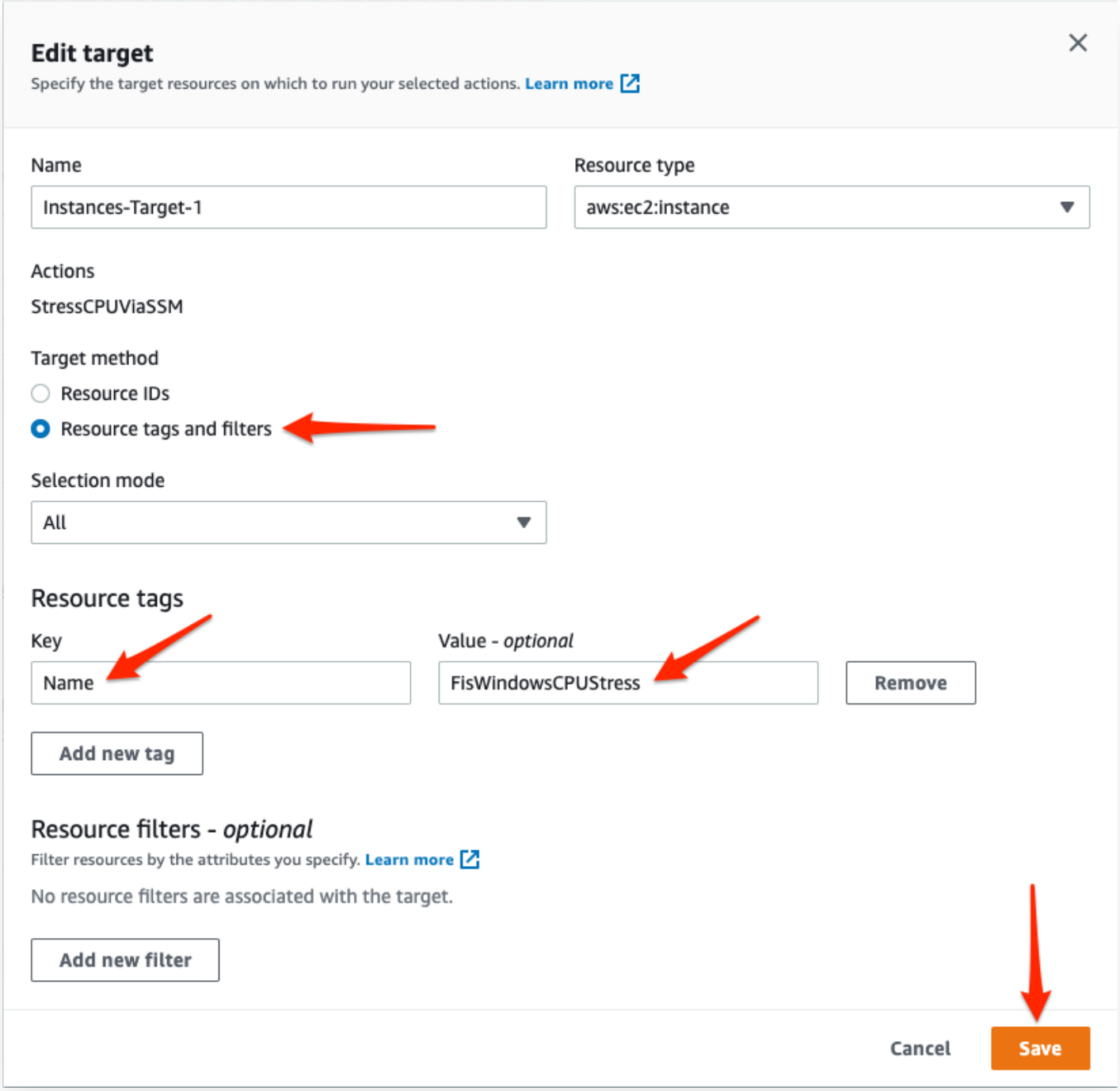
Enter a "Name" of `StressCPUViaSSM`, and under "Action Type" select the `aws:ssm:send-command` action. Currently there is not an out of box Action for Windows CPU Stress Testing, so we are using the send-command action along with a command document that was deployed by our CloudFormation template. To view this document please reference the `WinStressDocument` resource in the CloudFormation template.

To find the ARN of the document that was created by the template, open a new tab and browse to the CloudFormation console, select "Stacks", click on the stack named "FisCpuStress", then select "Outputs". Copy the value of the `WinStressDocumentArn` entry as you will need it in the next step.

Return to the FIS console and enter the ARN you copied into the "documentArn" field. Then set the "documentParameters" field to `{"durationSeconds":120}` which is passed to the script and the "duration" field to `2` which tells FIS how long to wait for a result. Finally click "Save". This action will use AWS Systems Manager Run Command to run the `FisCpuStress-WinStressDocument` document against our targets for two minutes.

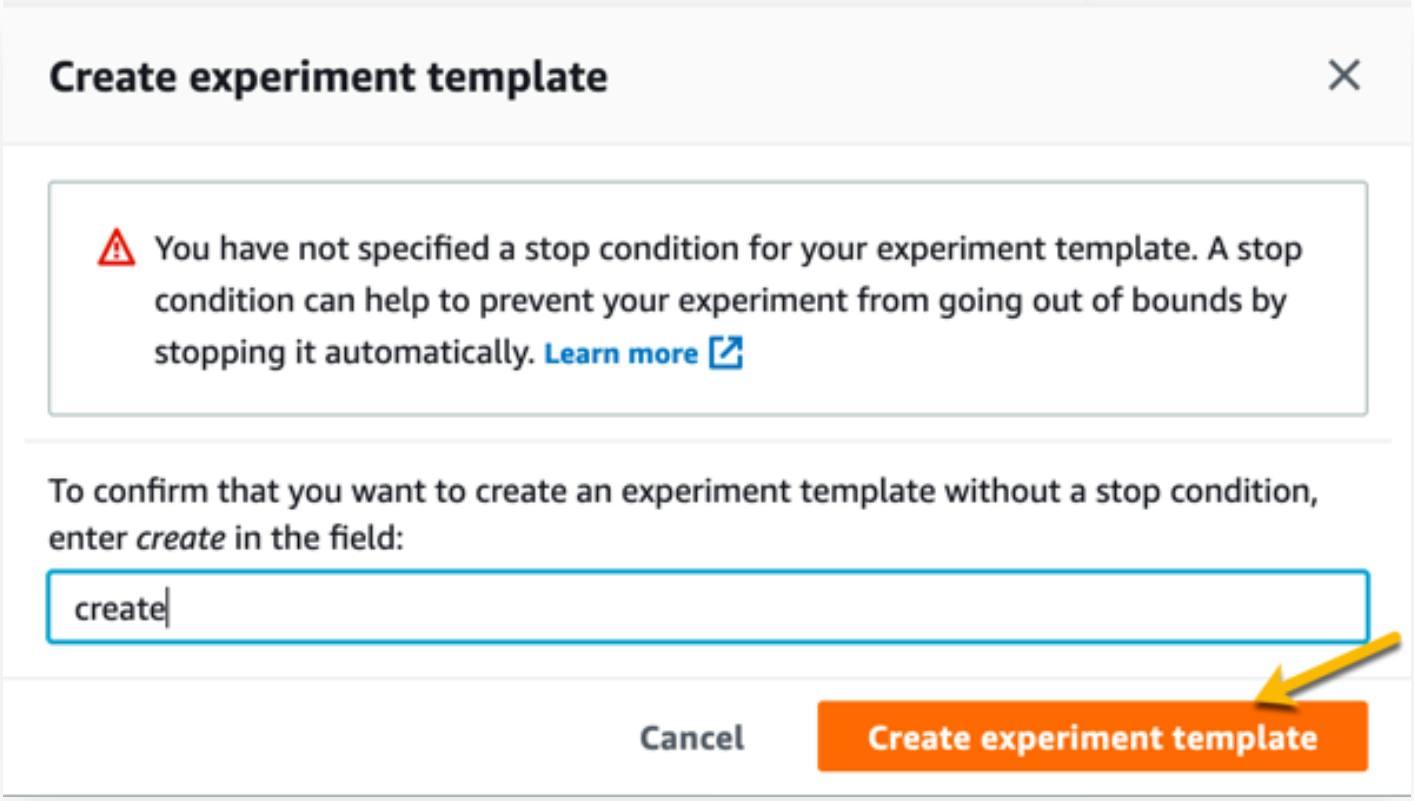


5. Once we have saved the action, let's edit our targets. Click on "Edit" in "Instaces-Target-1" card inside the "Targets" card. To select our target instances by tag select "Resource tags and filters" and keep selection mode `ALL`. Click "Add new tag" and enter a "Key" of `Name` and a "Value" of `FisWindowsCPUStress`. Finally click "Save".



6. Once we have actions and targets specified we can click on the "Create Experiment" button toward the bottom of the console to create our template.

Note: For this experiment we did not assign a stop condition, for a workshop or lab this is acceptable. However, it would be considered best practice to have stop conditions on your experiments so they don't go out of bounds. Because we do not have a stop condition we are being asked to confirm creation of this experiment. Type in `create` and then hit the "Create Experiment" button again to confirm.



We have created our Windows CPU stress experiment template, now lets connect to our EC2 Instance.

Validation procedure

We will use the Windows task manager to observe increased CPU load. To do this we now need to connect to our EC2 Instance so we can observe the CPU being stressed.

Use AWS Systems Manager Run Command to reset Password

When we deployed the instance we didn't use SSH Keys, and we don't know the password. However, with the SSM Agent along with the right IAM privileges we have a break glass scenario where we can reset the password. Please adjust the value of `TMP_PASSWORD` and use the commands below to find the InstanceId of the `FisWindowsCPUStress` instance and help you reset the admin password to the password of choice.

```
# For readability - passing passwords this way is not secure
TMP_PASSWORD=ENTER_NEW_PASSWORD_HERE
```

```
# For readability and convenience
TMP_INSTANCE=$(  aws ec2 describe-instances --filter
Name=tag:Name,Values=FisWindowsCPUStress --query
'Reservations[*].Instances[0].InstanceId' --output text )

# Reset password on instance - this is not a secure method,
# in real life use AWS-PasswordReset document
aws ssm send-command \
  --document-name "AWS-RunPowerShellScript" \
  --document-version "1" \
  --targets '[{"Key":"InstanceIds","Values":["'${TMP_INSTANCE}'"]}]' --parameters
'{"workingDirectory":[""],"executionTimeout":["3600"],"commands":["net user
administrator '${TMP_PASSWORD}'"]}' \
  --timeout-seconds 600 \
  --max-concurrency "50" \
  --max-errors "0" \
  --cloud-watch-output-config '{"CloudWatchOutputEnabled":false}'
```

Use AWS Systems Manager Session Manager to connect to Target Instance

We now need to connect to our EC2 Instance so we can observe the CPU being stressed. We are going to do this by using the port forwarding capability of AWS Systems Manager Session Manager and using RDP.

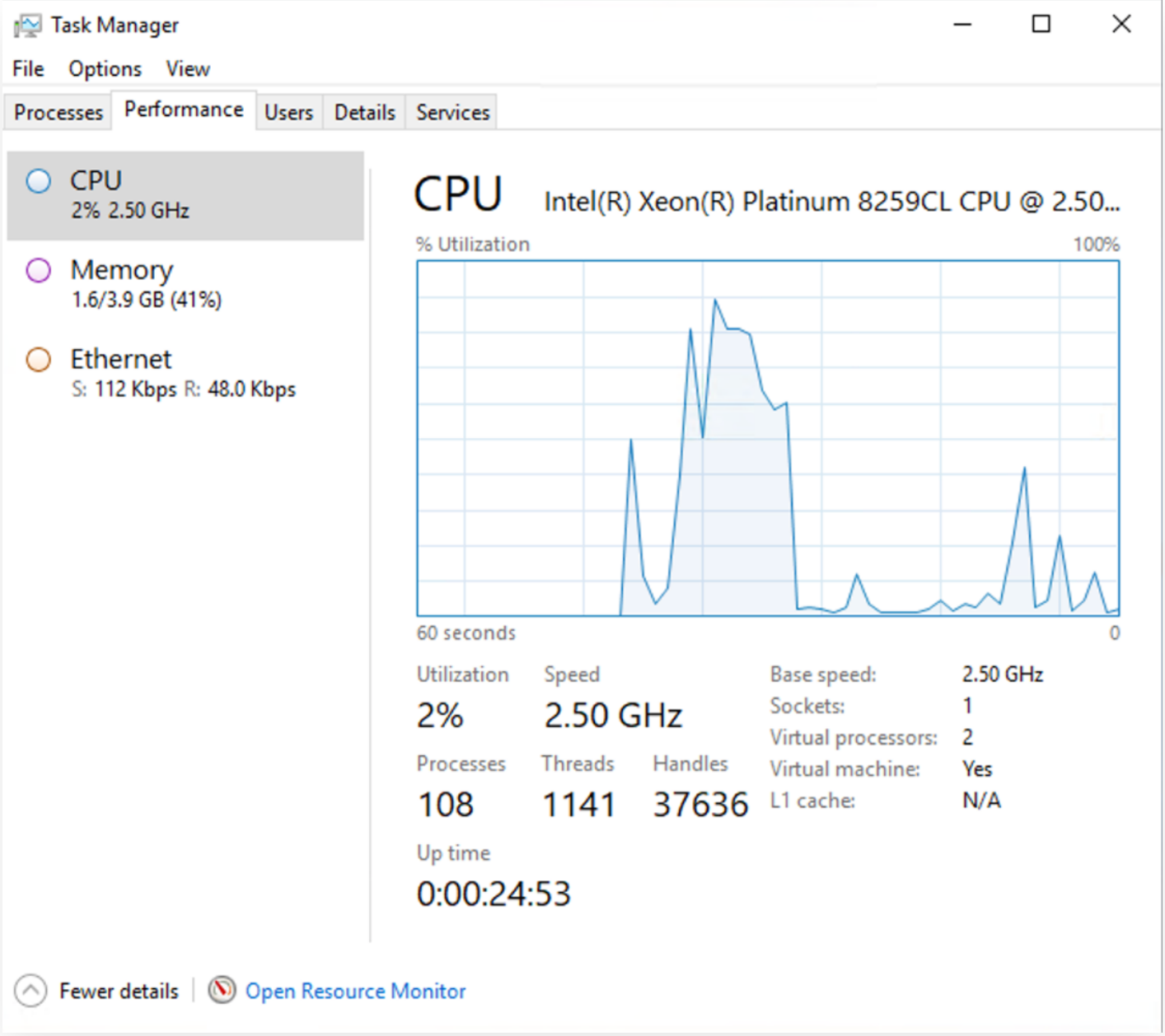
1. First make sure that the Session Manager plugin for the AWS CLI is installed on your local machine.
2. Run the following command first, this will forward local port 56788 to port 3389 on the Windows EC2 Instance. Replace the with the instance ID of the Windows Instance.

   ```
   # This presumes you set TMP_INSTANCE (see above)
   aws ssm start-session --target ${TMP_INSTANCE} --document-name AWS-
   StartPortForwardingSession --parameters '{"portNumber":
   ["3389"],"localPortNumber":["56788"]}'
   ```

3. Once the command says waiting for connections you can launch the RDP client and enter `localhost:56788` for the server name and login as `administrator` with the password you set in the previous section.

   › Troubleshooting connectivity

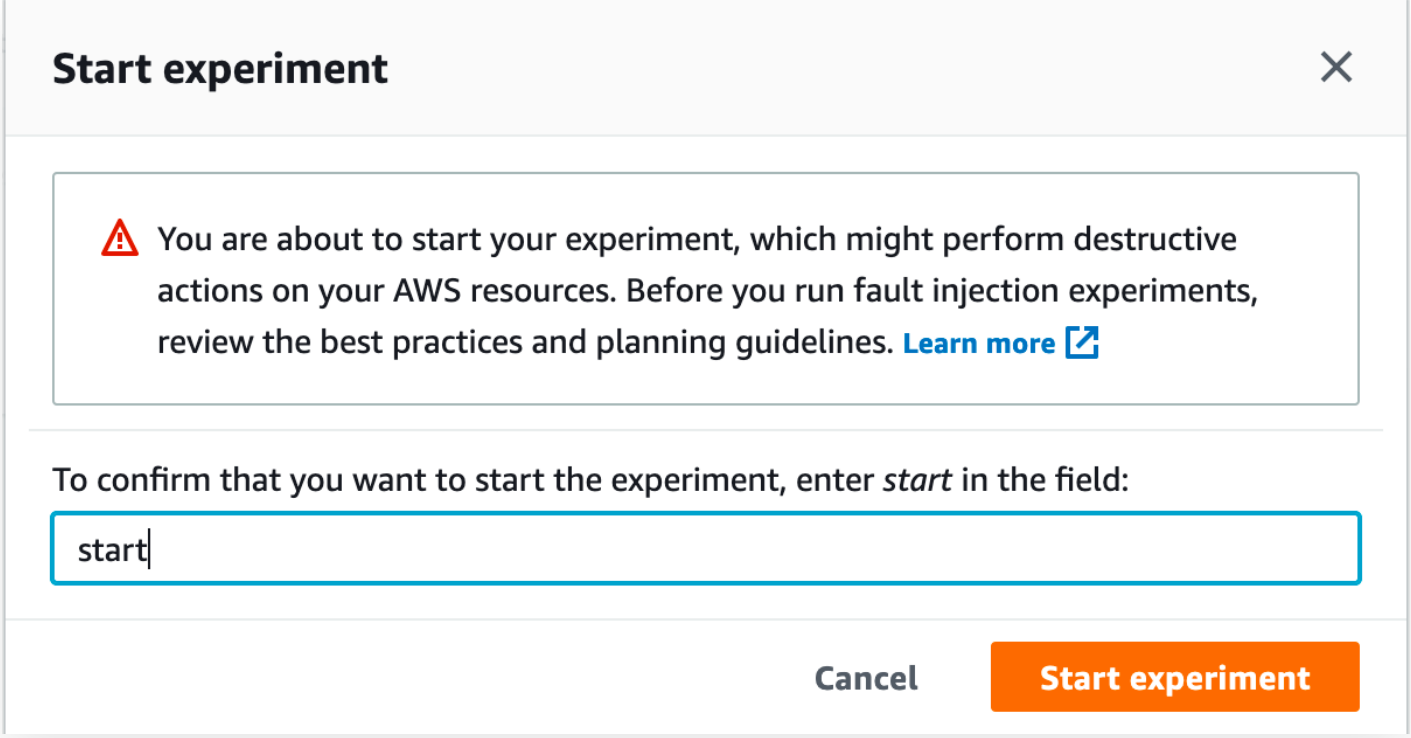
4. Once you have RDP'ed into the Windows Instance, launch task manager by right clicking on the menu bar and selecting "Task Manager". Click on "More details" button and then on the "Performance" tab so you can see the CPU graph as shown below.



Run CPU Stress Experiment

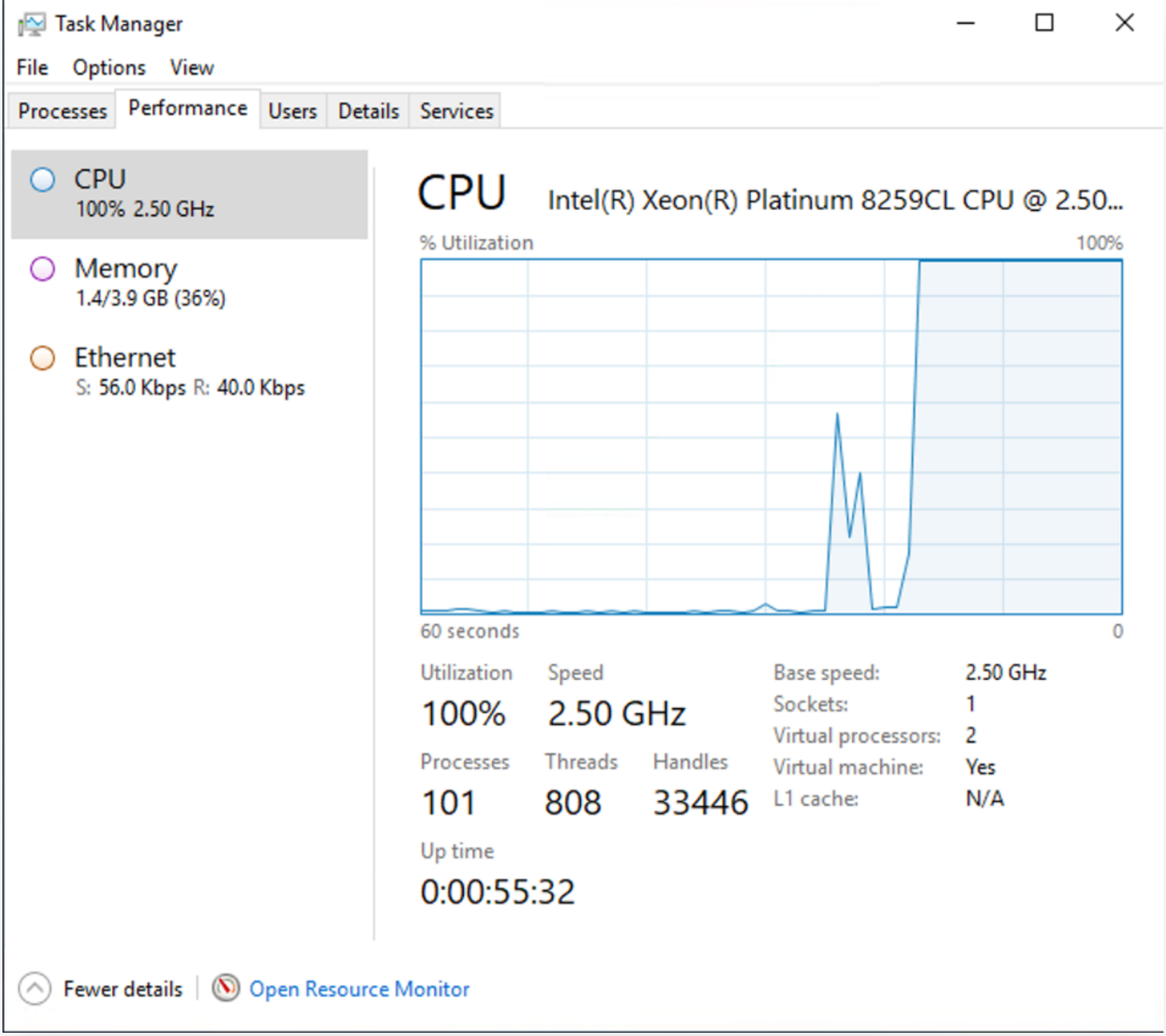
Let's head back to the AWS Fault Injection Simulator Console.

1. Once in the Fault Injection Simulator console, lets click on "Experiment templates" again on the left side pane.
2. Select the experiment with the `WindowsBurnCPUviaSSM` description, then click on the "Actions" button and select "Start". This will allow us to enter additional tags before starting our experiment. Then click on the "Start experiment" button.
3. Next type in `start` and click on "Start Experiment" again to confirm you want to start the experiment.



This will take you to the running experiment that is started from the template. In the detail section of the experiment check `State` and you should see the experiment is initializing. Once the experiment is running, lets go back to the RDP session and observe the task manager graph.

Watch the CPU percentage, it should hit 100% for a few minutes and then return back to 0%. Once we have observed the action we can logout of the Windows Instance and hit CTRL + C on the window you ran the port forwarding command to close the session.



Learning and improving

Congrats for completing this lab! In this lab you walked through running an experiment that took action within a Windows EC2 Instance using AWS Systems Manager and a custom run command. Using the integration between Fault Injection Simulator and AWS Systems Manager you can run scripted actions within an EC2 Instance. Through this integration you can script events against your applications or run other chaos engineering tools and frameworks.

Since this instance wasn't doing anything there aren't any actions. To think about how to use this to test a hypothesis and make an improvement consider building custom SSM scripts to run custom scenarios. We will cover some of these in the **Common Scenarios** section.

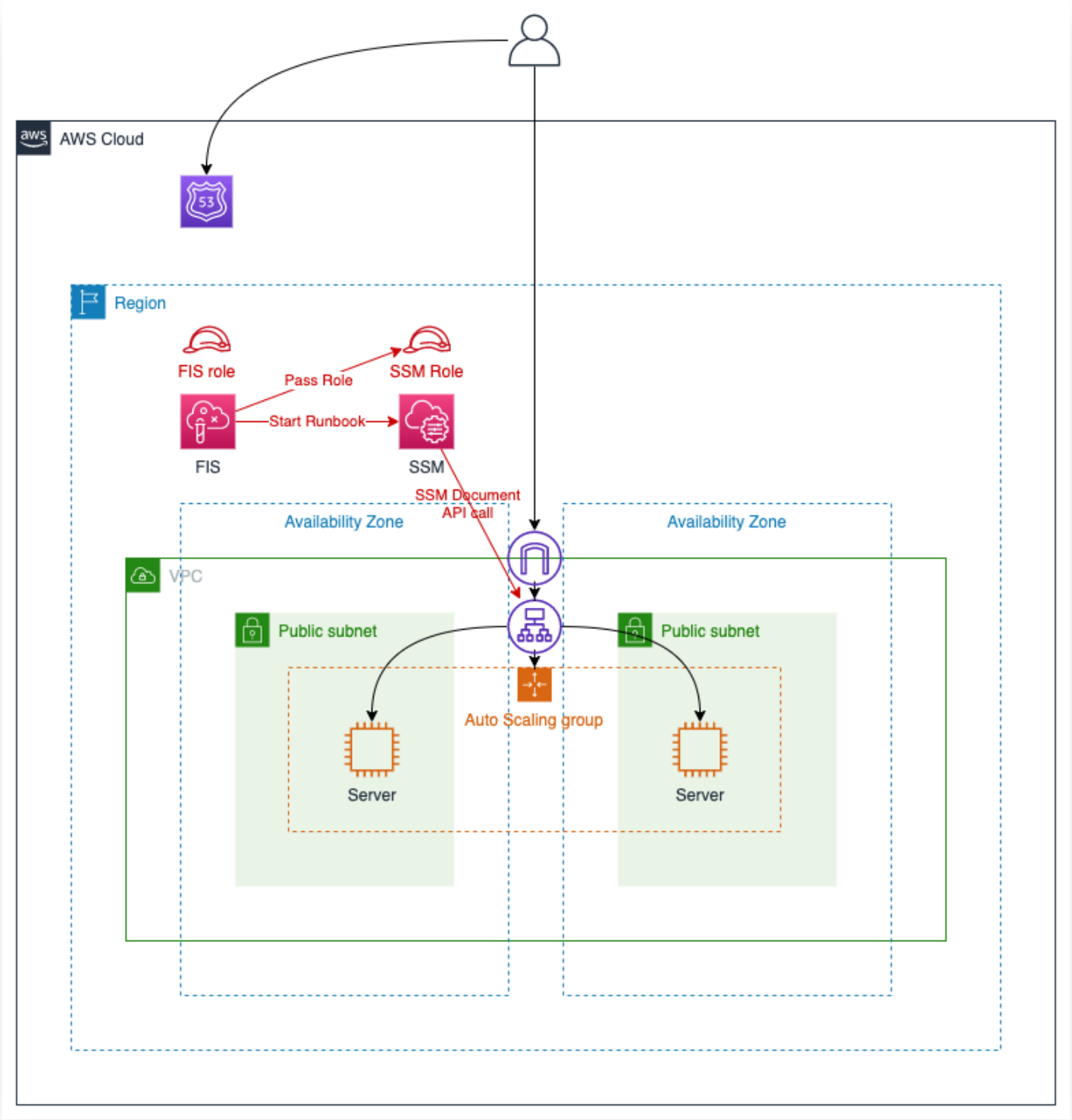
Cleanup

If you created an additional `CpuStress` CloudFormation stack in the **FIS SSM Setup** section, make sure to delete that stack to avoid incurring additional costs.

FIS SSM START AUTOMATION SETUP

In the previous sections we used FIS actions to directly interact with AWS APIs to terminate EC2 instances, and the SSM SendCommand option to execute code directly on our virtual machines.

In this section we will cover how to execute additional actions against AWS APIs that are not yet supported by FIS by using SSM Runbooks.



Configure permissions

In the Configuring Permissions section we defined a service role `FisWorkshopServiceRole` that granted us access to running the FIS `aws:ssm:send-command` on our instances. To use the `aws:ssm:start-automation-execution` action we will need to update our permissions

Create SSM role

As shown in the image above, SSM Runbooks require us to define and pass a separate role. Let's say we want to create an SSM document that can terminate instances in an autoscaling group. A policy for that might need the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableAsgDocument",
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:SuspendProcesses",
                "autoscaling:ResumeProcesses",
                "ec2:DescribeInstances",
                "ec2:DescribeInstanceStatus",
                "ec2:TerminateInstances"
            ],
            "Resource": "*"
        }
    ]
}
```

Since SSM needs to be able to assume this role for running an SSM document we also need to define a trust policy:

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Principal": {
            "Service": "ssm.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }
}
```

To create a role, save the two JSON block above into files named `iam-ec2-demo-policy.json` and `iam-ec2-demo-trust.json` and run the following CLI commands to create a role named `FisWorkshopSsmEc2DemoRole`

```
ROLE_NAME=FisWorkshopSsmEc2DemoRole
```

```
aws iam create-role \
  --role-name ${ROLE_NAME} \
  --assume-role-policy-document file://iam-ec2-demo-trust.json

aws iam put-role-policy \
  --role-name ${ROLE_NAME} \
  --policy-name ${ROLE_NAME} \
  --policy-document file://iam-ec2-demo-policy.json
```

Note the ARN of the created role as we will need it below.

› Troubleshooting Security Token Invalid when Creating IAM Role

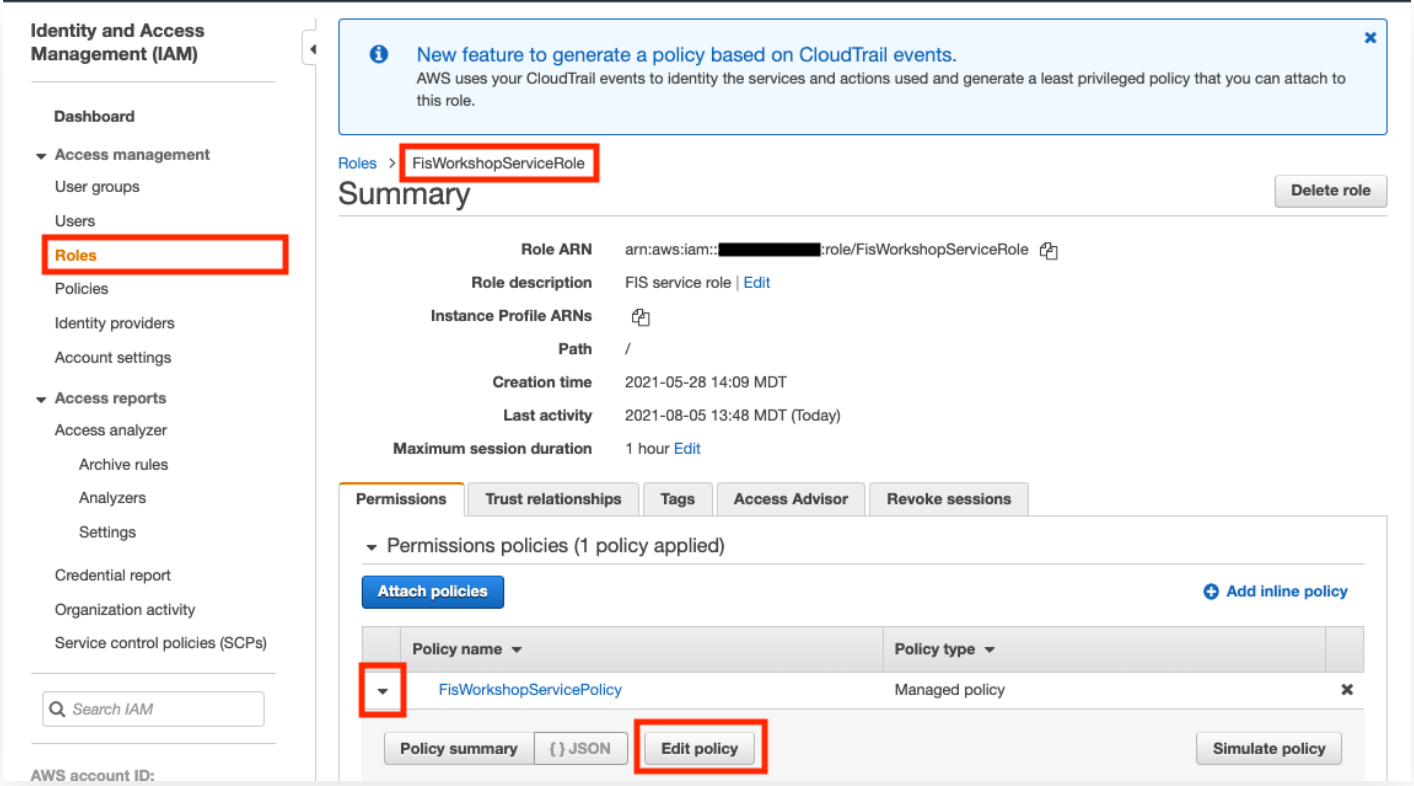
Update FIS service role

The `FisWorkshopServiceRole` we defined in the Configuring Permissions only grants limited access to SSM so we need to add the following two policy statements.

```
        {
            "Sid": "EnableSSMAutomationExecution",
            "Effect": "Allow",
            "Action": [
                "ssm:GetAutomationExecution",
                "ssm:StartAutomationExecution",
                "ssm:StopAutomationExecution"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowFisToPassListedRolesToSsm",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "PLACE_ROLE_ARN_HERE"
        },
```

The first statement allows FIS to use SSM actions. The second statement defines the role that SSM will use. Make sure to insert the ARN of the role you created above.

