INTRODUCTION

This workshop provides an introduction to chaos engineering using Amazon Web Services (AWS) tooling, with a focus on AWS Fault Injection Simulator (FIS). It introduces the core elements of chaos engineering:

- form a hypothesis (plan),
- introduce stress (do),
- observe (check), and
- improve (act).

You will learn how to use FIS and other AWS tools to inject faults in your 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 practices 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 a 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

Core sections

For an introductory workshop we recommend the following core sections:

- Baselining and Monitoring
- Synthetic User Experience
- First Experiment > Configuring Permissions
- First Experiment > Experiment (Console)
- AWS Systems Manager Integration > FIS SSM Send Command Setup

- AWS Systems Manager Integration > Linux CPU Stress Experiment
- AWS Systems Manager Integration > Working with SSM documents
- AWS Systems Manager Integration > Optional - Windows CPU Stress Experiment
- AWS Systems Manager Integration > FIS SSM Start Automation Setup
- AWS Systems Manager Integration > SSM Additional resources
- Databases > RDS DB Instance Reboot

When run in a prepared AWS account these core sections of the workshop will take about 2-3h. When run in a customer account, deploying the workshop's core infrastructure will require an additional 45min.

Additional sections

All remaining sections are intended as independent modules that can be added based on customer need and interest. All sections require the roles created in

- First Experiment > Configuring Permissions
- AWS Systems Manager Integration > FIS SSM Start Automation Setup

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 links: depending on whether you are…

- **Running the workshop in your own account**
- **Running in an AWS provided account (using AWS provided hashes)**

Once you have completed one of the setup paths above, 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**.

Next step:

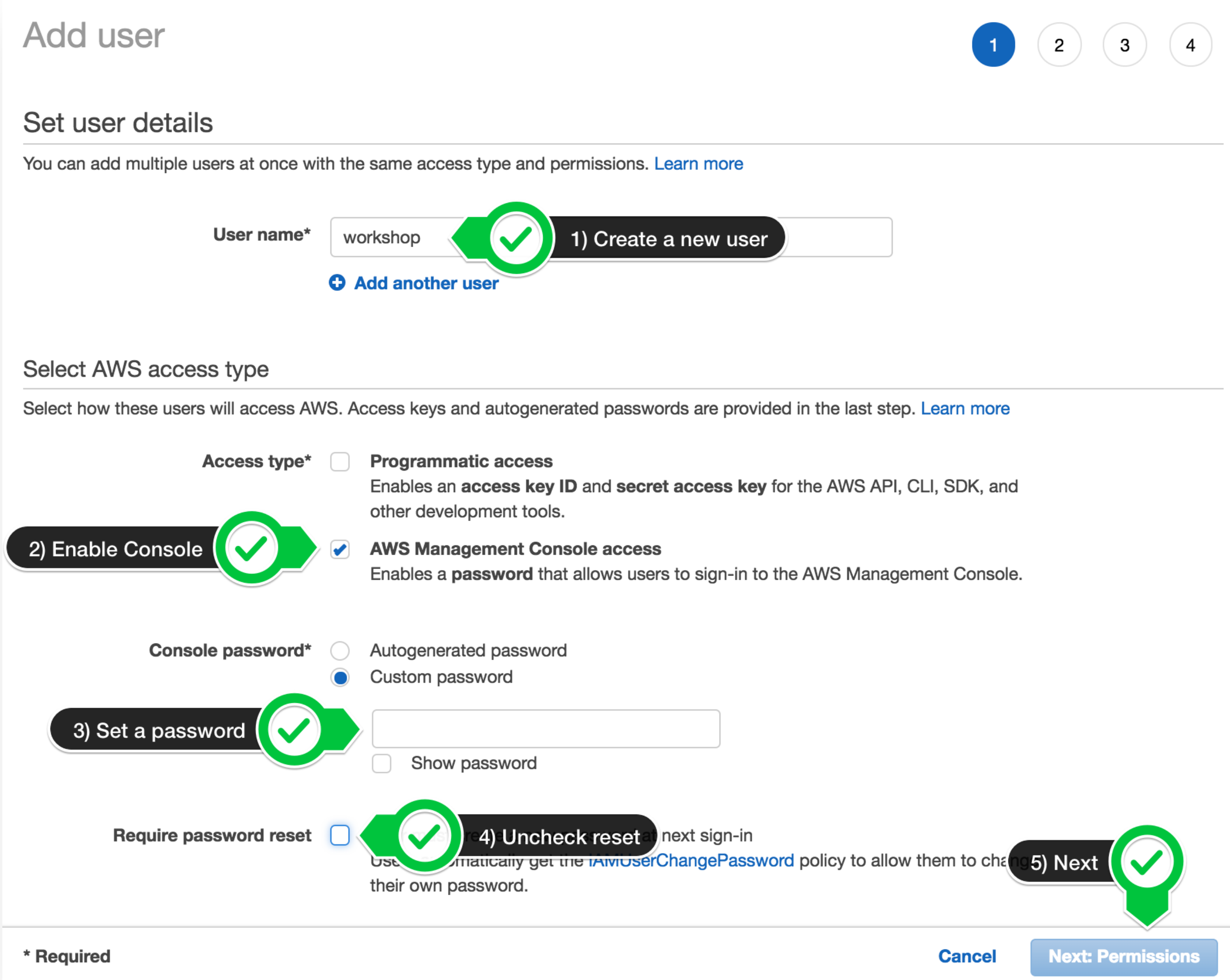
- Create an AWS account
- Region selection
- Create a Workspace
- Provision AWS resources

CREATE AN AWS ACCOUNT

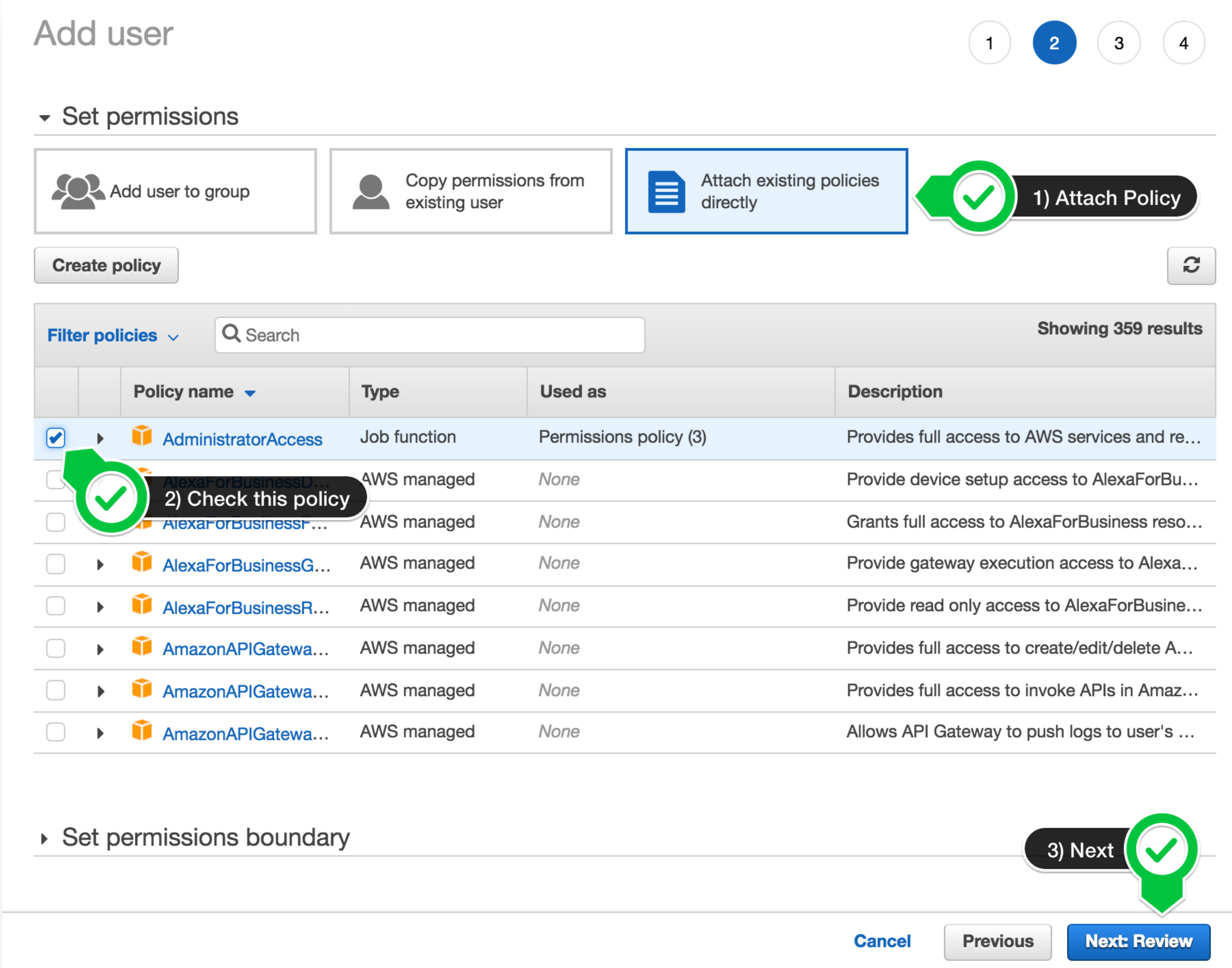
Warning

Your account must have the ability to create new AWS Identity and Access Management (IAM) roles and scope other IAM permissions.

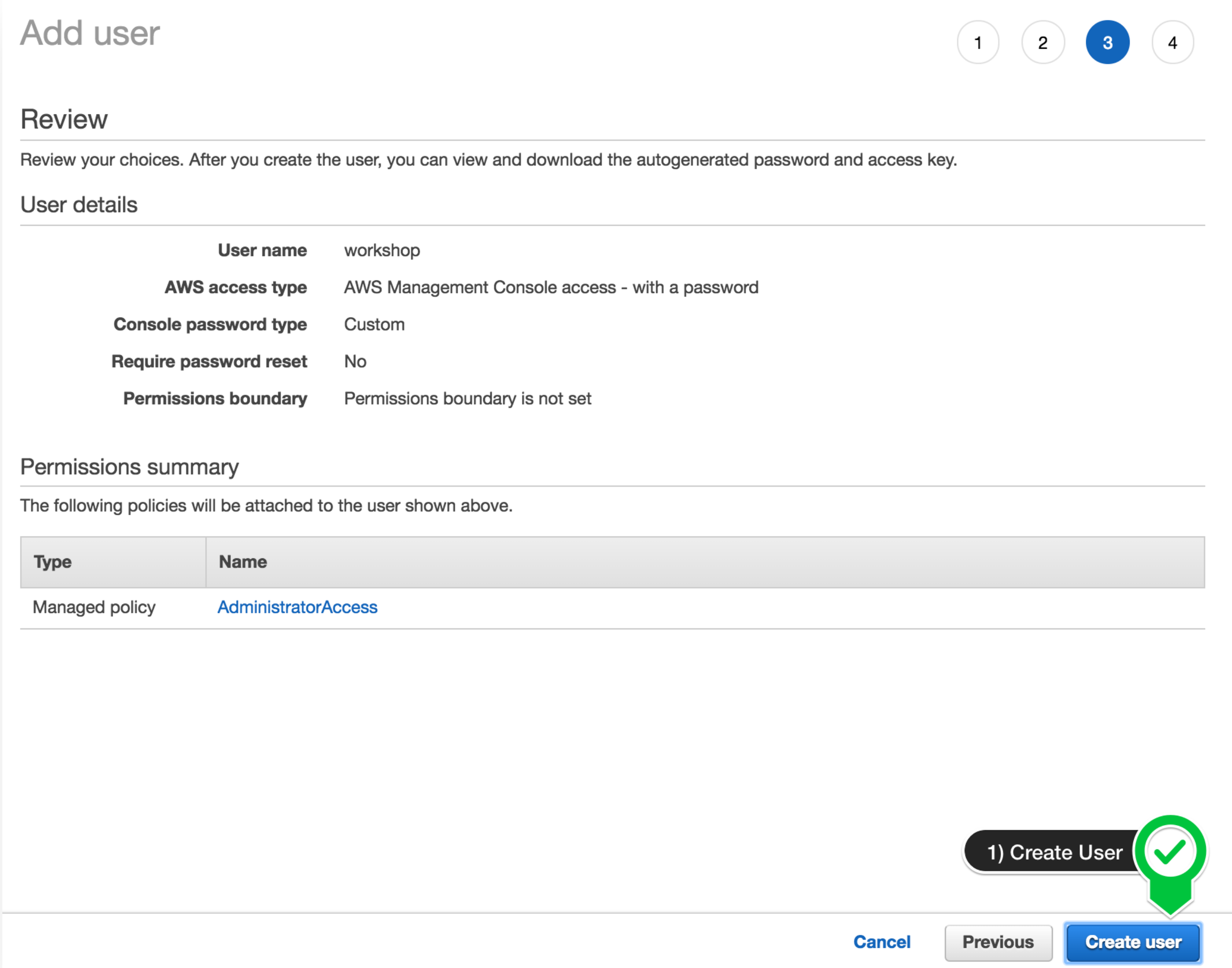
1. If you don't already have an AWS account with **"Administrator"** access: **Create an AWS account by clicking here**.
2. Once you have an AWS account, ensure you are following the remaining workshop steps as an IAM user with administrator access to the AWS account: **Create a new IAM user to use for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



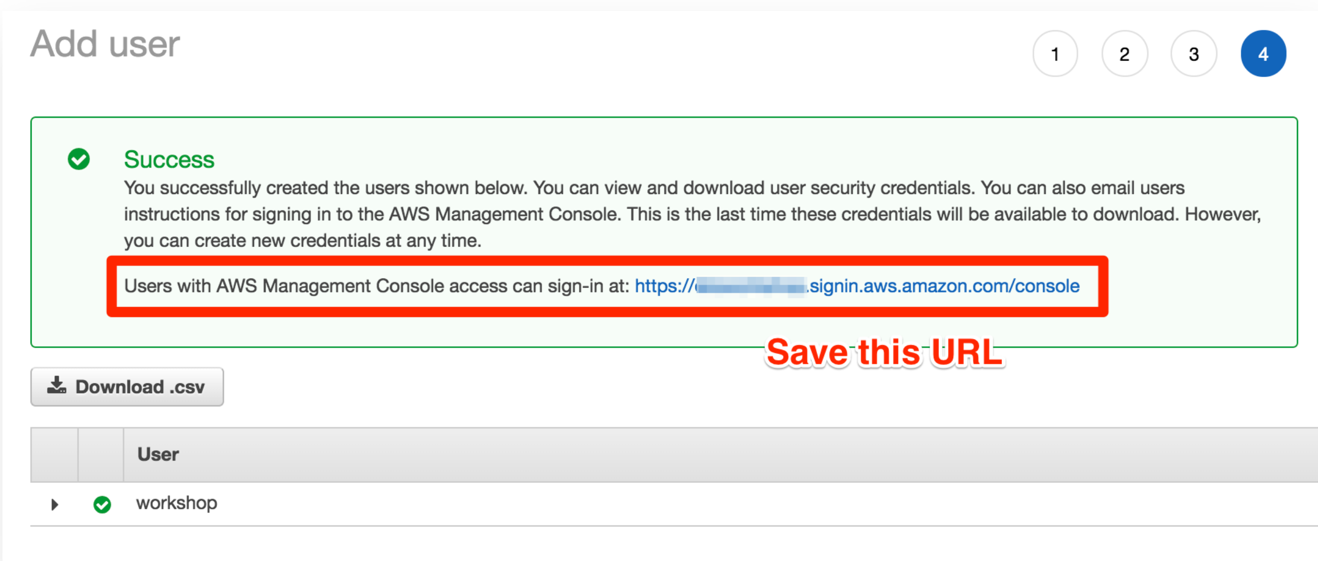
4. Attach the **"AdministratorAccess"** IAM Policy:



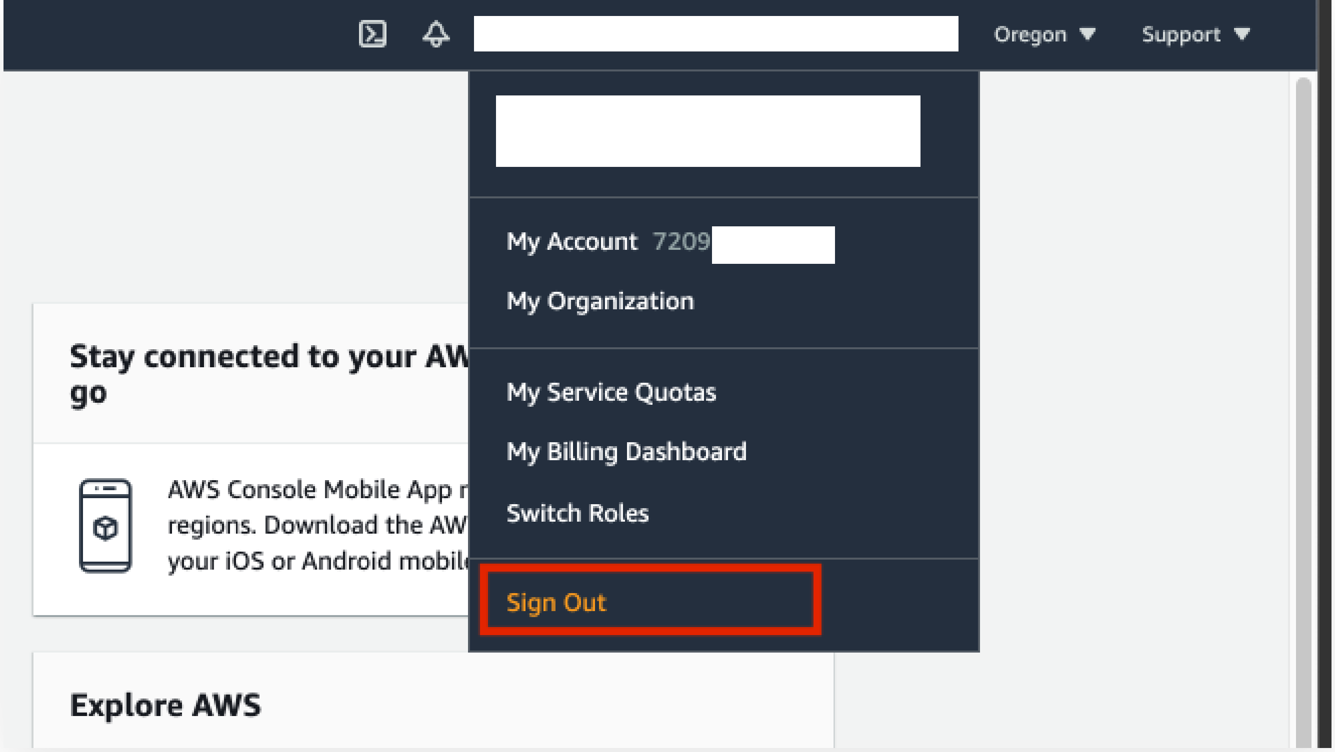
5. Select **"Create user"**:



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



7. Sign out of your current AWS Console session: on the top menu, select your login and select **"Sign out"**



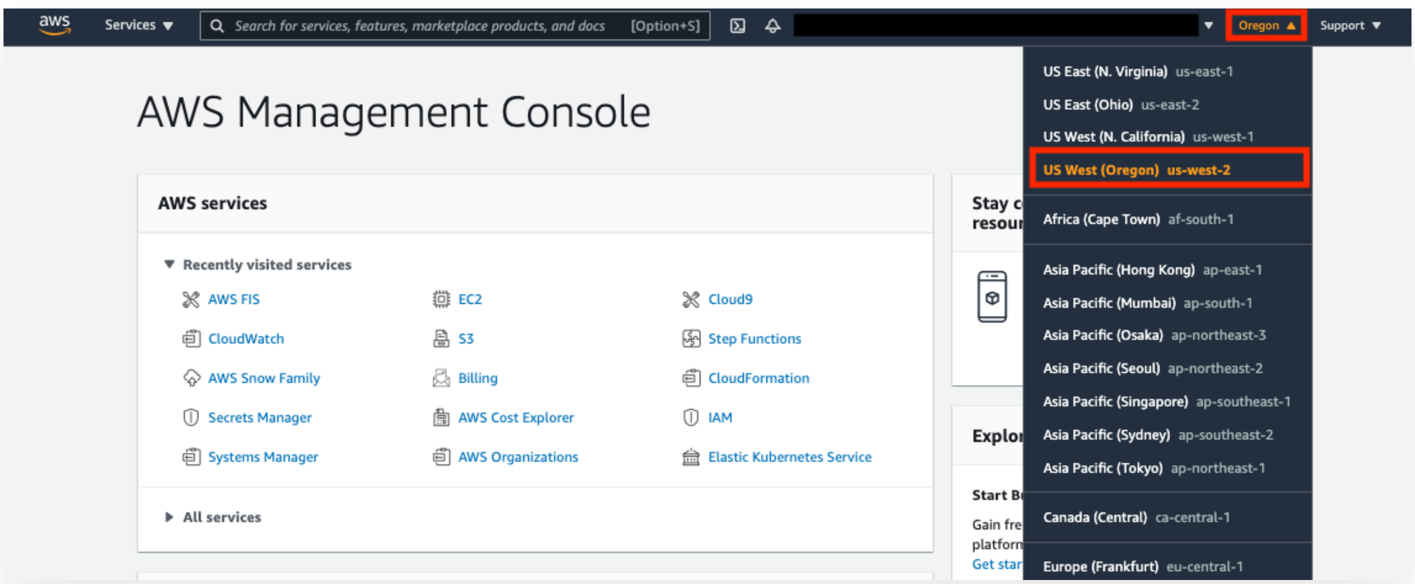
8. Sign in to a new AWS Console session by using the sign-in URL saved and the newly created user credentials.