To update the `FisWorkshopServiceRole`, navigate to the IAM console, select “Roles” on the left, and search for `FisWorkshopServiceRole`.



Expand the `FisWorkshopServicePolicy` and select “Edit Policy”. Then select the JSON tab and copy the above JSON block just above the first statement `AllowFISExperimentRoleReadOnly`:

Then select “Review policy” and “Save Changes”.

Tip

If the policy editor shows errors, check that you have separated blocks with commas, and that you have updated the Role ARN to a valid value.

Create SSM document

For this section we will replicate the FIS terminate instance action using SSM. Copy the YAML below into a file named `ssm-terminate-instances-asg-az.yaml`

```
---
description: Terminate all instances of ASG in a particular AZ
schemaVersion: '0.3'
assumeRole: "{{ AutomationAssumeRole }}"
parameters:
  AvailabilityZone:
    type: String
    description: "(Required) The Availability Zone to impact"
  AutoscalingGroupName:
    type: String
    description: "(Required) The names of the autoscaling group"
  AutomationAssumeRole:
    type: String
    description: "The ARN of the role that allows Automation to perform
      the actions on your behalf."
mainSteps:
# Find all instances in ASG
- name: DescribeAutoscaling
  action: aws:executeAwsApi
  onFailure: 'step:ExitReview'
  onCancel: 'step:ExitReview'
  timeoutSeconds: 60
  inputs:
    Service: autoscaling
    Api: DescribeAutoScalingGroups
    AutoScalingGroupNames:
        - "{{ AutoscalingGroupName }}"
  outputs:
    - Name: InstanceIds
      Selector: "$..InstanceId"
      Type: StringList
# Find all ASG instances in AZ
- name: DescribeInstances
  action: aws:executeAwsApi
  onFailure: 'step:ExitReview'
  onCancel: 'step:ExitReview'
  timeoutSeconds: 60
  inputs:
```

```
    Service: ec2
    Api: DescribeInstances
    Filters:
    - Name: "availability-zone"
      Values:
        - "{{ AvailabilityZone }}"
    - Name: "instance-id"
      Values: "{{ DescribeAutoscaling.InstanceIds }}"
  outputs:
     - Name: InstanceIds
       Selector: "$..InstanceId"
       Type: StringList
# Terminate 100% of selected instances
- name: TerminateEc2Instances
  action: aws:changeInstanceState
  onFailure: 'step:ExitReview'
  onCancel: 'step:ExitReview'
  inputs:
    InstanceIds: "{{ DescribeInstances.InstanceIds }}"
    DesiredState:  terminated
    Force: true
# Wait for up to 90s to make sure instances have been terminated
- name: VerifyInstanceStateTerminated
  action: aws:waitForAwsResourceProperty
  onFailure: 'step:ExitReview'
  onCancel: 'step:ExitReview'
  timeoutSeconds: 90
  inputs:
    Service: ec2
    Api: DescribeInstanceStatus
    IncludeAllInstances: true
    InstanceIds: "{{ DescribeInstances.InstanceIds }}"
    PropertySelector: "$..InstanceState.Name"
    DesiredValues:
      - terminated
# On normal exit or failure list instances in ASG/AZ
- name: ExitReview
  action: aws:executeAwsApi
  timeoutSeconds: 60
  inputs:
    Service: ec2
    Api: DescribeInstances
    Filters:
    - Name: "availability-zone"
      Values:
        - "{{ AvailabilityZone }}"
    - Name: "instance-id"
      Values: "{{ DescribeAutoscaling.InstanceIds }}"
  outputs:
     - Name: InstanceIds
       Selector: "$..InstanceId"
       Type: StringList
outputs:
- DescribeInstances.InstanceIds
- ExitReview.InstanceIds
```

Use the following CLI command to create the SSM document and export the document ARN:

```
SSM_DOCUMENT_NAME=TerminateAsgInstancesWithSsm

# Create SSM document
aws ssm create-document \
  --name ${SSM_DOCUMENT_NAME} \
  --document-format YAML \
  --document-type Automation \
  --content file://ssm-terminate-instances-asg-az.yaml

# Construct ARN
REGION=$(aws ec2 describe-availability-zones --output text --query
'AvailabilityZones[0].[RegionName]')
ACCOUNT_ID=$(aws sts get-caller-identity --output text --query 'Account')
DOCUMENT_ARN=arn:aws:ssm:${REGION}:${ACCOUNT_ID}:document/${SSM_DOCUMENT_NAME}
echo $DOCUMENT_ARN
```

Create FIS Experiment Template

Finally we have to create the FIS experiment template to call the SSM document. Copy the following JSON into a file called `fis-terminate-instances-asg-az.json`. You will need to replace the following:

- DOCUMENT_ARN - use the ARN from the previous step
- AZ_NAME - use the name of your target AZ, e.g. `us-east-1a` if you are working in `us-east-1`
- ASG_NAME - navigate to the EC2 console, select the ASG starting with `FisStackAsg`, then copy the full name of the ASG, e.g. `FisStackAsg-ASG46ED3070-1RAQ30VBKLWE1`
- SSM_ROLE_ARN - use the role ARN from the first step of this section. You can also find this by navigating to the IAM console, searching for `FisWorkshopSsmEc2DemoRole`, clicking on the role and copying the "Role ARN"
- FIS_WORKSHOP_ROLE_ARN - use the role ARN from the second step of this section. You can also find this by navigating to the IAM console, searching for `FisWorkshopServiceRole`, clicking on the role and copying the "Role ARN"

```
{
    "description": "Terminate All ASG Instances in AZ",
    "stopConditions": [
        {
            "source": "none"
        }
```

```
    ],
    "targets": {
    },
    "actions": {
        "terminateInstances": {
            "actionId": "aws:ssm:start-automation-execution",
            "description": "Terminate Instances in AZ",
            "parameters": {
                "documentArn": "DOCUMENT_ARN",
                "documentParameters": "{\"AvailabilityZone\": \"AZ_NAME\",
\"AutoscalingGroupName\": \"ASG_NAME\", \"AutomationAssumeRole\":
\"SSM_ROLE_ARN\"}",
                "duration": "PT3M"
            },
            "targets": {
            }
        }
    },
    "roleArn": "FIS_WORKSHOP_ROLE_ARN"
}
```

Once this is done, create the experiment template with this AWS CLI command:

```
aws fis create-experiment-template \
   --cli-input-json file://fis-terminate-instances-asg-az.json
```

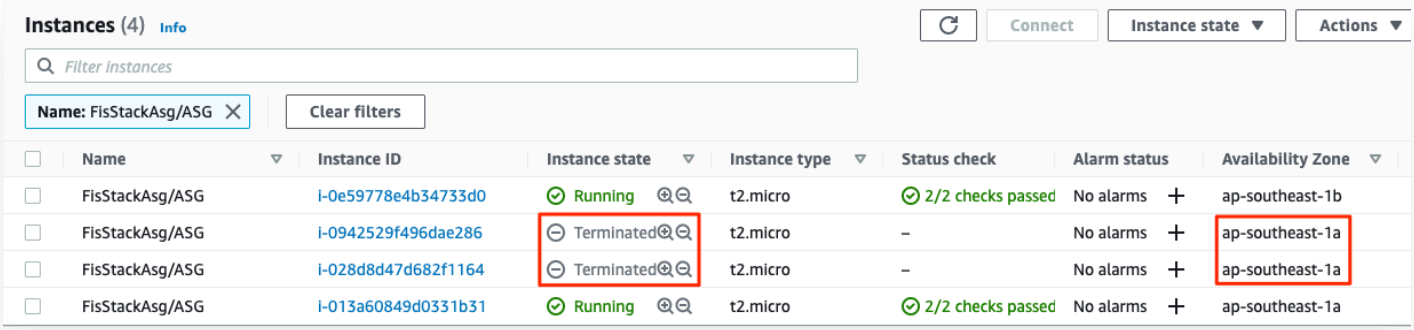
Note the experiment template ID as we will use this to start the experiment next.

Run FIS experiment using SSM automation

Using the experiment template ID from the previous step, run the following AWS CLI command to start the experiment:

```
aws fis start-experiment \
  --tags Name=DemoSsmAutomationDocument \
  --experiment-template-id TEMPLATE_ID
```

Let's get back to EC2 console and check what's happening to our EC2 instances in particular AZ. If the experiment runs successfully, all of our instances in particular AZ will be terminated, and spin back up after some time.



Troubleshooting

If you run into issues with your FIS experiment failing check the following:

- Experiment fails with “Unable to start SSM automation, not authorized to perform required action” - you probably didn’t update your FIS role to enable SSM AutomationExecution and allow PassRole. You can search the “Event history” in the CloudTrail console for “Event name” `StartAutomationExecution`. Note that events can take up to 15min to appear in CloudTrail.
- Experiment fails with “Unable to start SSM automation. A required parameter for the document is missing, or an undefined parameter was provided.” - make sure that you properly replaced all the document parameters. You can check this by editing the experiment template. This can also be caused by a role misconfiguration that prevents SSM from assuming the execution role. You can search the “Event history” in the CloudTrail console for “Event name” `StartAutomationExecution`. Note that events can take up to 15min to appear in CloudTrail.
- Experiment fails with “Automation execution completed with status: Failed.” - this can be caused by insufficient privileges on the role passed to SSM for execution. This can also happen if there are no instances found in the selected AZ. You can examine the history and output of SSM automation runs by navigating to the Systems Manager console and selecting “Automation” on the left. Then click on the automation run associated with your failed experiment and examine the output of the individual steps for more detail.
- Experiment succeeds but SSM automation status shows “Cancelled” steps. This can happen if you set the “Duration” in the FIS action to be shorter than the time it takes for the SSM document to finish. In this situation FIS will call a “Cancel” action on the SSM document. Ensure that you allow enough time in FIS for the SSM document to finish.

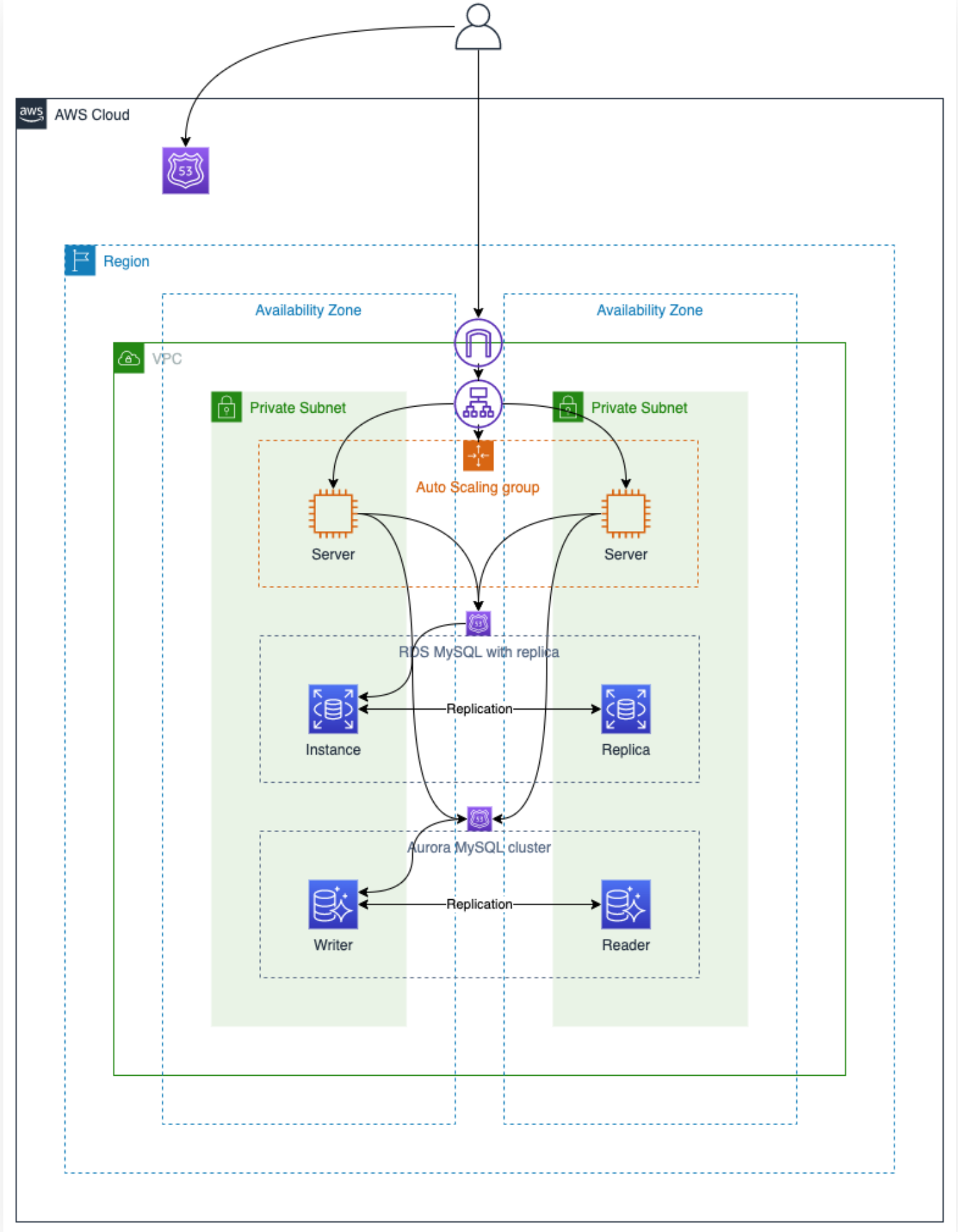
SSM ADDITIONAL RESOURCES

For additional SSM automation resources see:

- SSM Workshop
- SSM Chaos Documents
- SSM Documents AWS documentation
- SSM working with inputs and outputs

DATABASES

In this section we will cover working with databases. For this setup we are adding RDS MySQL and Aurora for MySQL to our test architecture:



Both RDS MySQL and Aurora for MySQL provide MySQL databases but they are different products. RDS MySQL is a managed service based on stock MySQL while Aurora is a custom built MySQL and PostgreSQL-compatible relational database with better performance and reliability.

Since these are different products they have slightly different failover patterns. They also use slightly different naming conventions:

- For RDS MySQL your dashboard will show “Instances” which may have “Replicas” attached for failover.
- For Aurora MySQL your dashboard will show “Clusters” with “Writers” and “Readers”.

For this workshop we are using a similar configuration that replicates data across two AZs for resilience.

RDS DB INSTANCE REBOOT

Experiment idea

In the previous section we ensured that we have a resilient front end of servers in an Auto Scaling group. Typically these servers would depend on a resilient database configuration. Let's validate this:

- **Given**: we have a managed database with a replica and automatic failover enabled
- **Hypothesis**: failure of a single database instance / replica may slow down a few requests but will not adversely affect our application

Experiment setup

Note

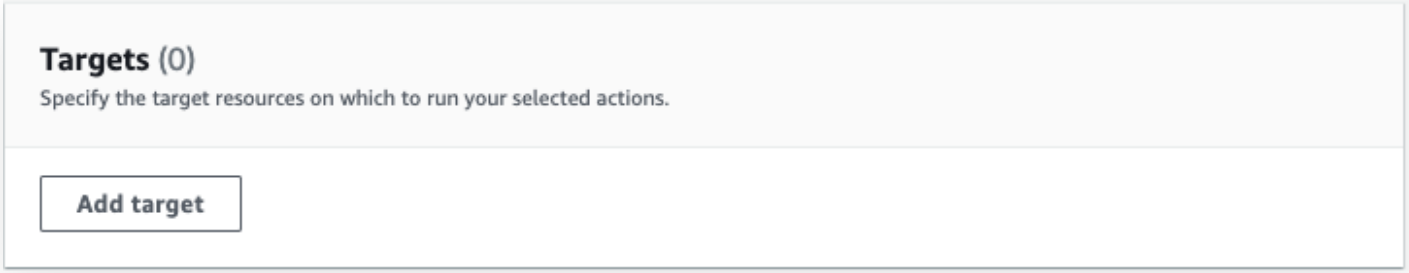
We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

General template setup

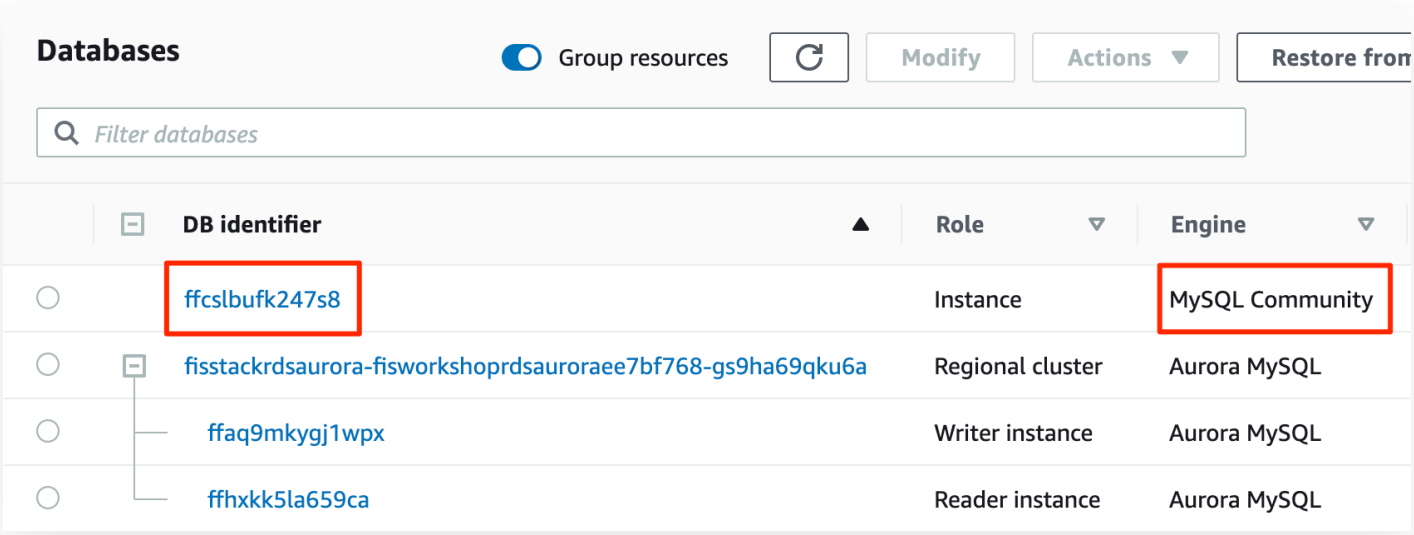
- Create a new experiment template
  - Add `Name` tag of `FisWorkshopRds1`
  - Add `Description` of `RebootRDSInstance`
  - Select `FisWorkshopServiceRole` as execution role

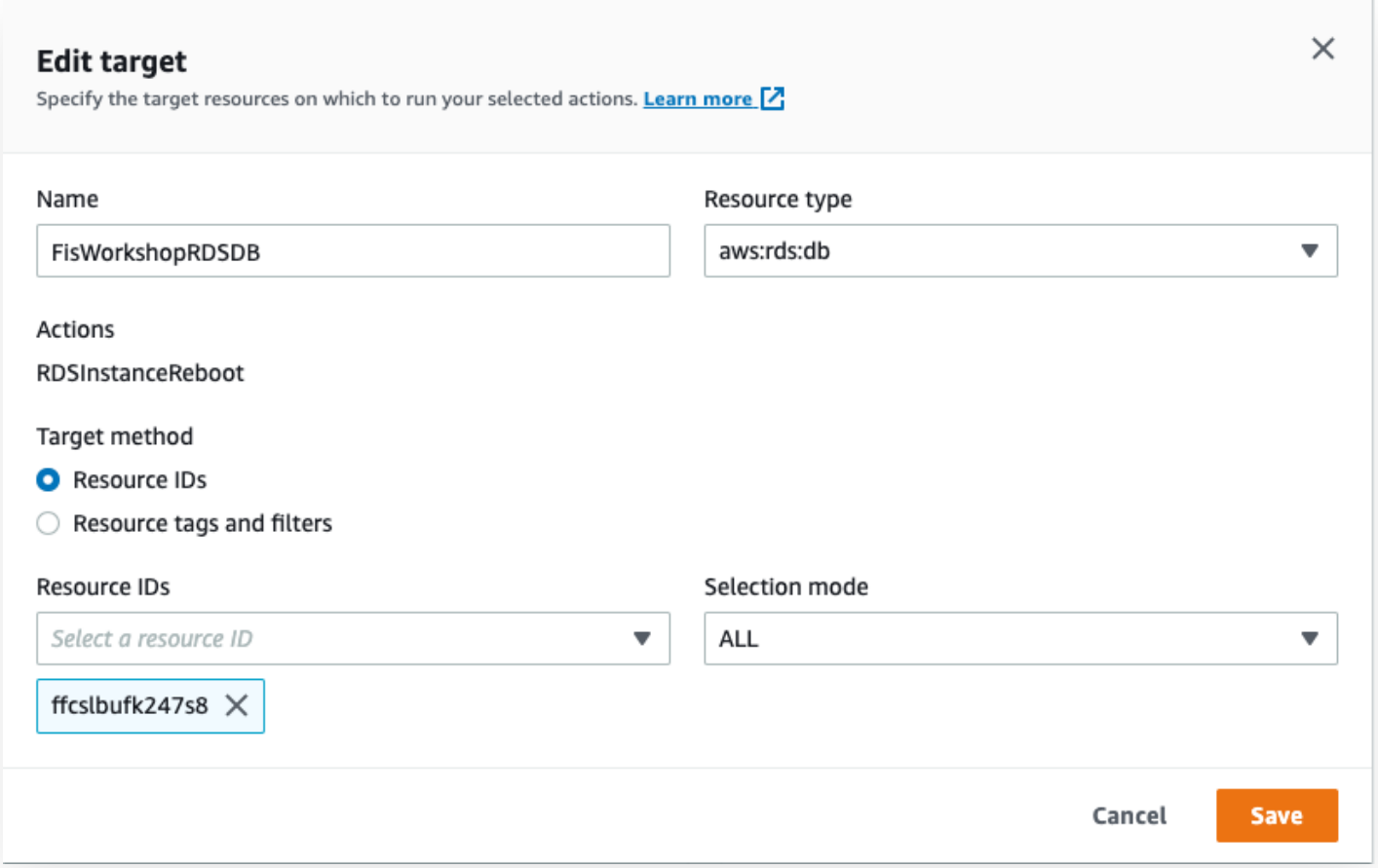
Target selection

Now we need to define targets. Scroll to the "Targets" section and select "Add Target"



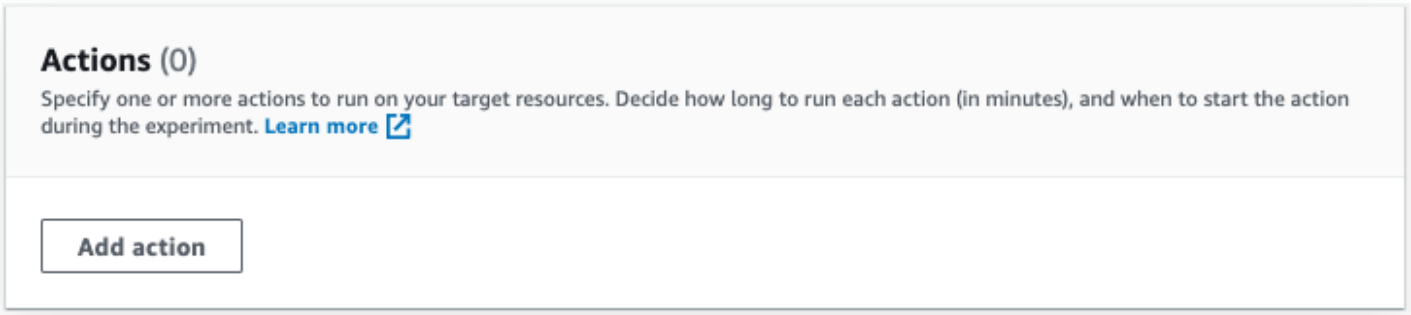
On the “Add target” popup, enter `FisWorkshopRDSDB` for name and select `aws:rds:db` for resource type. For “Target method” we will select resources based on the ID. Select the `Resource IDs` checkbox. Pick the target DB instance (the one with “MySQL Community” engine - check yours in RDS console), then pick `All` from “Selection mode”. Select “Save”.





Action definition

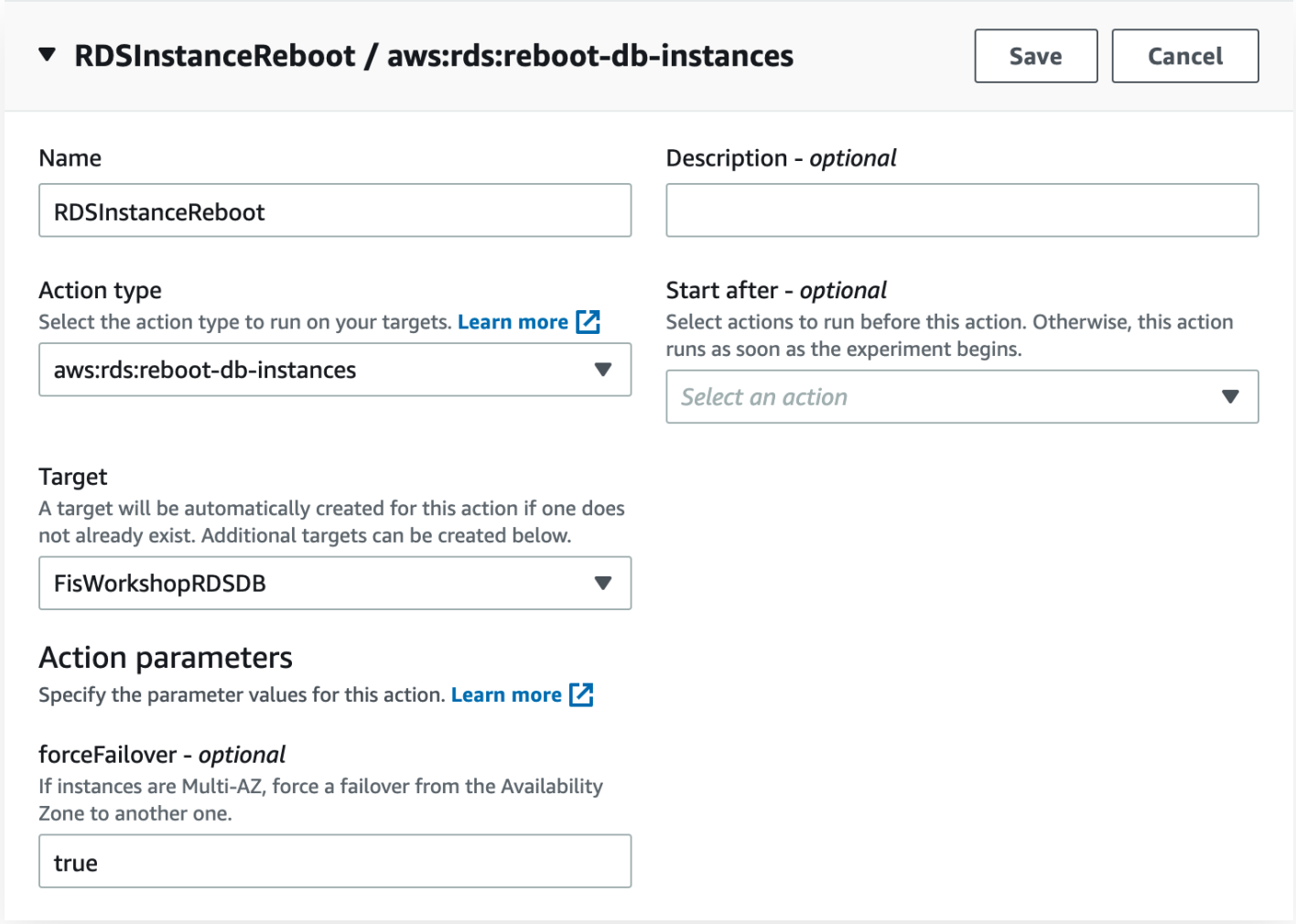
With targets defined we define the action to take. Scroll to the “Actions” section and select “Add Action”



For “Name” enter `RDSInstanceReboot` and you can skip the Description. For “Action type” select `aws:rds:reboot-db-instances`.

For this experiment we are using a Multi-AZ database and we want to force a failover to the standby instance to minimize outage time. To do this, set the `forceFailover` parameter to `true`.

Under “Target” select the `FisWorkshopRDSDB` target created above. Select “Save”.



Creating template without stop conditions

- Confirm that you wish to create the template without stop condition

Validation procedure

Before running the experiment we should consider how we will define success. How will we know that our failover was in fact non-impacting. For this workshop we have installed a python script that will read and write data to the database, conceptually like this but with some added safeguards:

```
import mysql.connector
mydb = mysql.connector.connect(...)
cursor = mydb.cursor()
while True:
    cursor.execute("insert into test (value) values (%d)" %
int(32768*random.random()))
    cursor.execute("select * from test order by id desc limit 10")
    for line in cursor:
        cursor.append("%-30s" % str(line))
```

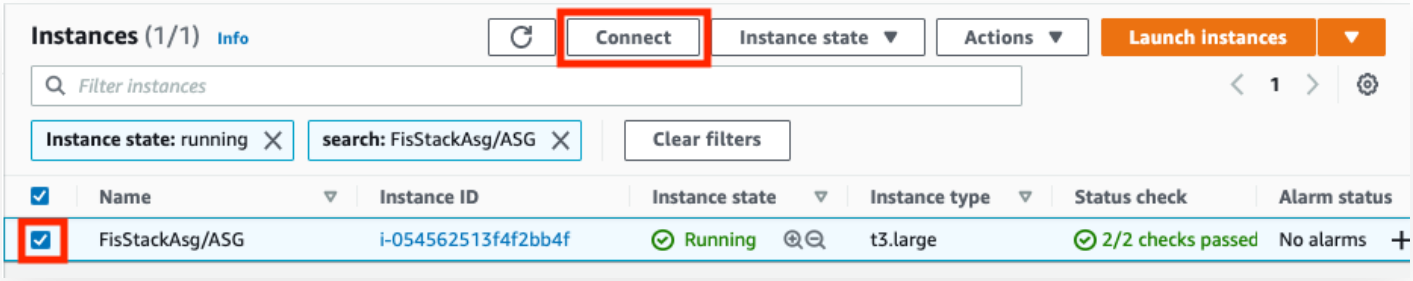
We would expect that this would keep writing output while the DB is available, stop while it's failing over and restart when the DB has successfully failed over.

Additionally because the DB connection does a DNS lookup our script will also print the IP address of the database it's currently connected to. A healthy output should look like this:

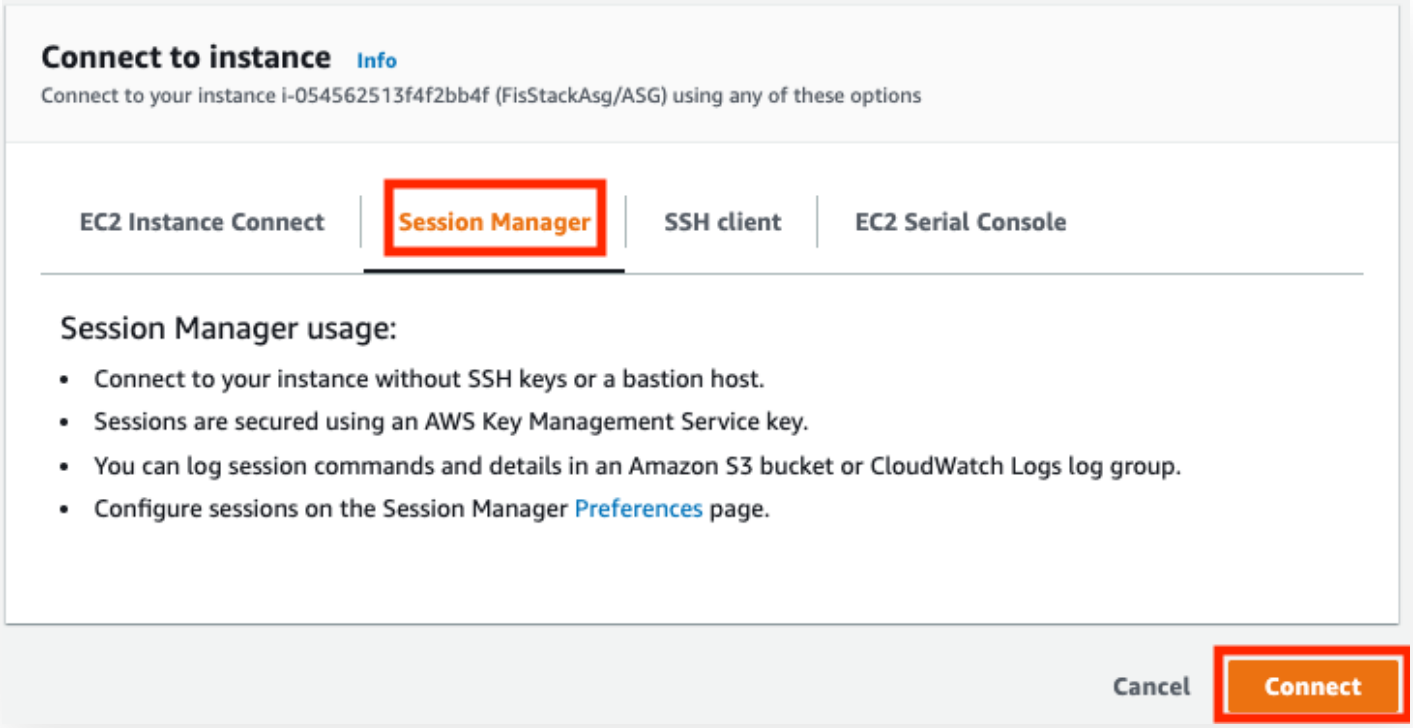
```
AURORA                          RDS
10.0.89.224                     10.0.95.247
(7711, 2282)                    (5419, 15189)
(7710, 5964)                    (5418, 15841)
(7709, 10634)                   (5417, 8071)
(7708, 4834)                    (5416, 21948)
(7707, 20291)                   (5415, 27256)
(7706, 9343)                    (5414, 8187)
(7705, 5496)                    (5413, 9359)
(7704, 30985)                   (5412, 6058)
(7703, 21808)                   (5411, 26174)
(7702, 20243)                   (5410, 21155)
```

Starting the validation procedure

Connect to one of the EC2 instances in your auto scaling group. In a new browser window - we need to be able to see this side-by-side with the FIS experiment later - navigate to your EC2 console and search for instances named `FisStackAsg/ASG`. Select one of the instances and click the "Connect" button:



On the next page select "Session Manager" and "Connect":



This will open a linux terminal session. In this session sudo to assume the `ec2-user` identity:

```
sudo su - ec2-user
```

If this is the first time you are doing this run the create_db.py script to ensure we can connect to the DB and we have created the required tables:

```
./create_db.py
```

If all went well you should see output similar to this:

```
AURORA                          RDS
10.0.89.224                     10.0.95.247
done
```

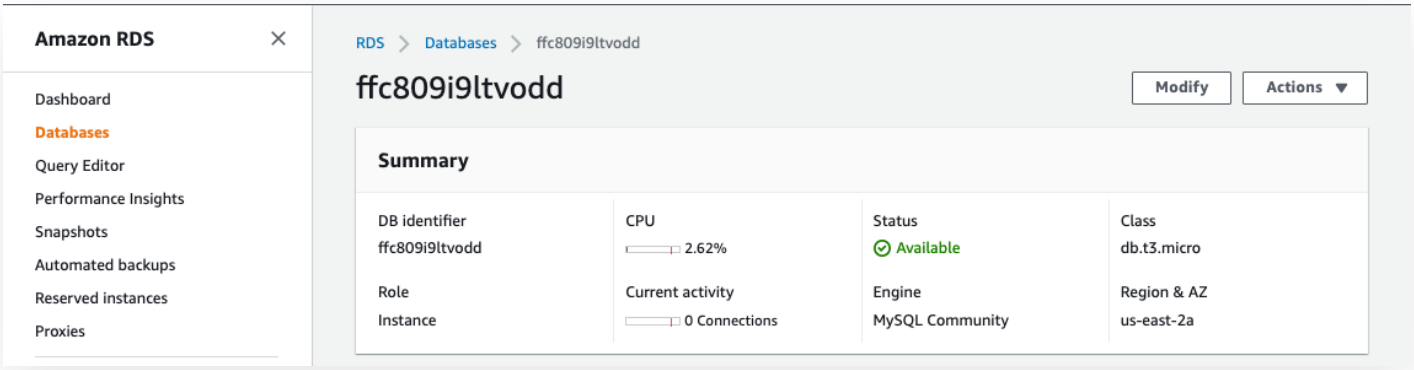
Now start the test script and leave it running:

```
./test_mysql_connector_curses.py
```

Run FIS experiment

Record current RDS state

Navigate to the RDS console, select “Databases” on the left menu, and select the “MySQL Community” instance. Note that the current instance state is “Available”:

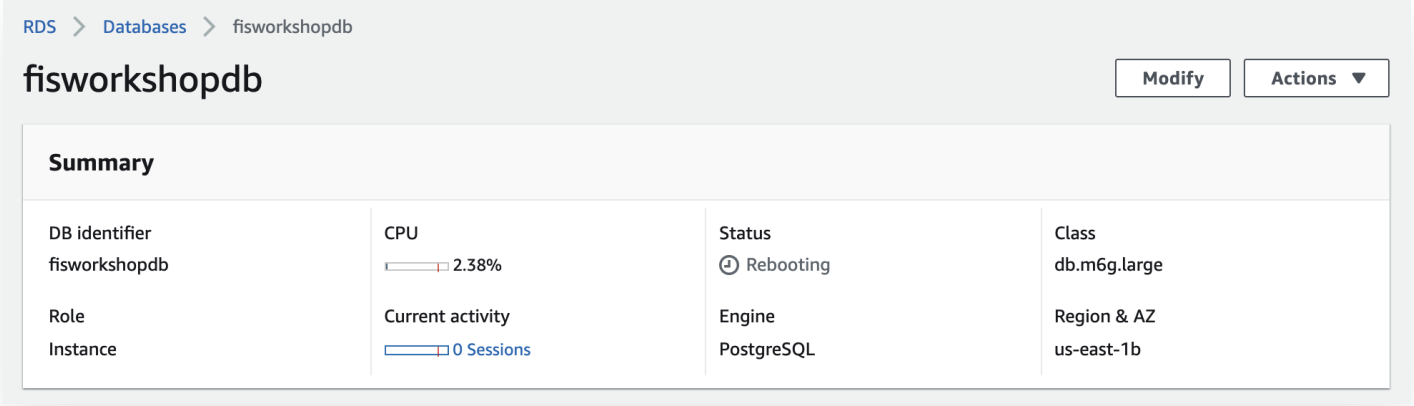


Start the experiment

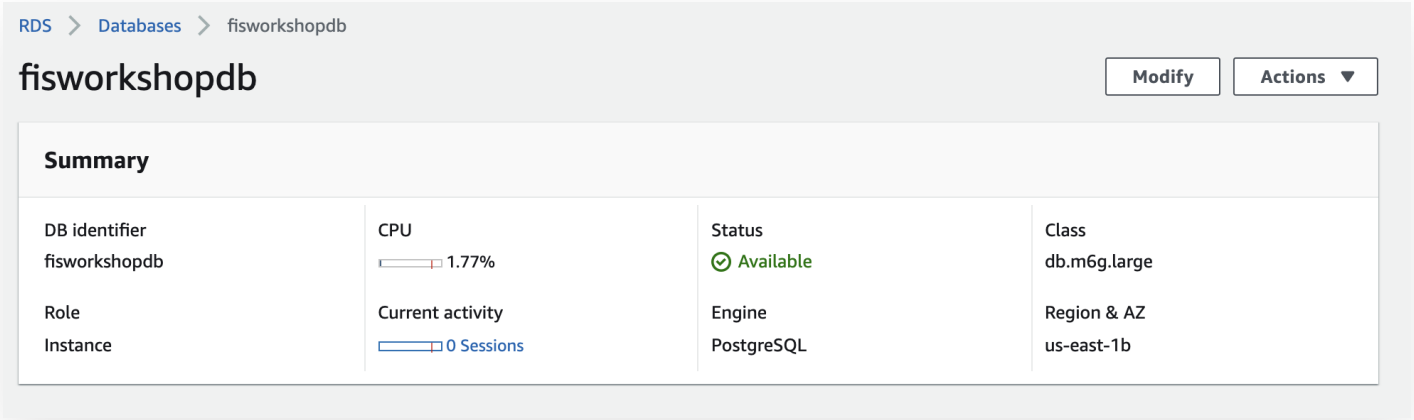
- Select the `FisWorkshopRds1` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopMysql1Run1`
- Confirm that you want to start an experiment
- Watch the output of your test script
- Check the state of your database in the RDS console

Review results

If all went “well” the status of the database in the RDS console should have changed from “Available” to “Rebooting”



and back to “Available”.



However, even though your database failed over successfully, your script should have locked up during the failover - no more updates to your data and it didn’t recover even after the DB successfully failed over. Discoveries like this are exactly why we are using Fault Injection Simulator!

Learning and Improving

What happened is that our script used a common MySQL database connector library that does not have a `read_timeout` setting. The database successfully failed over but the `INSERT` or `SELECT` statement that was in flight during the failover never timed out and locked our code into waiting forever.

Fortunately there is another common library that has very similar configuration and does implement `read_timeout`. For your convenience we have provided an updated script. CTRL-C out of the hung script and repeat the experiment but this time running

```
./test_pymysql_curses.py
```

This time you should see almost no interruption in your code's ability to interact with the database.

To end the session, hit CTRL+C to stop the script, and click Terminate button.

AURORA CLUSTER FAILOVER

Experiment idea

In the previous section we ensured that we have a resilient front end of servers in an Auto Scaling group. Typically these servers would depend on a resilient database configuration. Let's validate this:

- **Given**: we have a managed database with a replica and automatic failover enabled
- **Hypothesis**: failure of a single database instance / replica may slow down a few requests but will not adversely affect our application

Experiment setup

Note

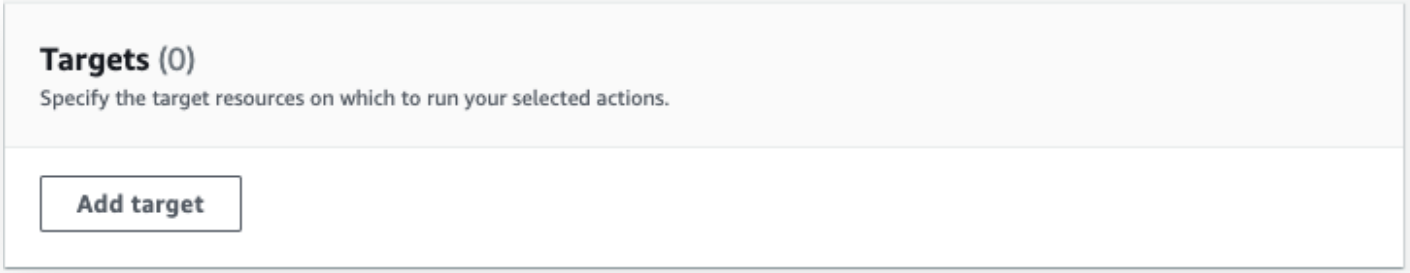
We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

General template setup

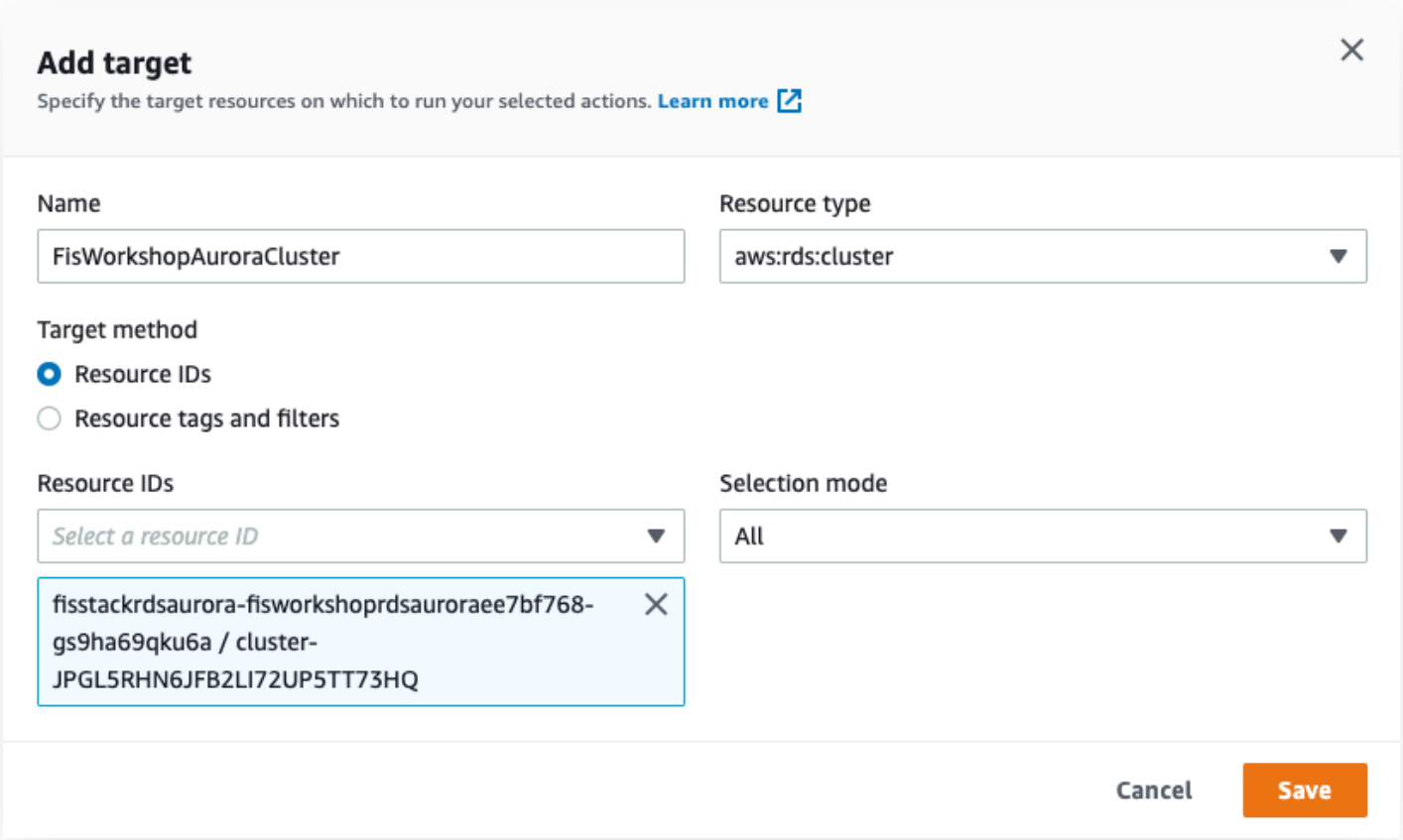
- Create a new experiment template
  - Add `Name` tag of `FisWorkshopAurora1`
  - Add `Description` of `FailoverAuroraCluster`
  - Select `FisWorkshopServiceRole` as execution role

Target selection

Now we need to define targets. Scroll to the "Targets" section and select "Add Target"

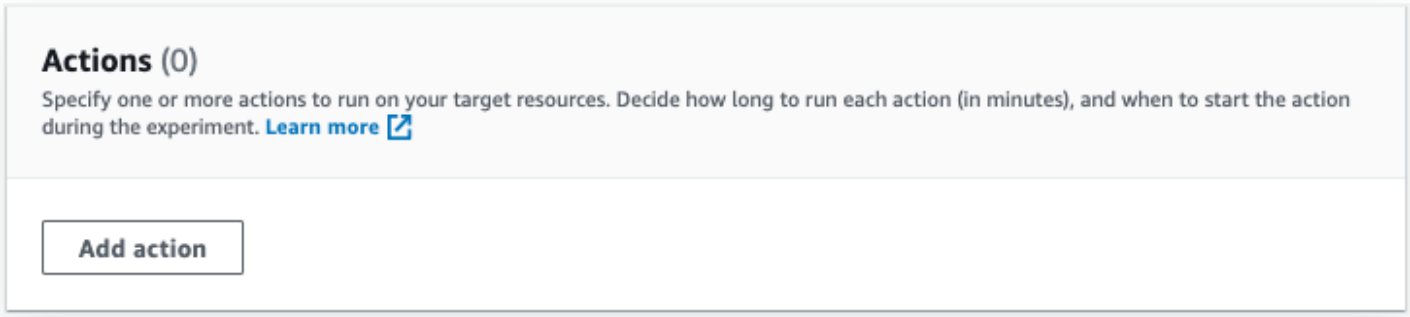


On the “Add target” popup enter `FisWorkshopAuroraCluster` for name and select `aws:rds:cluster`. For “Target method” we will select resources based on the ID. Select the `Resource IDs` checkbox. Pick the target cluster, then pick `All` from “Selection mode”. Select “Save”.



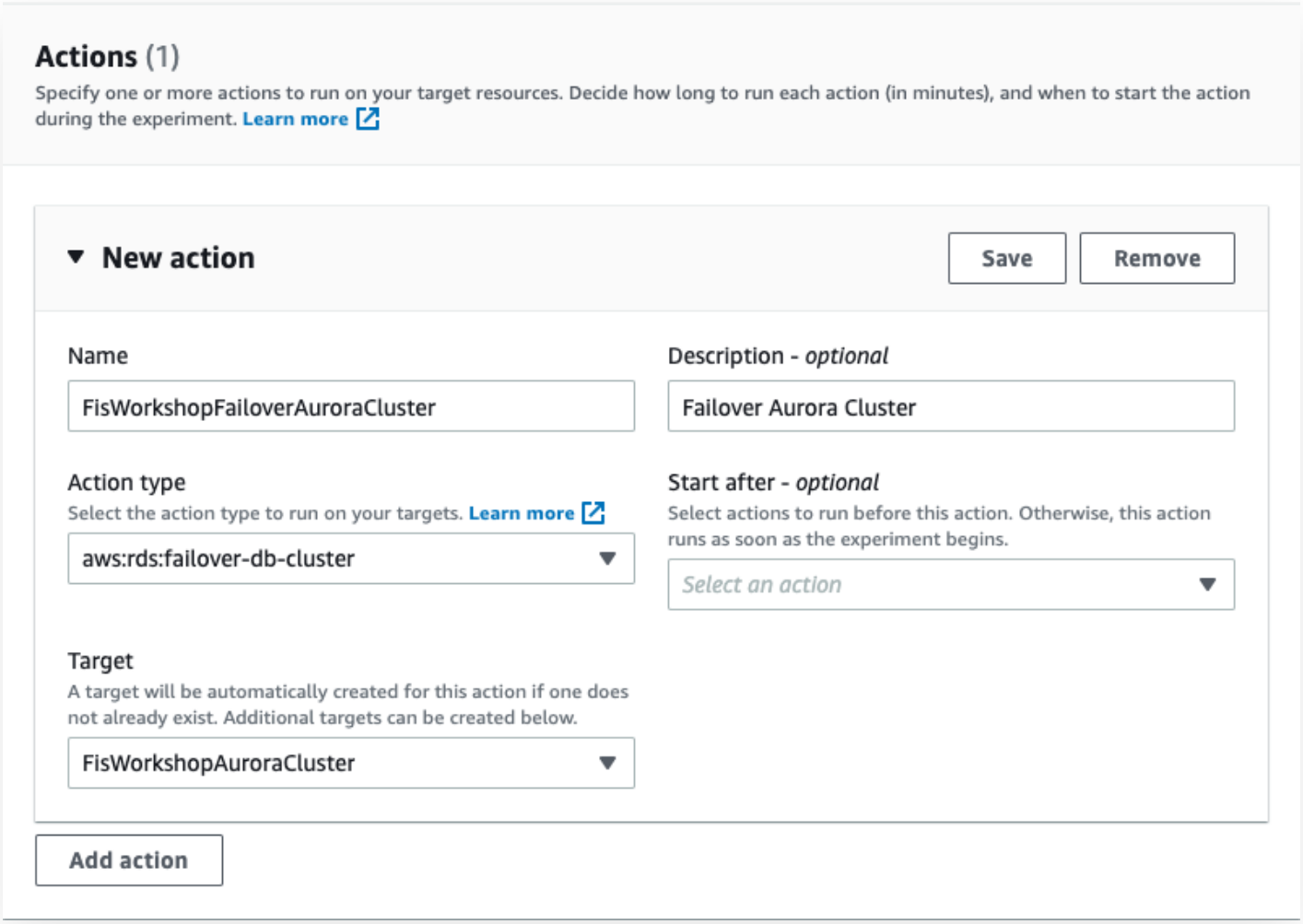
Action definition

With targets defined we define the action to take. Scroll to the “Actions” section and select “Add Action”



For “Name” enter `FisWorkshopFailoverAuroraCluster` and add a “Description” like `Failover Aurora Cluster`. For “Action type” select `aws:rds:failover-db-cluster`.

Under “Target” select the `FisWorkshopAuroraCluster` target created above. Select “Save”.



Creating template without stop conditions

- Confirm that you wish to create the template without stop condition

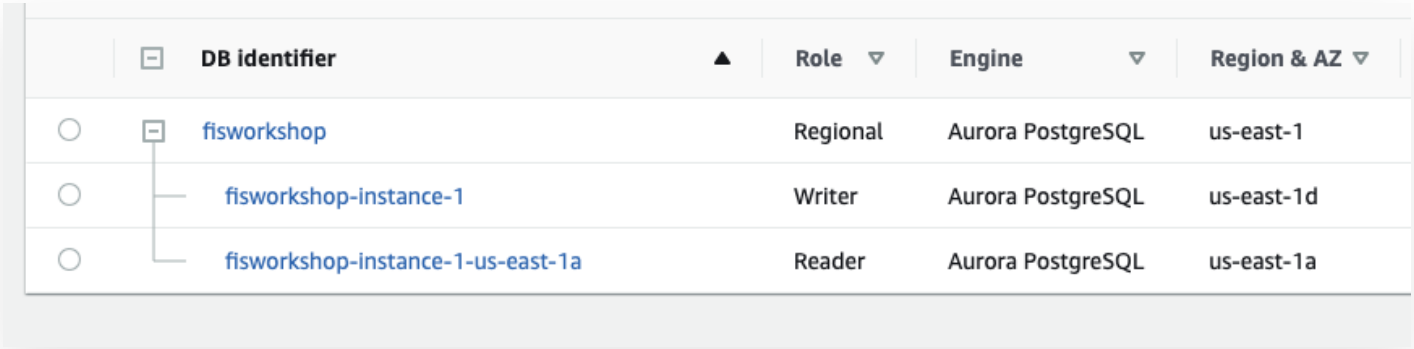
Validation procedure

› Identical to RDS MySQL procedure - click to expand

Run FIS experiment

Record current Aurora state

Navigate to the RDS console, select “Databases” on the left menu, and search for “fisworkshop”. Take a screenshot or write down the “Reader” and “Writer” AZ information, e.g.:



DB identifier	Role	Engine	Region & AZ
fisworkshop	Regional	Aurora PostgreSQL	us-east-1
fisworkshop-instance-1	Writer	Aurora PostgreSQL	us-east-1d
fisworkshop-instance-1-us-east-1a	Reader	Aurora PostgreSQL	us-east-1a

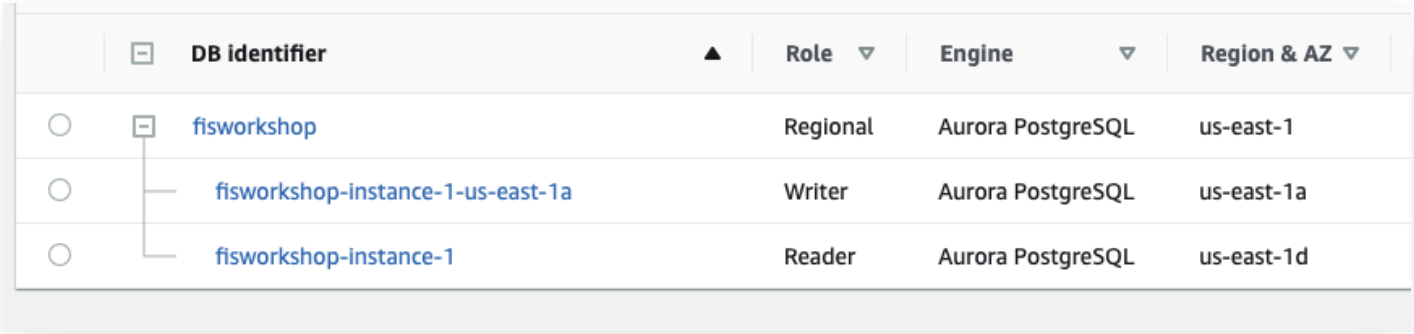
Start the experiment

- Select the `FisWorkshopAurora1` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopAurora1Run1`
- Confirm that you want to start an experiment
- Watch the output of your test script

Review results

Verify that the experiment worked. If you are not already on the pane viewing your experiment, navigate to the FIS console, select “Experiments”, and select the experiment ID for the experiment you just started. This should show success.

Verify that the failover actually happened. Navigate to the RDS console again and about a minute after you started the experiment you’ll see the “Reader” and “Writer” instances flipped to the other AZ:



DB identifier	Role	Engine	Region & AZ
fisworkshop	Regional	Aurora PostgreSQL	us-east-1
fisworkshop-instance-1-us-east-1a	Writer	Aurora PostgreSQL	us-east-1a
fisworkshop-instance-1	Reader	Aurora PostgreSQL	us-east-1d

If all went well, the “Reader” and “Writer” instances should have traded places.

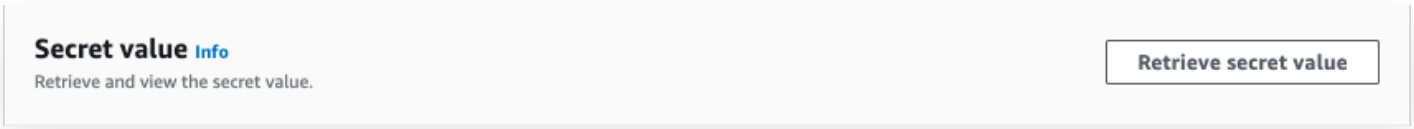
If you were watching the output of your test script carefully you might also have noticed that for a short period of time DNS returns no value for Aurora. To address this our code already contains an additional try/except block for DB reconnection.

Learning and improving

As this was essentially the same as the previous **RDS DB Instance Reboot** section there are not new learnings here.

However, you may want to experiment further built-in Aurora fault injection queries.

To access your Aurora database, you can extract the connection information from the AWS Secrets Manager console by selecting the `FisAuroraSecret` and selecting “Retrieve secret value”:



Using the information you can open another terminal, e.g. from the same instance you were using for testing, and connect to your Aurora database with the retrieved secret values:

```
mysql -h HOST -u USERNAME -p DBNAME
```

you can then run fault injection queries as further explained in this blog post and observe the effect on the test script, e.g.:

```
ALTER SYSTEM CRASH NODE;
```

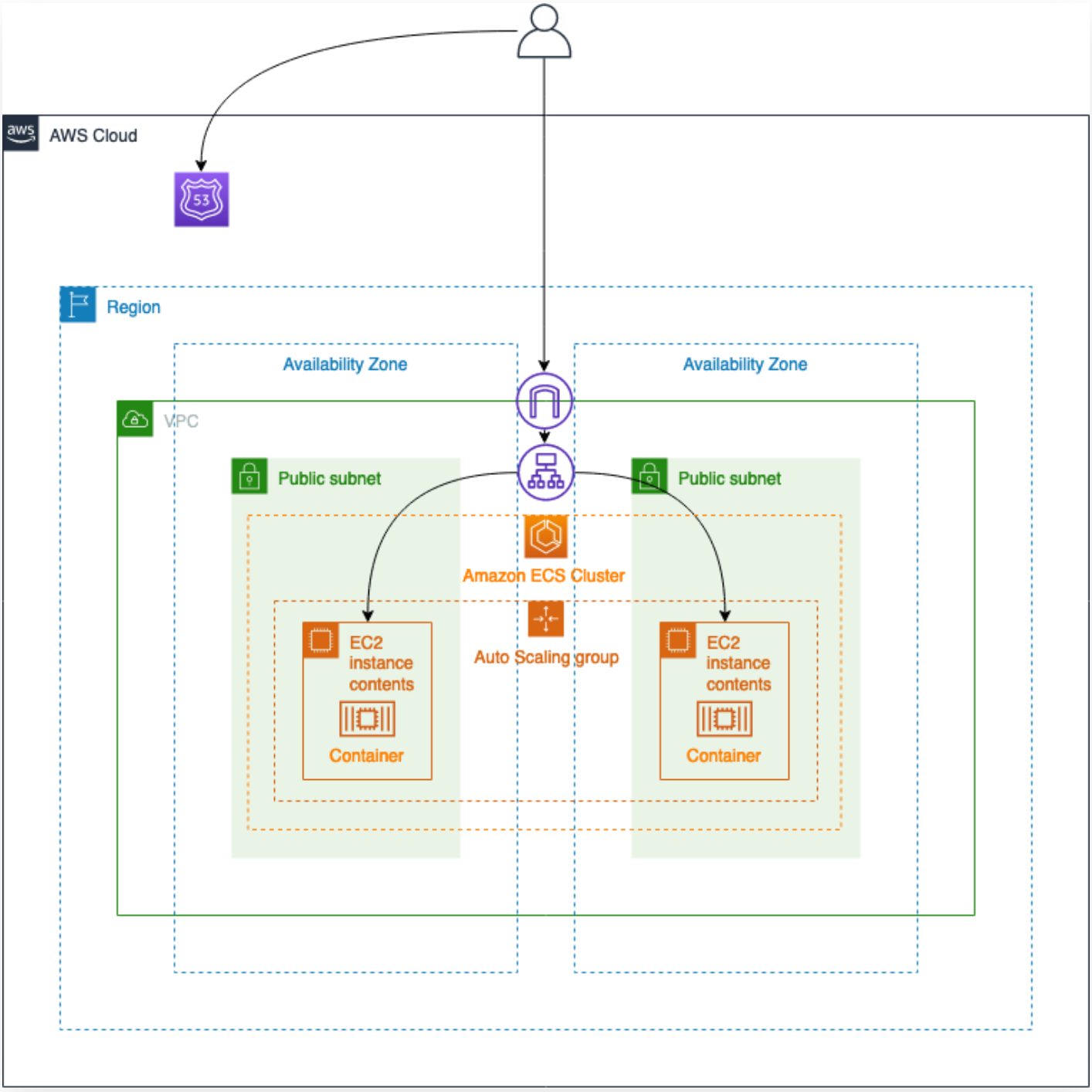
CONTAINERS

In addition to the ability to run containers on virtual machines, AWS provides managed services to run containers at scale. This section covers fault injection experiments on:

- Amazon ECS
- Amazon EKS

AMAZON ECS

In this section we will cover working with containers running on Amazon Elastic Container Service (Amazon ECS). For this setup we'll be using the following test architecture:



Amazon ECS is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. It deeply integrates with the rest of the AWS platform to provide a secure and easy-to-use solution for running container workloads in the cloud and now on your infrastructure with Amazon ECS Anywhere.

HYPOTHESIS & EXPERIMENT

Experiment idea

In this section we want to ensure that our containerized application running on Amazon ECS is designed in a fault tolerant way, so that even if an instance in the cluster fails our application is still available. Let's validate this:

- **Given**: we have a containerized application running on Amazon ECS exposing a web page.
- **Hypothesis**: failure of a single container instance will not adversely affect our application. The web page will continue to be available.

Experiment setup

Note

We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

General template setup

Note

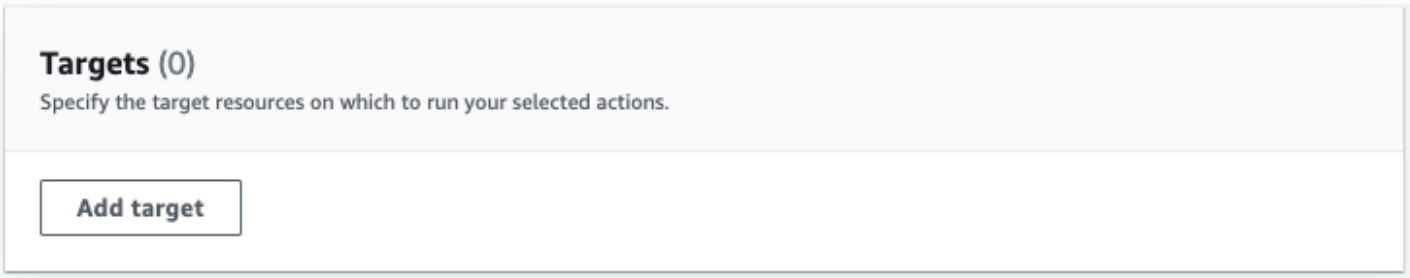
We are assuming that you have already set up an IAM role for for this workshop. If you haven't, see the **Create FIS Service Role** section.