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

REGION SELECTION

This workshop relies heavily on AWS Fault Injection Simulator (FIS) and assumes you will be running all your experiments in the same region. If not otherwise instructed, please **choose a region** in which FIS is currently available.

To select this region for navigate to the **AWS Console** and select the desired region from the drop-down menu on the top right:



Note

A number of services, in particular AWS Identity and Access Management (IAM), are region independent and will show **"Global"** as the selection.

CREATE A WORKSPACE

Info

A list of supported browsers for AWS Cloud9 (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 specific domains by following **these instructions**.

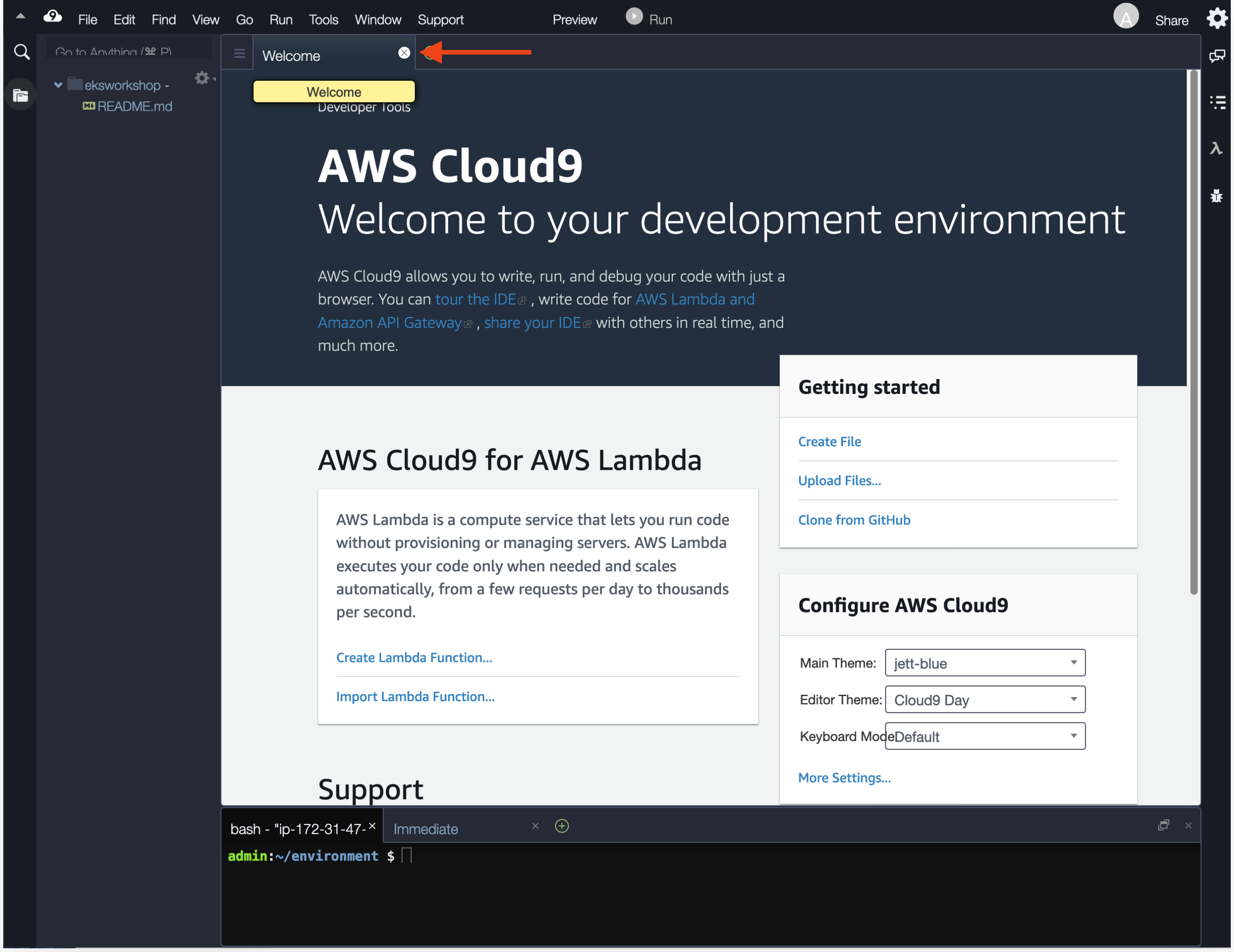
Launch Cloud9 in the region selected previously

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

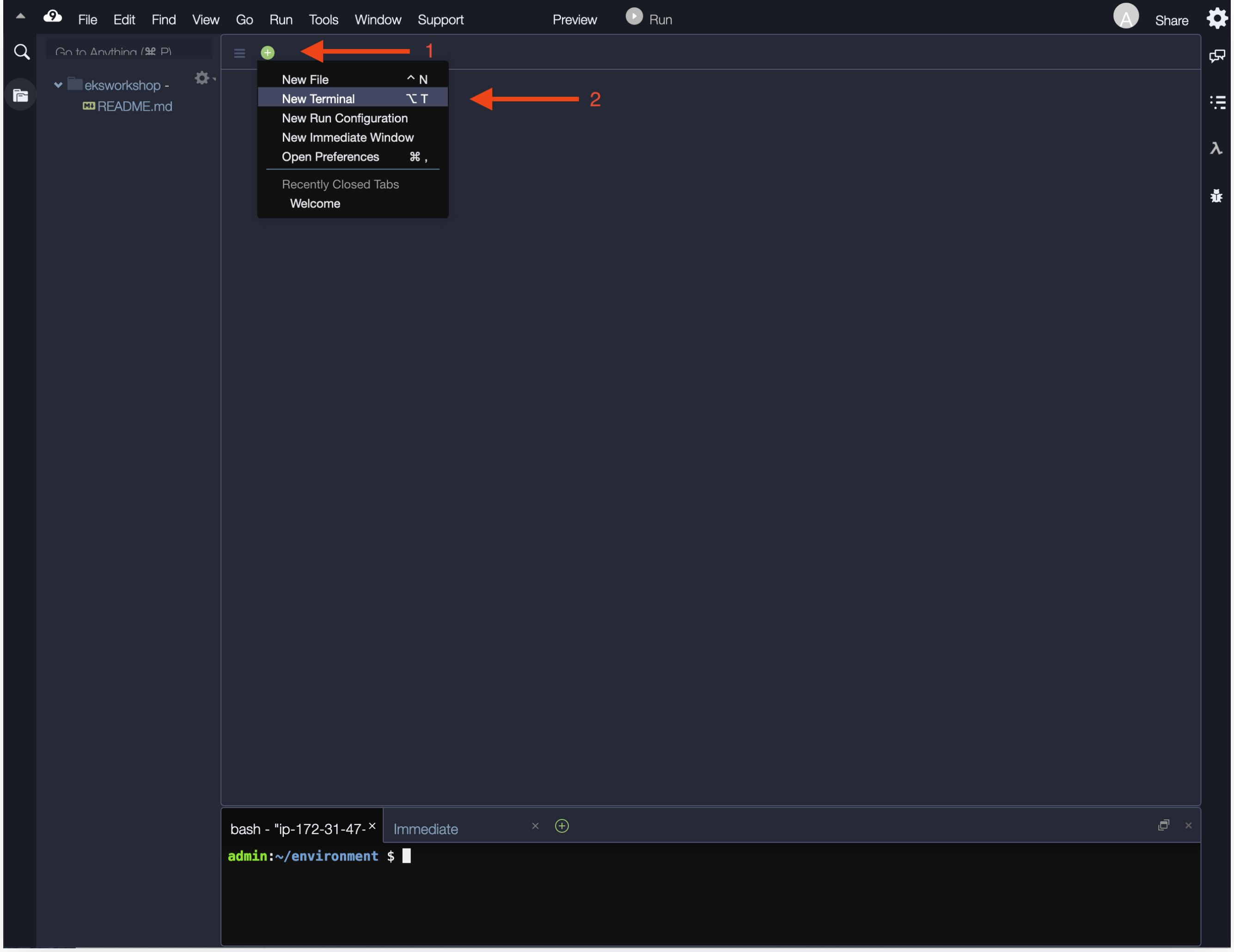
- Select **Create environment**
- Name it `fisworkshop` and select **Next step**.
- Since we only need to access our Cloud9 environment via web browser, please select the **Create a new no-ingress EC2 instance for environment (access via Systems Manager)** under the Environment Type.
- Select “Other Instance Types” and choose `t3.medium` (you can type to search) for instance type, go through the wizard with the default values. Finally select **Create environment**

When it comes up, customize the environment by:

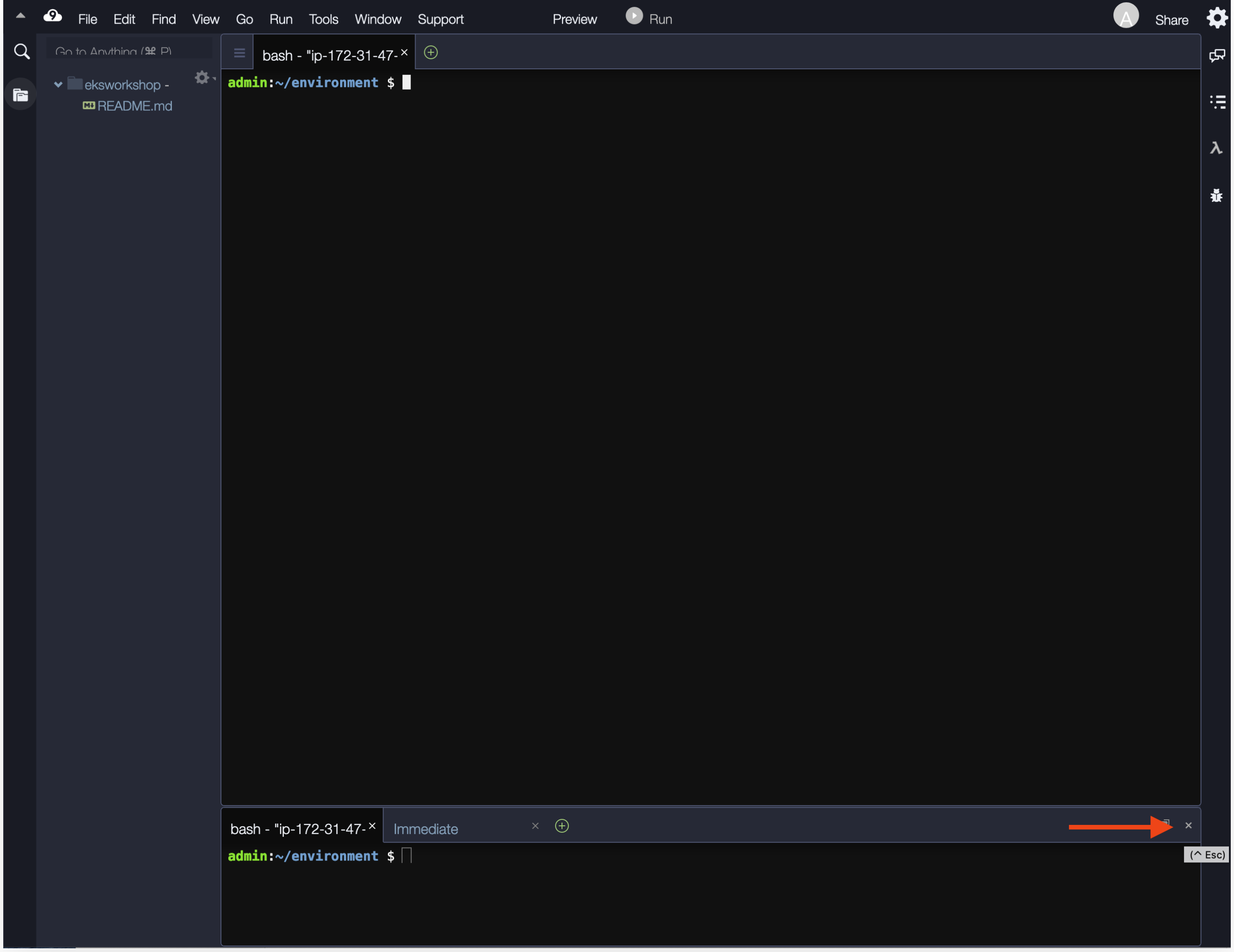
- Closing the **Welcome tab**



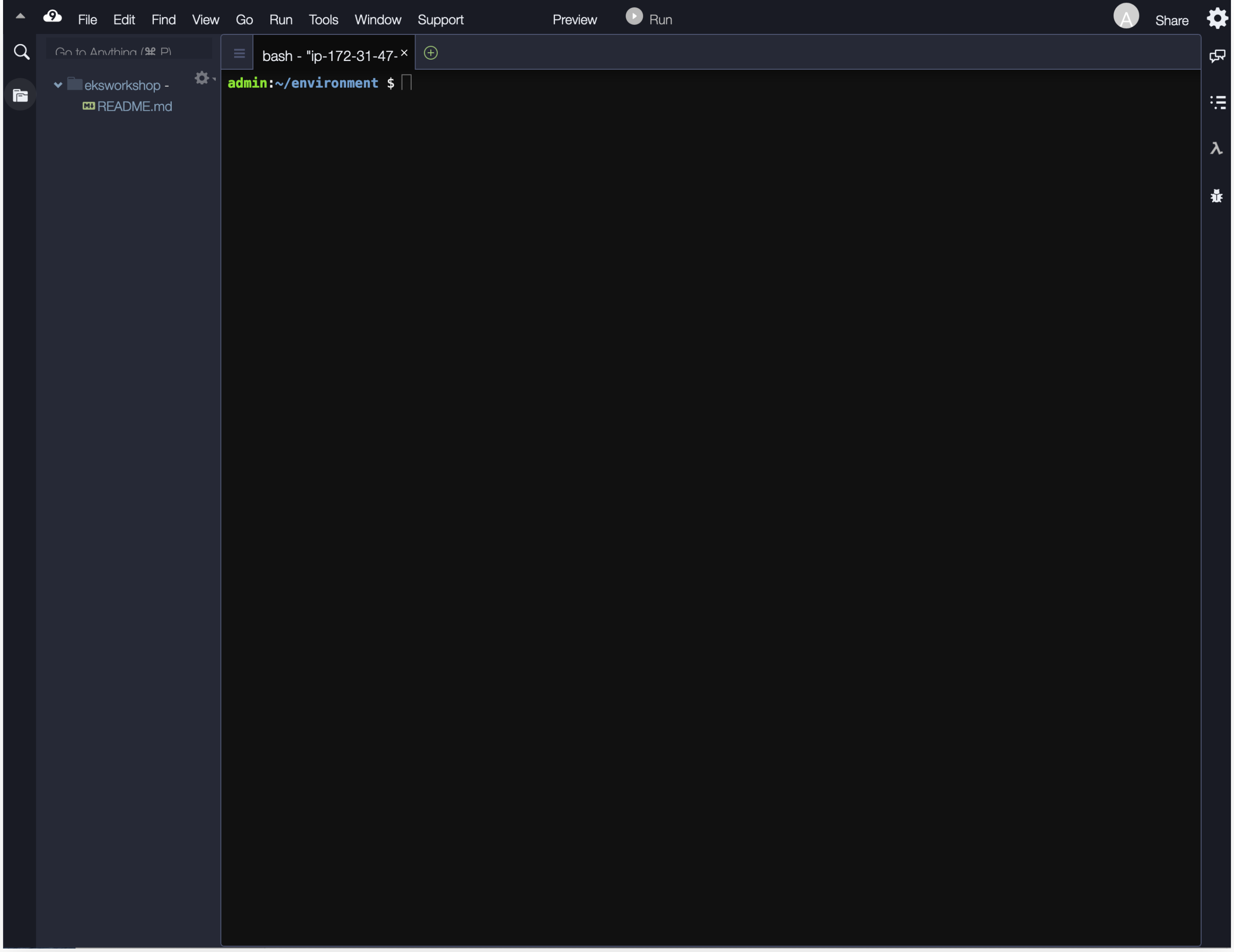
- Opening a new **Terminal** tab in the main work area



- Closing the lower work area



Your workspace should now look like this



Increase the disk size on the Cloud9 instance

Info

Some commands in this workshop require more than the default disk allocation on a Cloud9 workspace. The following command adds more disk space to the root volume of the Amazon EC2 (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 Integrated Development Environment (IDE) to come back online.

```
# Ensure we have newest boto3 installed
pip3 install --user --upgrade boto3

# Identify instance ID of the Cloud9 environment
export instance_id=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
```

```
# Use API to identify attached volume and increase size
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))"

# Reboot - on restart the cloud-init will adjust FS size
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 an AWS 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 --lts
nvm use --lts

# Install typescript
```

```
npm install -g typescript

# Install CDK
npm install -g aws-cdk

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

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
```

Note

Instantiating all resources will take about 30 minutes. This might be a good time to read ahead at **Baselining and Monitoring** or go for coffee.

Review the deploy output. It should similar to this:

```
Substack vpc SUCCEEDED
Substack goad-cdk SUCCEEDED
Substack access-controls SUCCEEDED
```

```
Substack serverless SUCCEEDED
Substack rds SUCCEEDED
Substack asg-cdk SUCCEEDED
Substack eks SUCCEEDED
Substack ecs SUCCEEDED
Substack cpu-stress SUCCEEDED
Substack api-failures SUCCEEDED
Substack spot SUCCEEDED
Overall install SUCCEEDED
```

If any of the substacks report as `FAILED` you can try to re-run the deployment script. If that still fails you can find some debugging information in files named `deploy-output.*.txt`.

...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**.

Next step:

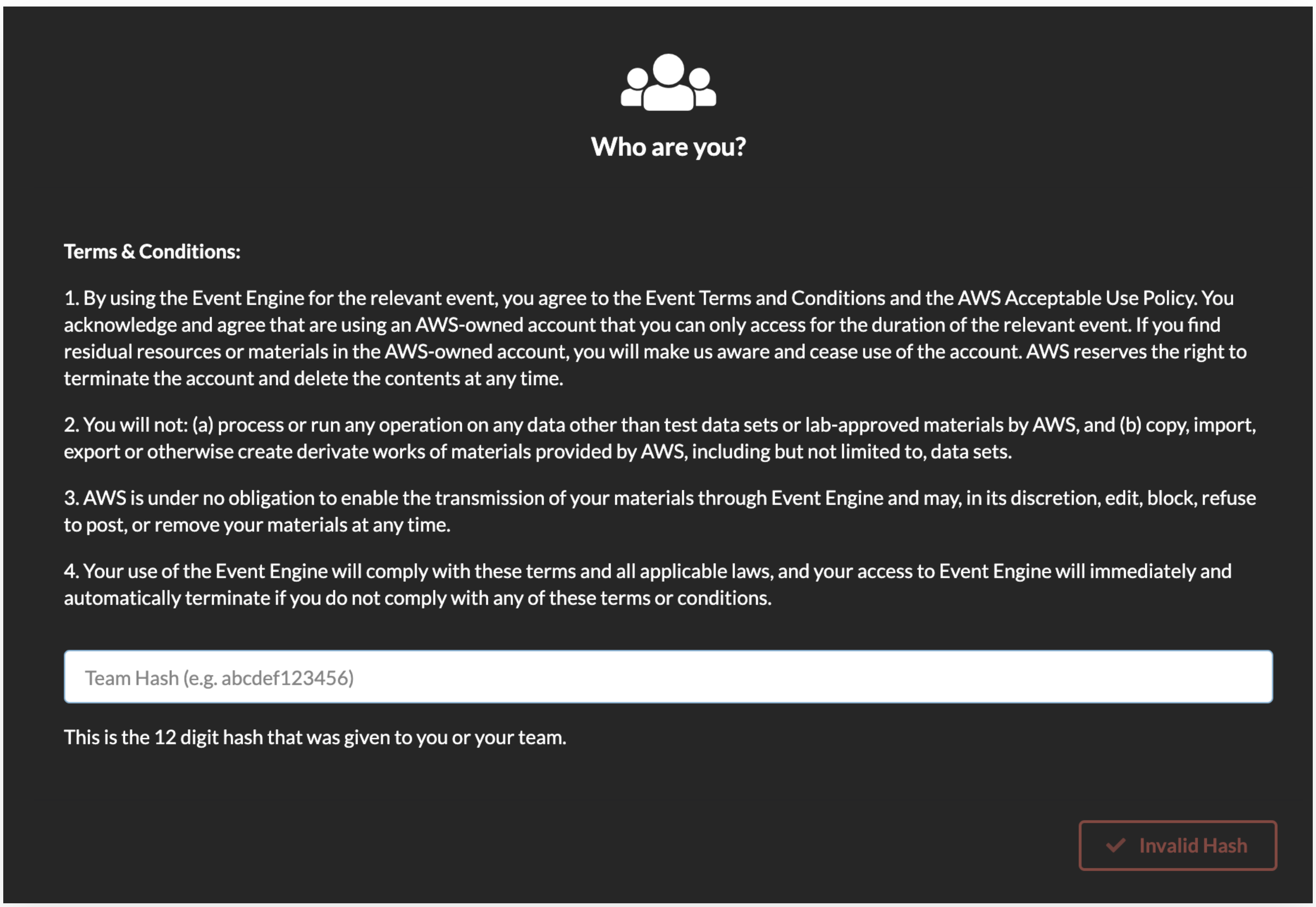
- AWS Workshop Portal
- Configure AWS CloudShell

AWS WORKSHOP PORTAL

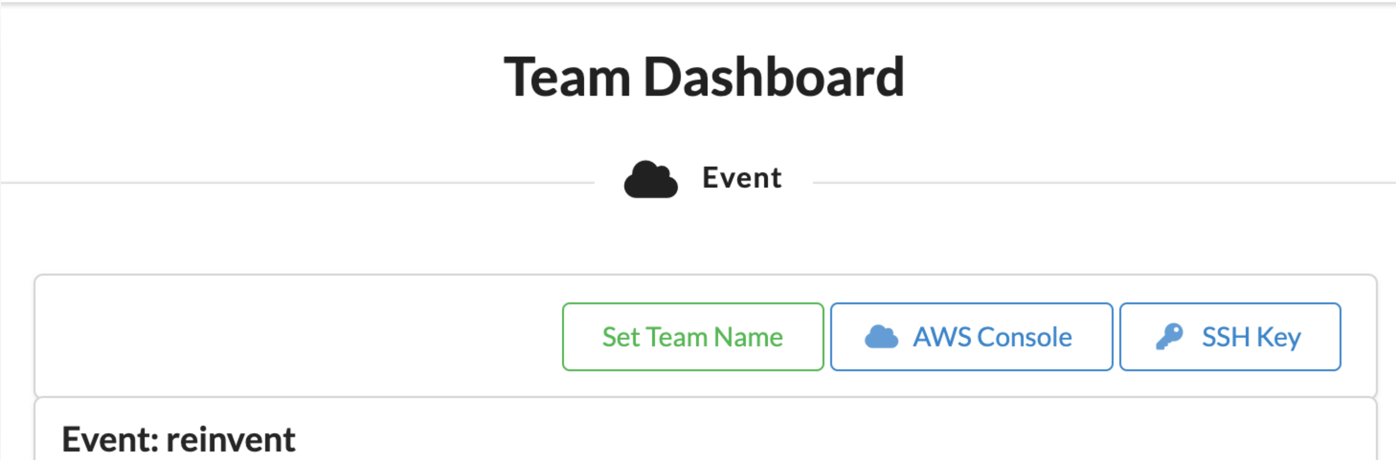
Login to AWS Workshop Portal

This workshop uses an AWS account and a AWS Cloud9 environment. You will need the **Participant Hash** provided by the event organizers and your email address to track your unique session.