Create a new experiment template:

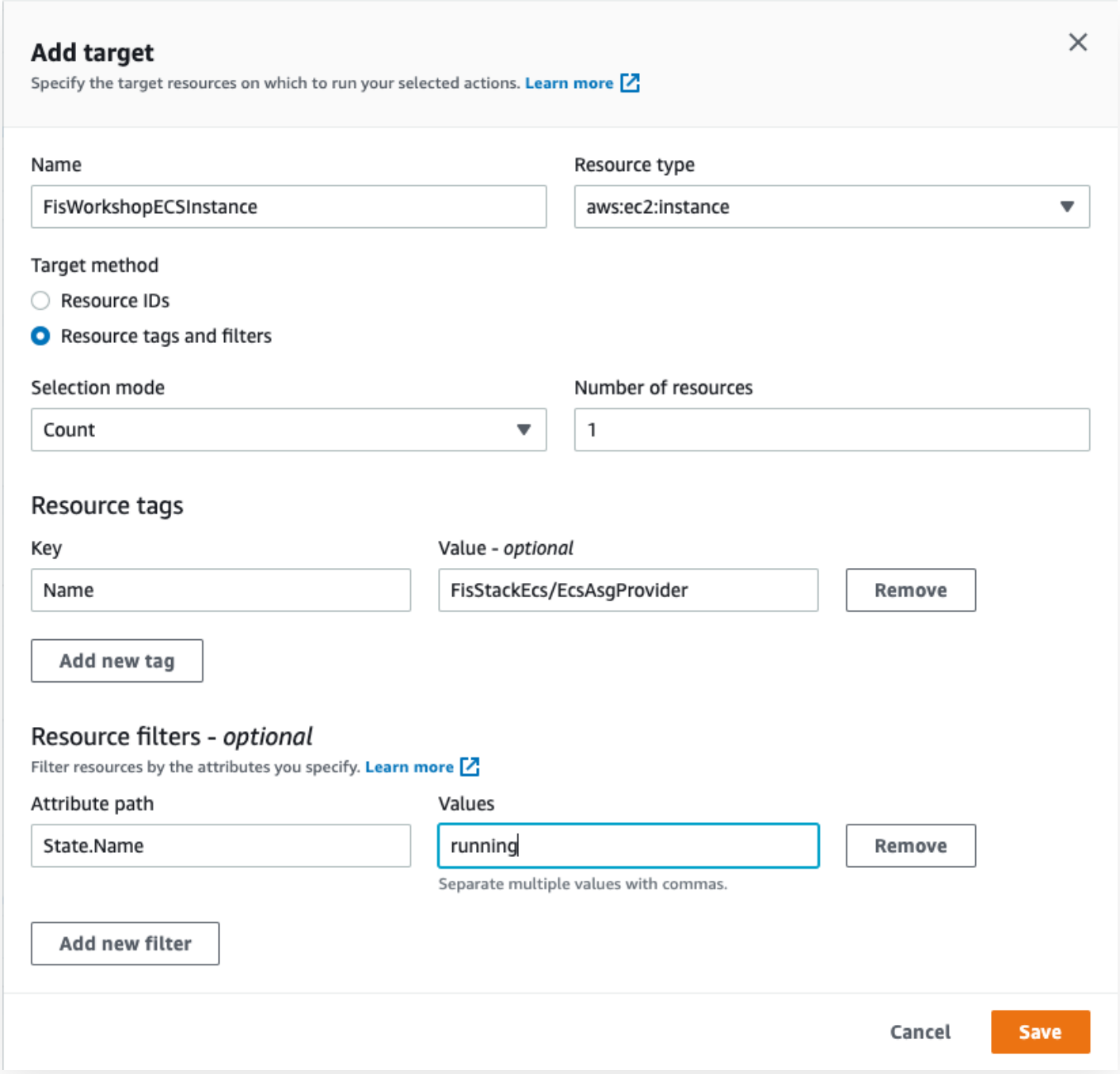
- add `Name` tag of `FisWorkshopECS`
- add `Description` of `Terminate ECS Cluster Instance`
- select `FisWorkshopServiceRole` as execution role

Target selection

Now we need to define targets. Scroll to the “Targets” section and select “Add Target”

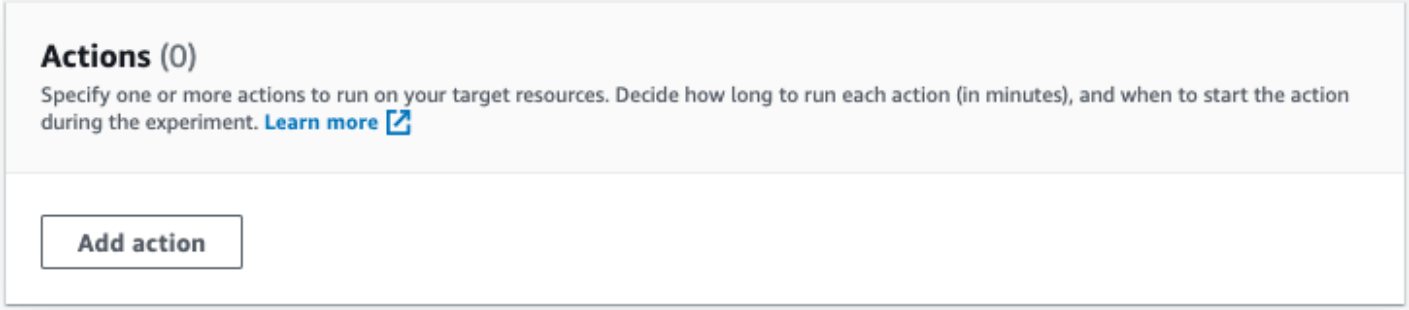


On the “Add target” popup enter `FisWorkshopECSInstance` for name and select `aws:ec2:instance` for resource type. For “Target method” we will dynamically select resources based on an associated tag. Select the `Resource tags and filters` checkbox. Pick `Count` from “Selection mode” and enter `1`. Under “Resource tags” enter `Name` in the “Key” field and `FisStackEcs/EcsAsgProvider` for “Value”. Under filters enter `State.Name` in the “Attribute path” field and `running` under “Values”. Select “Save”.



Action definition

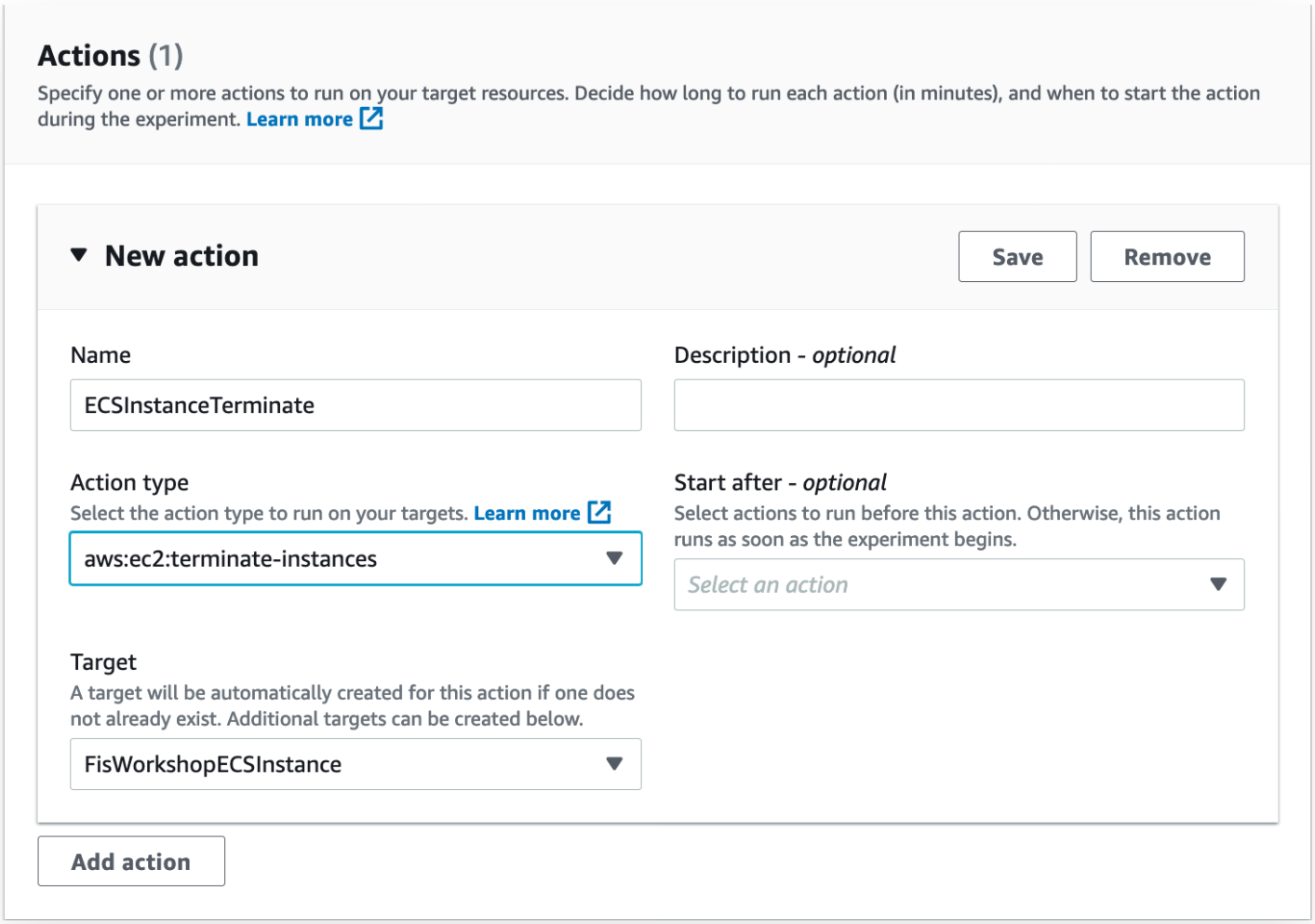
With targets defined we define the action to take. Scroll to the “Actions” section" and select “Add Action”



For “Name” enter `ECSInstanceTerminate` and you can skip the Description. For “Action type” select `aws:ec2:terminate-instances`.

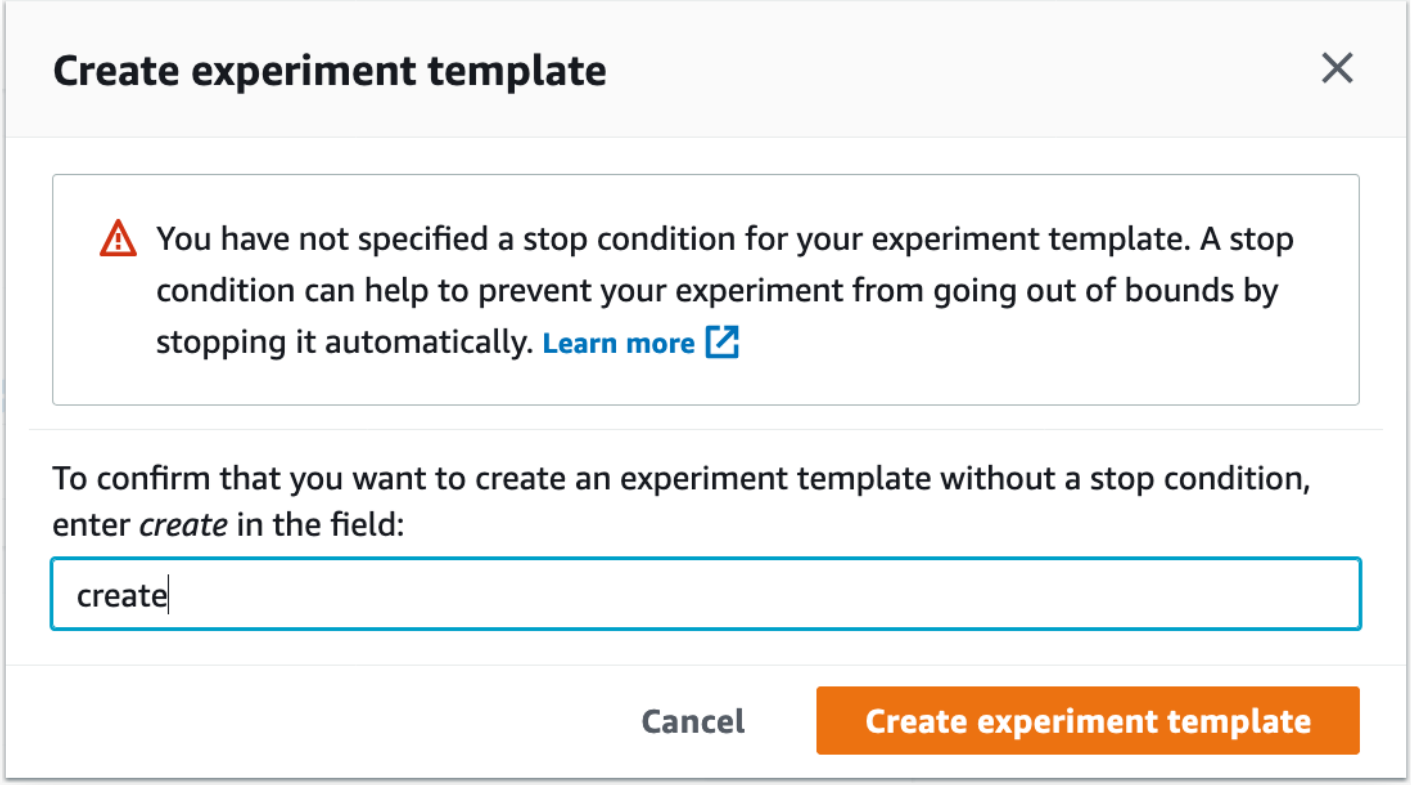
We will leave the “Start after” section blank since the instances we are terminating are part of an auto scaling group and we can let the auto scaling group create new instances to replace the terminated ones.

Under “Target” select the `FisWorkshopECSInstance` target created above. Select “Save”.



Creating template without stop conditions

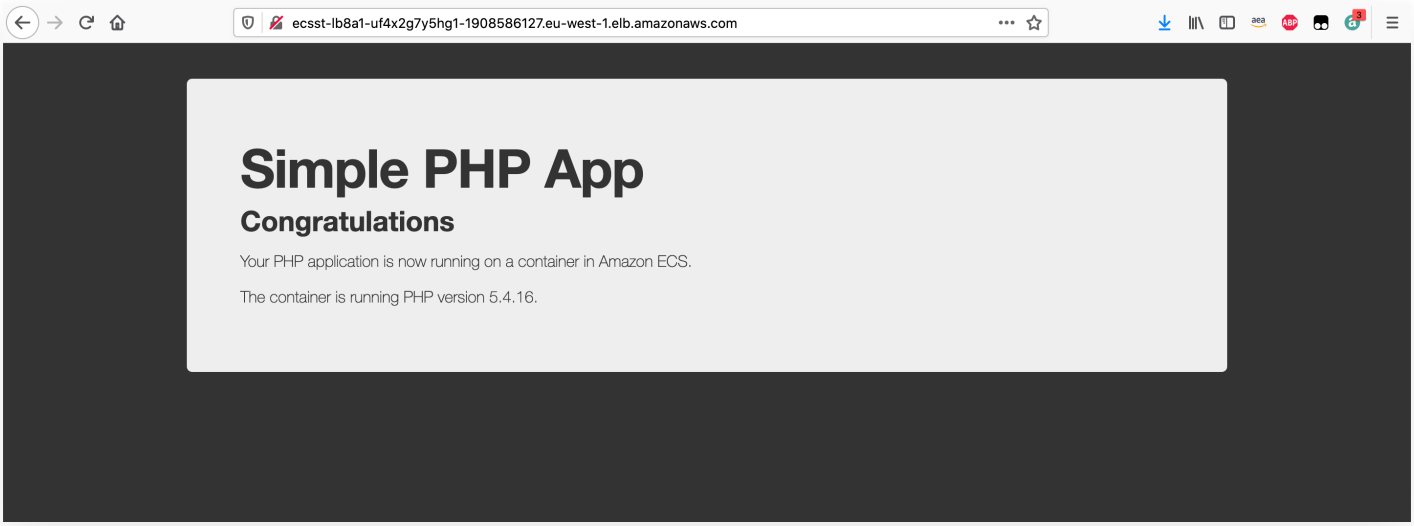
Confirm that you wish to create the template without stop condition.



Validation procedure

Before running the experiment we should consider how we will define success. Let's check the webpage we are hosting. To find the URL of the webpage navigate to the CloudFormation console, select the `FisStackEcs` stack, Select "Outputs", and copy the value of "FisEcsUrl".

Open the URL in a new tab to validate that our website is in fact up and running:



How will we know that our instance failure was in fact non-impacting? For this workshop we'll be using a simple Bash script that continuously polls our application.

Starting the validation procedure

In your local terminal, run the following script. For your convenience we are automating the query for the load balancer URL but you could also paste the URL you've found above:

```
# Query URL for convenience
ECS_URL=$( aws cloudformation describe-stacks --stack-name FisStackEcs --query
"Stacks[*].Outputs[?OutputKey=='FisEcsUrl'].OutputValue" --output text )

# Busy loop queries. CTRL-C to end loop
while true; do
  curl -sLo /dev/null -w 'Code %{response_code} Duration %{time_total} \n'
${ECS_URL}
done
```

We would expect that all requests will return a HTTP 200 OK code with some variability in the request duration, meaning the application is still responding successfully. Healthy output should look like this:

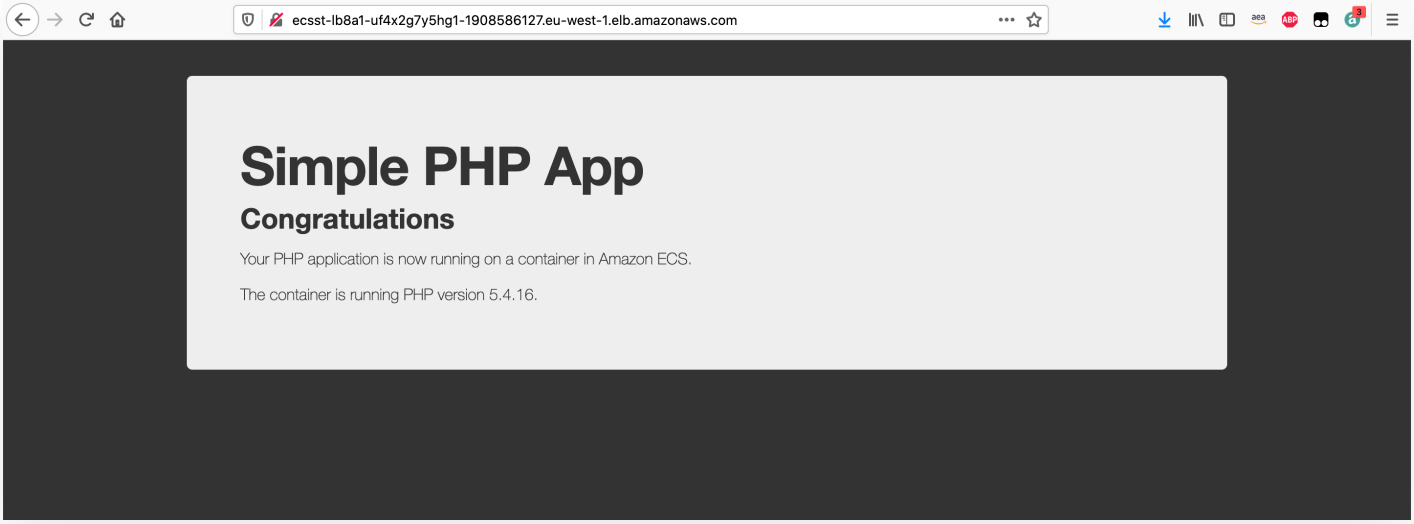
```
Code 200 Duration 0.140314
Code 200 Duration 0.086206
Code 200 Duration 0.085946
Code 200 Duration 0.084102
Code 200 Duration 0.085972
```

Leave the script running while we run the FIS experiment next.

Run FIS experiment

Record current application state

In a new browser window navigate to the load balancer URL you copied earlier, this is your application endpoint. Notice that the application is currently running:

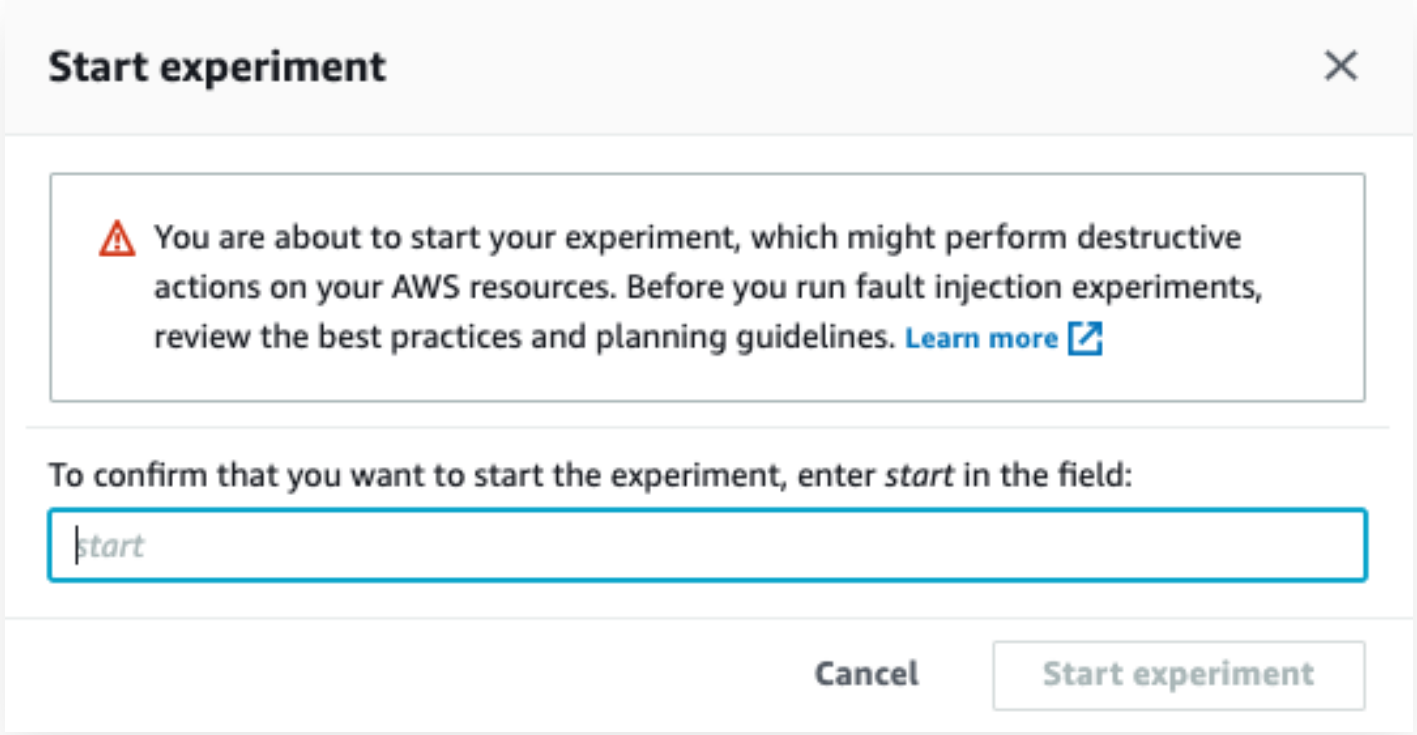


You can also verify the HTTP return code using this command, replacing `REPLACE_WITH_ECS_SERVICE_ALB_URL` with the load balancer DNS name you copied earlier:

```
curl -IL <REPLACE_WITH_ECS_SERVICE_ALB_URL> | grep "^HTTP\/"
```

Start the experiment

- Select the `FisWorkshopECS` experiment template you created above
- Select **Start experiment** from the **Action** drop-down menu
- Add a `Name` tag of `FisWorkshopECSRun1`
- Confirm that you want to start an experiment


Start experiment
You are about to start your experiment, which might perform destructive actions on your AWS resources. Before you run fault injection experiments, review the best practices and planning guidelines. Learn more
To confirm that you want to start the experiment, enter start in the field:
start
Cancel
Start experiment

OBSERVE THE SYSTEM

Review results

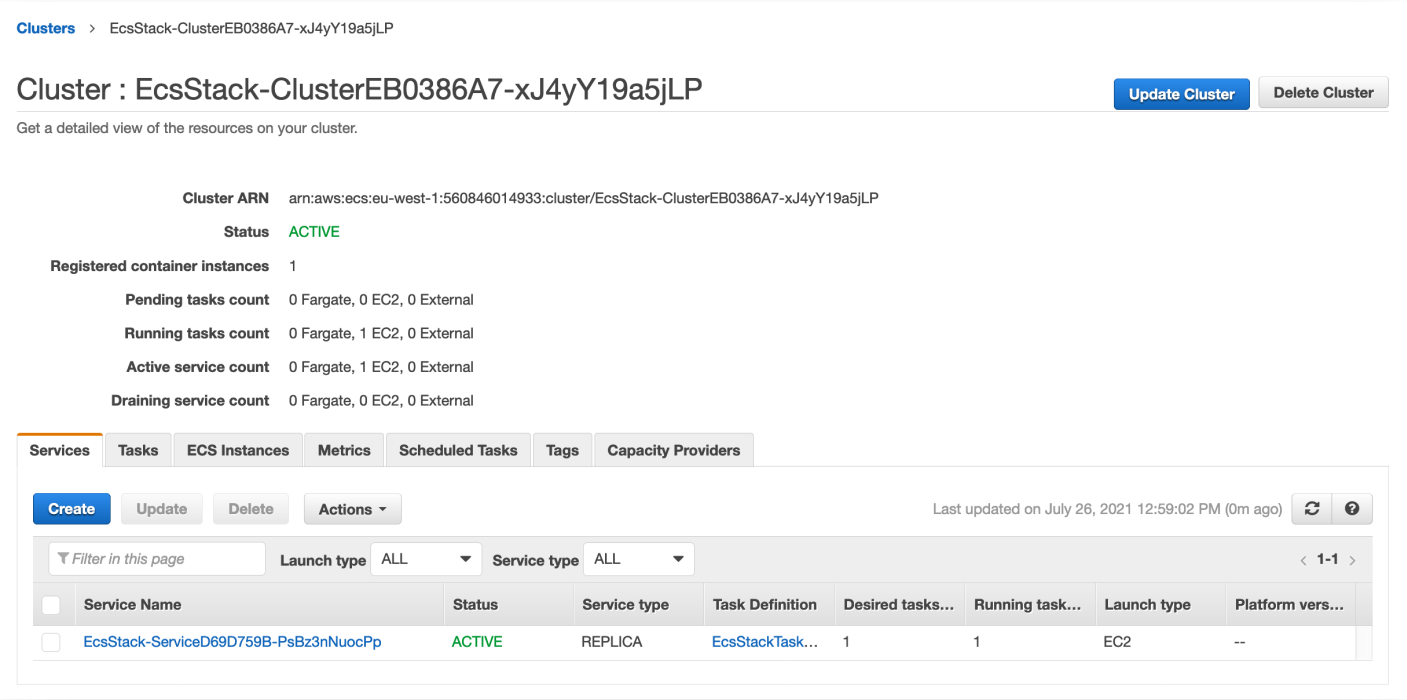
Let's take a look at the output in the terminal window where your Bash script is running:

```
Code 200 Duration 0.137204
Code 200 Duration 0.080911
Code 200 Duration 0.081539
Code 200 Duration 0.077265
Code 200 Duration 0.085331
Code 200 Duration 0.081634

...

Code 503 Duration 0.083001
Code 503 Duration 0.088983
Code 502 Duration 0.085972
Code 502 Duration 0.086619
Code 502 Duration 0.086554
Code 503 Duration 0.083428
Code 502 Duration 0.084929

...

Code 200 Duration 0.082434
Code 200 Duration 0.081427
Code 200 Duration 0.087983
Code 200 Duration 0.081950
Code 200 Duration 0.082790
```

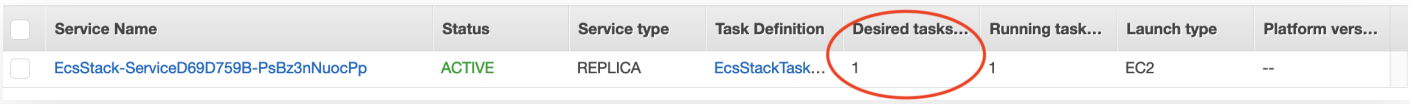
You'll notice that as not all the requests were successful. As the FIS experiment starts you should see some HTTP 502 "Bad Gateway" and HTTP 503 "Service Unavailable" return codes. This means our application was not available for a period of time. This is not what we were expecting, so let's dive a bit deeper to find out why it happened.

Check number of containers

In a new browser window navigate to the *Clusters* section in the ECS console and search for the cluster named `FisStackEcs-Cluster...`, e.g. `FisStack-ClusterEB0386A7-xJ4yY19a5jLP`. Click on the cluster name and look at the ECS services running on this cluster:

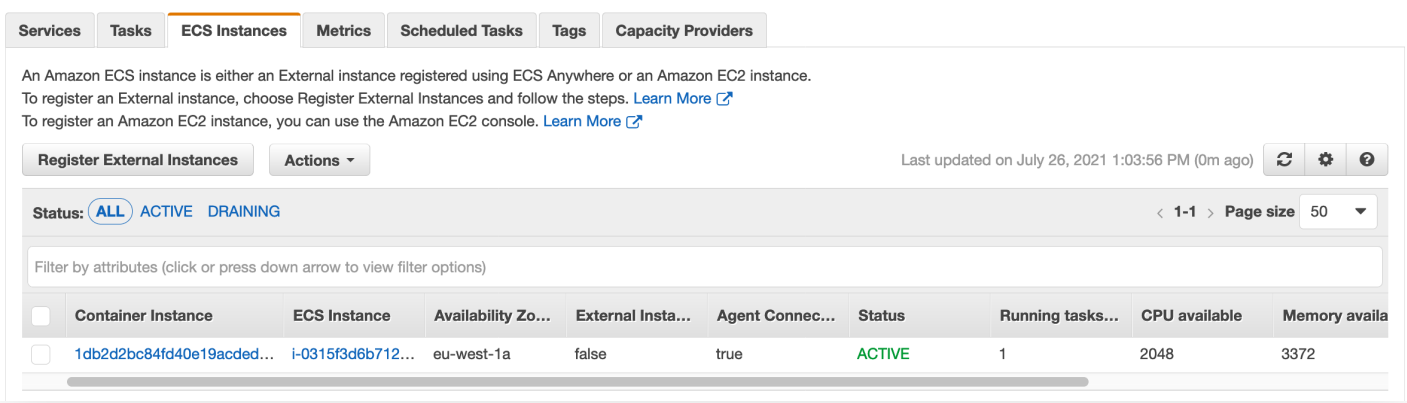


You'll notice that the service named `FisStackEcs-SampleAppService...`, e.g. `FisStackEcs-SampleAppServiceD69D759B-PsBz3nNuocPp` - i.e. our application - only has **one** desired task, meaning that only one copy of our containerized application will be running at any time.



Check number of instances

Now click on the "ECS Instances" tab. You'll see here that there's only one instance registered with our cluster.



Observations

This configuration is not optimal:

- A cluster with a single instance means that if that instance fails, all the containers running on that instance will also be killed. This is what happened during our experiment and the reason why we observed some HTTP 503 "Service Unavailable" return codes. We should change this so that our cluster has more than one instance across multiple Availability Zones (AZs).
- Having an ECS Service with **one** desired task also means that if that task fails, there aren't any other tasks to continue serving requests. We can modify this by adjusting the desired task capacity to `2` (or any number greater than `1`).

Now that we have identified some issues with our current setup, let's move to the next section to fix them.

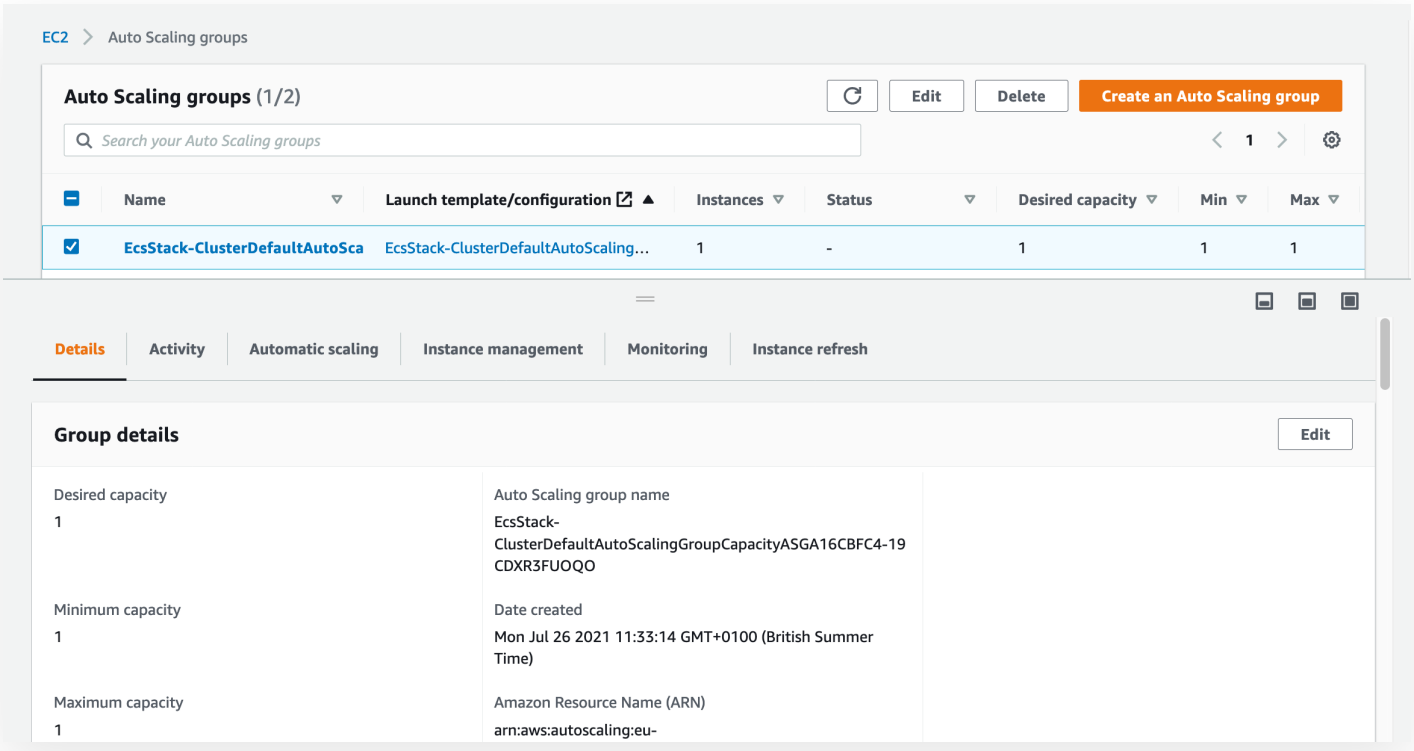
IMPROVE & REPEAT

Learning and Improving

In the previous section we have identified some issues with our current setup: our ECS cluster only had **one** instance and our application's ECS Service desired capacity was set to `1`. Now, let's improve our infrastructure setup.

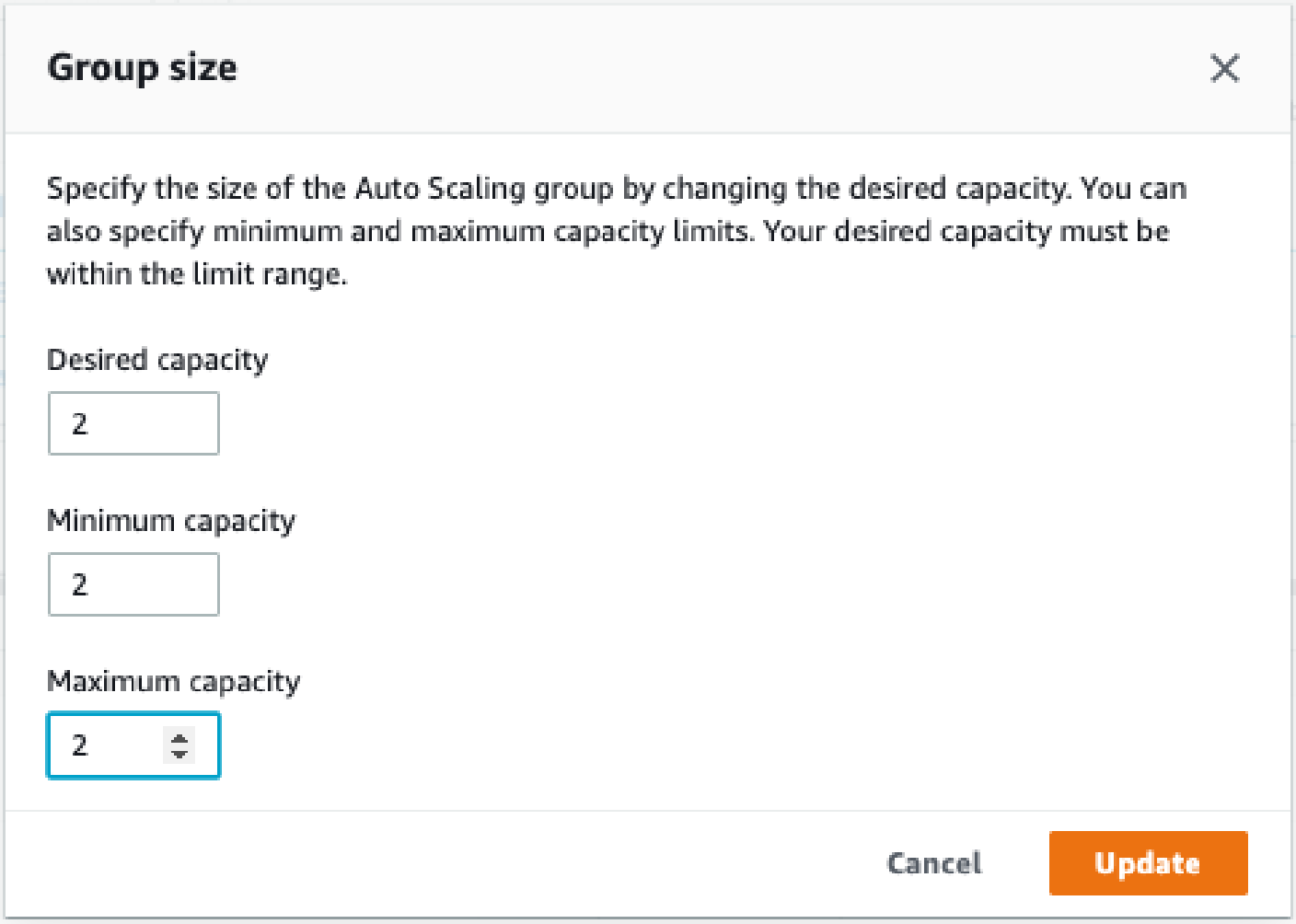
Increase the number of instances

In our ECS configuration we have chosen to use EC2 with an auto scaling group as our capacity provider. To adjust desired instance capacity open a browser window and navigate to the *Auto Scaling Groups* section in the EC2 console and search for an auto scaling group named `FisStackEcs-EcsAsgProvider...`, e.g. `FisStackEcs-EcsAsgProviderASG51CCF8BD-4LO6D3044727`. Select the check box next to our Auto Scaling group. A split pane opens up in the bottom part of the Auto Scaling groups page, showing information about the group that's selected.



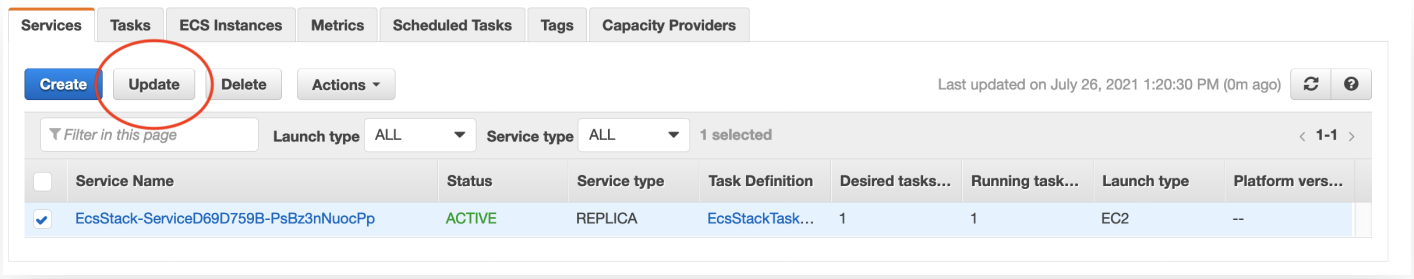
In the lower pane, in the **Details** tab and under **Group details** section, click the **Edit** button.

- Change the current settings for “minimum” to `2` to ensure we always have at least 2 instances available for redundancy. Note: if you only increase “desired” and “maximum” then the scaling policy for the auto scaling group could decrease the “desired” value back to `1` during low load periods.
- Set “desired” and “maximum” to `2` or more. Note: setting the desired value to more than the number of tasks (see below) will leave you with idle instances.
- Click **Update** to complete the changes:

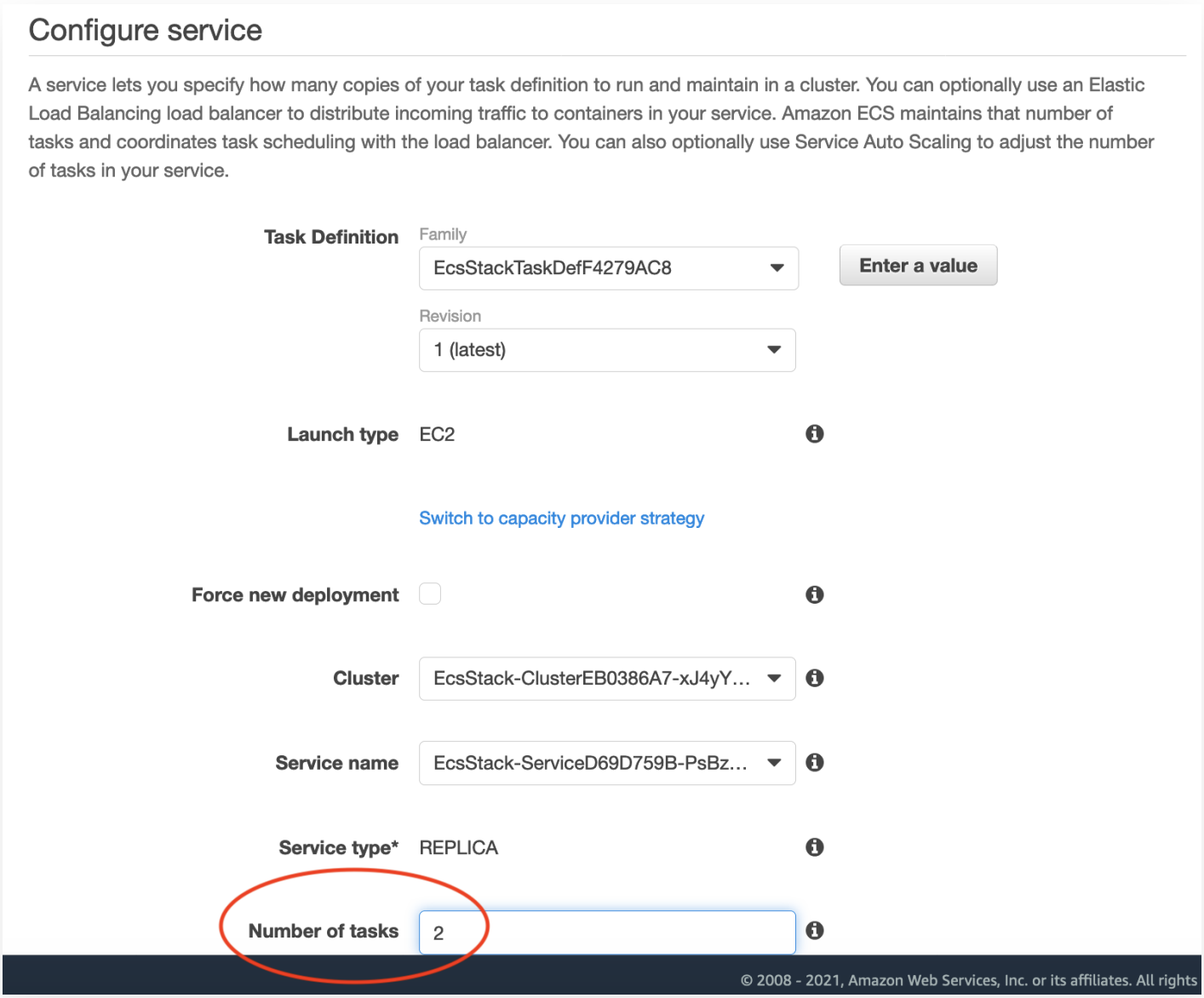


Increase the number of tasks

Navigate to the *Clusters* section in the ECS console and search for the cluster named `FisStackEcs-Cluster...`, e.g. `FisStackEcs-ClusterEB0386A7-xJ4yY19a5jLP`. Click on the cluster name and look at the ECS service named `FisStackEcs-SampleAppService...`, e.g. `FisStackEcs-SampleAppServiceD69D759B-PsBz3nNuocPp`, running on this cluster. Select the check box next to our ECS Service and click **Update**:



Scroll to the bottom of the *Configure service* screen and change the value of the **Number of tasks** setting from `1` to `2`. Click **Skip to review** and complete the process by selecting **Update Service**.



Repeat the experiment

Now that we have improved our configuration, let's re-run the experiment. Before starting review the ECS Cluster to ensure that the instance capacity has increased to `2` and that the number of running tasks is `2`.

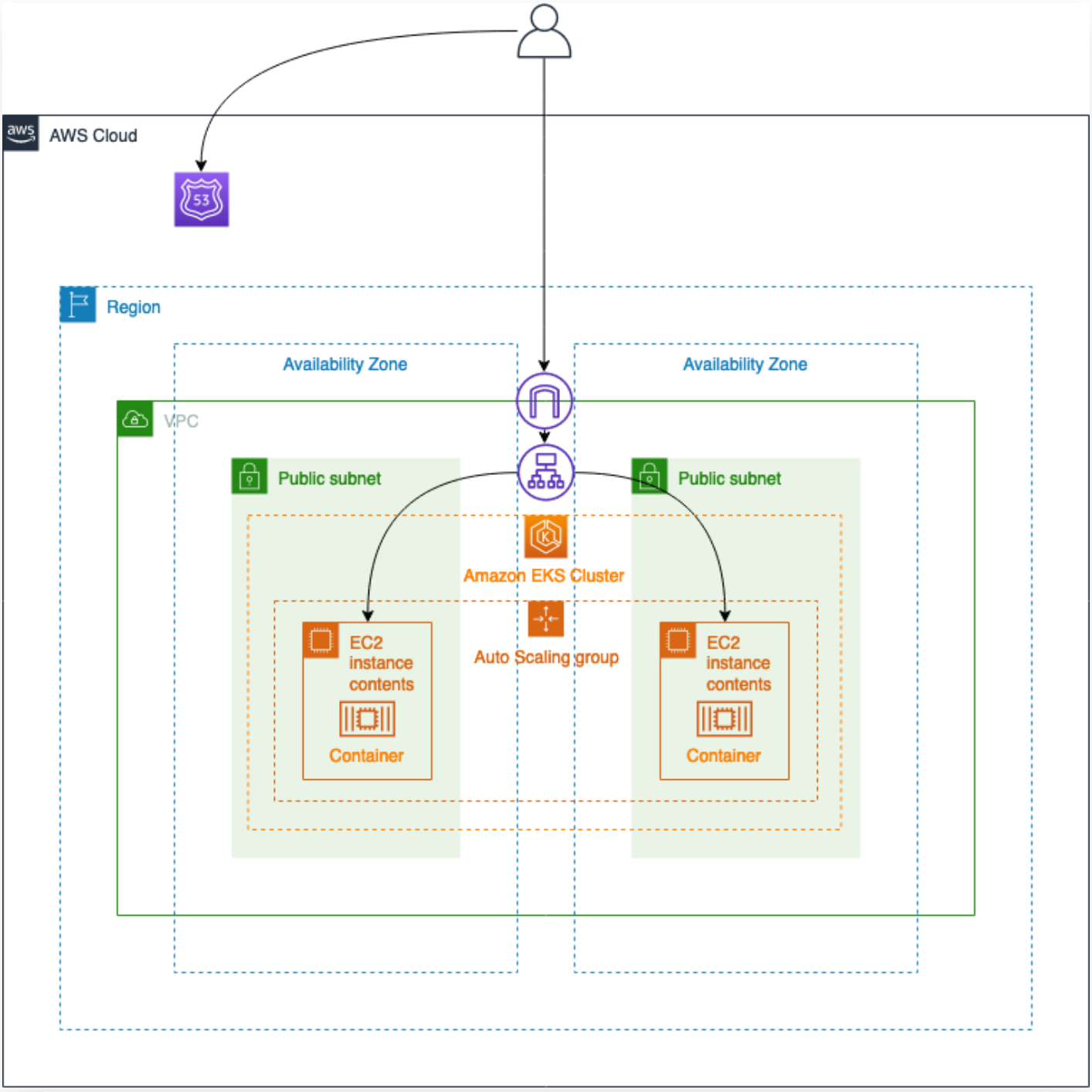
This time we should observe that, even when one of the container instances gets terminated, our application is still available and successfully serving requests. In the output of the Bash script there we should no longer see the HTTP 503 “Service Unavailable” return codes.

ECS FURTHER LEARNING

For more on ECS configurations see the ECS workshop.

AMAZON EKS

In this section we will cover working with containers running on Amazon Elastic Kubernetes Service (Amazon EKS). For this setup we'll be using the following test architecture:



Amazon EKS gives you the flexibility to start, run, and scale Kubernetes applications in the AWS cloud or on-premises. Amazon EKS helps you provide highly-available and secure clusters and automates key tasks such as patching, node provisioning, and updates. EKS runs upstream Kubernetes and is certified Kubernetes conformant for a predictable experience. You can easily migrate any standard Kubernetes application to EKS without needing to refactor your code.

Note

For this section, make sure you have `kubectl` installed in your local environment. Follow these steps if you need to install `kubectl`.

HYPOTHESIS & EXPERIMENT

Experiment idea

In this section we want to ensure that our containerized application running on Amazon EKS is designed in a fault tolerant way, so that even if an instance in the cluster fails our application is still available. Let's validate this:

- **Given**: we have a containerized application running on Amazon EKS exposing a web page.
- **Hypothesis**: failure of a single worker node instance will not adversely affect our application. The web page will continue to be available.

Experiment setup

Note

We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

General template setup

Note

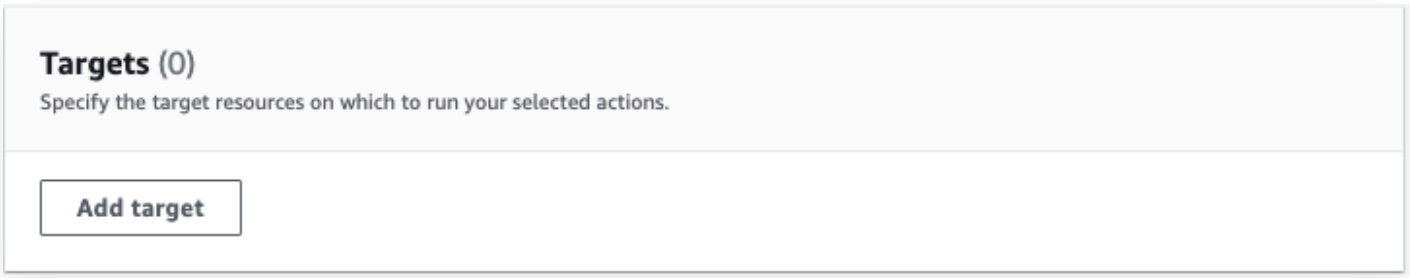
We are assuming that you have already set up an IAM role for for this workshop. If you haven't, see the **Create FIS Service Role** section.

Create a new experiment template:

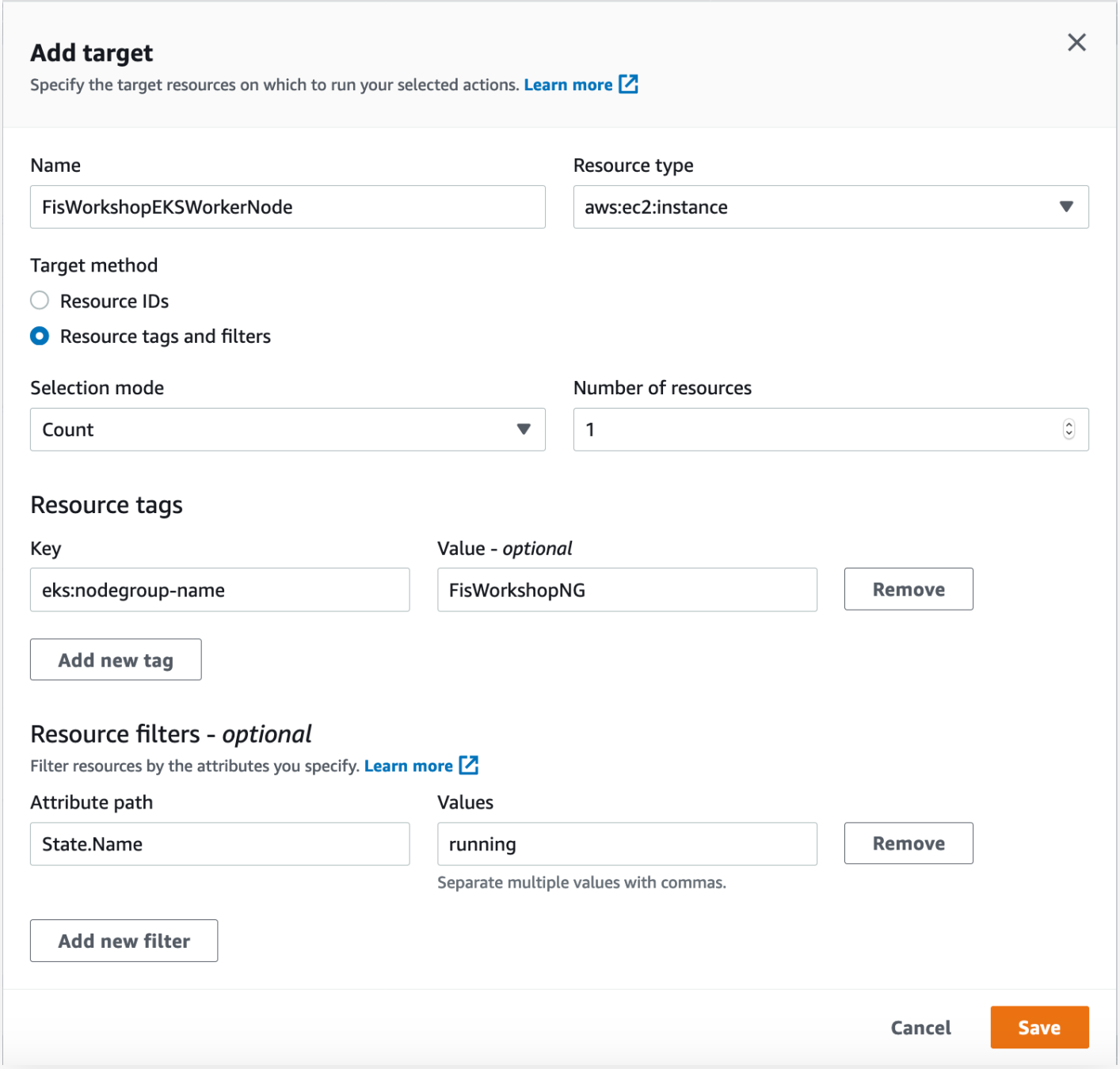
- add `Name` tag of `FisWorkshopEKS`
- add `Description` of `Terminate EKS Worker Node`
- select `FisWorkshopServiceRole` as execution role

Target selection

Now we need to define targets. Scroll to the “Targets” section and select “Add Target”



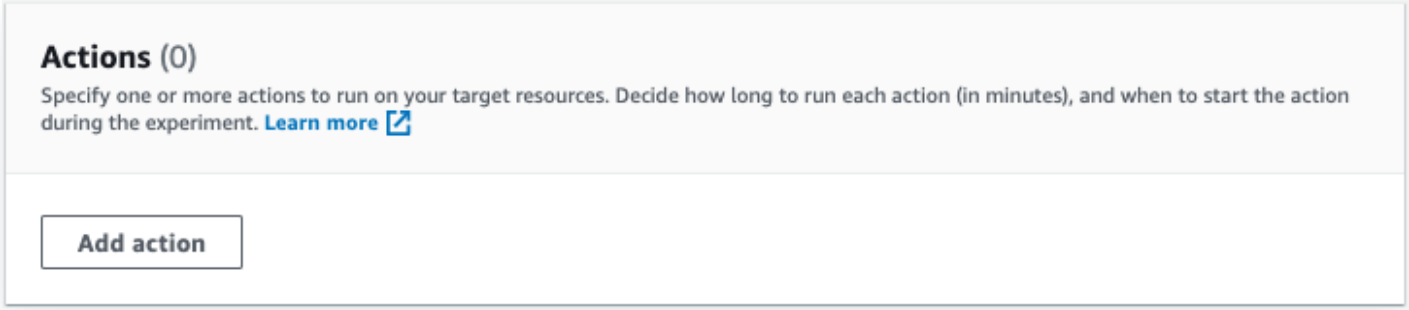
On the “Add target” popup enter `FisWorkshopEKSWorkerNode` for name and select `aws:ec2:instance`. For “Target method” we will dynamically select resources based on an associated tag. Select the `Resource tags and filters` checkbox. Pick `Count` from “Selection mode” and enter `1`. Under “Resource tags” enter `eks:nodegroup-name` in the “Key” field and `FisWorkshopNG` for “Value”. Under filters enter `State.Name` in the “Attribute path” field and `running` under “Values”. Select “Save”.



Note: we are using the `aws:ec2:instance` action instead of the `aws:eks:nodegroup` action because currently the latter cannot terminate a single running worker node.

Action definition

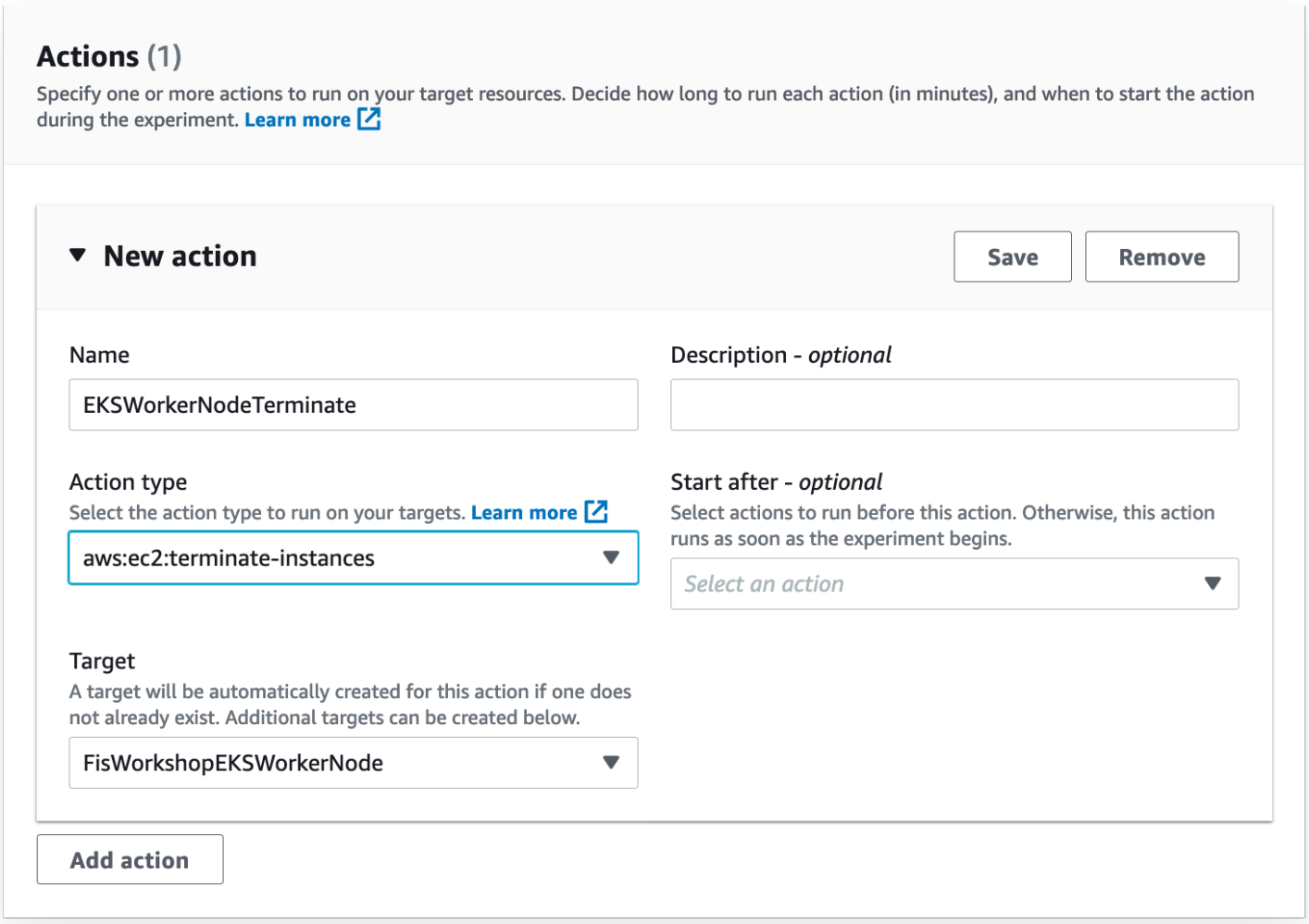
With targets defined we define the action to take. Scroll to the “Actions” section" and select “Add Action”



For “Name” enter `EKSWorkerNodeTerminate` and you can skip the Description. For “Action type” select `aws:ec2:terminate-instances`.

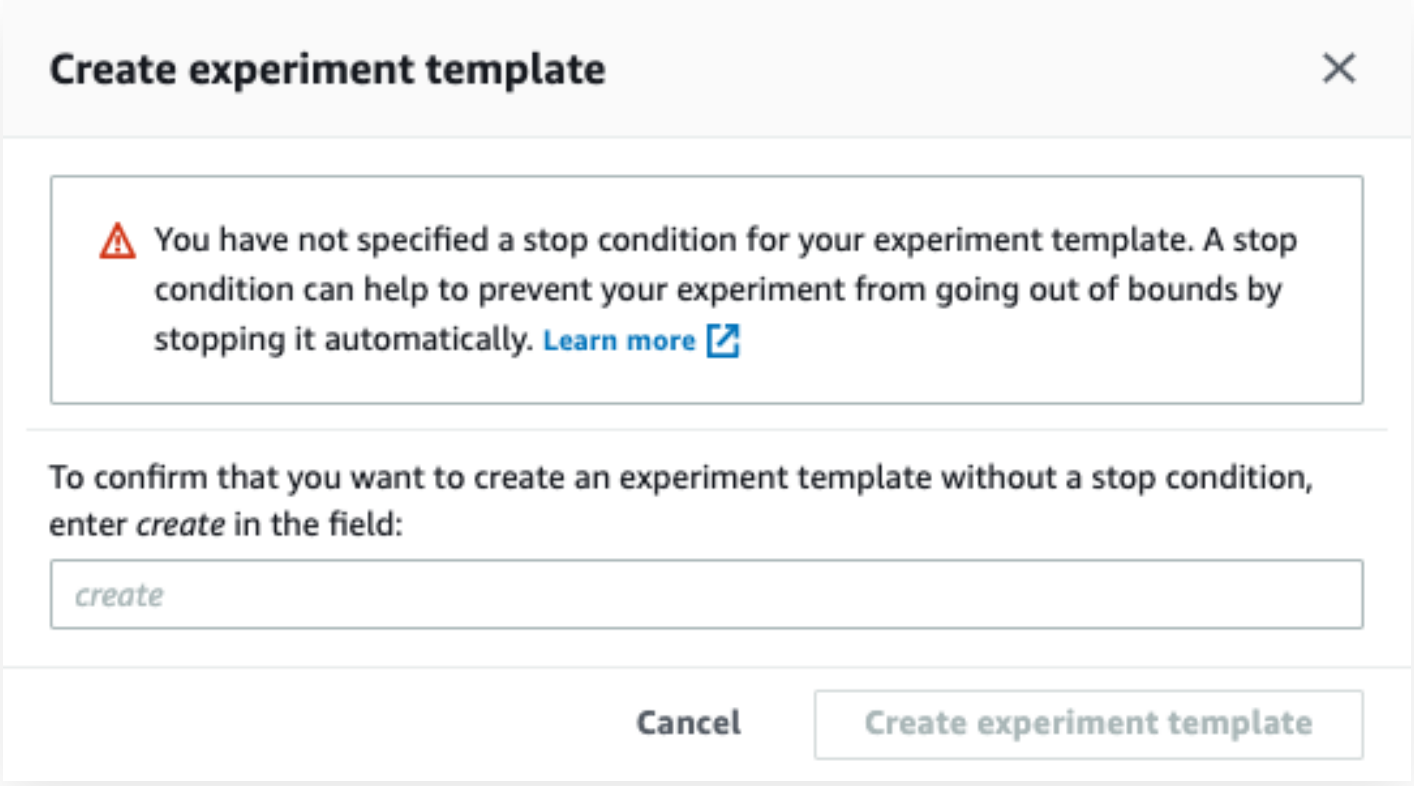
We will leave the “Start after” section blank since the instances we are terminating are part of an EKS Managed Node Group and we can let the Managed Node Group create new instances to replace the terminated ones.

Under “Target” select the `FisWorkshopEKSWorkerNode` target created above. Select “Save”.