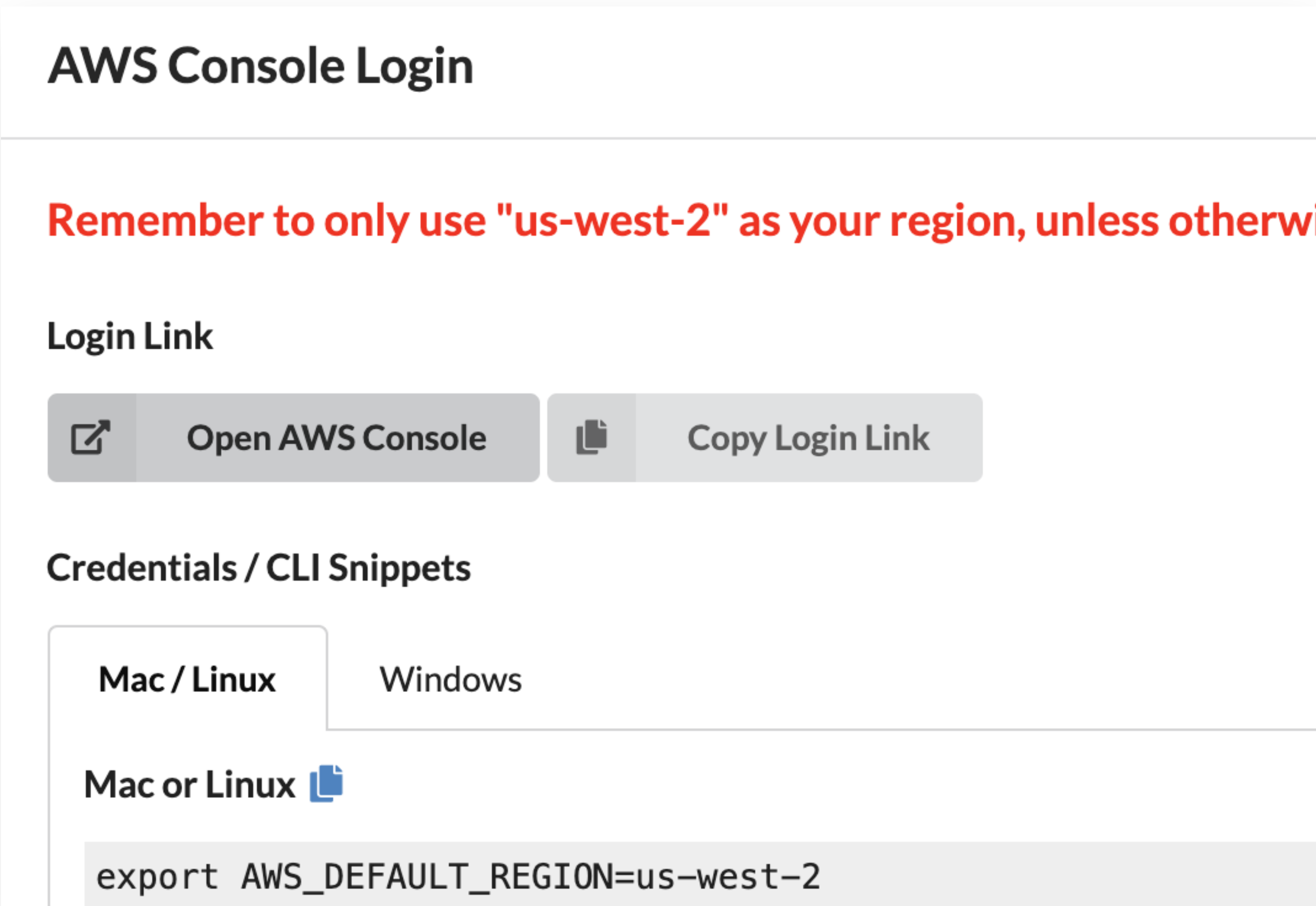
Connect to the portal by following instructions sent by the organizers or by browsing to **https://dashboard.eventengine.run/**. You should see the following screen:



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



Select **AWS Console** on dashboard.



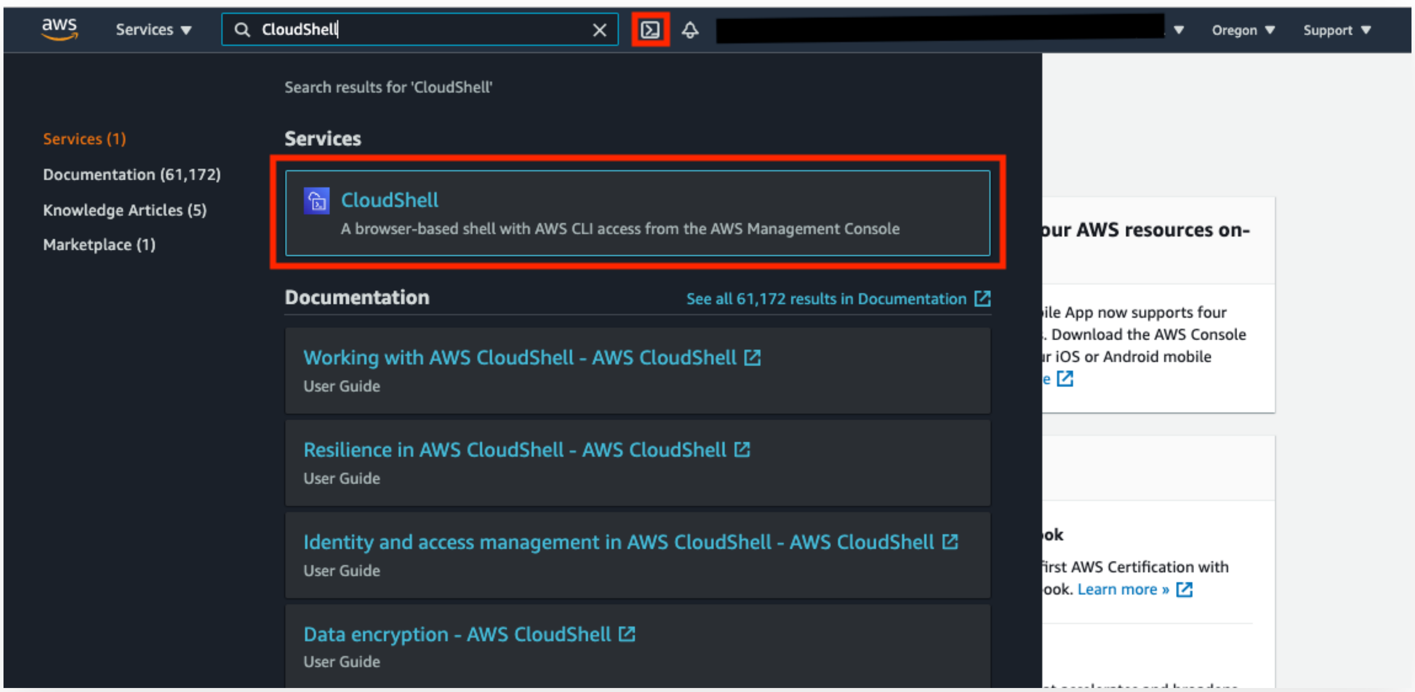
Keep the defaults and select **Open AWS Console**. This will open AWS Console in a new browser tab.

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

CONFIGURE AWS CLOUDSHELL

While it is possible to do this workshop from your desktop, the instructions in this workshop will assume that you are using AWS CloudShell (AWS events) or AWS Cloud9 (in your own account).

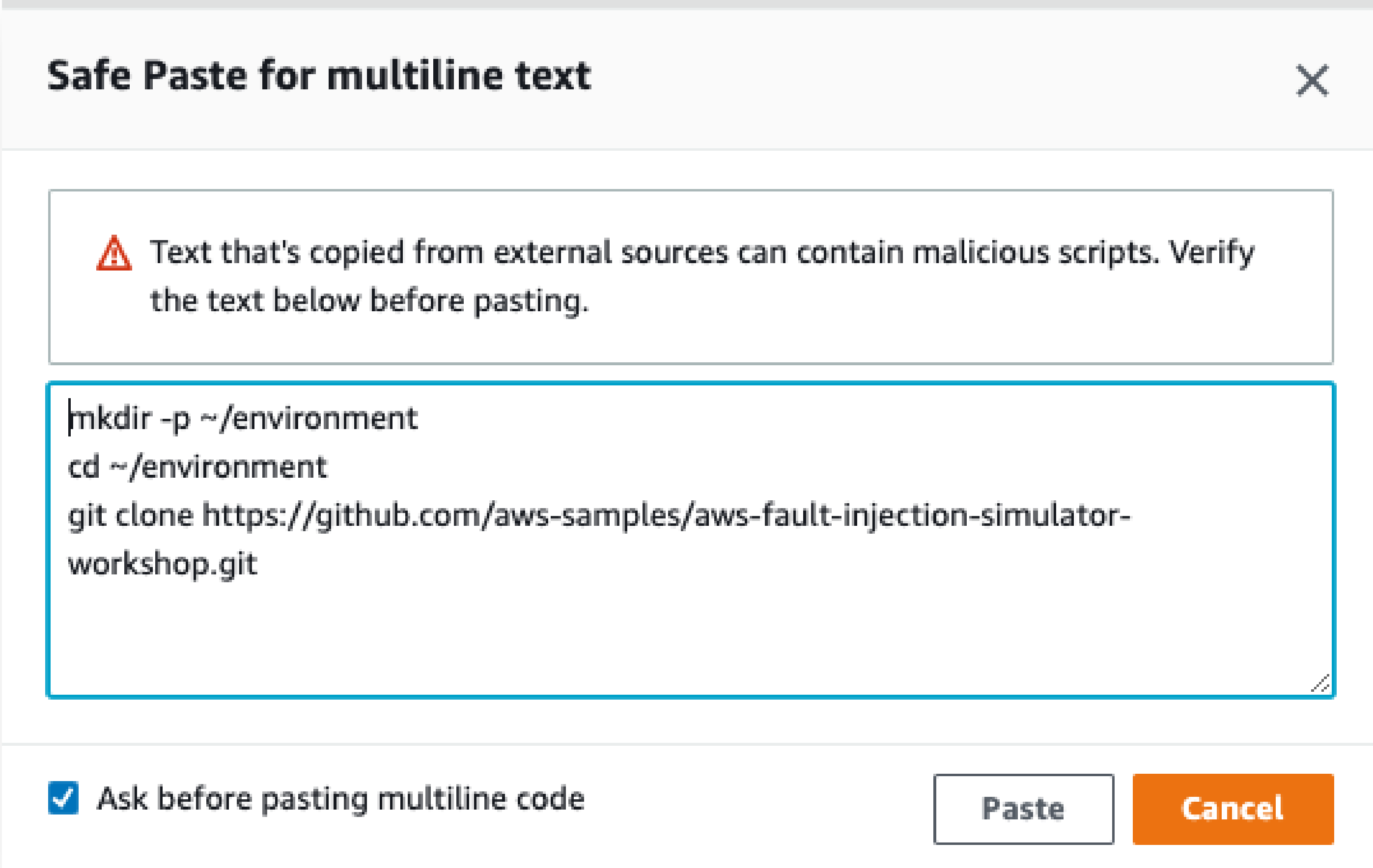
To open CloudShell, navigate to the AWS console and either search for “CloudShell” or click on the CloudShell icon in the menu bar:



Once the CloudShell terminal opens, we need to check out the GitHub repository. Paste the following into your CloudShell:

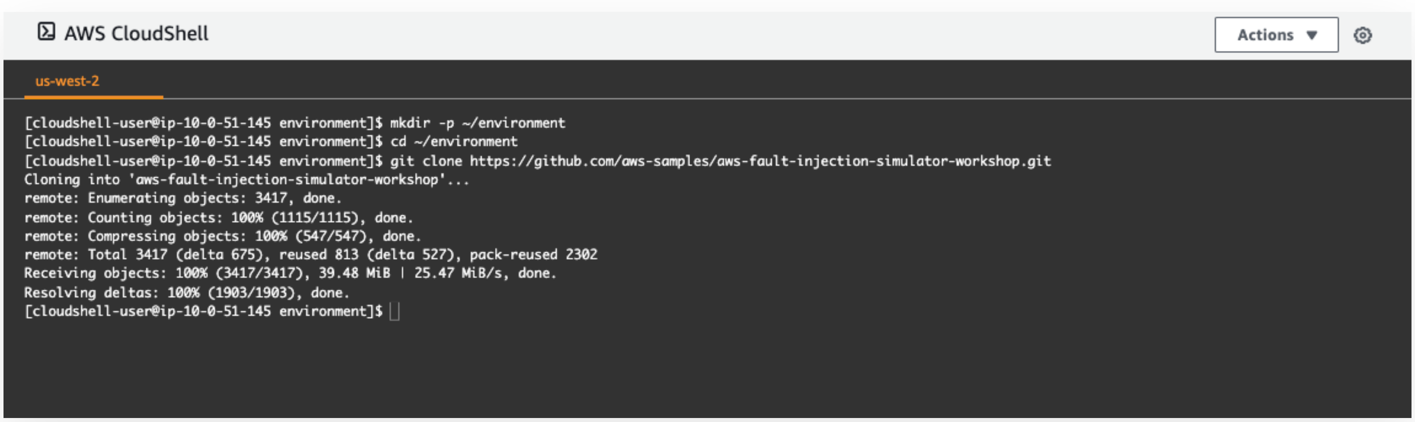
```
mkdir -p ~/environment
cd ~/environment
git clone https://github.com/aws-samples/aws-fault-injection-simulator-workshop.git
```

If this is this first time you are using CloudShell you may receive a dialog box asking to confirm a multi-line paste:



Optionally uncheck the “Ask before pasting multiline code” checkbox. Then select “Paste”.

You should see a git clone like this:



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 CloudShell:

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

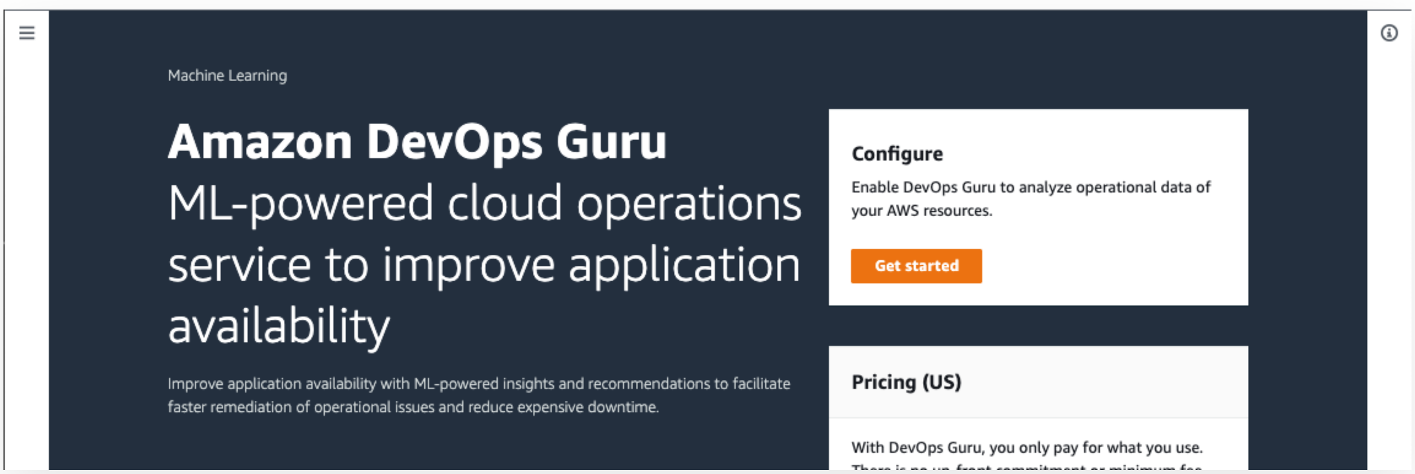
```
# Update to the latest stable release of npm and nodejs.
sudo npm install -g stable

# Install typescript
sudo npm install -g typescript

# Install CDK
sudo npm install -g aws-cdk

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

OPTIONAL: SETUP FOR AMAZON DEVOPS GURU

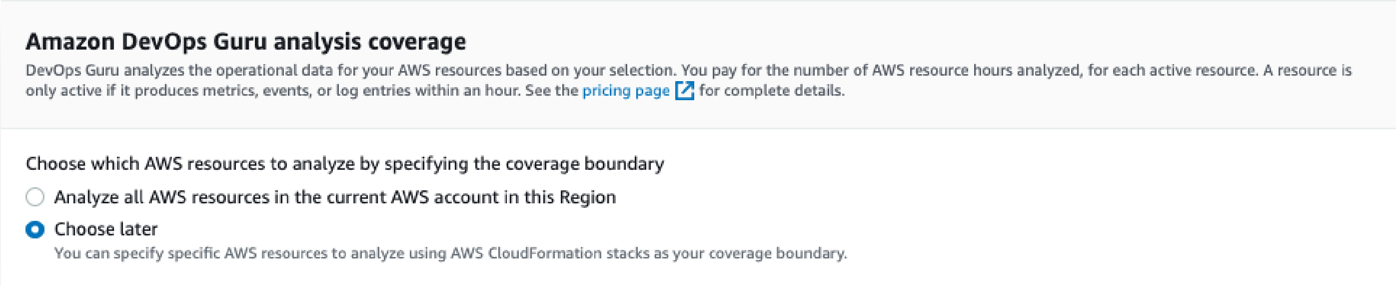
Warning

Only complete this section if you are planning to explore the Amazon DevOps Guru (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 select the **"Get Started"** button:



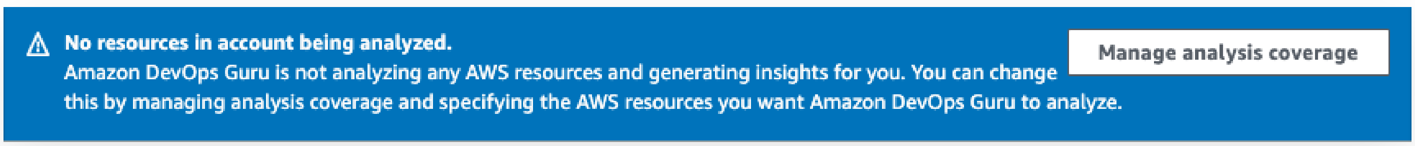
For "Amazon DevOps Guru analysis coverage" select **"Choose later"** if you will only be exploring as part of this workshop. Otherwise you can select "Analyze all AWS resources in the current AWS account in this Region" but it may take more time and incur more cost to get started.



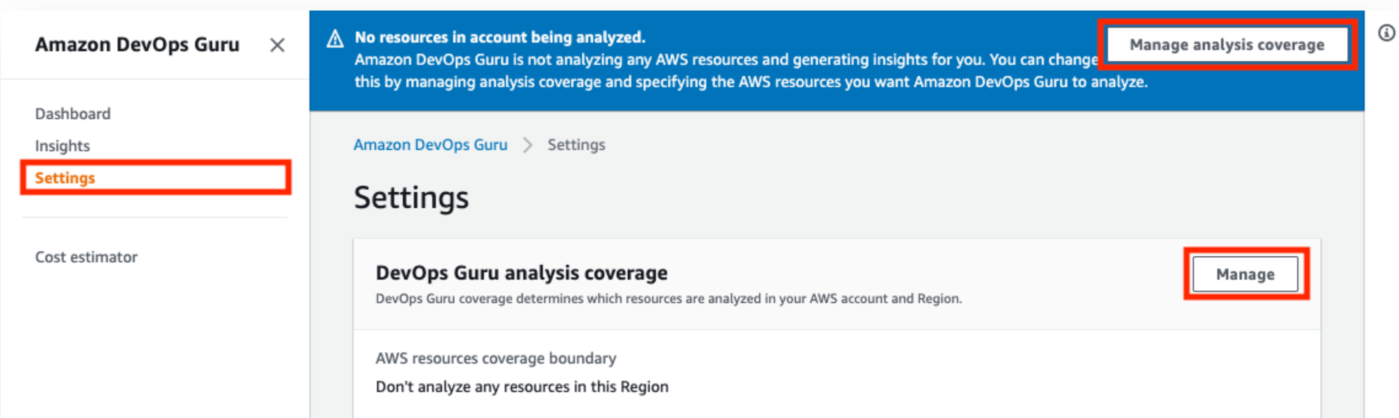
During this workshop we will not be exploring Amazon Simple Notification Service (SNS) notifications and thus don't need to specify an SNS topic.

Select **"Enable"**.

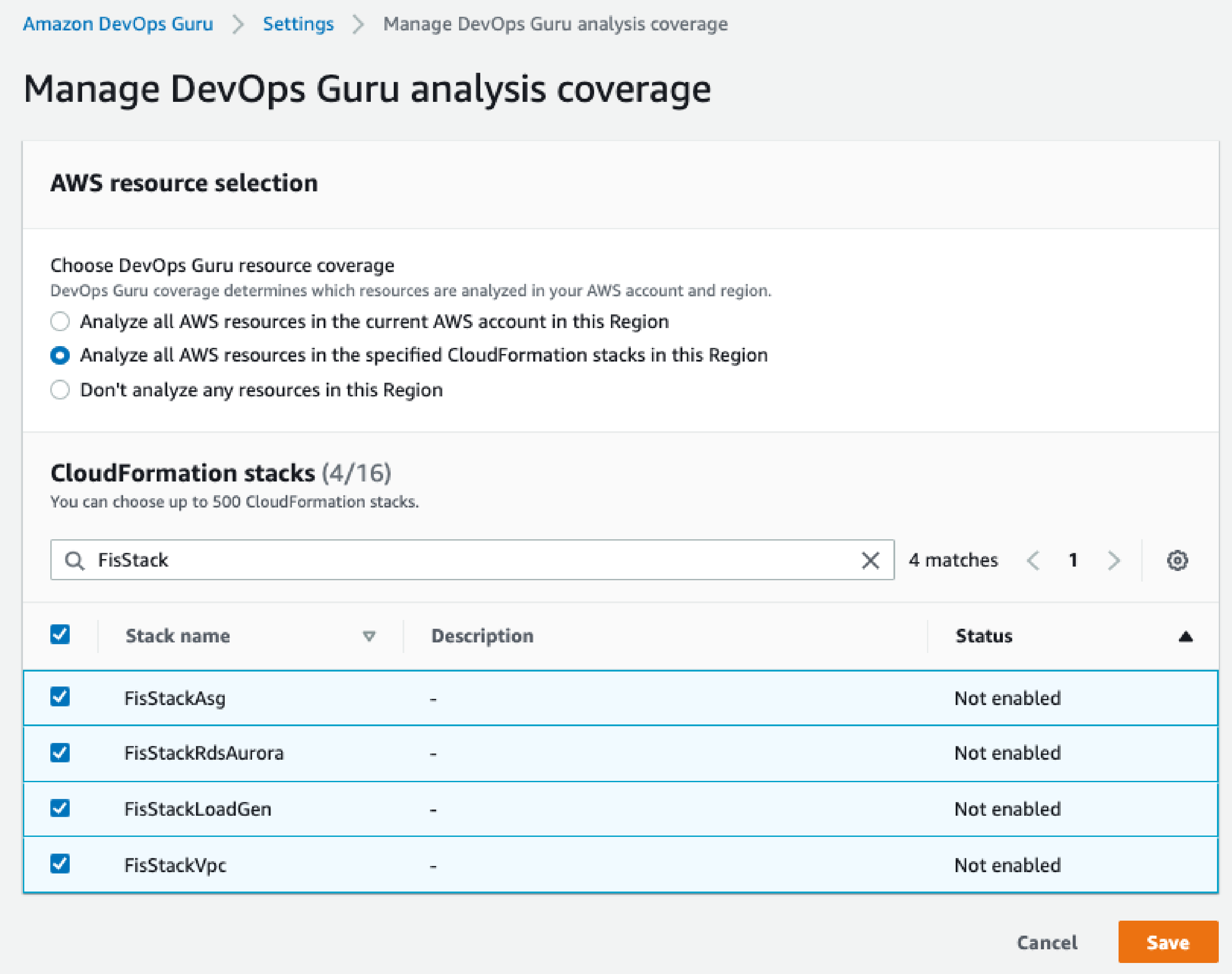
If you set coverage to "Choose later" you should now see an information banner notifying you that you have not yet selected resources:



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 stacks with names starting with `Fis`:


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”**.

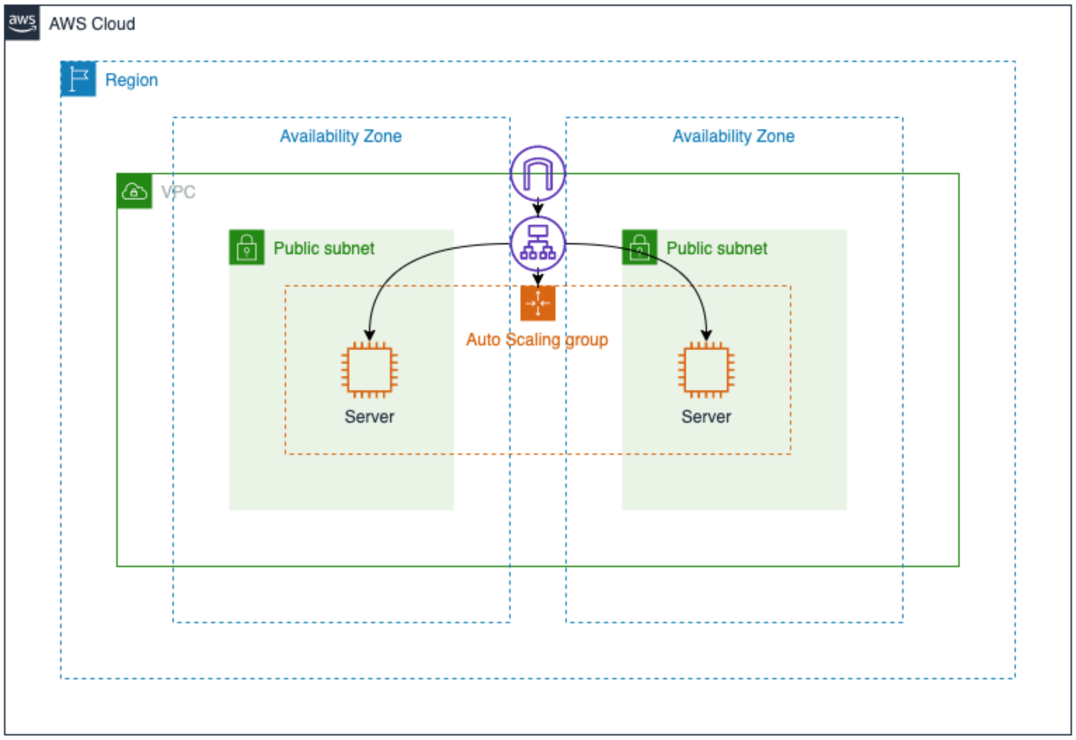
WORKSHOP

This workshop is broken into multiple chapters. The chapters are designed to be done in sequence with each chapter assuming familiarity with some concepts from previous chapters and focusing on new learnings. We include refresher links to relevant prior sections to help you skip over materials you are already familiar with.

Chapters:

Architecture Diagrams

This workshop is focused on how to inject fault into an existing infrastructure. For this purpose the template in the **Provision AWS resources** 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 hosting a demo

BASELINING AND MONITORING

Before we start 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 teams 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.

To disrupt production - or not

Chaos engineering was popularized by Netflix who famously ran it in production. This view of chaos engineering being a production practice is so entrenched that it was even spelt out in the wikipedia definition.

> *Chaos engineering is the discipline of experimenting on a software system* ***in production*** *in order to build confidence in the system's capability to withstand turbulent and unexpected conditions.*

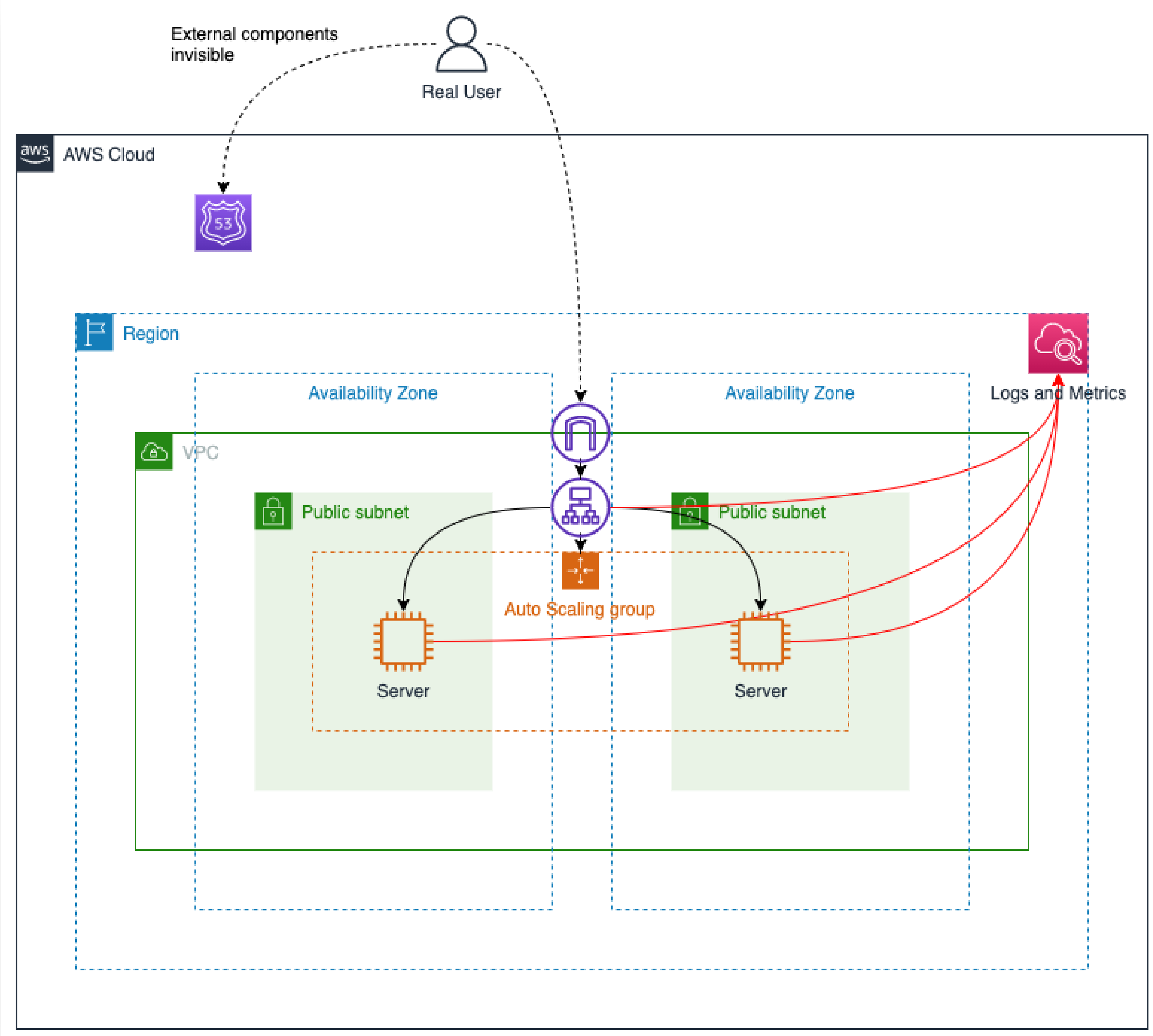
This is so counterintutive that Gene Kim used to have a section in his presentations where he would spell this out to immediate audience laughter:

> *One of the things people don't tell you about chaos engineering: before you do it in production, do it in dev/test.*

Once you stop laughing, stop to think: if *you* ran a chaos experiment in dev/test, would you have the same monitoring and alerting? Would you know if anything broke?

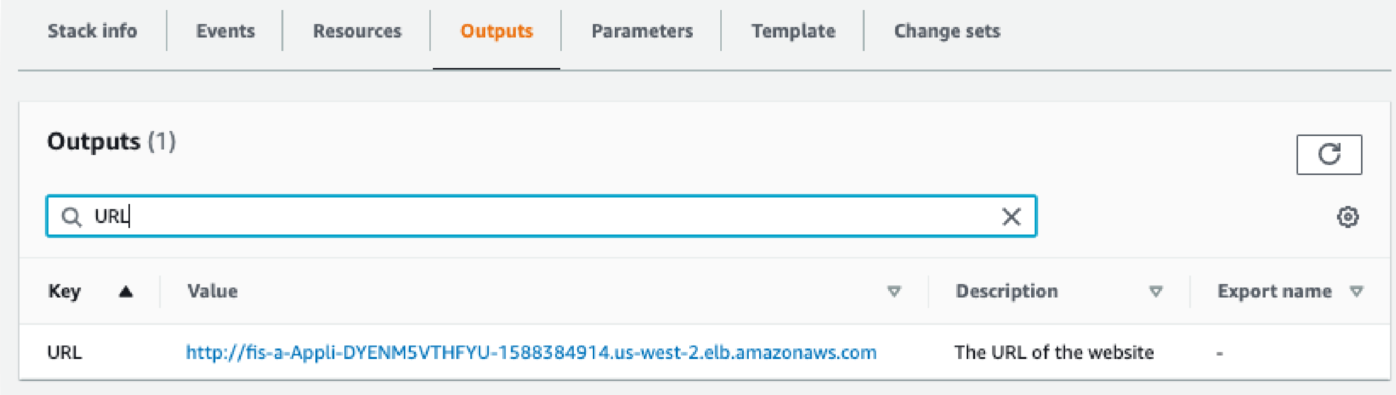
Setting up for fault injection

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

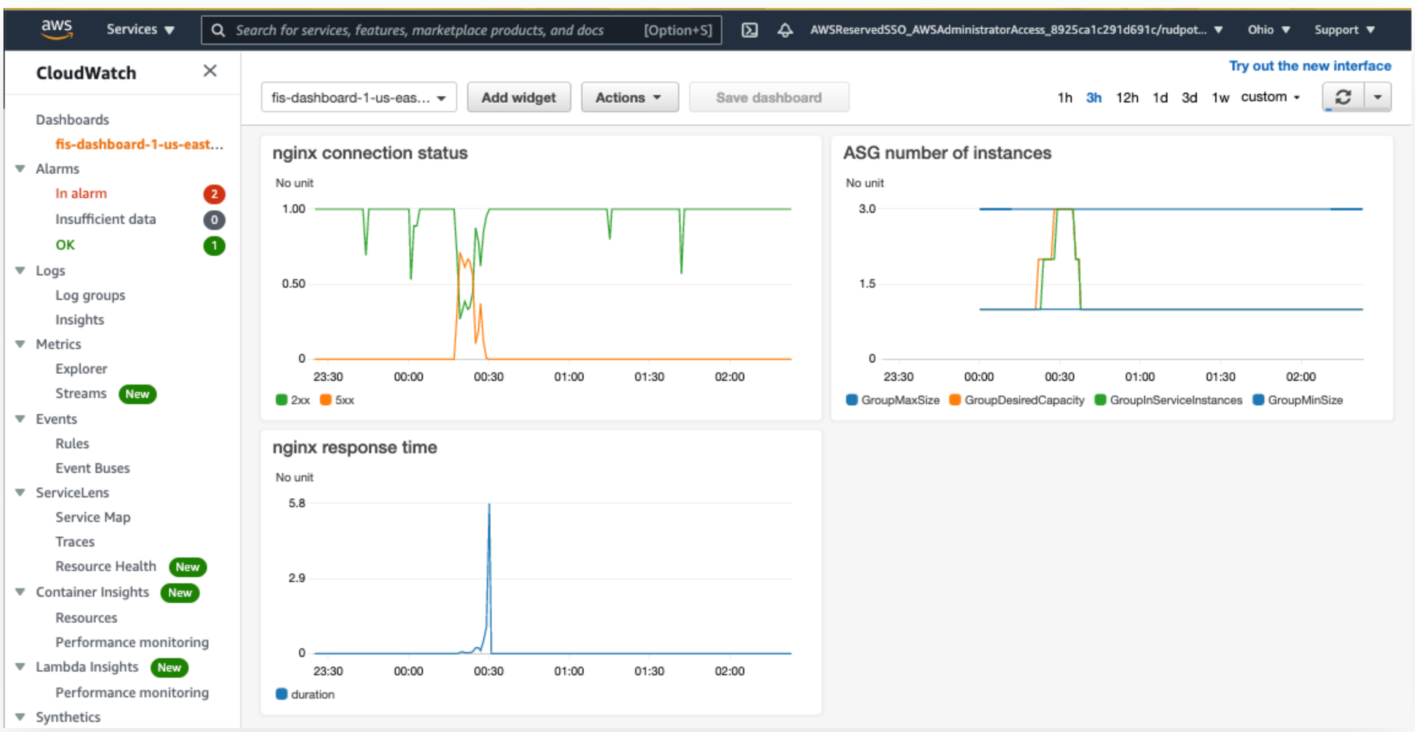


We have a user trying to access a website running on AWS. We have designed it for high availability. We used EC2 instances with an Auto Scaling group and a load balancer to ensure that users can always reach our website even under heavy load or if an instance suddenly fails.

Once you've created the resources as described in **Provision AWS resources** you can navigate to **CloudFormation**, select the `FisStackAsg` stack and select 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 **AWS CloudWatch agent** to export our web server logs to **AWS CloudWatch Logs** and we created **AWS 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 in which we deployed. If you chose a region other than `us-west-2` the dashboard's name will be different. The dashboard also shows the number of instances in our Auto Scaling Group (ASG).



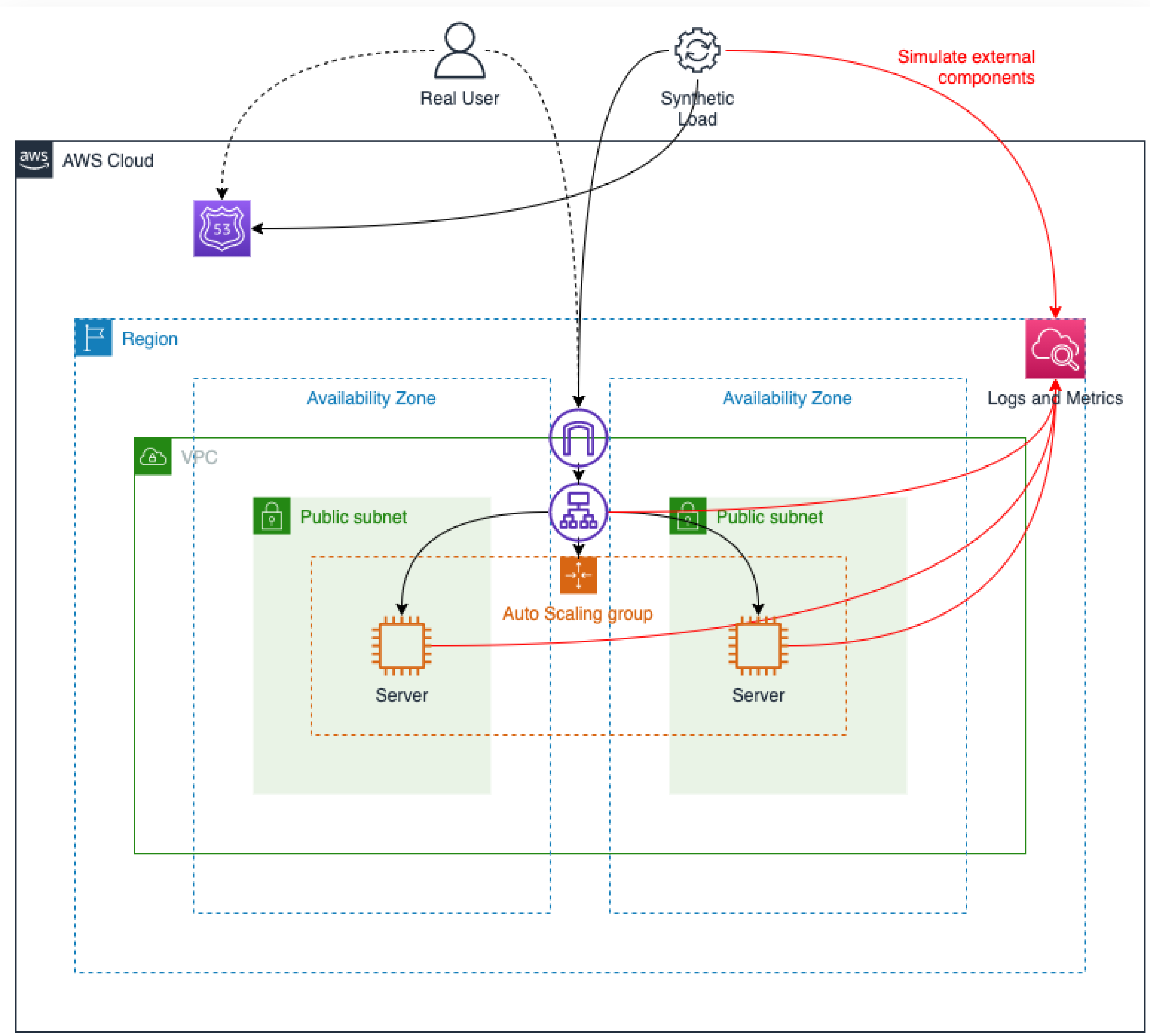
› Accessing the dashboard from the console

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

SYNTHETIC USER EXPERIENCE

In the previous section we showed you a typical configuration to collect sysops data but without visibility into the actual user experience. To gain end-to-end 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.

In this section we will show you how to generate and record synthetic load to reflect the user experience:

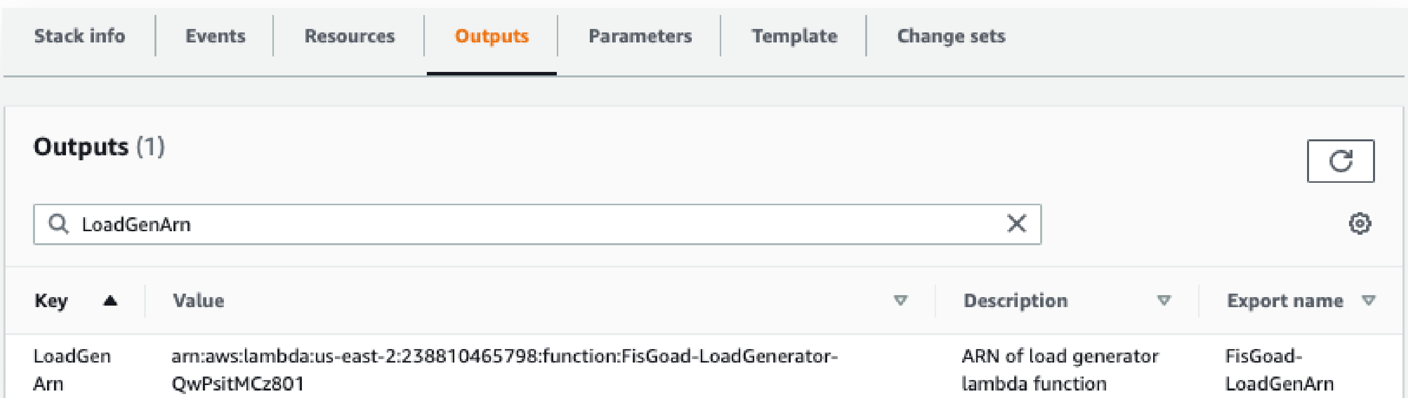


Generating load against our website

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 an AWS Lambda (Lambda) function that will make HTTP GET calls to our website and log performance data to Amazon CloudWatch (CloudWatch). To find the Lambda function, navigate to the AWS CloudFormation (CloudFormation) **console**, 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:

```
# Workaround for AWS CLI v1/v2 compatibility issues
CLI_MAJOR_VERSION=$( aws --version | grep '^aws-cli' | cut -d/ -f2 | cut -d. -f1 )
if [ "$CLI_MAJOR_VERSION" == "2" ]; then FIX_CLI_PARAM="--cli-binary-format raw-in-base64-out"; else unset FIX_CLI_PARAM; fi

# Run load for 3min
aws lambda invoke \
```

```
    --function-name ${LAMBDA_ARN} \
    --payload "{
          \"ConnectionTargetUrl\": \"${URL_HOME}\",
          \"ExperimentDurationSeconds\": 180,
          \"ConnectionsPerSecond\": 1000,
          \"ReportingMilliseconds\": 1000,
          \"ConnectionTimeoutMilliseconds\": 2000,
          \"TlsTimeoutMilliseconds\": 2000,
          \"TotalTimeoutMilliseconds\": 2000
      }" \
    $FIX_CLI_PARAM \
    --invocation-type Event \
    /dev/null
```

Info

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. This notice is for troubleshooting, the code above should work for both CLI versions.

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.

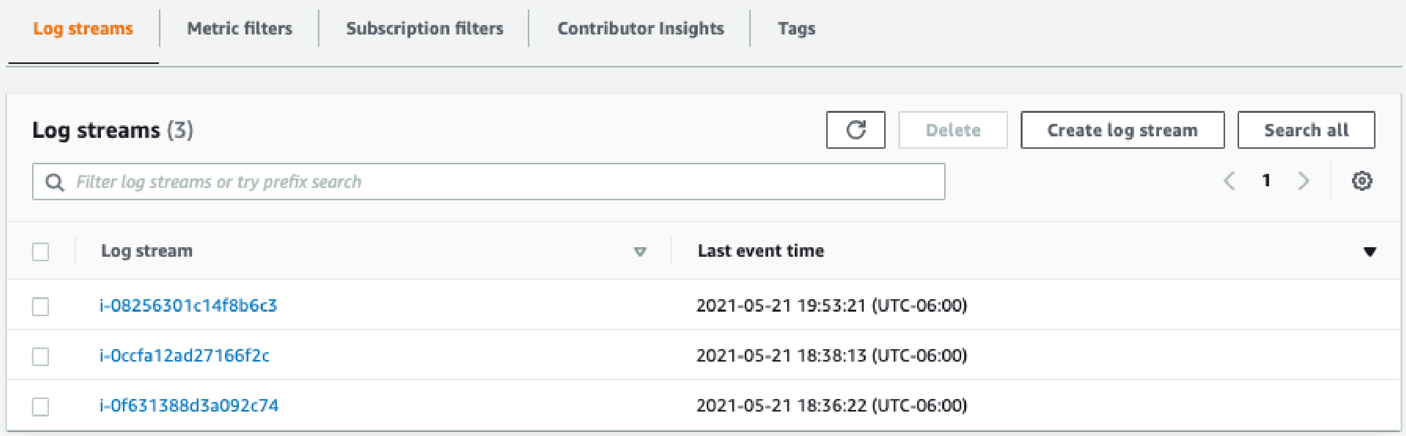
Explore impact of load

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

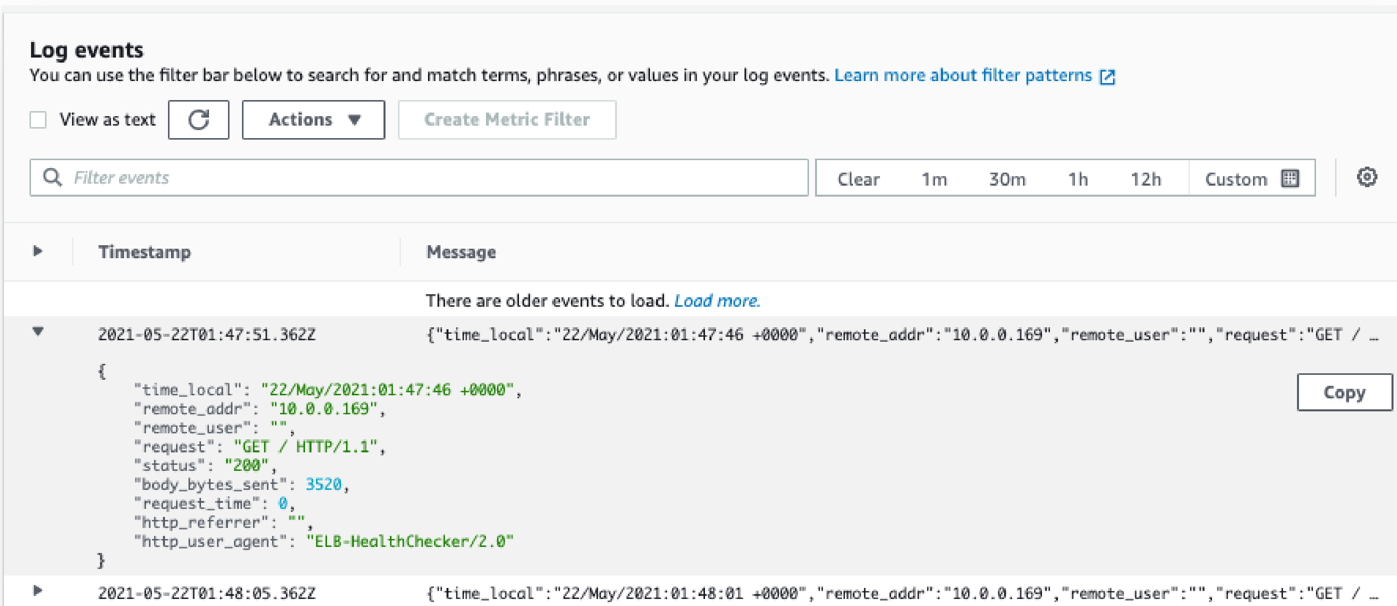
Webserver logs and metrics

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

› Navigating to CloudWatch Log Groups

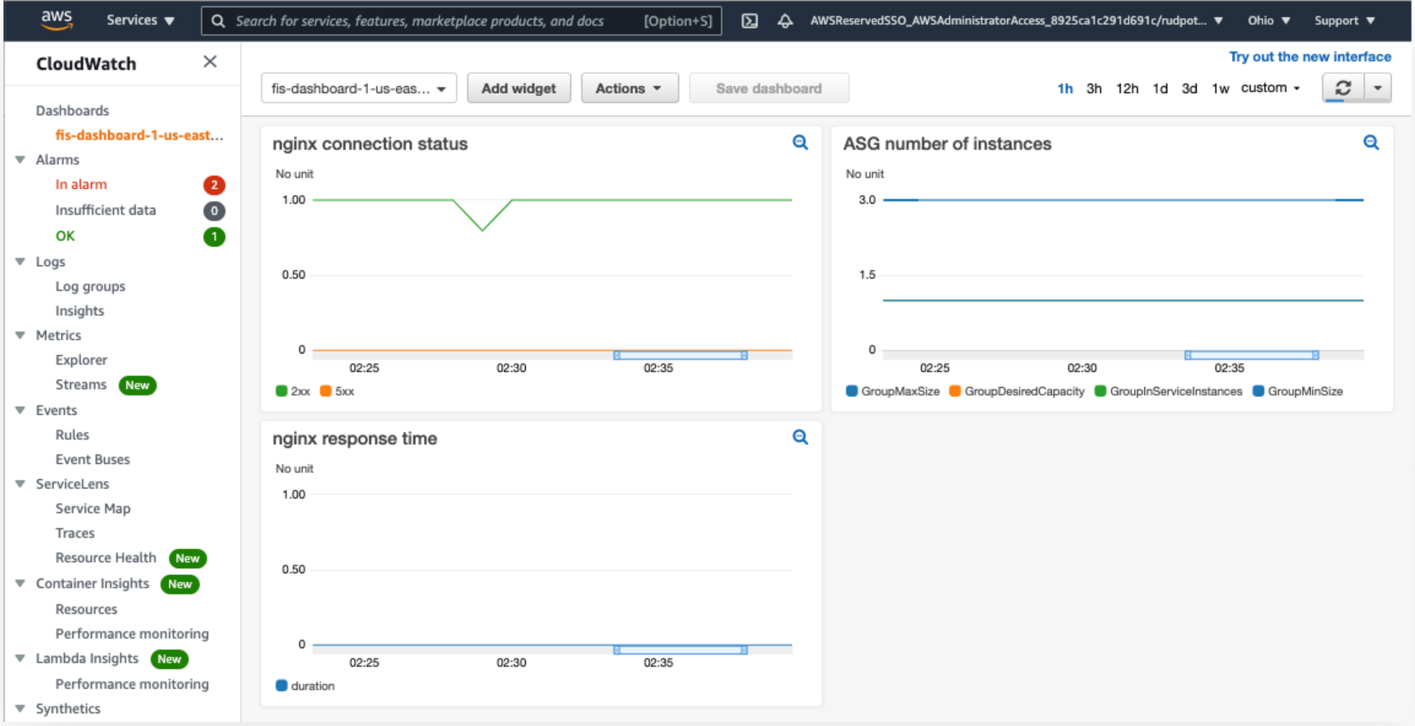


Click through on the topmost entry and expand any of the log lines. You may notice that we've modified the Nginx output format to use JSON instead of the default format:

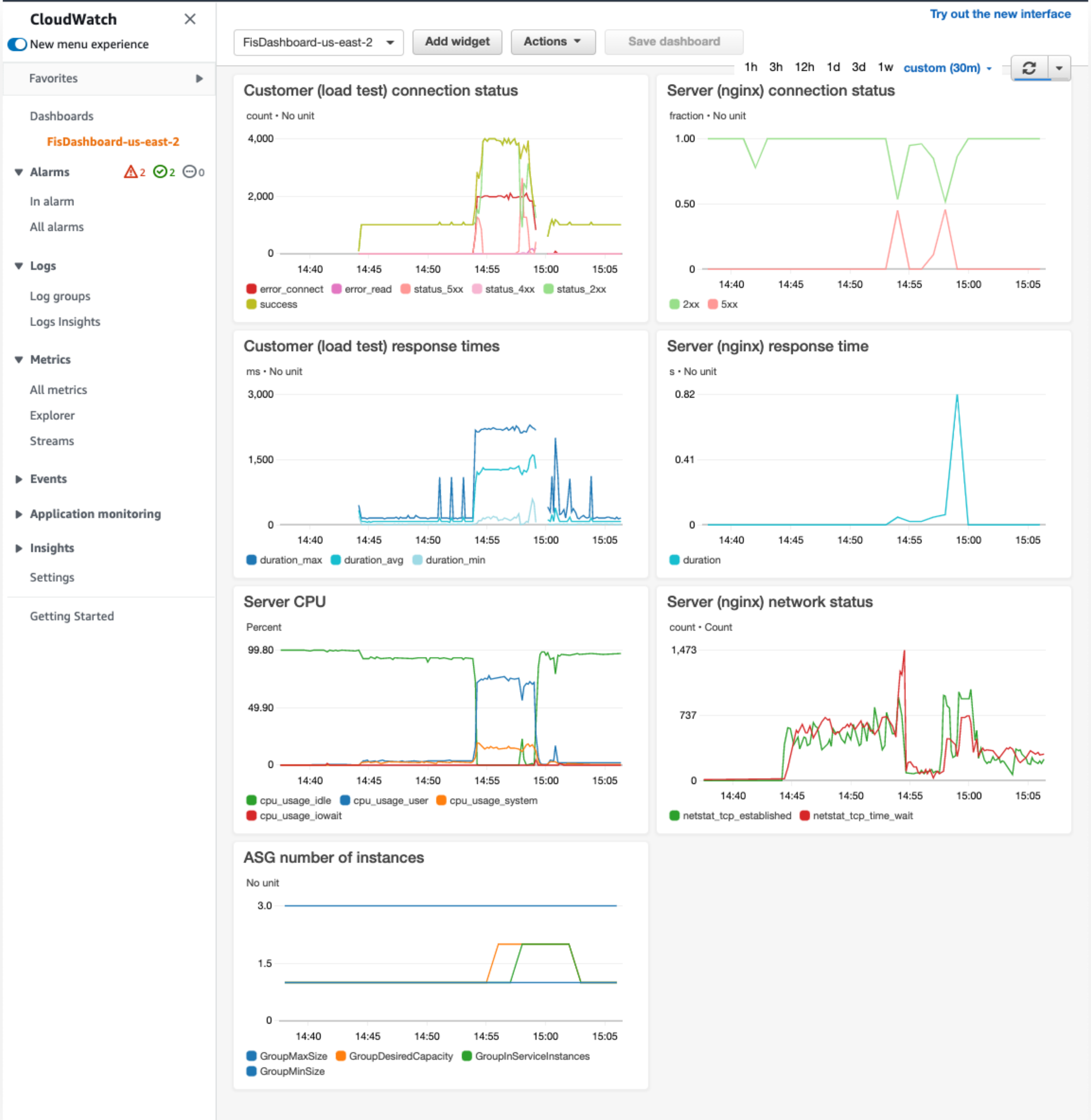


While not necessary, this makes it easy to create **Metric Filters**. Navigate back to the `/fis-workshop/asg-access-log` log group and select the "**Metric filters**" tab. You will see that we have created filters to extract the count of responses with HTTP `status` codes in the `2xx` (good responses) 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 `Server (nginx) connection status` and `Server (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

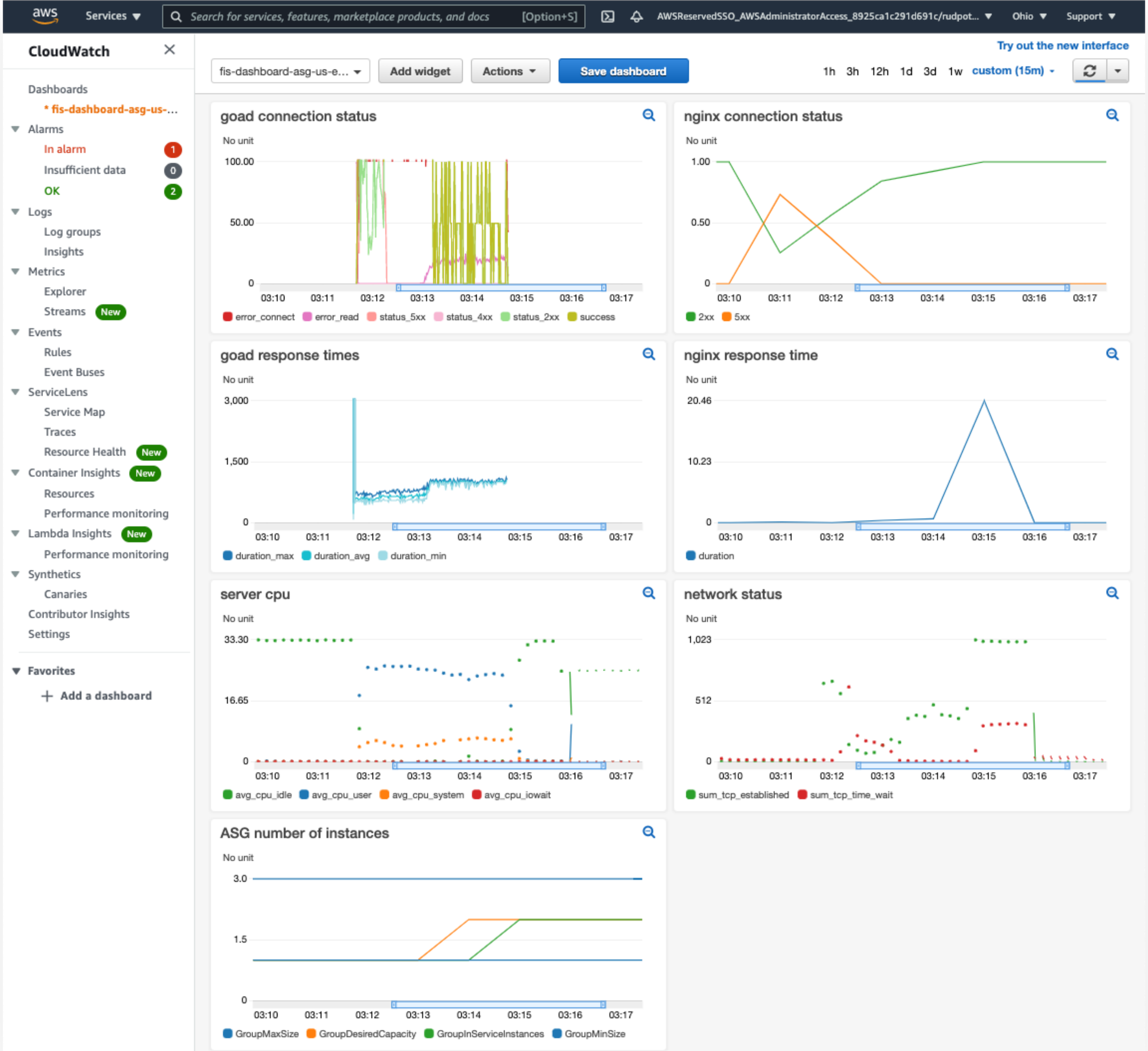
Clearly, hitting a static page isn't a good test to validate our Auto Scaling configuration works as intended. Fortunately, the server also exposes a `phpinfo.php` page. Let's try loading that instead. Define another environment variable and run the load test against the new URL. Since we want to see the Auto Scaling group adjust capacity, let's run more than one copy:

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

# Workaround for AWS CLI v1/v2 compatibility issues
CLI_MAJOR_VERSION=$( aws --version | grep '^aws-cli' | cut -d/ -f2 | cut -d. -f1
)
if [ "$CLI_MAJOR_VERSION" == "2" ]; then FIX_CLI_PARAM="--cli-binary-format raw-
in-base64-out"; else unset FIX_CLI_PARAM; fi

# Run load for 5min, 3x in parallel because max per lambda is 1000
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
      }" \
    $FIX_CLI_PARAM \
    --invocation-type Event \
    /dev/null
done
```

Info

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. This notice is for troubleshooting, the code above should work for both CLI versions.

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



According to the dashboards, we've now generated enough load to force a 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 by the client 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.

Note

If you are working in CloudShell you terminal may expire throughout this workshop. To save your environment variables from this section so they re-populate when you restart your terminal, paste this into your shell:

```
source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.
```

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.