Creating template without stop conditions

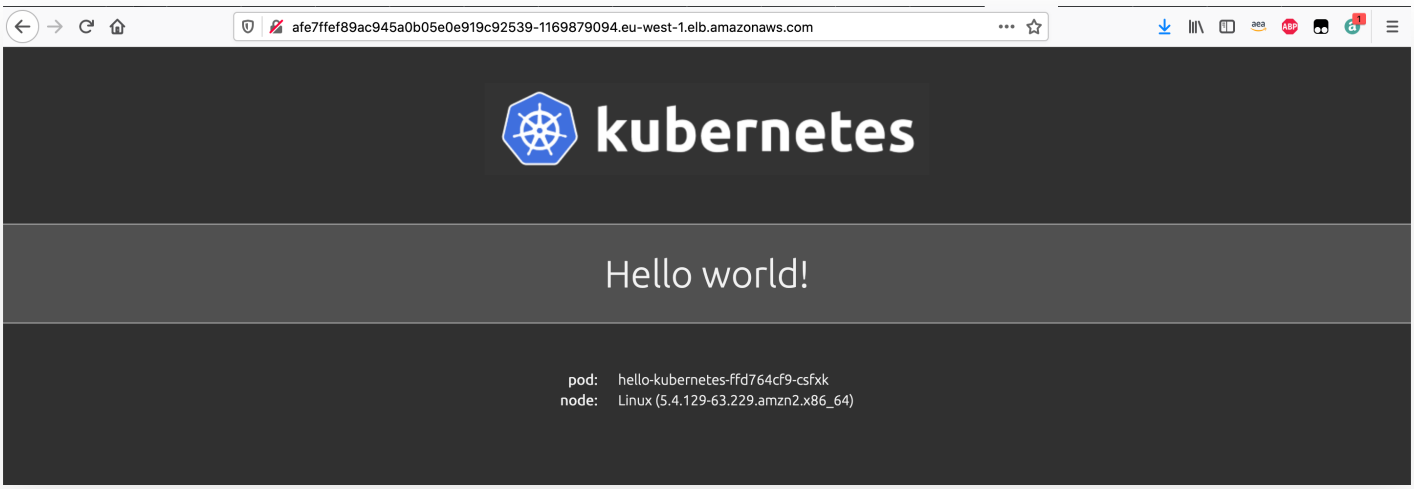
Confirm that you wish to create the template without stop condition.



Validation procedure

Before running the experiment we should consider how we will define success. Let's check the webpage we are hosting. To find the URL of the webpage navigate to the CloudFormation console, select the `FisStackEks` stack, Select "Outputs", and copy the value of "FisEksUrl".

Open the URL in a new tab to validate that our website is in fact up and running:



How will we know that our instance failure was in fact non-impacting? For this workshop we'll be using a simple Bash script that continuously polls our application.

Starting the validation procedure

In your local terminal, run the following script. For your convenience we are automating the query for the load balancer URL but you could also paste the URL you've found above:

```
# Query URL for convenience
EKS_URL=$( aws cloudformation describe-stacks --stack-name FisStackEks --query
"Stacks[*].Outputs[?OutputKey=='FisEksUrl'].OutputValue" --output text )

# Busy loop queries. CTRL-C to end loop
while true; do
  curl -sLo /dev/null -w 'Code %{response_code} Duration %{time_total} \n'
${EKS_URL}
done
```

We would expect that all requests will return a HTTP 200 OK code with some variability in the request duration, meaning the application is still responding successfully. Healthy output should look like this:

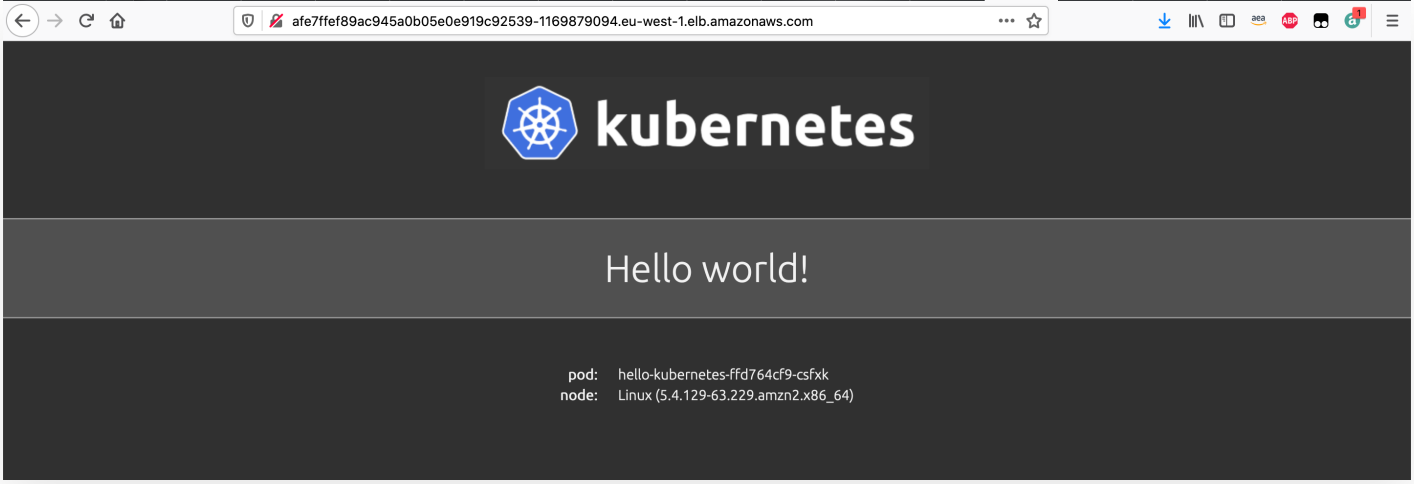
```
Code 200 Duration 0.140314
Code 200 Duration 0.086206
Code 200 Duration 0.085946
Code 200 Duration 0.084102
Code 200 Duration 0.085972
```

Leave the script running while we run the FIS experiment next.

Run FIS experiment

Record current application state

In a new browser window navigate to the load balancer URL you copied earlier, this is your application endpoint. Notice that the application is currently running:

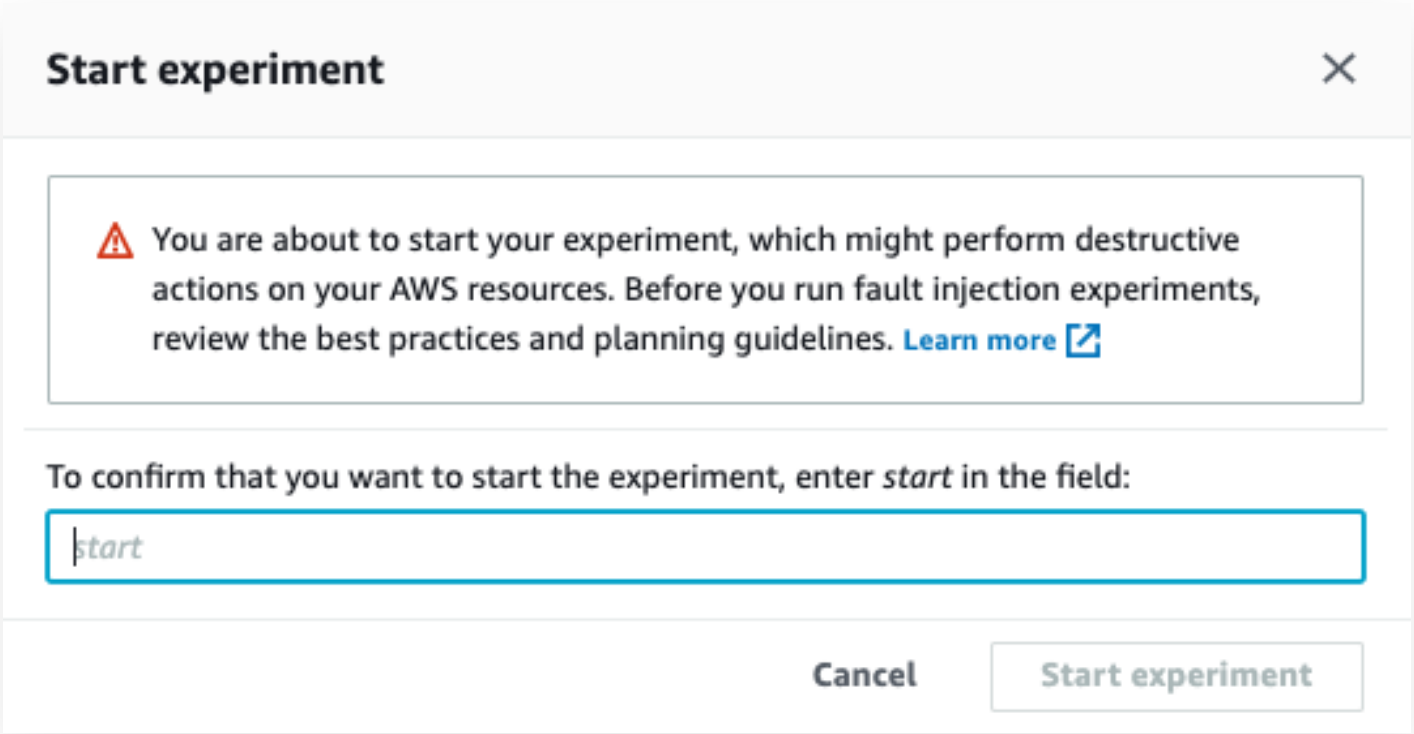


You can also verify the HTTP return code using this command, replacing `REPLACE_WITH_EKS_SERVICE_ALB_URL` with the load balancer DNS name you copied earlier:

```
curl -IL <REPLACE_WITH_EKS_SERVICE_ALB_URL> | grep "^HTTP\/"
```

Start the experiment

- select the `FisWorkshopEKS` experiment template you created above
- select **Start experiment** from the **Action** drop-down menu
- add a `Name` tag of `FisWorkshopEKSRun1`
- confirm that you want to start an experiment



Review results

Let's take a look at the output in the terminal window where your Bash script is running:

```
Code 200 Duration 0.137204
Code 200 Duration 0.080911
Code 200 Duration 0.081539
Code 200 Duration 0.077265
Code 200 Duration 0.085331
Code 200 Duration 0.081634

...

Code 000 Duration 0.093033
Code 000 Duration 0.088688
Code 000 Duration 0.086454
Code 000 Duration 0.088505
Code 000 Duration 0.097665

...

Code 200 Duration 0.082434
Code 200 Duration 0.081427
Code 200 Duration 0.087983
Code 200 Duration 0.081950
Code 200 Duration 0.082790
```

You'll notice that not all the requests were successful, As the FIS experiment starts you should see some `000` return codes. This is not a legal HTTP response code. If we just ran curl as

```
curl $EKS_URL
```

we would see an error message indicating that the server just closed the connection on us.

```
curl: (52) Empty reply from server
```

In practice this means our application was not available for a period of time. This is not what we were expecting, so let's dive a bit deeper to find out why it happened.

Configure kubectl

Note

Make sure you have `kubectl` installed in your local environment. Follow these steps if you need to install `kubectl`.

We will follow these steps to update the `kubectl` configuration to securely connect to the EKS cluster. The cluster is named `FisWorkshop-EksCluster`. To find the ARN of the kubectl access role, navigate to the CloudFormation console, select the `FisStackEks` stack, Select "Outputs", and copy the value of "FisEksKubectlRole".

From a local terminal, run the following command to configure kubectl:

```
# verify you have aws CLI installed
aws --version

# Retrieve the role ARN
KUBECTL_ROLE=$( aws cloudformation describe-stacks --stack-name FisStackEks --
query "Stacks[*].Outputs[?OutputKey=='FisEksKubectlRole'].OutputValue" --output
text )

# Configure kubectl with cluster name and ARN
aws eks update-kubeconfig --name FisWorkshop-EksCluster --role-arn
${KUBECTL_ROLE}
```

Note

If you get the message **"error: You must be logged in to the server (Unauthorized)"** when running `kubectl` command, please follow these steps to troubleshoot the problem.

Check number of containers

From a local terminal, run the following command to check our application service configuration:

```
kubectl get pods
```

You'll notice that there's only one pod named `hello-kubernetes-...` - e.g. `hello-kubernetes-ffd764cf9-zwnq7` - meaning that only one copy of our containerized application is running at any time.

```
NAME                                 READY   STATUS    RESTARTS   AGE
hello-kubernetes-ffd764cf9-zwnq7   1/1     Running   0          8m34s
```

Check number of instances

In the same terminal, run the following command to check the nodes in our cluster:

```
kubectl get nodes
```

In the output you'll see that our cluster only has a single worker node.

```
NAME                                          STATUS   ROLES    AGE   VERSION
ip-10-0-150-147.eu-west-1.compute.internal   Ready    <none>   12m   v1.20.4-eks-
6b7464
```

Observations

This configuration is not optimal:

- A cluster with a single worker node means that if that instance fails, all the containers running on that instance will also be killed. This is what happened during our experiment and the reason why we observed some `curl: (52) Empty reply from server` messages. We should change this so that our cluster has more than one instance across multiple Availability Zones (AZs).
- An EKS workload with **one** pod also means that if that pod fails, there aren't any other pods to continue serving requests. We can modify this by adjusting the pod count to `2` (or any number greater than `1`).

Now that we have identified some issues with our current setup, let's move to the next section to fix them.

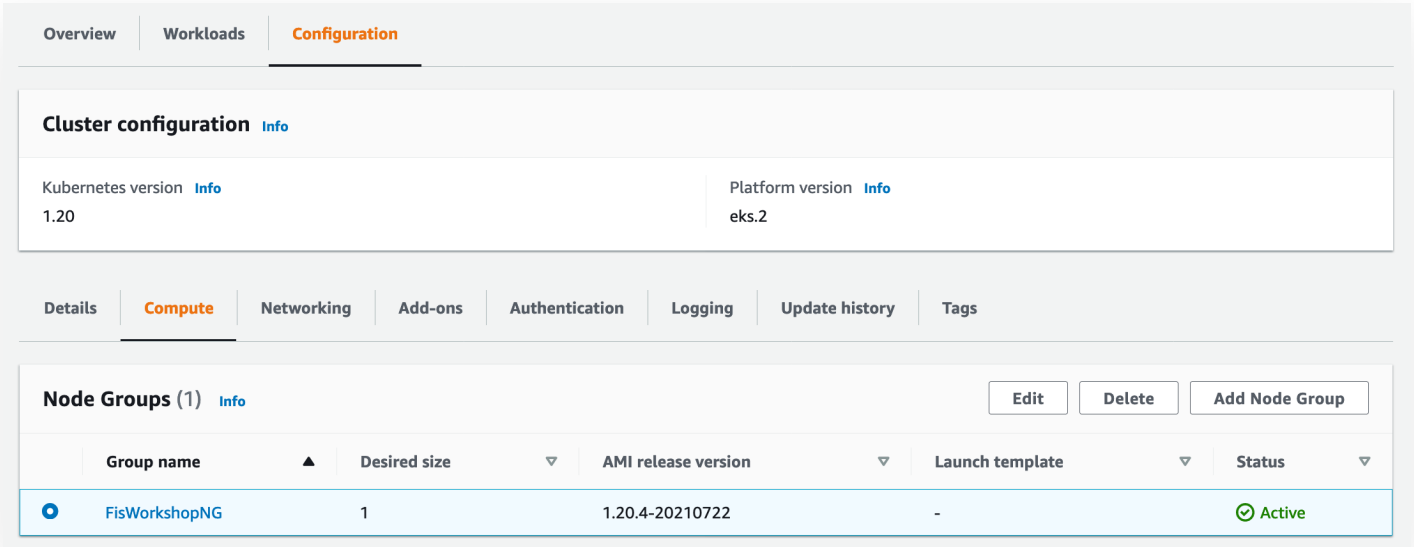
IMPROVE & REPEAT

Learning and Improving

In the previous section we have identified some issues with our current setup: our EKS cluster only had **one** worker node and our application's pod count was set to `1`. Now, let's improve our infrastructure setup.

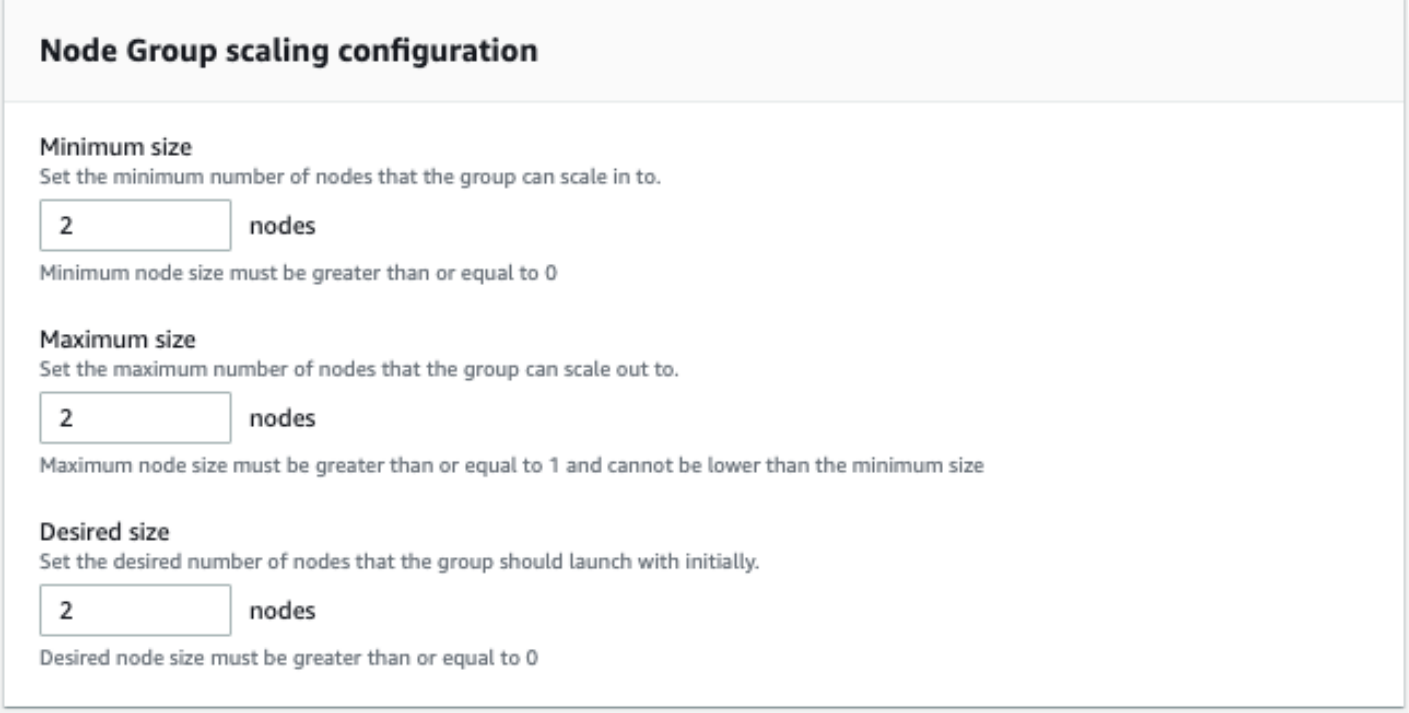
Increase the number of instances

In a browser window navigate to the *Clusters* section in the EKS console and search for the cluster named `FisWorkshop-EksCluster`. Click on the cluster name, select the *Configuration* tab and then the *Compute* tab. In the *Node Groups* section, select the round check box next to the group named `FisWorkshopNG` and click **Edit**.



On the *Edit node group* page

- Change the current settings for "minimum" to `2` to ensure we always have at least 2 instances available for redundancy. Note: if you only increase "desired" and "maximum" then the scaling policy for the auto scaling group could decrease the "desired" value back to `1` during low load periods.
- Set "desired" and "maximum" to `2` or more. Note: setting the desired value to more than the number of tasks (see below) will leave you with idle instances.



When you're finished editing, scroll to the bottom and choose **Save changes**.

Increase the number of containers

From a local terminal, run the following command to update the application's pod count to **2**:

```
kubectl scale --current-replicas=1 --replicas=2 deployment/hello-kubernetes
```

To verify, you can run `kubectl get pods` and `kubectl get deployments`. Here's the sample output.

```
NAME                                 READY   STATUS    RESTARTS   AGE
hello-kubernetes-ffd764cf9-5v7z9   1/1     Running   0          25s
hello-kubernetes-ffd764cf9-6bdbn   1/1     Running   0          4m43s
```

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
hello-kubernetes   2/2     2            2           46h
```

Repeat the experiment

Now that we have improved our configuration, let's re-run the experiment. Before starting review the EKS Cluster to ensure that the instance capacity has increased to `2` and that the number of running containers is `2`.

This time we should observe that, even when one of the container instances gets terminated, our application is still available and successfully serving requests. In the output of the Bash script there should be no `curl: (52) Empty reply from server` messages.

EKS/k8s cluster auto scaling

In this workshop we used manual scaling of both worker nodes and pods. In a production setup you would likely configure kubernetes / EKS to use

- a Cluster Autoscaler that is aware of scaling needs based on pod configuration.
- a Horizontal Pod Autoscaler to dynamically manage the number of pods .
- a Vertical Pod Autoscaler to dynamically manage CPU and memory allocation on your pods.

For more on EKS configurations see the EKS workshop.

SERVERLESS

FIS currently does not support disrupting serverless execution in AWS Lambda. It is, however, possible to inject chaos actions by decorating the code executed within AWS Lambda. For a deeper introduction to the topic see the Serverless Chaos workshop.

CI/CD

In this section, we will discuss how to integrate a Fault Injection Simulator experiment with a CICD pipeline.

A natural progression for customers as they start to adopt chaos engineering is to integrate experiments into their existing CICD pipelines. Just like unit tests, integrations tests, and load tests, chaos experiments are a valuable tool to determining the robustness of a new piece of software. By running an experiment as part of the pipeline, you can ensure that every new release meets your company's defined quality gates of reliability and performance. Remember that chaos engineering should not be viewed as a replacement for other types of test, but an enhancement to your existing testing strategy.

Fault Injection Simulator allows us to programmatically execute an experiment through an API. We will use this API call as part of the pipeline to start the experiment on each checkin to our repository. Next a process will run and monitor the execution of the pipeline. At this point, we can make a determination if the release (experiment passing) was successful and continue our deployment.

In this next section, we will create a CICD pipeline using the AWS Code Suite of tools. From there we will use this pipeline to apply a CloudFormation Template on each checkin that provisions our sample infrastructure and our Fault Injection Experiment. The last stage in out pipeline is to run the experiment against our infrastructure.

SETUP

In this section, we will integrate a Fault Injection Simulator experiment with a CICD pipeline.

Create The Pipeline

We will use the AWS CDK to provision our CICD pipeline.
First, in your cloud9 terminal, start by cloning the repository for the workshop.

```
cd ~/environment
git clone https://github.com:aws-samples/aws-fault-injection-simulator-
workshop.git
```

Next change directory into the CICD CDK project and restore the npm packages used for the pipeline.

```
cd aws-fault-injection-simulator-workshop/resources/code/cdk/cicd/
npm install
```

Finally lets deploy our stack.

```
cdk deploy --require-approval never
```

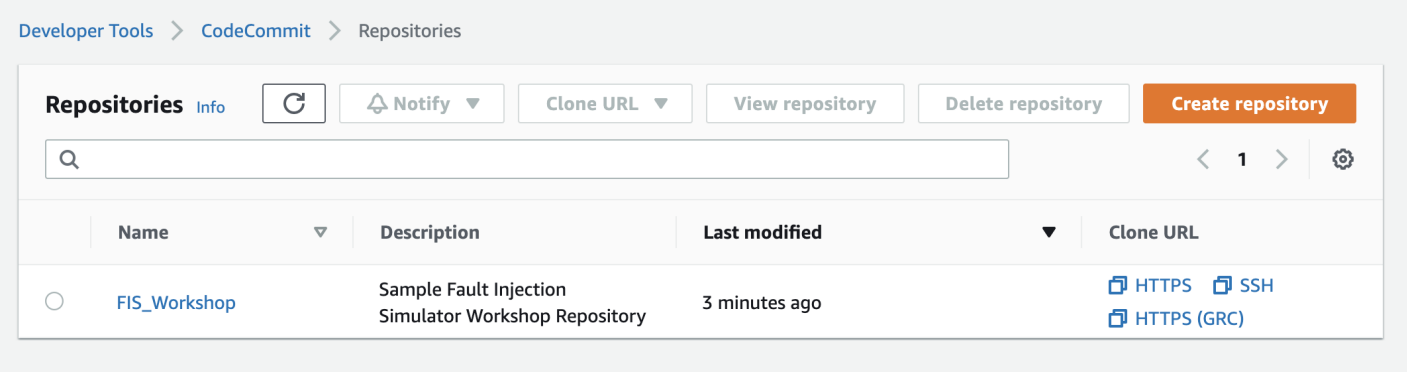
The stack will take a few minutes to complete. You can monitor the progress from the CloudFormation Console.
Continue to the next section.

REVIEW THE PIPELINE

Lets review the components our previous section created.

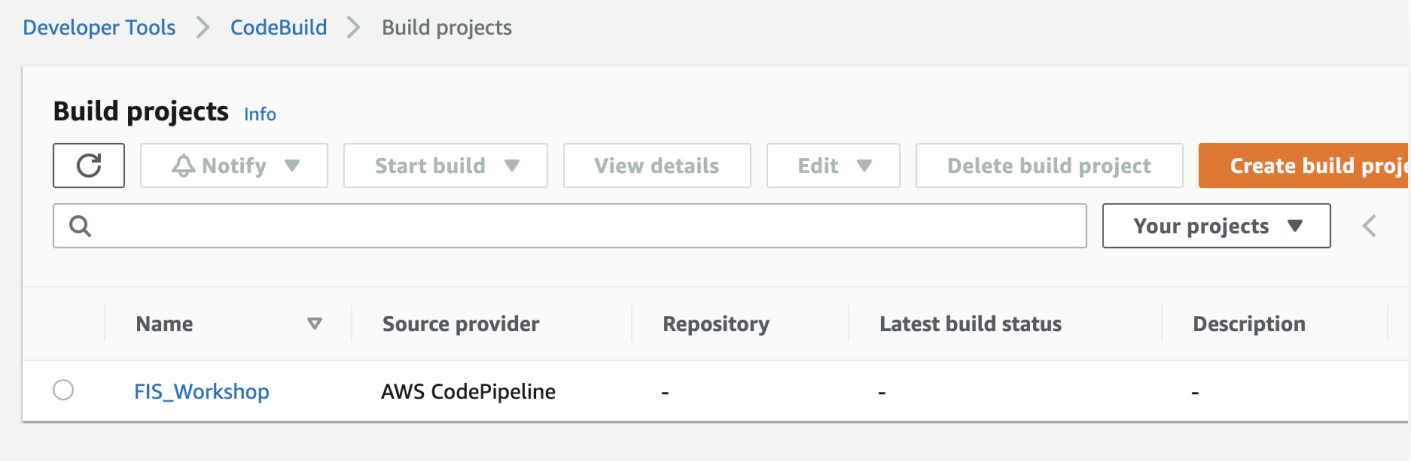
CodeCommit

Open the AWS CodeCommit Console. You should see the newly created `FIS_Workshop` repository.



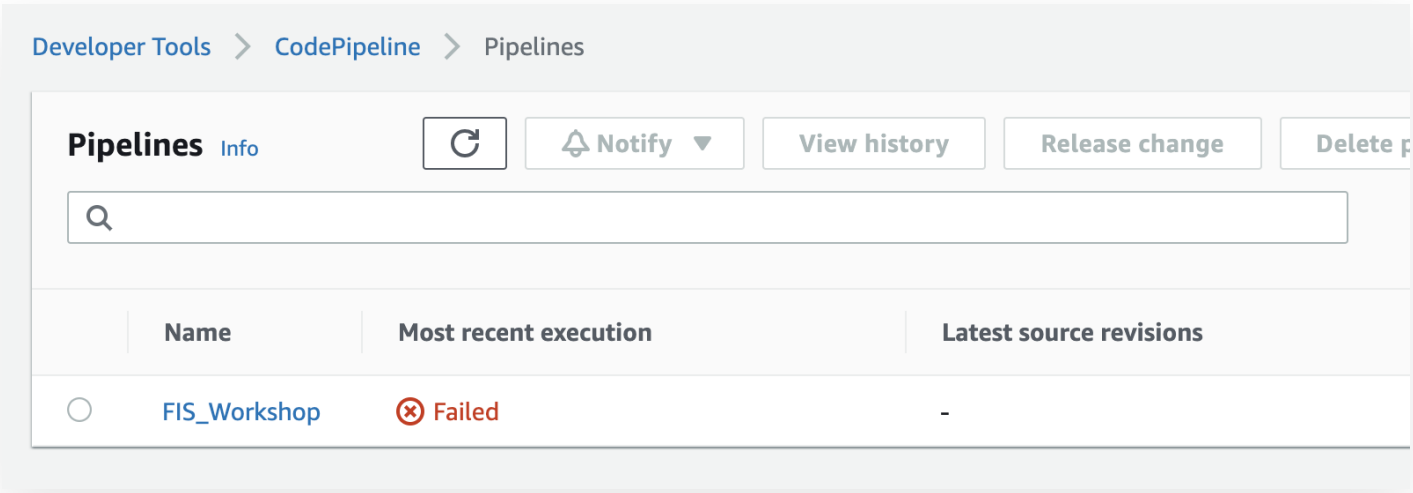
CodeBuild

Open the AWS CodeBuild Console. You should see the `FIS_Workshop` build project.



CodePipeline

Open the AWS CodePipeline Console. You should now see the `FIS_Workshop` pipeline.

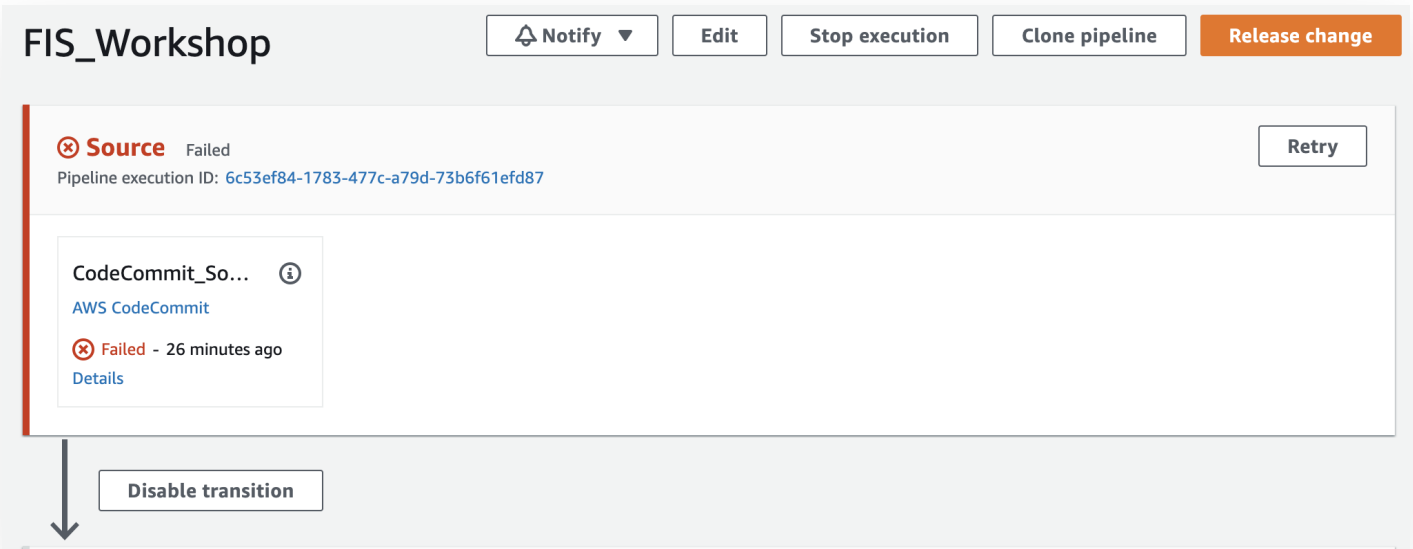


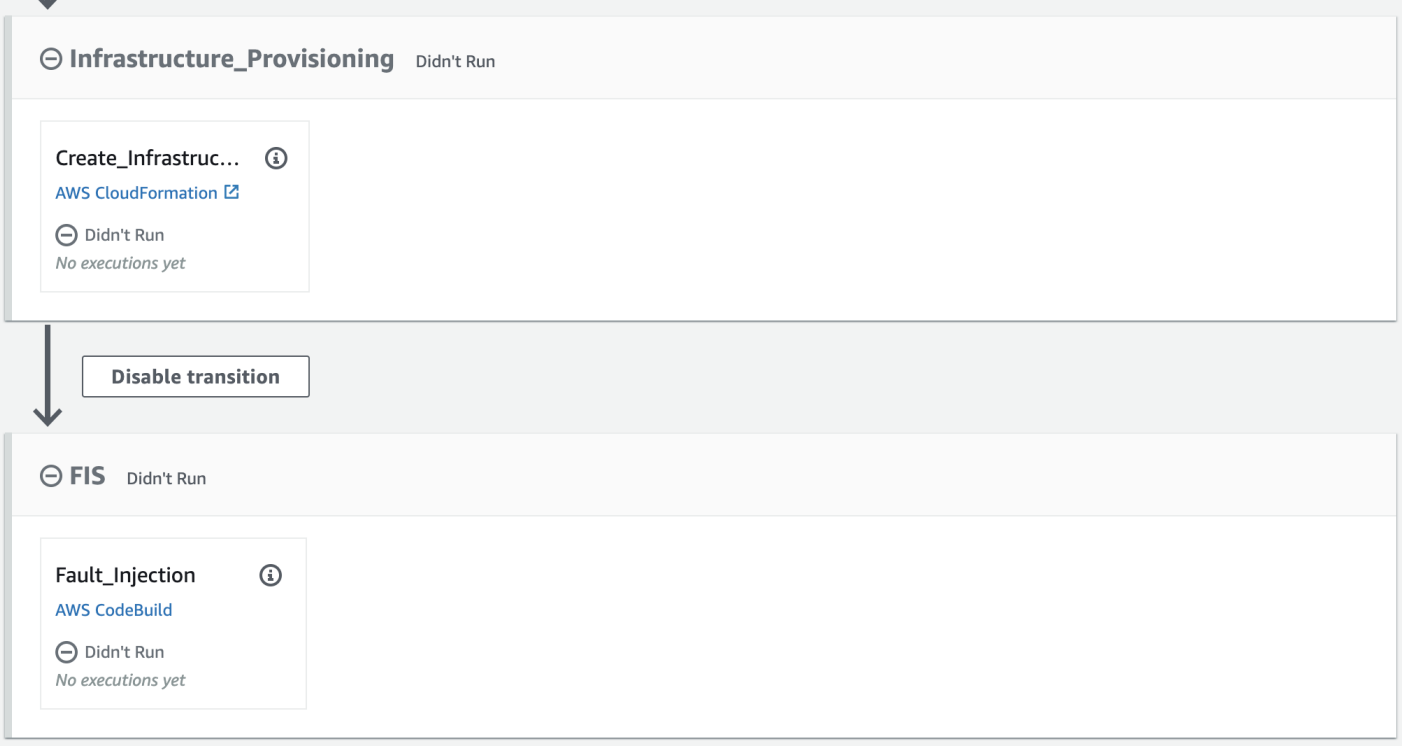
The pipeline will start in a failed state, since we have not uploaded any files to our repository.

Click on the pipeline name, to review.

This pipeline has 3 stages.

1. **Source**: This stage will trigger the pipeline when a commit occurs in our repository.
2. **Infrastructure_Provisioning**: This stage will create our test infrastructure and create our experiment templates.
3. **FIS**: This stage will use the code build project to run our experiment and monitor the results.





Continue to the next section to start the pipeline.

START THE PIPELINE

To start our pipeline we need to commit files to our CodeCommit repository.
The commit action will trigger the pipeline that provisions our infrastructure and then runs our experiment.

Commit Files

In your cloud9 terminal, start by cloning the `FIS_Workshop` repository.
Open the AWS Code Commit Console. Click the HTTPS link next to the repository name. See the below command for an example, when working in the `US-EAST-1` region.