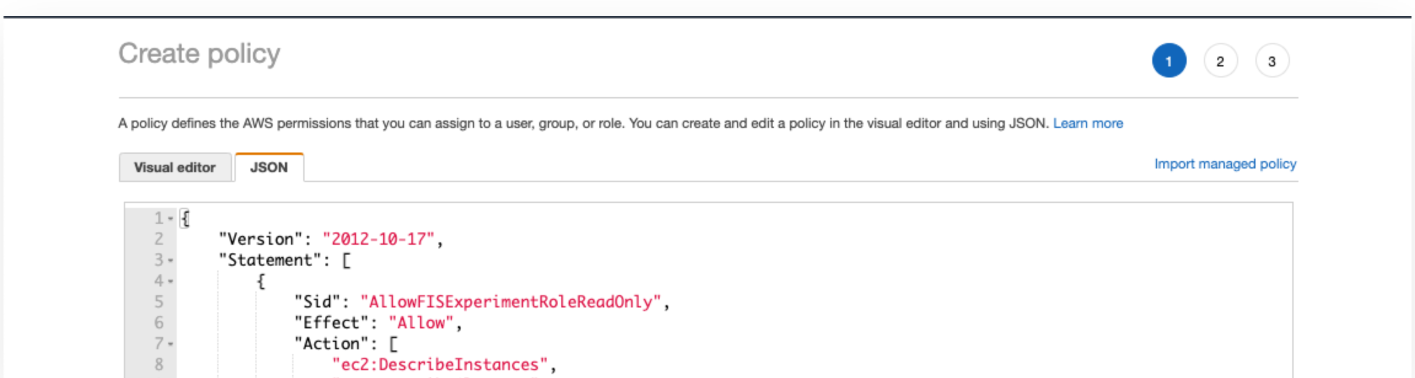
Note

> Please note that FIS uses a **service linked role** to perform some of the internal tasks FIS does on your behalf. If you have sufficient privileges like during this workshop, specifically if you are permitted to perform the `iam:CreateServiceLinkedRole` action, this role will be automatically created the first time you use FIS. If you plan on configuring FIS in an account that is fully managed by Infrastructure as Code (IaC) and where all FIS users do not have the above permission, please make sure to create the service linked role as part of your IaC setup.

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 IAM policy. On the “Create Policy” page select the **JSON** tab



and paste the following policy. This policy is designed to allow you to freely test during the workshop but take the time to look at how broad these permissions are. We suggest limiting this policy using resource names and conditions before using FIS in production:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowFISExperimentLoggingActionsCloudwatch",
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:PutResourcePolicy",
                "logs:DescribeResourcePolicies",
                "logs:DescribeLogGroups"
            ],
            "Resource": "*"
        },
        {
            "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/*"
```

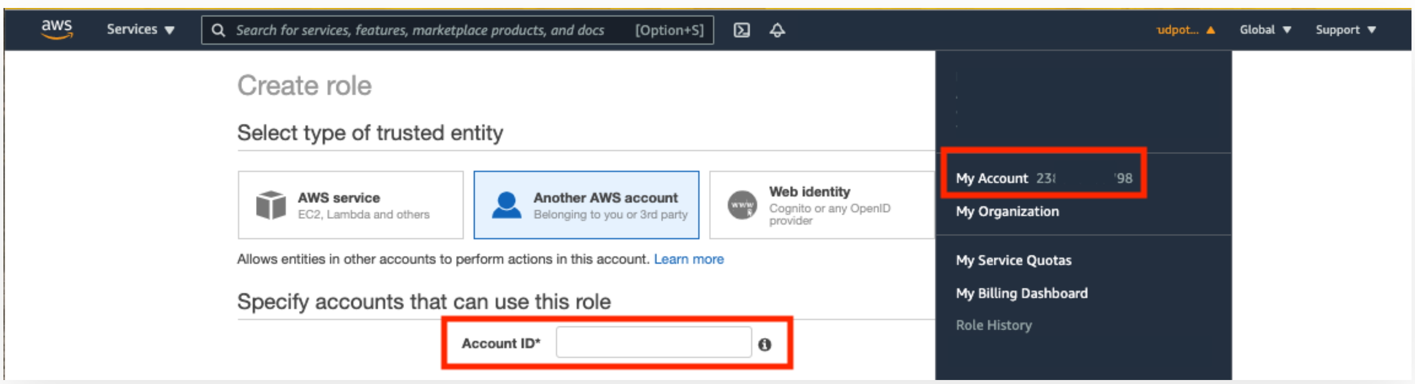
```
        },
        {
            "Sid": "AllowFISExperimentRoleEKSActions",
            "Effect": "Allow",
            "Action": [
                "ec2:TerminateInstances"
            ],
            "Resource": "arn:aws:ec2:*:*:instance/*"
        },
        {
            "Sid": "AllowFISExperimentRoleFISActions",
            "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": "*"
        }
```

```
        ]
    }
```

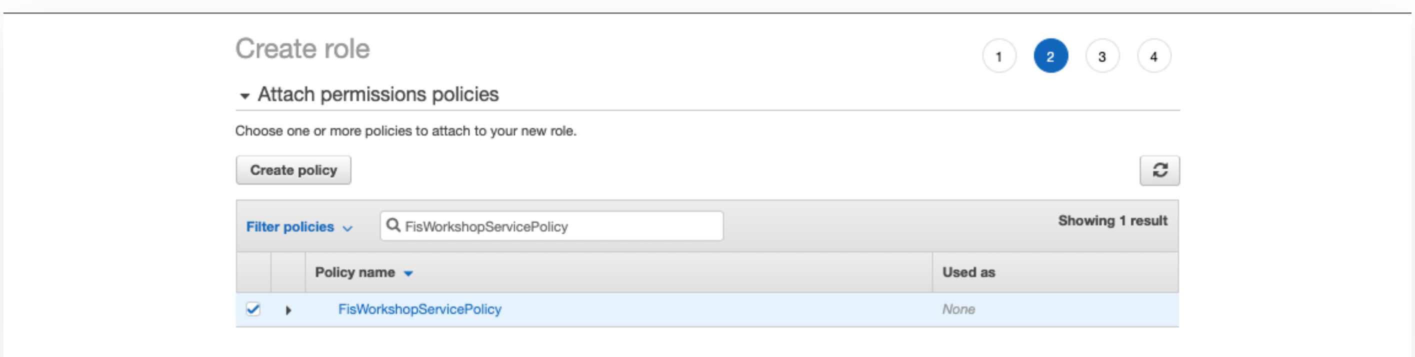
Click on **Next: Tags** to move to the next screen, adding any Tags as you'd wish. In the **Review Policy** page, save this policy as `FisWorkshopServicePolicy` and add any description you would like. Complete the policy creation by clicking on **Create Policy**.

Navigate to the **IAM console** page and create a new **Role**.

On the "Select type of trusted entity" page AWS FIS does not exist as a trusted service yet. We shall add an account trust as a placeholder and replace this with AWS FIS later. Select **"Another AWS Account"** and add the current account number. You can find the AWS account number in the drop-down menu at the top right of the page as shown:



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

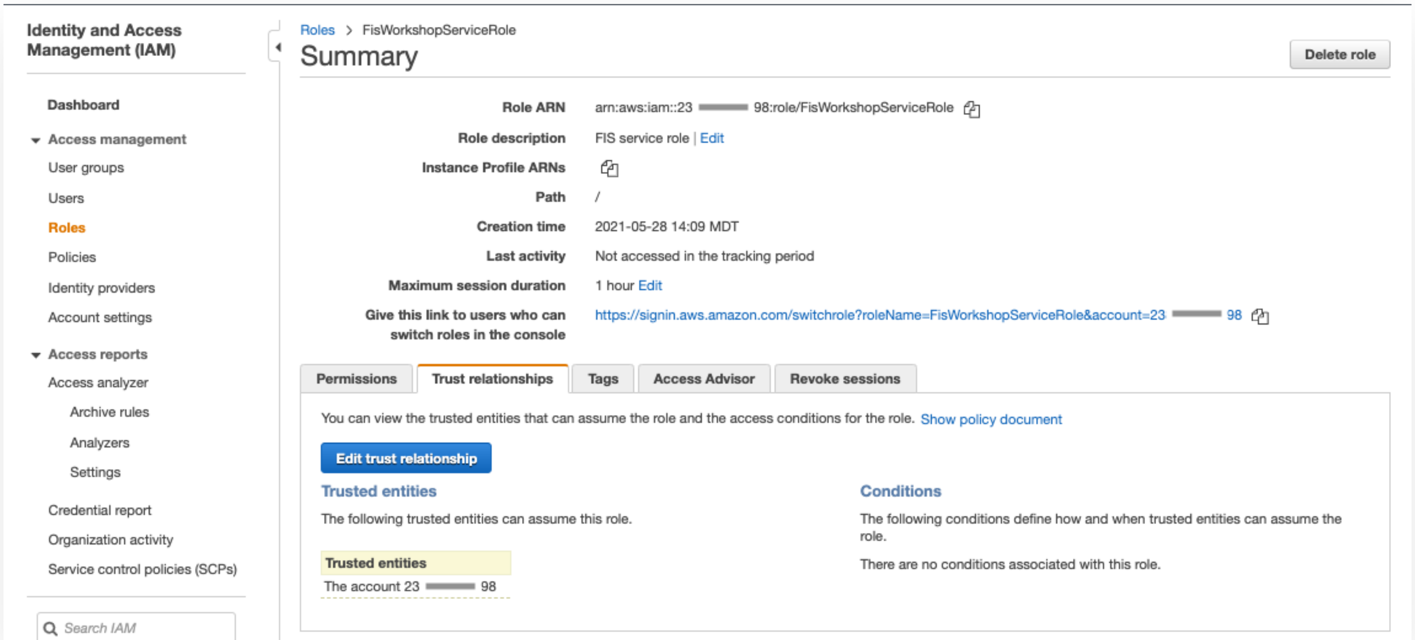


Click on **Next: Tags** and add any Tags you would like for this role.

Click on **Next: Review** and save the role name as `FisWorkshopServiceRole`. Add any description you would like for this role.

Complete the Role creation by clicking on **Create 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": {}
        }
    ]
}
```

Click on **Update Trust Policy** to complete updating the Role.

EXPERIMENT (CONSOLE)

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

Note

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

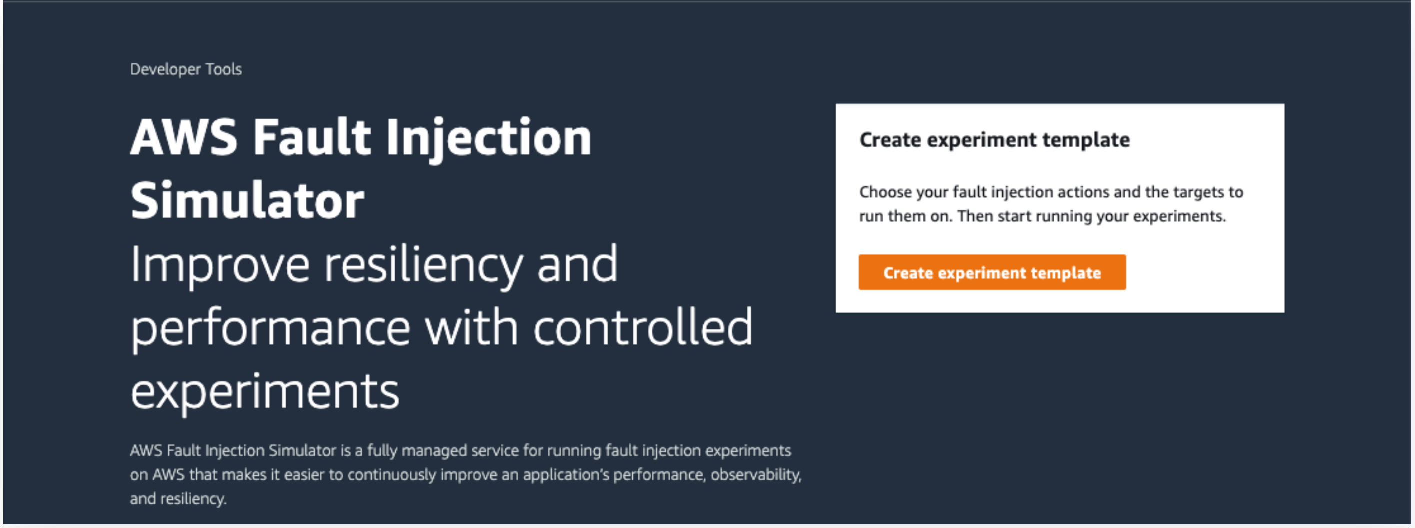
Experiment setup

To create a fault injection experiment, we first need to create an AWS FIS template defining:

- Name (optional)
- Description (optional)
- Template permissions
- **Actions**
- **Targets**
- **Stop Conditions** (optional but strongly recommended)
- Tags

Create an AWS FIS experiment template

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

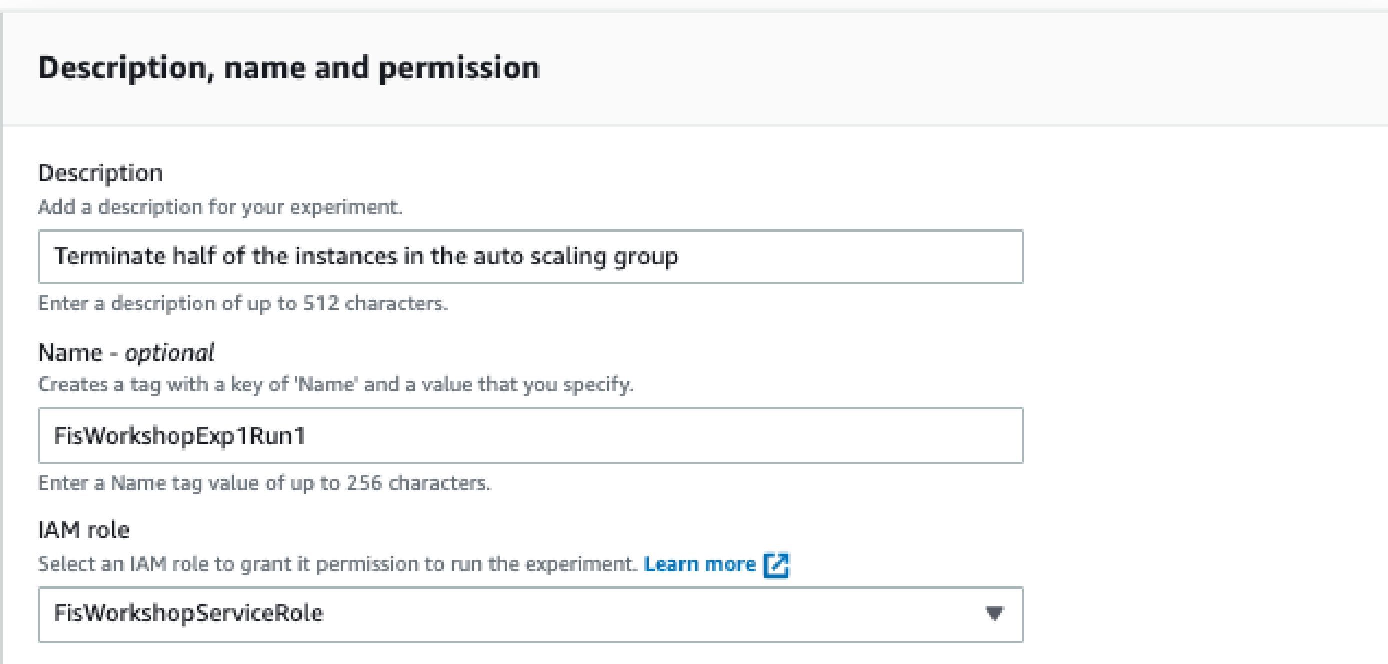


Note

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

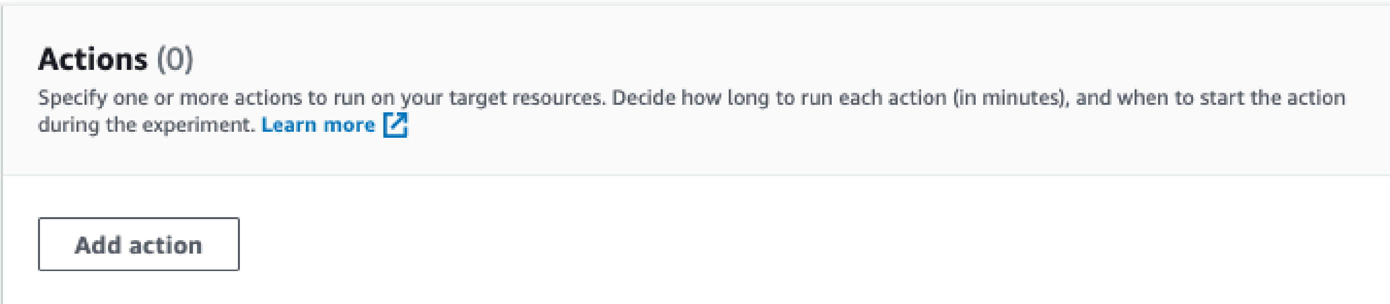
Template description, name, and permissions

Let's write a description for our experiment template and select an IAM role to use when performing the experiment. Go to the "Description, name and permission" section. For "Description" enter `Terminate half of the instances in the auto scaling group`, for "Name" enter `FisWorkshopExp1Run1` and for "IAM Role" select the `FisWorkshopServiceRole` role you created previously.



Action definition

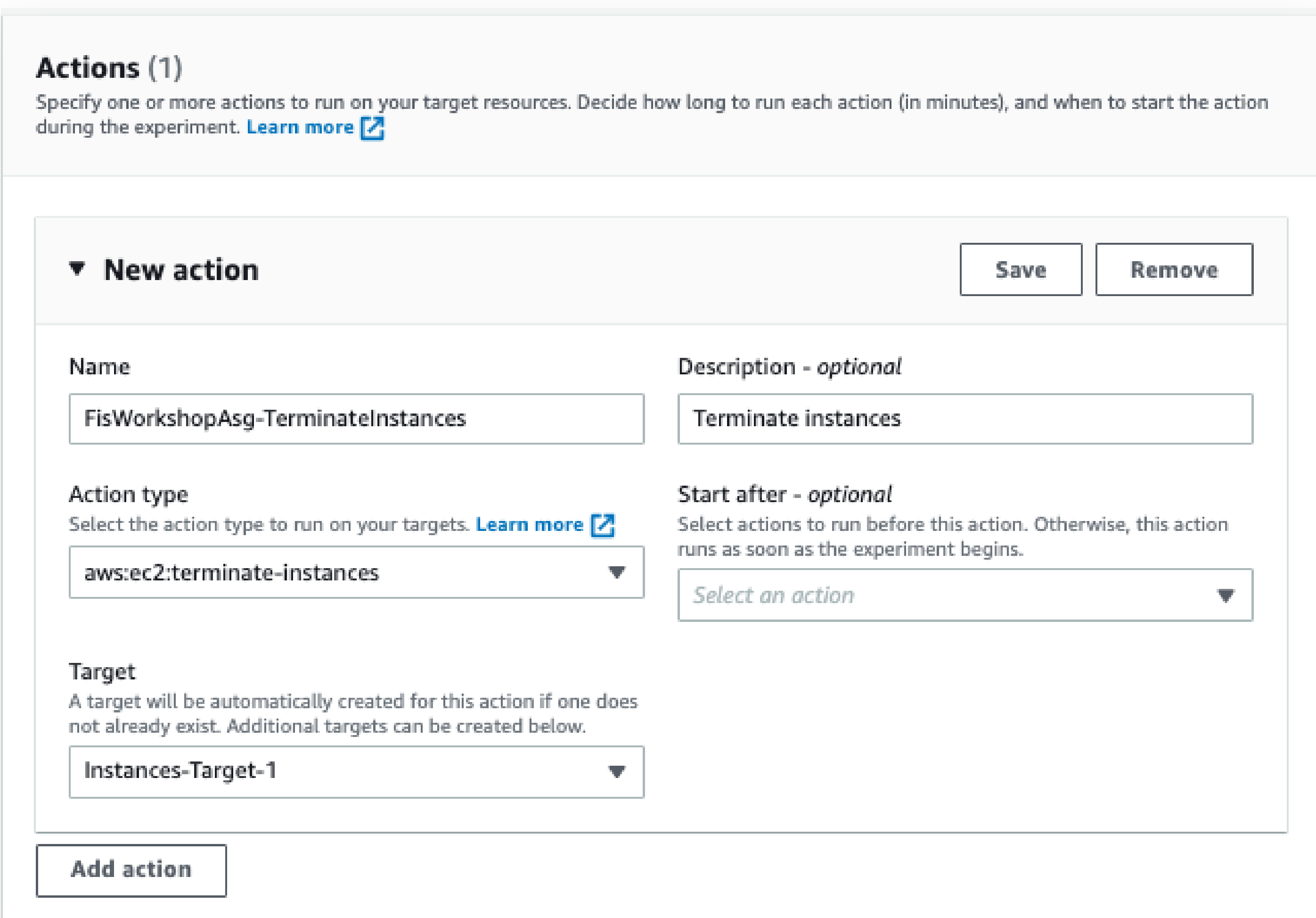
Here we select the type of fault we wish to inject, the action to take. To test the hypothesis that we can safely impact half the instances in our Auto Scaling group, we will terminate those instances. Go to the "Actions" section and select **"Add Action"**.



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 we are only taking a single action in this experiment template.

Leave the default "Target" `Instances-Target-1` and select **"Save"**.