```
cd ~/environment
git clone https://git-codecommit.us-east-1.amazonaws.com/v1/repos/FIS_Workshop
cd FIS_Workshop
```

Copy the sample files from the resources section into the newly cloned repository.

```
cp ~/environment/aws-fault-injection-simulator-
workshop/resources/code/cdk/cicd/resources/* ~/environment/FIS_Workshop/
```

Since this is the first time working with code commit, we should setup our username and email for the commit history. Run the below commands, be sure to replace the details with your information.

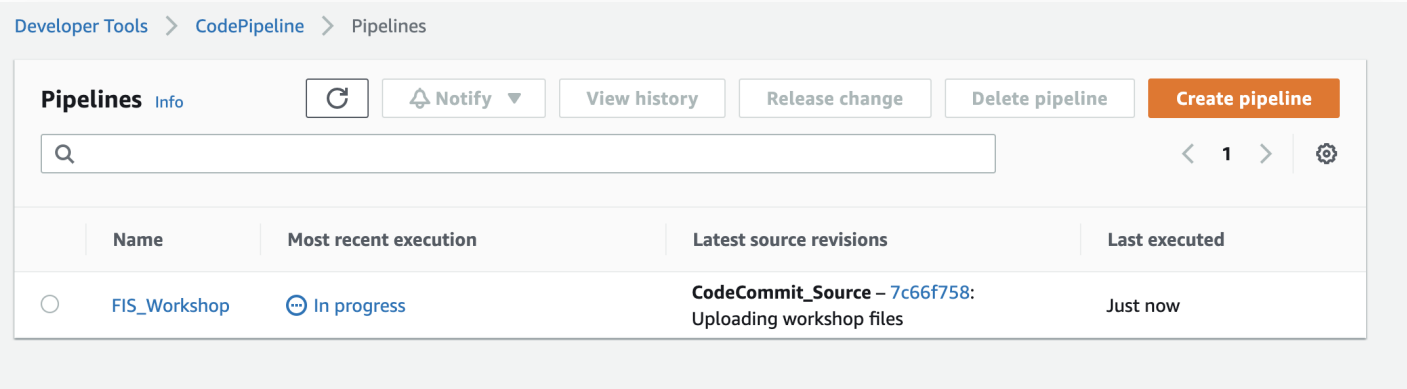
```
git config --global user.name "Your Name"
git config --global user.email you@example.com
```

Finally commit the files to start the pipeline.

```
git add .
git commit -am "Uploading Workshop files"
git push -u
```

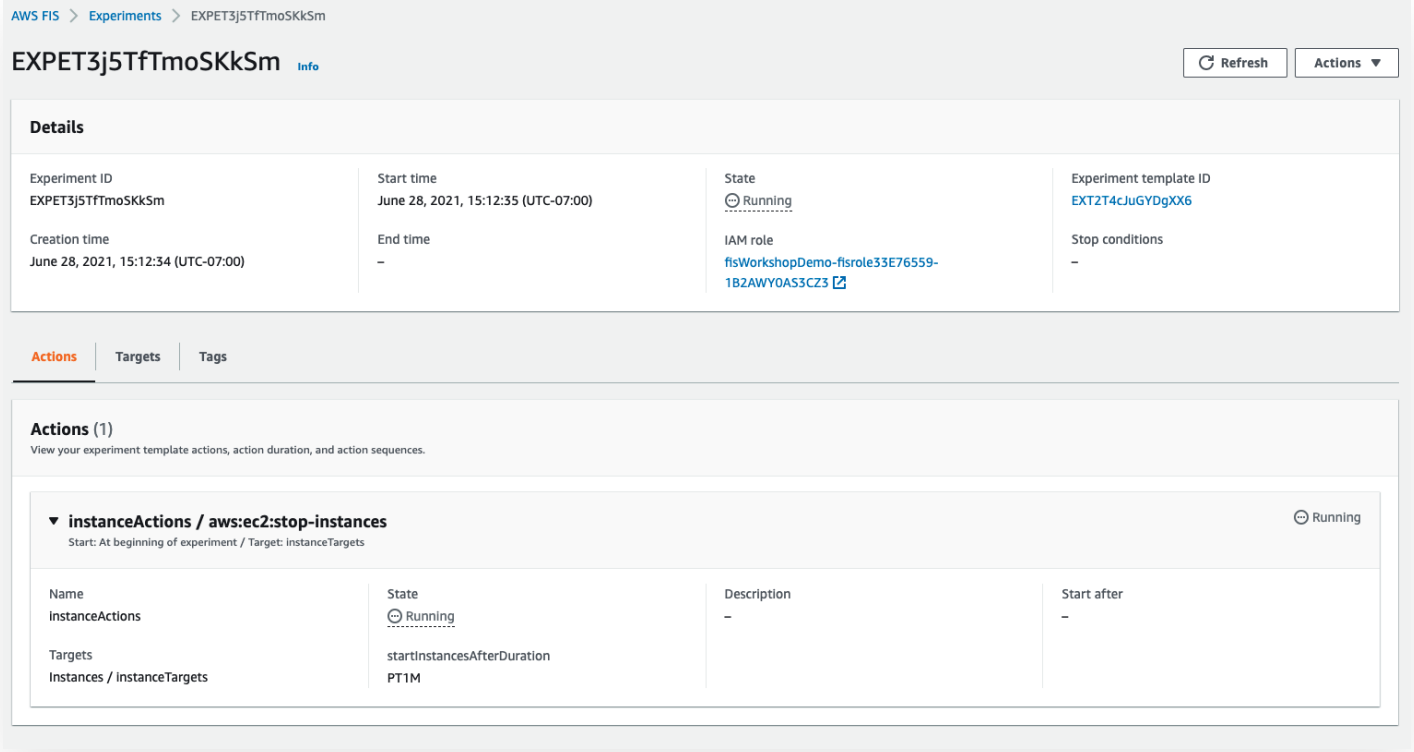
View Progress

After you commit the files, the pipeline will start. Open the AWS CodePipeline Console. You should now see the `FIS_Workshop` pipeline is in progress. Click on the pipeline name to view the step details.

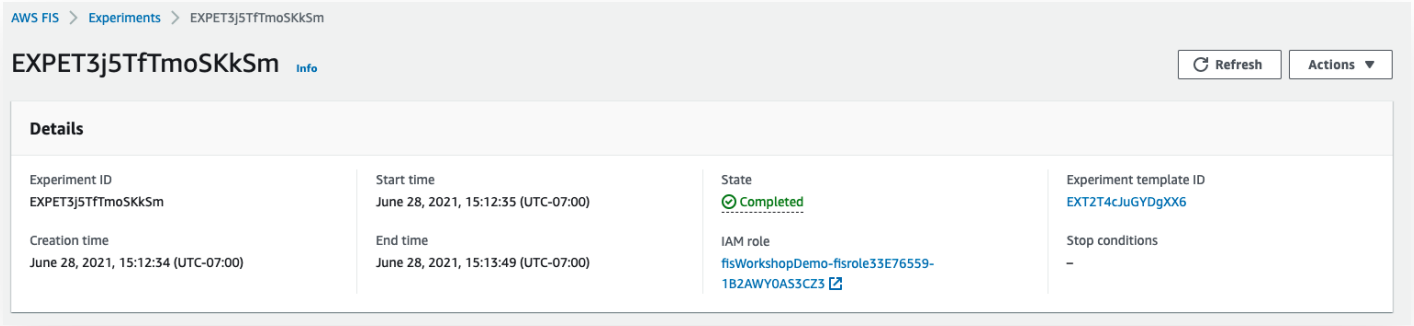


Wait for the infrastructure provisioning step to complete. After this step, our Experiment will start. You can monitor the progress of your experiment from both the CodePipeline details page or from the FIS console.

Click on the running experiment. You should see the experiment in a running status.



After a few minutes refresh the page. You should see the experiment is completed successfully.



Finally navigate back to the AWS CodePipeline Console. You should also see that your pipeline has completed successfully.

Infrastructure_Provisioning Succeeded

Pipeline execution ID: 45b360c4-c02a-43b8-83c8-9f709ecfcdba

Create_Infrastructure

AWS CloudFormation

Succeeded - 10 minutes ago

Details

1c211c6d CodeCommit_Source: updated template

Disable transition

FIS Succeeded

Pipeline execution ID: 45b360c4-c02a-43b8-83c8-9f709ecfcdba

Fault_Injection

AWS CodeBuild

Succeeded - 7 minutes ago

Details

1c211c6d CodeCommit_Source: updated template

Congratulations! You have successfully integrated a Fault Injection Simulator Experiment into a CI/CD pipeline. In this scenario, we completed a happy path to ensure that our infrastructure and experiment completed without error. Continue on to the next section, where we will deploy a new version of our CloudFormation template and force our experiment (and pipeline) fail.

FORCE A PIPELINE ERROR

In this section we will update our repository to deploy a new version of our experiment and infrastructure. Then while our pipeline is running, we will force an error to test that our pipeline will stop on a failed experiment.

Change the Infrastructure Template

We will be making a change to our CloudFormation template that creates our EC2 Instance and defines our experiment. Open the AWS CodeCommit Console and select the `FIS_Workshop` repository. Click on `cfn_fis_demos.yaml` and select edit in the upper right hand corner. Edit the file as below to enable a CloudWatch alarm as a Stop Condition.

Before:

```
121         StopConditions:
122           - Source: none
123         #   - Source: aws:cloudwatch:alarm
124         #     Value:
125         #       Fn::GetAtt:
126         #         - cwalarm8A77F56F
127         #         - Arn
```

After:

```
121         StopConditions:
122         #   - Source: none
123             - Source: aws:cloudwatch:alarm
124               Value:
125                 Fn::GetAtt:
126                   - cwalarm8A77F56F
127                   - Arn
```

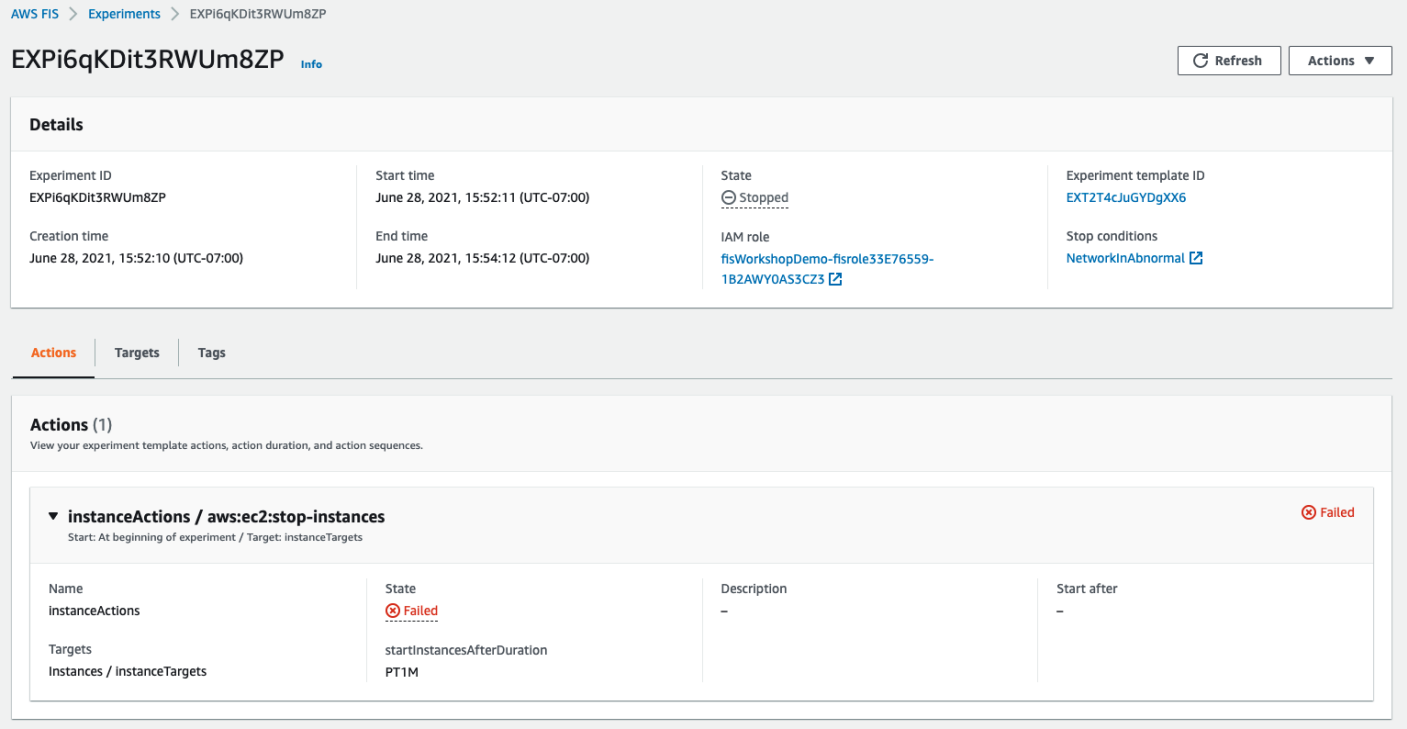
Finally, enter your name and email at the bottom of the page and commit the change. This will trigger the pipeline to start immediately.

Forcing an Error

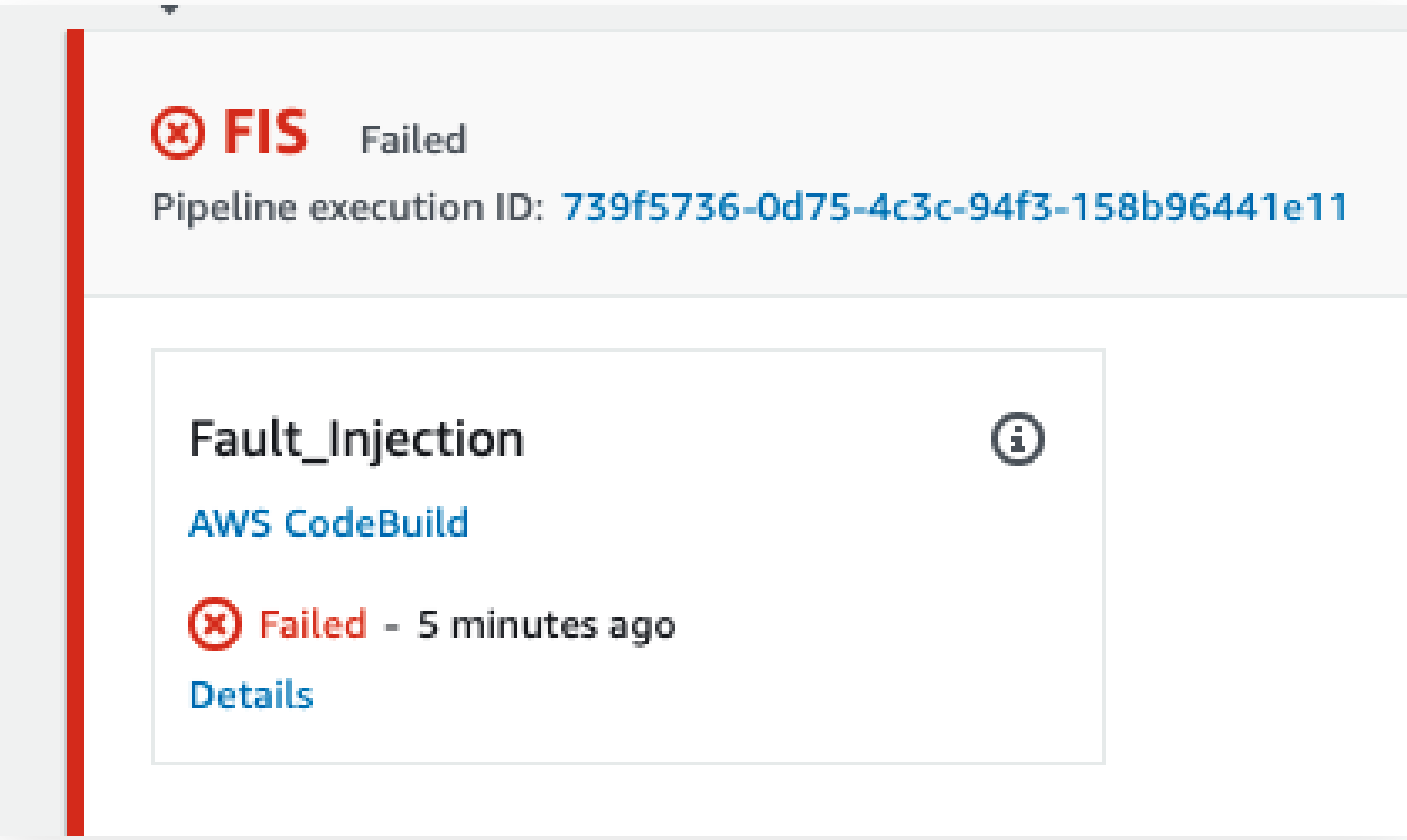
To simulate an failed experiment, we will manually set our CloudWatch alarm to an error state. Navigate back to the AWS CodePipeline Console and watch the pipeline status. Once the FIS section changes to in progress, run the below command from your workstation to force an error.

```
aws cloudwatch set-alarm-state --alarm-name "NetworkInAbnormal" --state-value
"ALARM" --state-reason "testing FIS"
```

By setting this CloudWatch alarm to an error state, this will stop the experiment. Open the FIS console. You should see that your latest experiment has failed do to the stop condition.



Finally navigate back to the AWS CodePipeline Console. You should see that your pipeline has also failed do to the experiment stopping.



Congratulations! We have now tested that a failure to our experiment will stop our pipeline. In a production scenario, after the experiment step, we would continue on with our deployment to the next stage in our pipeline.

More mature deployments could also leverage various experiments each with more disruptive behavior against different environments until our application makes it to production.

COMMON SCENARIOS

This section covers common scenarios customers ask about.

- Simulating AZ Issues

SIMULATING AZ ISSUES

A common ask we hear is for “AZ outage” simulation. Because AWS has spent more than a decade working to *prevent* exactly those scenarios and to self-heal any disruption, there is currently no easy button solution to simulate this.

In this section we will present failure paths you can group together to build an experiment that approximates an AZ failure for your particular workload.

BACKGROUND

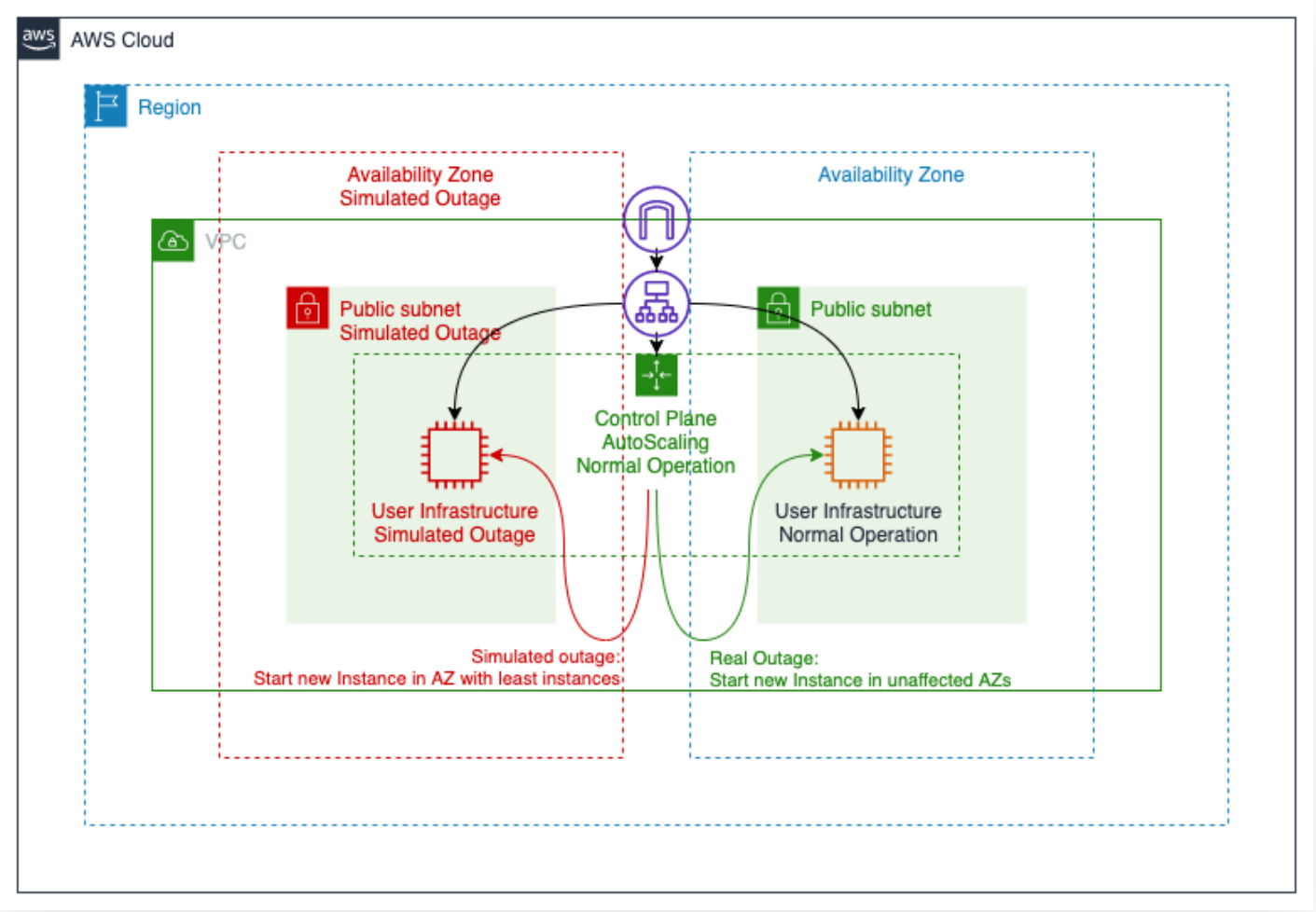
Before attempting to simulate an AZ failure it's worth considering what we mean by "AZ failure".

AZ vs. datacenter

Many of our customers phrase their idea of an AZ failure as "the whole datacenter goes away" but AWS Availability Zones are "one or more discrete data centers with redundant power, networking, and connectivity in an AWS Region" so even a full "datacenter" outage at AWS may not have the level of impact you would expect on-prem.

Control plane vs. active infrastructure

An important thing to consider is the difference between the AWS backplane and the provisioned customer infrastructure. A full datacenter outage would create awareness in the AWS control plane that resources are unavailable. In contrast a *simulated* outage to just the provisioned customer resources has to ensure that only the resources for *a single* customer are affected.



For example in the auto scaling setup we built for the **First Experiment** section, we can target EC2 instances in a given AZ for termination by filtering on `Placement.AvailabilityZone`. We *expect* that the "control plane", in this case the associated Auto Scaling group, will start new instances to replace those terminated. However, since there is no actual AZ failure and the Auto Scaling group thus has no awareness of our experiment, the new instances will most likely be re-created in the AZ for which we wanted to simulate a failure.

Simulating AZ outage options

In the following sections we will cover how to approximate AZ outages for different configurations and how to build that into a bigger experiment.

IMPACT EC2/ASG

This section covers approaches to simulating AZ issues for EC2 instances and Auto Scaling groups.

Warning

This section relies on the use of SSM Automation documents. Please review the FIS SSM Start Automation Setup when you need additional details.

Standalone EC2

Standalone EC2 instances can be directly targeted based on availability zone placement using the target filter and set `Placement.AvailabilityZone` to the desired availability zone.

EC2 with Auto Scaling

We can use `Placement.AvailabilityZone` to target instances that are part of an Auto Scaling group as well. However, as mentioned in the background section, Auto Scaling groups (ASGs) will try to rebalance instances and will likely create new instances in the "affected" AZ.

Workaround: prevent Auto Scaling

If you only need to verify continued availability you can instruct to ASG to suspend activity and not add any new instances.

For this we can extend the SSM Automation approach shown in FIS SSM Start Automation Setup.

Similar to the `aws:ec2:terminate-instances` FIS action, the updated SSM document below will terminate EC2 instances that are members of a specified Auto Scaling group and are in the selected AZ. Additionally this document will use the Auto Scaling API to suspend and re-enable auto-scaling activity:

```
---
description: Terminate all instances of ASG in a particular AZ
```

```
schemaVersion: '0.3'
assumeRole: "{{ AutomationAssumeRole }}"
parameters:
  AvailabilityZone:
    type: String
    description: "(Required) The Availability Zone to impact"
  AutoscalingGroupName:
    type: String
    description: "(Required) The names of the Auto Scaling group"
  AutomationAssumeRole:
    type: String
    description: "The ARN of the role that allows Automation to perform
      the actions on your behalf."
  Duration:
    type: String
    description: (Optional) The duration of the attack in minutes (default=5)
    default: '5'
mainSteps:
# Find all instances in ASG
- name: DescribeAutoscaling
  action: aws:executeAwsApi
  onFailure: 'step:Rollback'
  onCancel: 'step:Rollback'
  timeoutSeconds: 60
  inputs:
    Service: autoscaling
    Api: DescribeAutoScalingGroups
    AutoScalingGroupNames:
        - "{{ AutoscalingGroupName }}"
  outputs:
    - Name: InstanceIds
      Selector: "$..InstanceId"
      Type: StringList
# Find all ASG instances in AZ
- name: DescribeInstances
  action: aws:executeAwsApi
  onFailure: 'step:Rollback'
  onCancel: 'step:Rollback'
  timeoutSeconds: 60
  inputs:
    Service: ec2
    Api: DescribeInstances
    Filters:
    - Name: "availability-zone"
      Values:
        - "{{ AvailabilityZone }}"
    - Name: "instance-id"
      Values: "{{ DescribeAutoscaling.InstanceIds }}"
  outputs:
     - Name: InstanceIds
       Selector: "$..InstanceId"
       Type: StringList
# Suspend ASG activity to prevent scaling
- name: SuspendAsgProcesses
  action: aws:executeAwsApi
```

```
    onFailure: 'step:Rollback'
    onCancel: 'step:Rollback'
    inputs:
      Service: autoscaling
      Api: SuspendProcesses
      AutoScalingGroupName: "{{ AutoscalingGroupName }}"
      ScalingProcesses: ['Launch','Terminate']
  # Terminate 100% of selected instances
  - name: TerminateEc2Instances
    action: aws:changeInstanceState
    onFailure: 'step:Rollback'
    onCancel: 'step:Rollback'
    inputs:
      InstanceIds: "{{ DescribeInstances.InstanceIds }}"
      DesiredState:  terminated
      Force: true
  # Wait for up to 90s to make sure instances have been terminated
  - name: VerifyInstanceStateTerminated
    action: aws:waitForAwsResourceProperty
    onFailure: 'step:Rollback'
    onCancel: 'step:Rollback'
    timeoutSeconds: 90
    inputs:
      Service: ec2
      Api: DescribeInstanceStatus
      IncludeAllInstances: true
      InstanceIds: "{{ DescribeInstances.InstanceIds }}"
      PropertySelector: "$..InstanceState.Name"
      DesiredValues:
        - terminated
  # Wait for duration specified before re-enabling autoscaling
  # Note that this is different of the FIS duration setting,
  # make sure that FIS duration setting is higher than this
  - name: WaitForDuration
    action: 'aws:sleep'
    onFailure: 'step:Rollback'
    onCancel: 'step:Rollback'
    inputs:
      Duration: 'PT{{Duration}}M'
  # Always re-enable autoscaling
  - name: Rollback
    action: aws:executeAwsApi
    inputs:
      Service: autoscaling
      Api: ResumeProcesses
      AutoScalingGroupName: "{{ AutoscalingGroupName }}"
      ScalingProcesses: ['Launch','Terminate']
    isEnd: true
  outputs:
  - DescribeInstances.InstanceIds
```

This SSM document requires an SSM role with the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableAsgDocument",
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:SuspendProcesses",
                "autoscaling:ResumeProcesses",
                "ec2:DescribeInstances",
                "ec2:DescribeInstanceStatus",
                "ec2:TerminateInstances"
            ],
            "Resource": "*"
        }
    ]
}
```

From here follow the “Create FIS Experiment Template” step shown in FIS SSM Start Automation Setup to add this as an action to your FIS experiment.