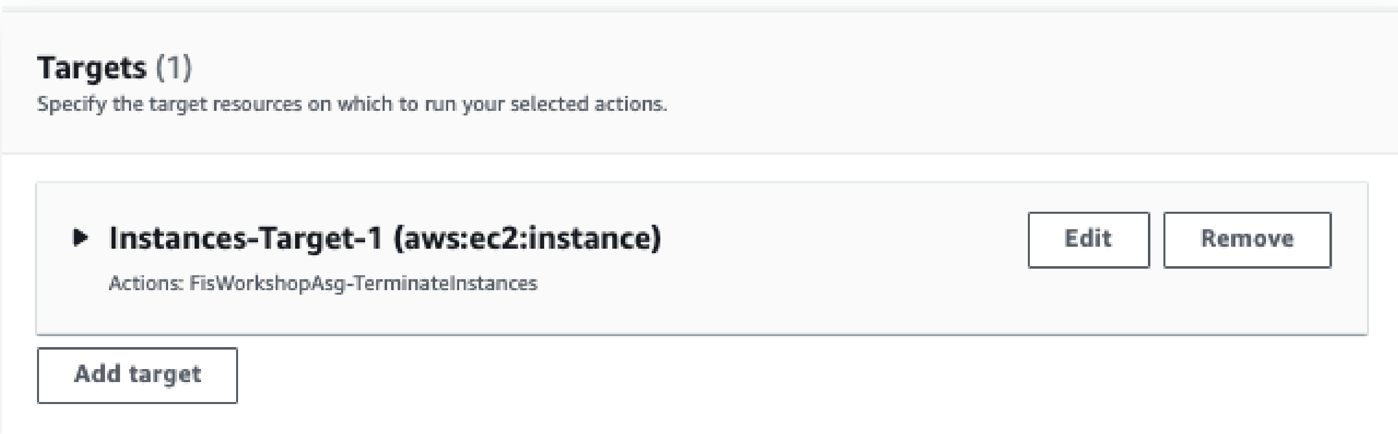
Note

`Instances-Target-1` was auto-generated for us because no appropriate target type existed in the experiment template. If one or more targets already exist, e.g. because we added actions before, then we will be presented with a drop down selector for existing targets instead.

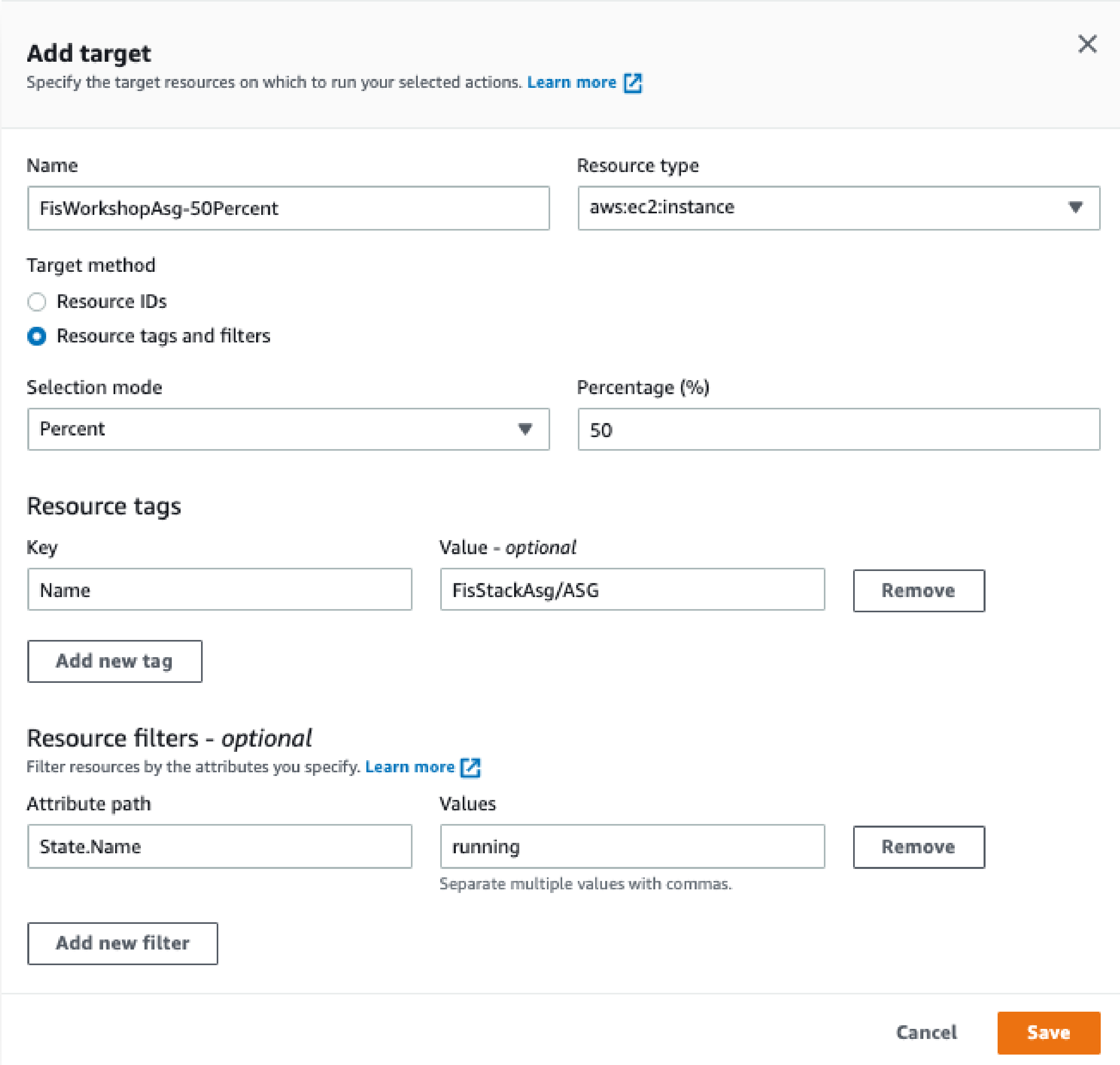
Target definition

For our action we are choosing to terminate EC2 instances. In the target section we define which instances to terminate. As a reminder, for this first experiment we want to prove the hypothesis that we can safely impact half the instances in our Auto Scaling group.

Go to the "Targets" section, select the `Instances-Target-1` section, and select **"Edit"**.

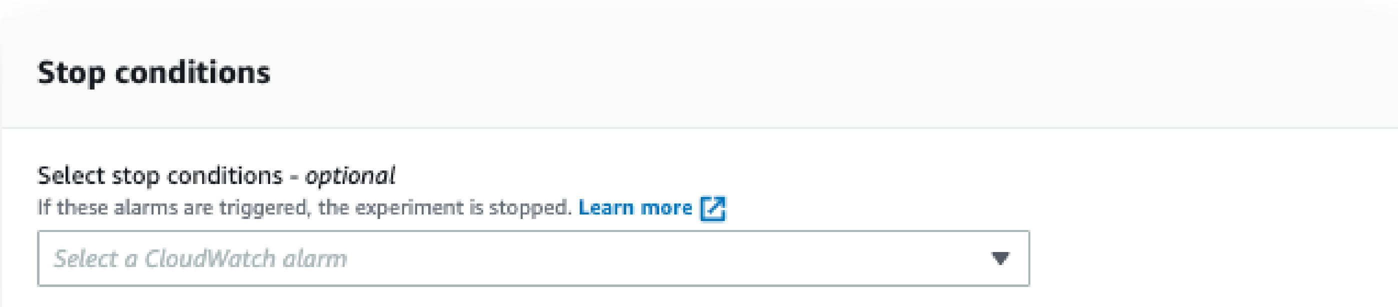


You may leave the default name `Instances-Target-1` but for maintainability we rcommend using descriptive target names. Change the name to `FisWorkshopAsg-50Percent` (this will automatically update the name in the action as well) and make sure "Resource type" is set to `aws:ec2:instances`. For "Target 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" to select only from instances associated with the desired Auto Scaling group. Under filters enter `State.Name` in the "Attribute path" field and `running` under "Values" to ensure we do not consider instances that are starting or stopping due to unrelated events. For more information on filters see the **documentation**. Select **"Save"**.



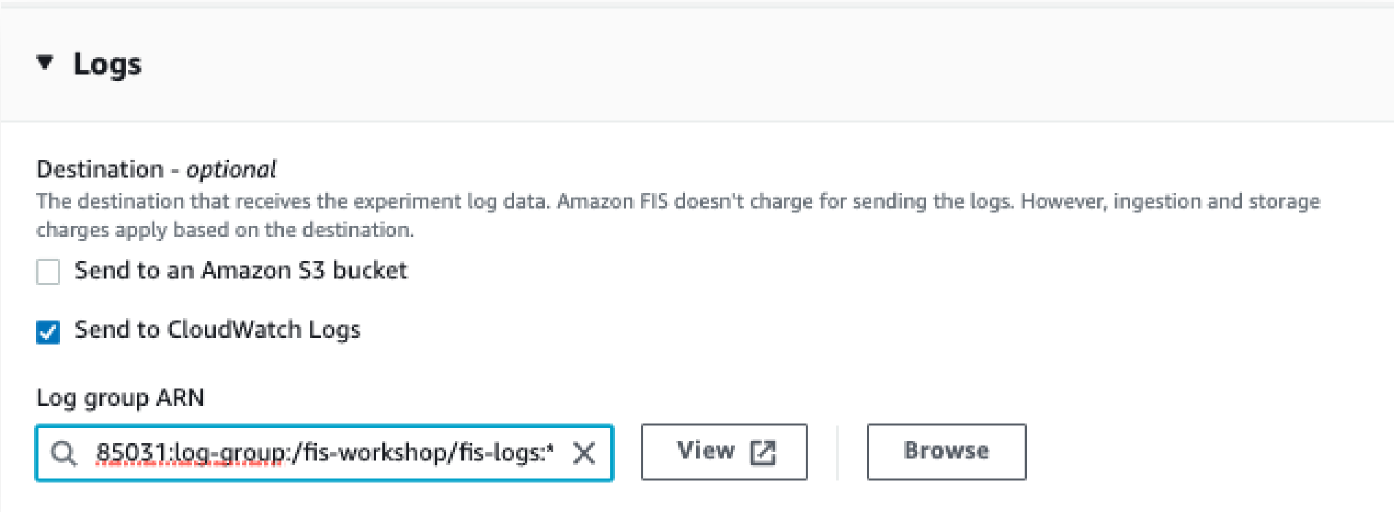
Stop conditions

AWS FIS provides stop conditions tied to **Amazon CloudWatch alarms** as a safeguard to minimize the impact of experiments that do not perform as expected. In this experiment we are performing a single action that cannot be reverted so we will leave this empty.

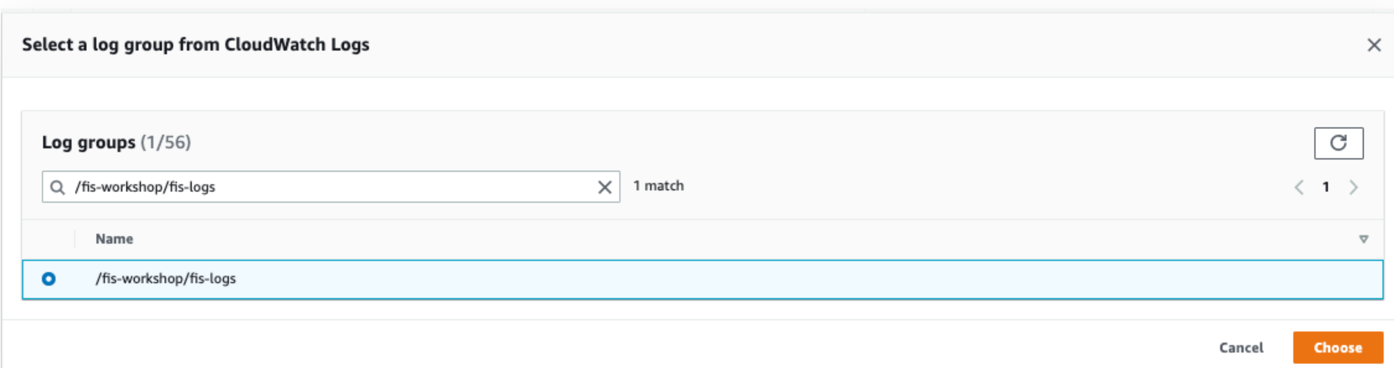


Logs

To write logs of FIS events to CloudWatch, expand the “Logs” card, check the “Send to CloudWatchLogs” checkbox, and select “Browse” to select the pre-created log group:



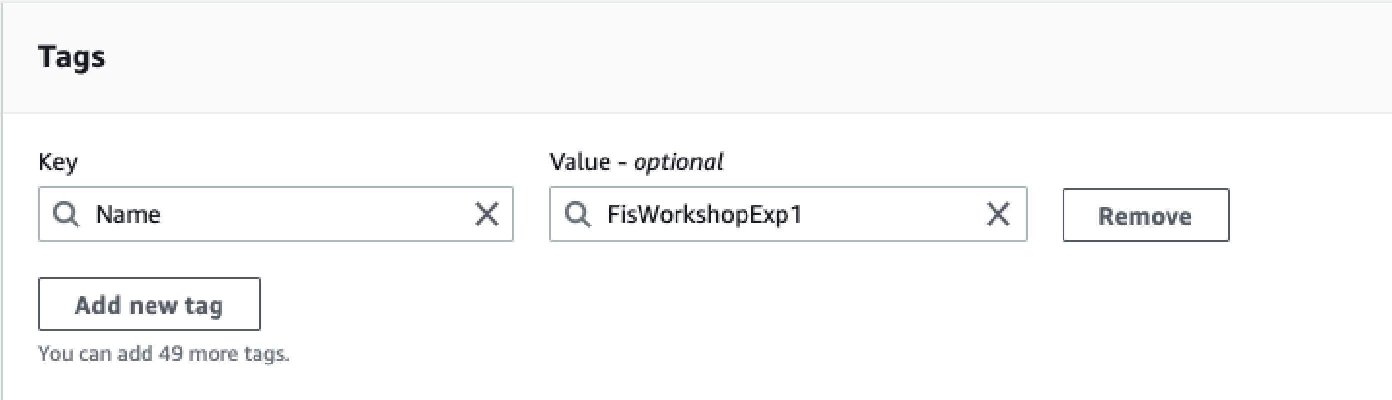
For “Log groups” enter `/fis-workshop/fis-logs` and select the relevant entry:



Template tags

AWS FIS tracks the template name as the special tag `Name` which is displayed in the “Name” field of the experiment template list view. In addition to the Name tag that propagated from setting it in the “Description, name and permission” card, we can optionally attach tags to our template. Tags can be used in IAM policy **condition keys** to control access to the experiment template.

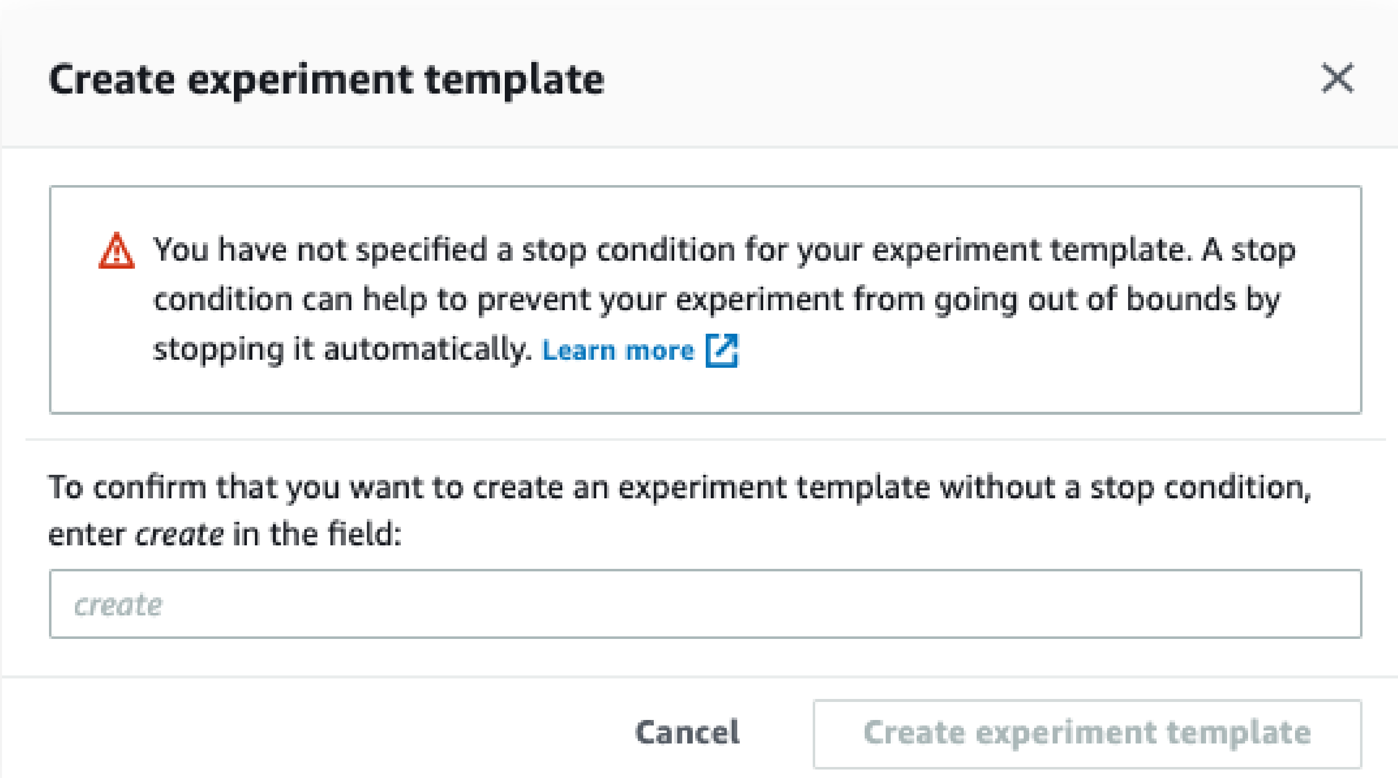
For this experiment we will make no changes here.



Creating template without stop conditions

Scroll to the bottom of the template definition page and select *"Create experiment template"*.

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"**.



Validation procedure

We will be using the AWS 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. For larger experiments we would add the load generator into our experiment.

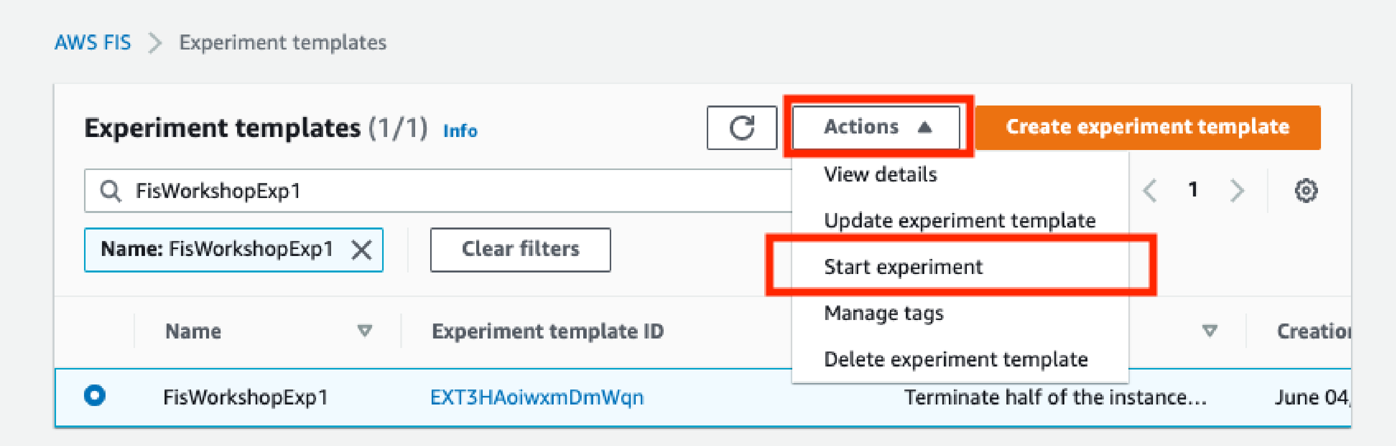
However, for this experiment we will manually trigger load generation on the system before starting the experiment, similar to what we did in the previous section. Here we have increased the run time to 5 minutes by setting `ExperimentDurationSeconds` to `300`:

```
# Please ensure that LAMBDA_ARN, URL_HOME, and FIX_CLI_PARAM are still set from
previous section

# Run load for 5min, 3x in parallel because max per lambda is 1000
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
      }" \
    $FIX_CLI_PARAM \
    --invocation-type Event \
    /dev/null
done
```

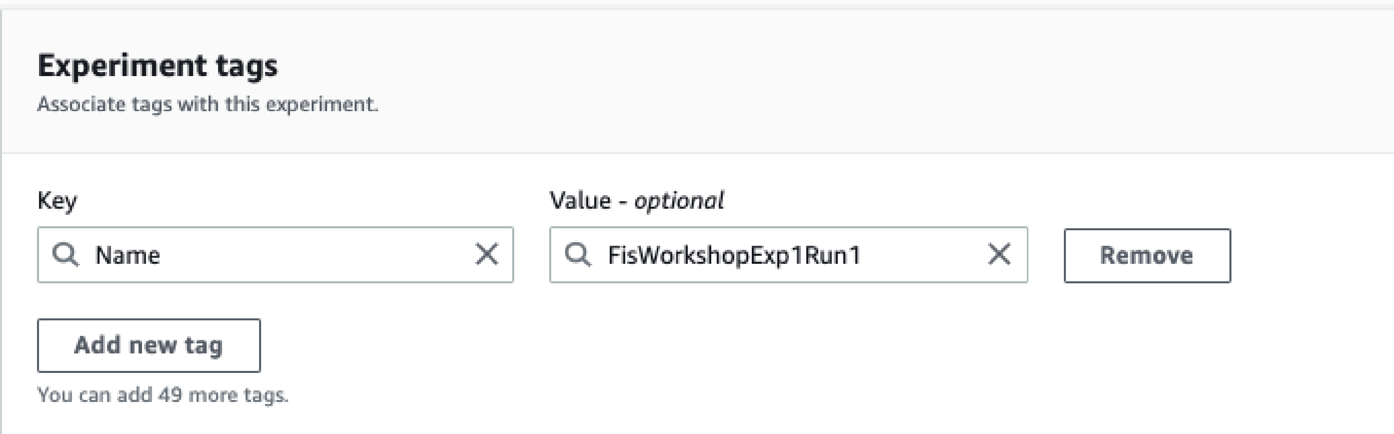
Warning

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.