Workaround: remove AZ from ASG / LB

If you need to model a situation in which EC2 instances in an AZ become unavailable but where the ASG will bring up replacement instances in the remaining AZs, you can modify the ASG to remove subnets associated with the AZ:

```
---
description: Terminate all instances of ASG in a particular AZ
schemaVersion: '0.3'
assumeRole: "{{ AutomationAssumeRole }}"
parameters:
  AvailabilityZone:
    type: String
    description: "(Required) The Availability Zone to impact"
  AutoscalingGroupName:
    type: String
    description: "(Required) The name of the autoscaling group"
  AutomationAssumeRole:
    type: String
    description: "The ARN of the role that allows Automation to perform
      the actions on your behalf."
  Duration:
    type: String
    description: (Optional) The duration of the attack in minutes (default=5)
    default: '5'
```

```
mainSteps:

# ---------------------------------------------------------------
# Query subnets attached to ASG. We will later match these to AZs
# for detaching and re-attaching operations
- name: DescribeAutoscaling
  action: aws:executeAwsApi
  onFailure: 'step:ExitList'
  onCancel: 'step:ExitList'
  timeoutSeconds: 60
  inputs:
    Service: autoscaling
    Api: DescribeAutoScalingGroups
    AutoScalingGroupNames:
        - "{{ AutoscalingGroupName }}"
  outputs:
    - Name: VPCZoneIdentifier
      Selector: "$.AutoScalingGroups[0].VPCZoneIdentifier"
      Type: String
    - Name: AvailabilityZones
      Selector: "$.AutoScalingGroups[0].AvailabilityZones"
      Type: StringList
    - Name: InstanceIds
      Selector: "$..InstanceId"
      Type: StringList

# ---------------------------------------------------------------
# Using ASG information, select subnets / AZs to remove from ASG
# and subnets / AZs to keep in ASG. This also makes an API call
# because the selection logic is more readable than using SSM
# JSONPATH / JMESPATH selectors.
- name: SubnetSelector
  action: aws:executeScript
  onFailure: 'step:ExitList'
  onCancel: 'step:ExitList'
  timeoutSeconds: 60
  inputs:
    Runtime: "python3.6"
    Handler: "script_handler"
    InputPayload:
      "vpcZoneIdentifier": "{{ DescribeAutoscaling.VPCZoneIdentifier }}"
      "affectAz": "{{ AvailabilityZone }}"
    Script: |
      import boto3
      client = boto3.client("ec2")
      def script_handler(events, context):
          asgSubnets = events.get("vpcZoneIdentifier","").split(",")
          affectAz = events.get("affectAz","")
          botoOut = client.describe_subnets(SubnetIds=asgSubnets).get("Subnets")
          affectSubnets  = [x["SubnetId"] for x in botoOut if
x["AvailabilityZone"] == affectAz]
          protectSubnets = [x["SubnetId"] for x in botoOut if
x["AvailabilityZone"] != affectAz]
          affectAzs      = [x["AvailabilityZone"] for x in botoOut if
x["AvailabilityZone"] == affectAz]
```

```
          protectAzs     = [x["AvailabilityZone"] for x in botoOut if
x["AvailabilityZone"] != affectAz]
          return {
              "SubnetIdArray": asgSubnets,
              "AffectSubnetsArray":  affectSubnets,
              "ProtectSubnetsArray": protectSubnets,
              "ProtectVpcZoneIdentifier": ",".join(protectSubnets),
              "AffectAzsArray":      affectAzs,
              "ProtectAzsArray":     protectAzs,
          }
  outputs:
    - Name: SubnetIds
      Selector: "$.Payload.SubnetIdArray"
      Type: StringList
    - Name: AffectSubnetsArray
      Selector: "$.Payload.AffectSubnetsArray"
      Type: StringList
    - Name: ProtectSubnetsArray
      Selector: "$.Payload.ProtectSubnetsArray"
      Type: StringList
    - Name: ProtectVpcZoneIdentifier
      Selector: "$.Payload.ProtectVpcZoneIdentifier"
      Type: String
    - Name: AffectAzsArray
      Selector: "$.Payload.AffectAzsArray"
      Type: StringList
    - Name: ProtectAzsArray
      Selector: "$.Payload.ProtectAzsArray"
      Type: StringList

# ---------------------------------------------------------------
# Remove subnets / AZs
- name: RemoveSubnets
  action: aws:executeAwsApi
  onFailure: 'step:Rollback'
  onCancel: 'step:Rollback'
  inputs:
    Service: autoscaling
    Api: UpdateAutoScalingGroup
    AutoScalingGroupName: "{{ AutoscalingGroupName }}"
    VPCZoneIdentifier: "{{ SubnetSelector.ProtectVpcZoneIdentifier }}"

# ---------------------------------------------------------------
# Wait in outage simulation state
- name: WaitForDuration
  action: 'aws:sleep'
  onFailure: 'step:Rollback'
  onCancel: 'step:Rollback'
  inputs:
    Duration: 'PT{{Duration}}M'

# ---------------------------------------------------------------
# Reset ASG subnets / AZs to original state before we started.
- name: Rollback
  action: aws:executeAwsApi
```

```
    onFailure: 'step:ExitList'
    onCancel: 'step:ExitList'
    inputs:
      Service: autoscaling
      Api: UpdateAutoScalingGroup
      AutoScalingGroupName: "{{ AutoscalingGroupName }}"
      VPCZoneIdentifier: "{{ DescribeAutoscaling.VPCZoneIdentifier }}"

  # ---------------------------------------------------------------
  # List state of ASG after all is done. Hopefully it's the same as
  # before we started.
  - name: ExitList
    action: aws:executeAwsApi
    timeoutSeconds: 60
    inputs:
      Service: autoscaling
      Api: DescribeAutoScalingGroups
      AutoScalingGroupNames:
          - "{{ AutoscalingGroupName }}"
    outputs:
      - Name: VPCZoneIdentifier
        Selector: "$.AutoScalingGroups[0].VPCZoneIdentifier"
        Type: String
      - Name: AvailabilityZones
        Selector: "$.AutoScalingGroups[0].AvailabilityZones"
        Type: StringList
      - Name: InstanceIds
        Selector: "$..InstanceId"
        Type: StringList
    isEnd: true

  outputs:
```

This SSM document requires an SSM role with the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableAsgDocument",
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:SuspendProcesses",
                "autoscaling:ResumeProcesses",
                "ec2:DescribeInstances",
                "ec2:DescribeInstanceStatus",
                "ec2:TerminateInstances"
            ],
            "Resource": "*"
        }
```

```
        ]
    }
```

From here follow the “Create FIS Experiment Template” step shown in FIS SSM Start Automation Setup to add this as an action to your FIS experiment.

Note that the above SSM document example limits itself to affecting the ASG and relying on the ASG to *cleanly* drain and remove instances from the LB. You can add extra steps to explicitly terminate instances and/or add NACLs to achieve more extreme failure scenarios on your instances.

Avoid: NACLs and SGs on their own

For EC2 instances in ASGs avoid the *exclusive* use Network Access Control Lists (NACLs) or security groups (SGs) as they will create untypical failure scenarios. In particular NACLs preventing access to an ASG or LB subnet will lead to churn when the ASG tries to spin up new instances and they fail to register as healthy.

If other aspects of your simulation require using NACLs or SGs we suggest combining them with the prevention Auto Scaling actions as described in the first workaround section above and/or with the removal of subnets from the ASG as shown in the second example.

OBSERVABILITY

A core aspect of chaos engineering is observability. In this section we will cover AWS tooling that supports FIS in providing observability for experiments and help in gaining the understanding needed to improve your system.

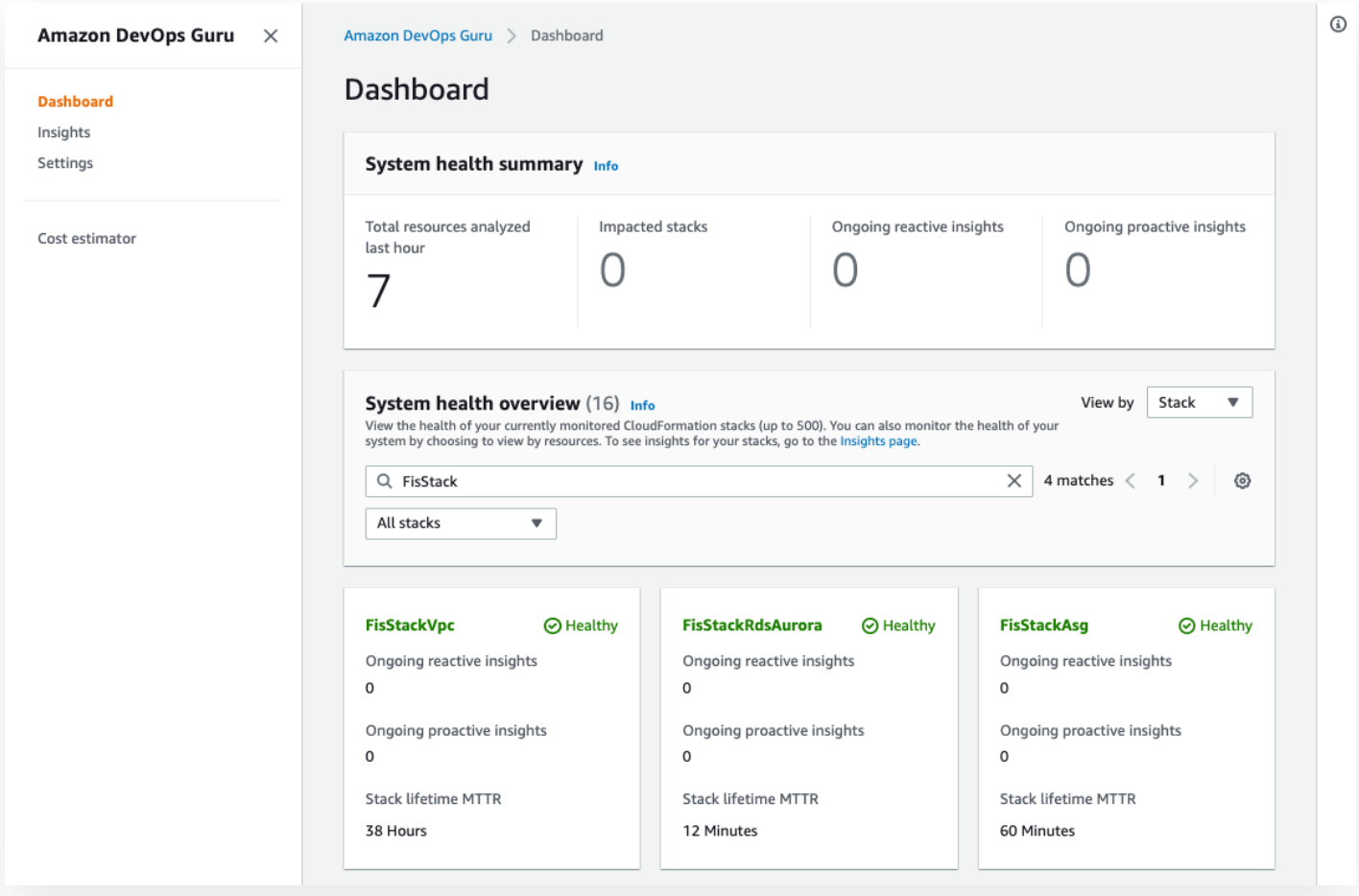
DEVOPS GURU

Warning

This section requires that you followed the setup instructions at the beginning of the workshop and allowed enough time for DevOps Guru to establish a baseline. This section also presumes that you followed the load generating steps in the **Synthetic user experience** section.

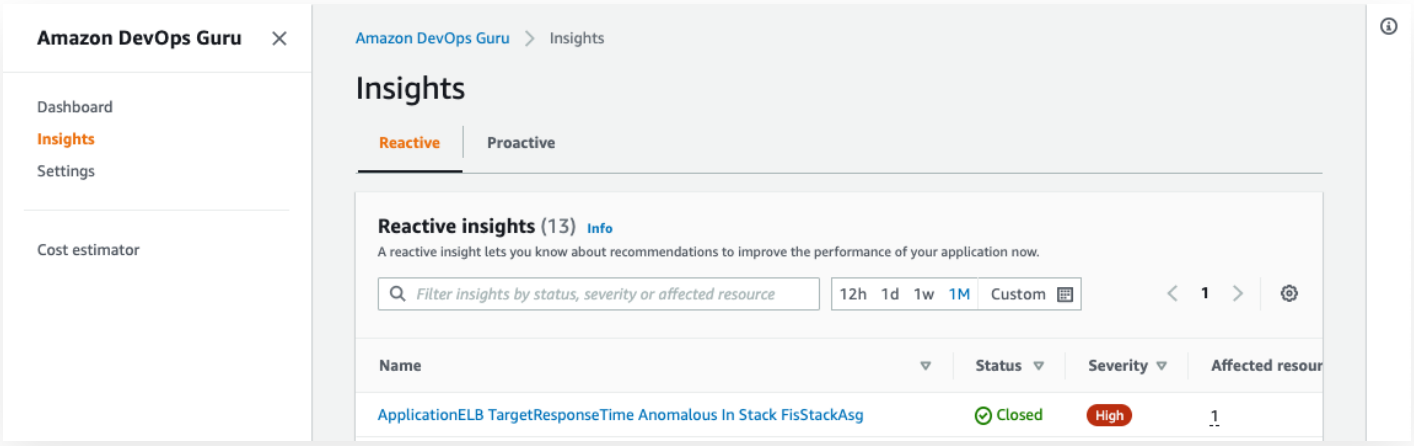
Dashboard overview

Navigate to the DevOps Guru console. Once enough time has passed for DevOps Guru to generate insights you should see a dashboard similar to this:



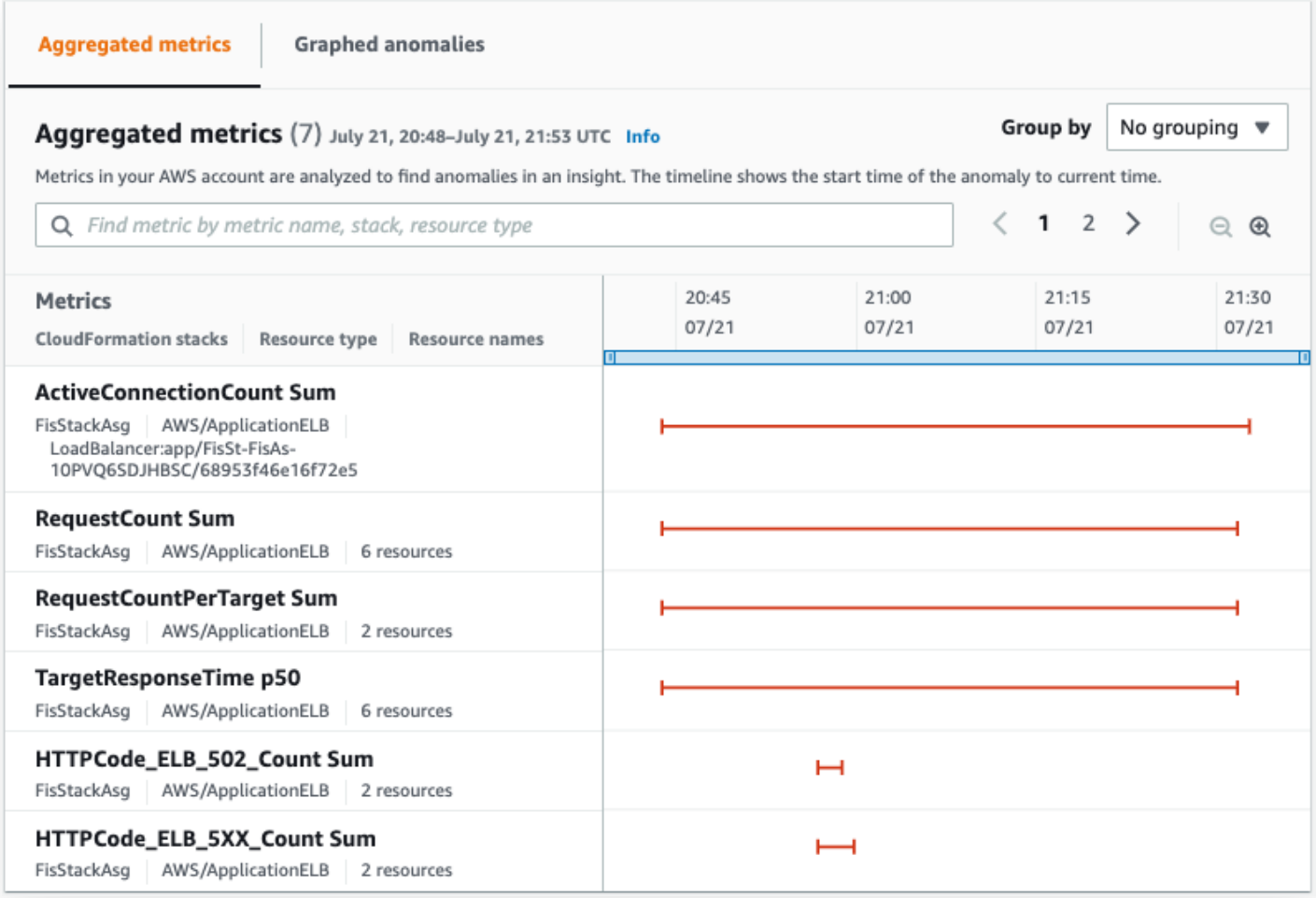
Reactive insights

Select “Insights” on the left and explore the reactive insights generated from our fault injection activities. You should see an event relating to “Application ELB” (depending on the exact order of events your dashboard may vary slightly):

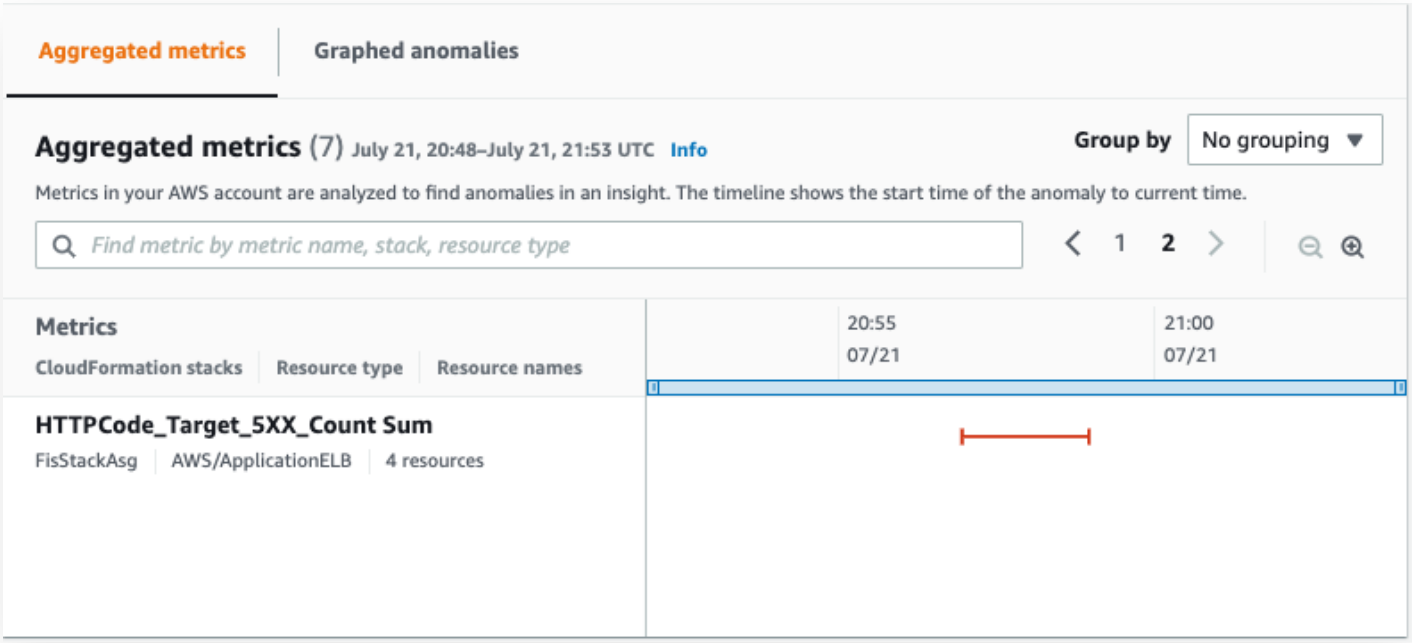


Visualizing anomalies

Selecting the event exposes more detailed information. The “Aggregated metrics” view will show timelines of different anomalous events that happened during the overall anomaly window:



Note that there may be multiple additional pages for additional events:



Examining the example above we see that during the event

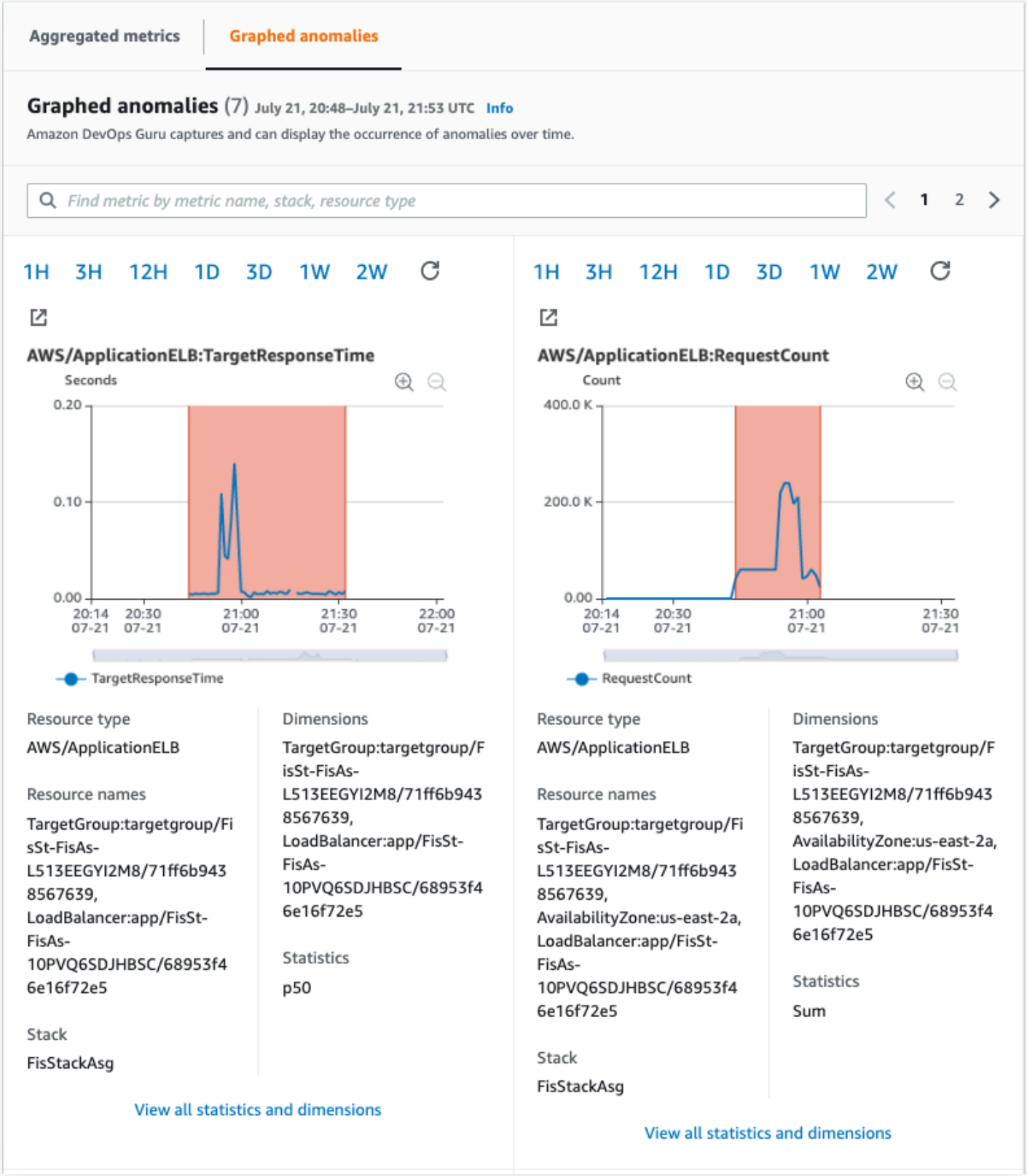
- an unusually high number of connections were made - by our external load testing tool,

- the high number of connections led to a high number of overall requests on the load balancer,
- the high number of connections led to a high number of connections to each target,
- the response time for the servers associated with the target increased substantially.

In addition to the expected direct impact of more connections, we also see unusual responses being sent:

- the number of HTTP 5xx errors increased at the load balancer,
- specifically the number of HTTP 502 error increased at the load balancer,
- the number of HTTP 5xx errors originated at the load balancer target, i.e. our web servers.

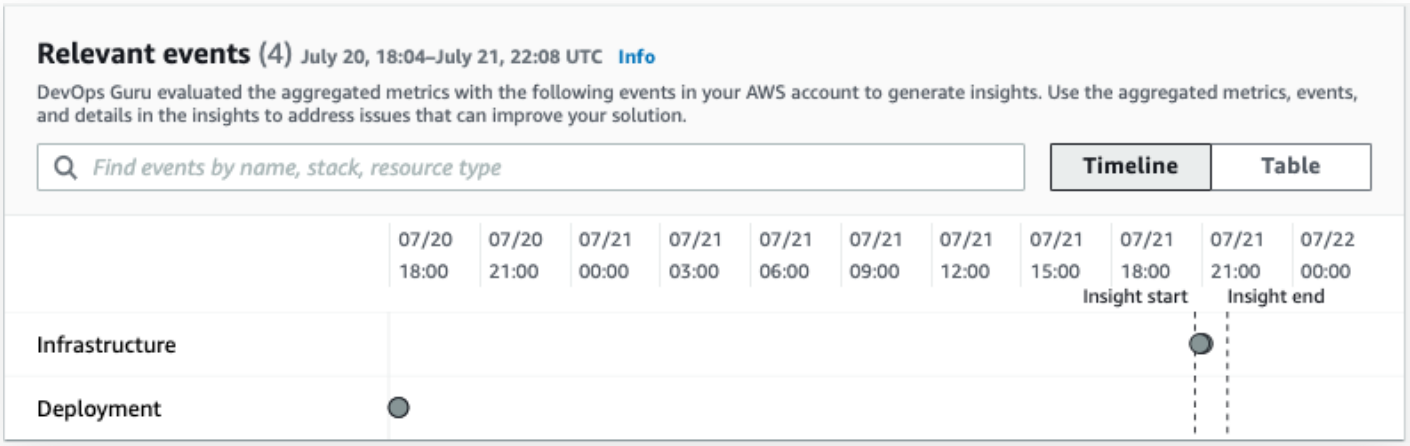
Switching to the “Graphed Anomalies” view shows the more detailed time data for each anomalous metric:



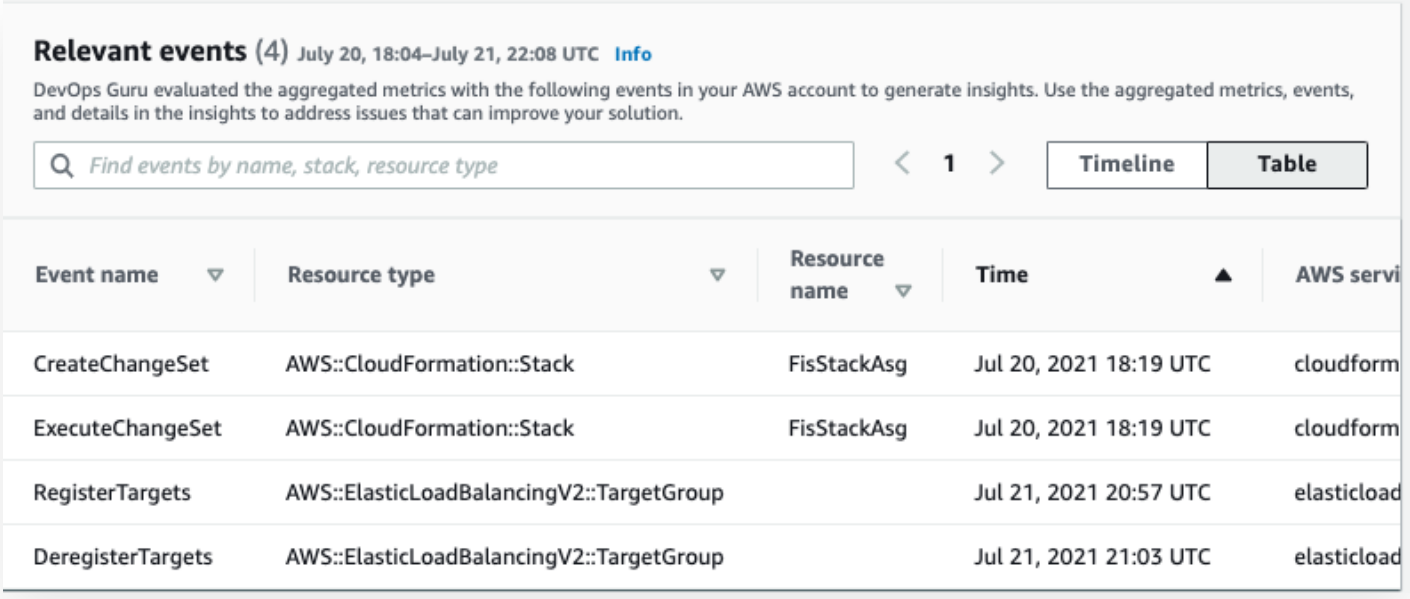
Note that in this view data outside the anomaly window are set to zero to allow focusing on the relevant details during the outage.

Contextualizing with infrastructure events

In our case the anomalies arose from external load but frequently anomalies are caused by changes to code or infrastructure configuration. To help you diagnose this, DevOps Guru provides visibility into deployment and infrastructure changes associated with the anomaly. These events can be visualized in a timeline view (you can get details by clicking on the dots):



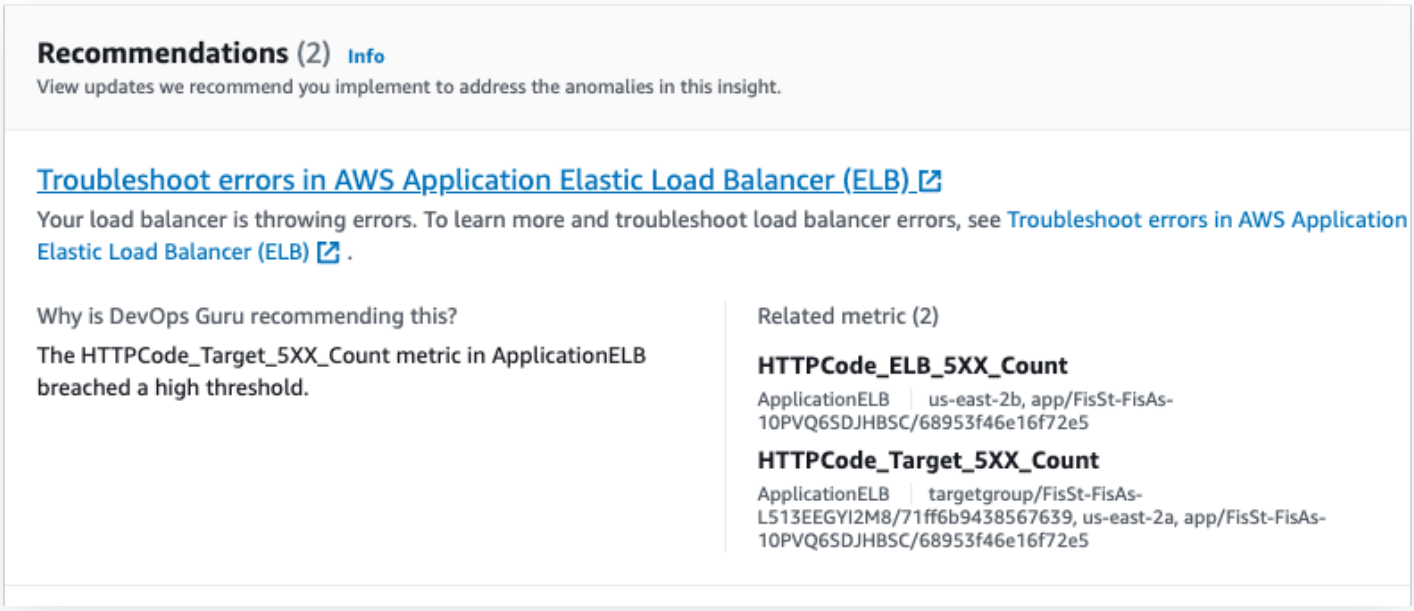
or in table format:



From the table format we can see that about 2h before the anomaly some changes were made to the stack configuration and deployed code. We can also see that around the time of the event instances were added to the load balancer in response to the increased load, and subsequently removed from the load balancer due to the external event subsiding.

Recommendations for improvement

Finally DevOps Guru provides “Recommendations”, links to relevant articles to help troubleshoot issues and improve overall system performance:



Further reading

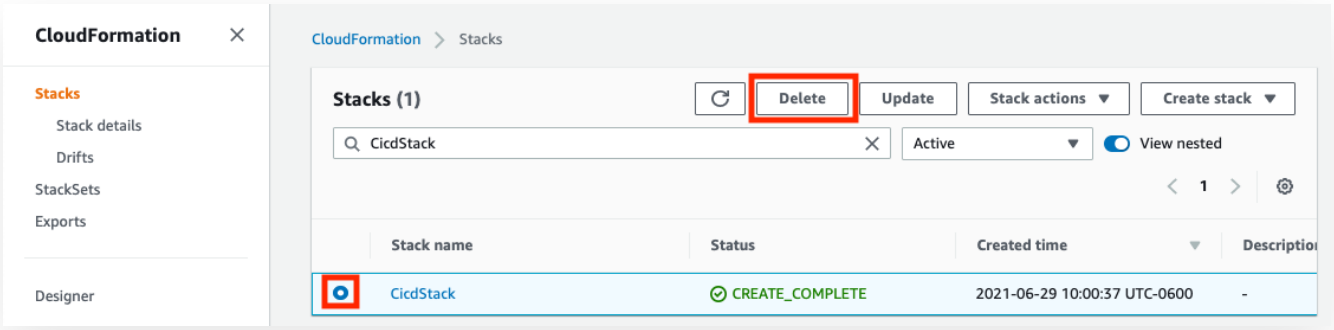
To learn more about DevOps Guru, see the documentation, and explore using DevOps guru on serverless infrastructure as well as larger deployment strategies.

CLEANUP

To ensure you don't incur any further costs after the workshop, please follow these instructions to delete the resources you created.

Manually

- If you created the CI/CD stack and ran the pipeline, first start by deleting the insfrastructure provisioned by the pipeline:
    - Navigate to the AWS CloudFormation console and find the stack named `fisWorkshopDemo`
    - Select the stack
    - Select "Delete"
- Once, the `fisWorkshopDemo` is deleted, following the same procedure as above, delete the `CicdStack` stack



- If you created the `CpuStress` stack in the **AWS Systems Manager Integration** section, delete it following the same procedure.
- Following the same procedure as above, delete the following stacks
    - `FisStackEks`
    - `FisStackEcs`
    - `FisStackRdsAurora`
    - `FisStackLoadGen`
    - `FisStackAsg`

- `FisStackVpc`

- Delete the CloudWatch log groups:
  - Navigate to the AWS CloudWatch console
  - Search for `fis-workshop`
  - Select the checkboxes
  - Under “Actions” select “Delete log group(s)”

- Delete Cloud9 Environments
  - Navigate to the AWS Cloud9 console
  - Delete the Cloud9 environment that you use during the workshop

Using a script

In your cloud9 terminal where you performed the **Provision AWS resources** step run the following commands:

```
cd ~/environment
```

```
cd aws-fault-injection-simulator-workshop
cd resources/templates
./cleanup.sh
```

Retained resources

CloudWatch metrics and FIS experiments will be retained until the end of their respective expiration periods.