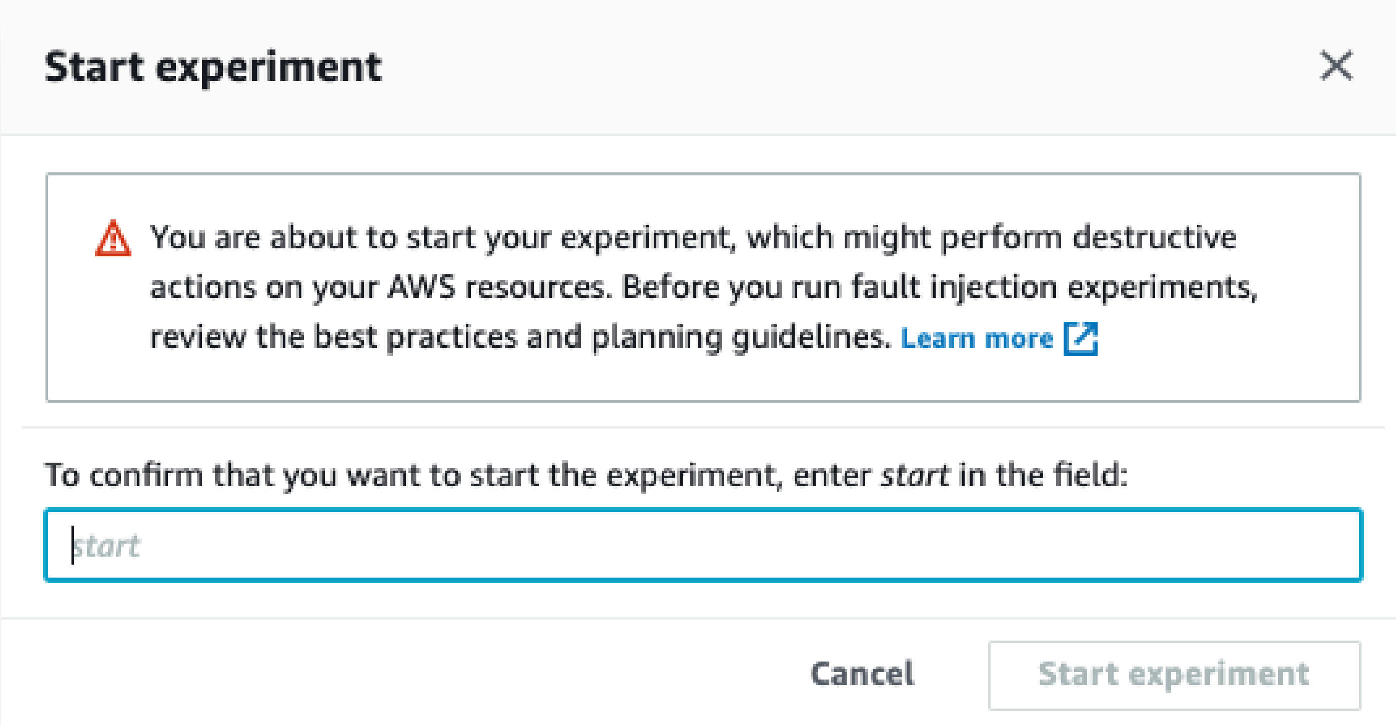
To start the experiment navigate to the **FIS console**, select the `FisWorkshopExp1` template we just created. Under **"Actions"** select **"Start experiment"**.



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"**.



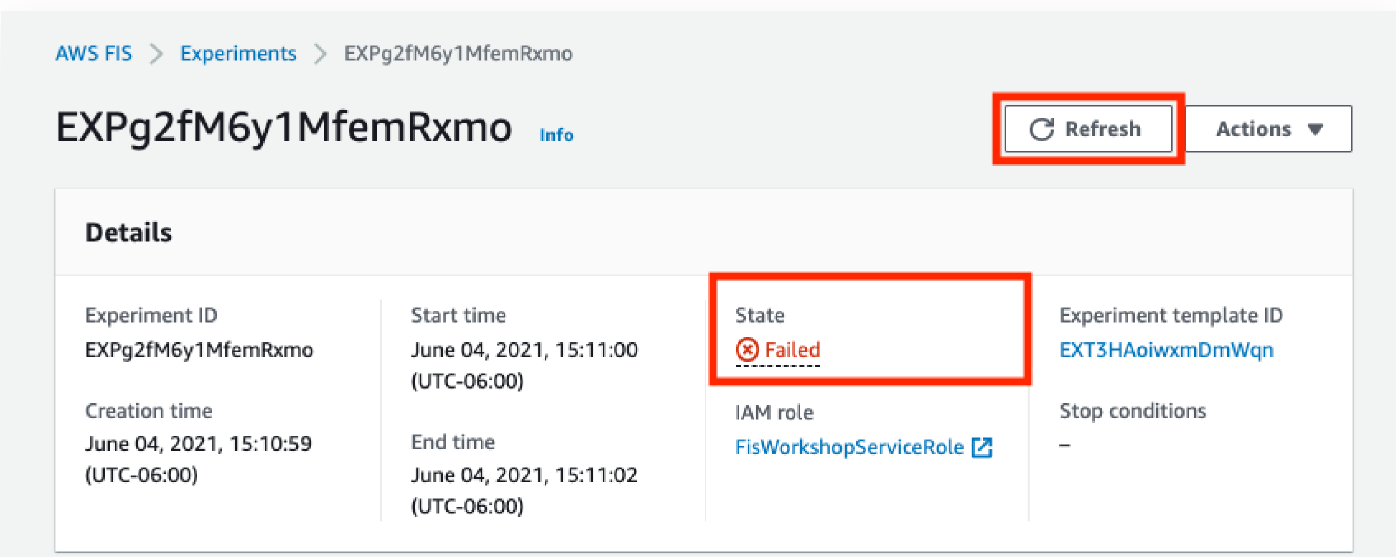
Because you are about to start a potentially destructive process, you will be asked to confirm that you really want to do this. Type `start` as directed and select **"Start experiment"**.



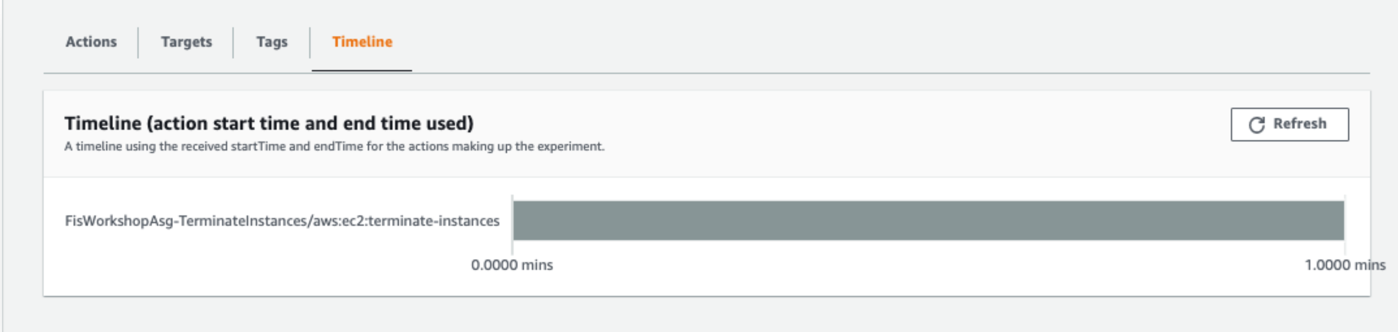
Review results

Navigate to the **FIS console**, select “Experiments”, and click 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`.

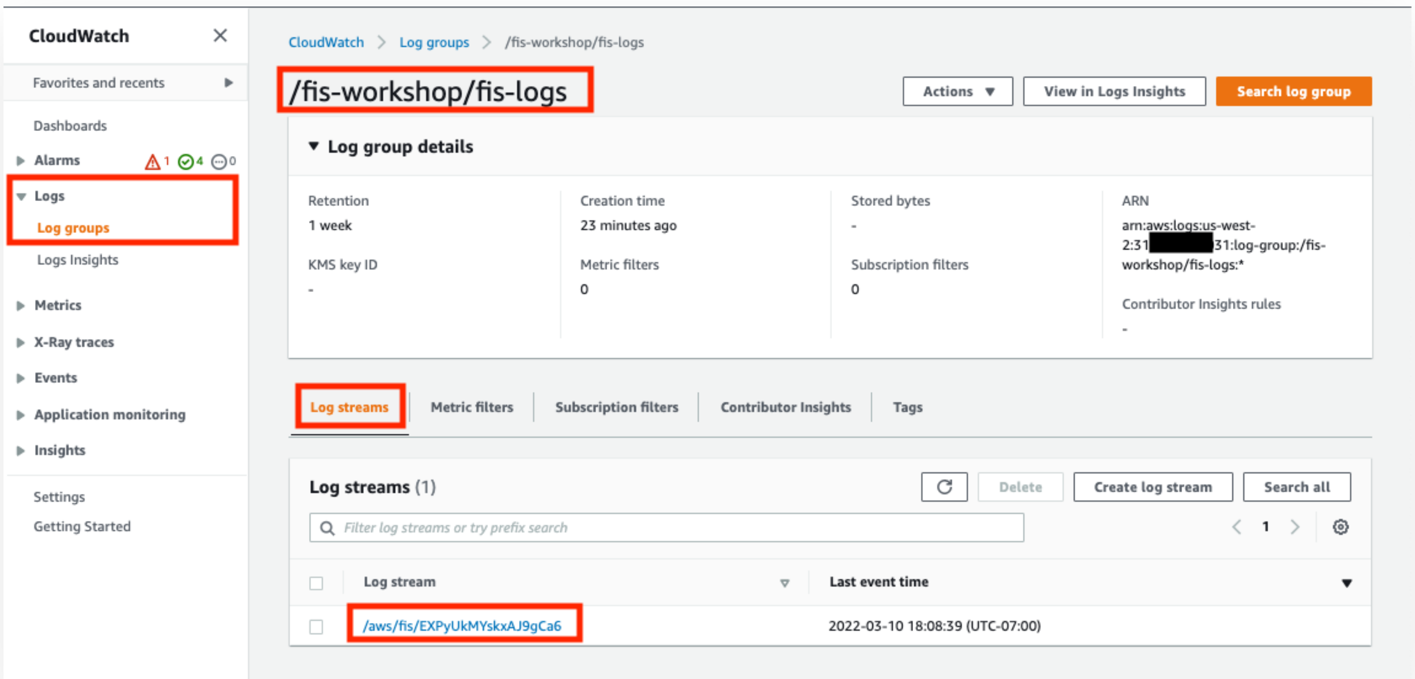


Click on the failed result to get more information about why it failed. The message should say `Target resolution returned empty set`. Scroll down further and select “Timeline”:

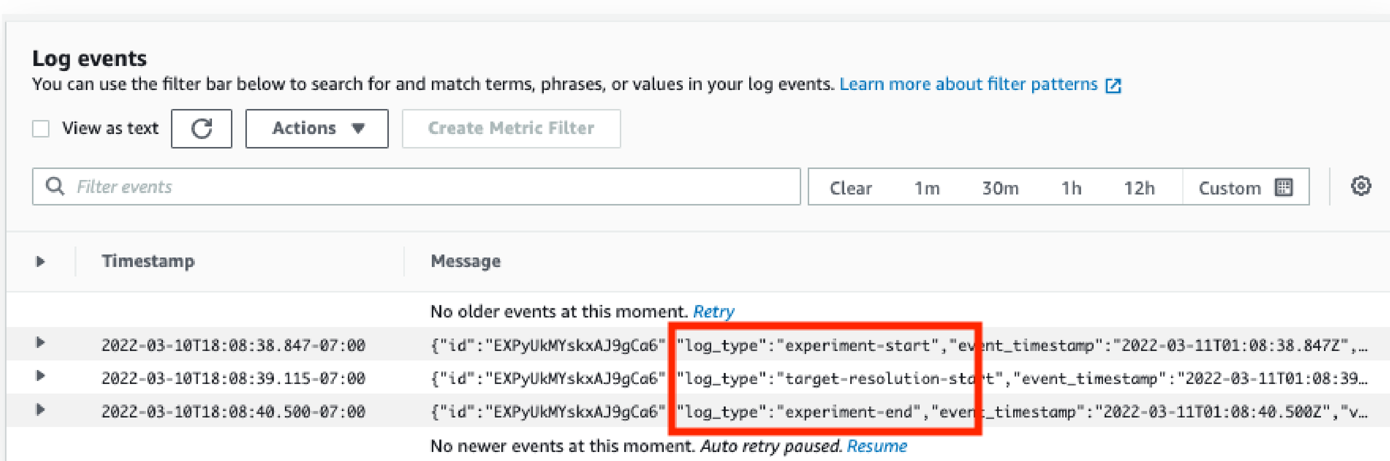


In this case this doesn't show anything because the experiment failed to run entirely, but for larger experiments you would see when each action was active in the timeline.

Next navigate to the **CloudWatch Logs console** and select the `/fis-workshop/fis-logs` log group

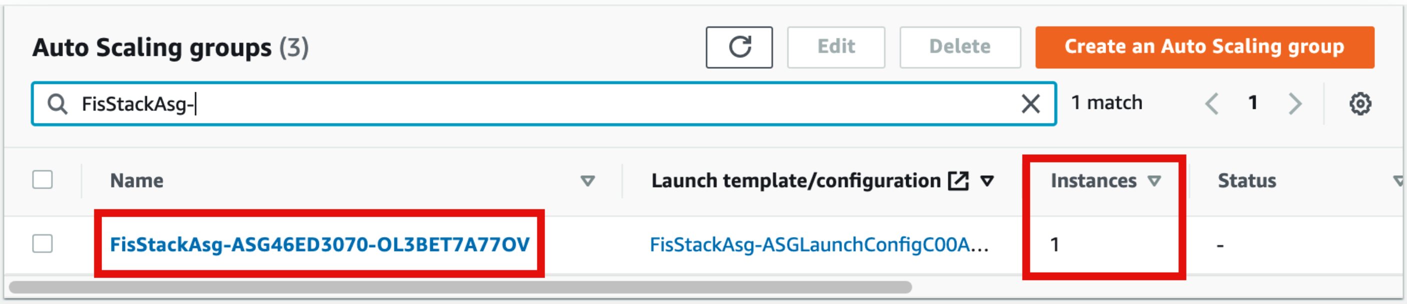


then expand the topmost stream under "Log streams"



All this shows that FIS failed to identify virtual machines that satisfied the condition of being “50% of instances with “Name” tag of `FisStackAsg/ASG`.

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 burger menu, and search for `FisStackAsg-`:

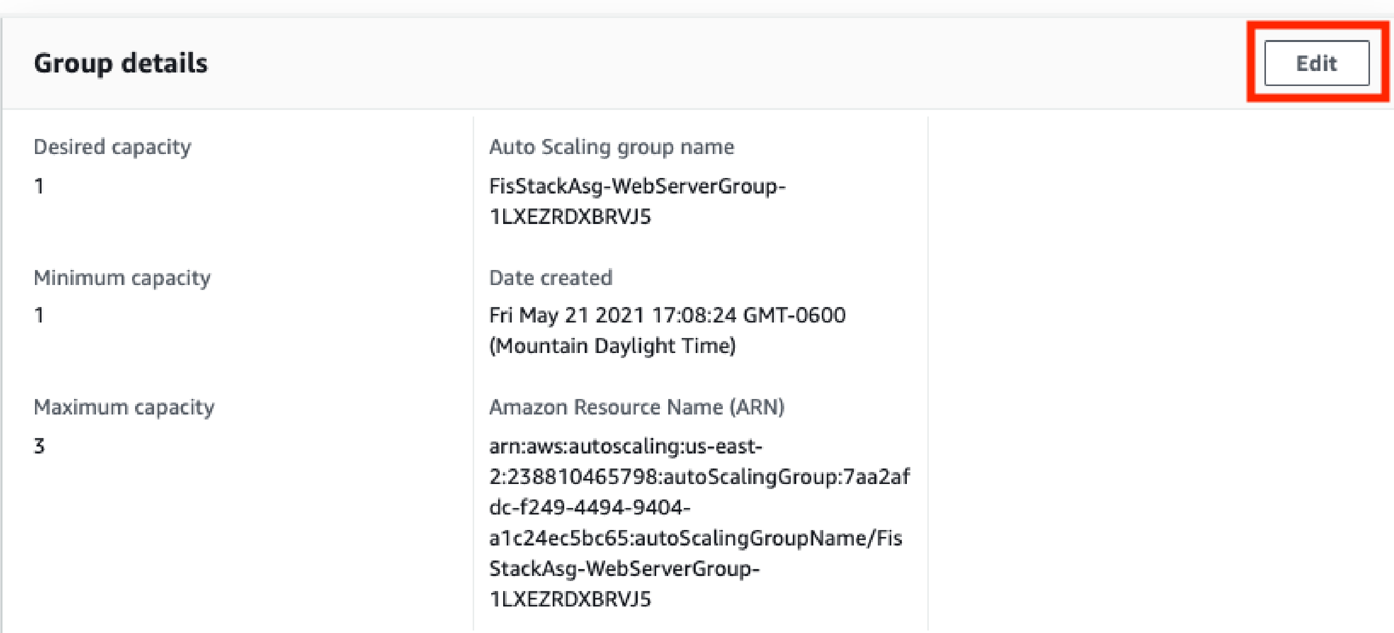


Learning and improving

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`.

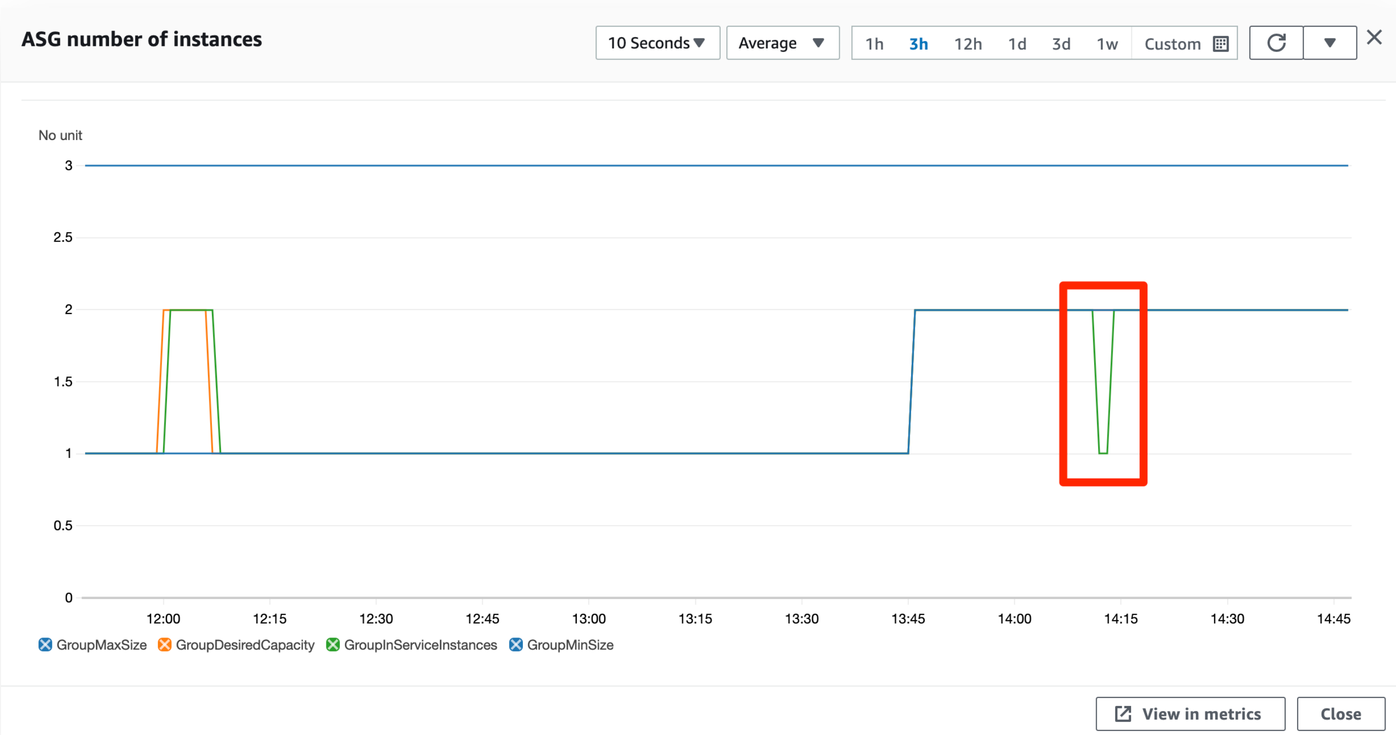


Check the ASG details or the CloudWatch Dashboard we explored in the previous section to make sure the active instances count has come up to 2.

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

From here you can explore how to set up experiments programatically using the **AWS CLI** or **AWS CloudFormation**, or move on exploring **more fault types** to inject.

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.

Note

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

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": [],
        "logConfiguration": {},
        "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**" in the AWS documentation, 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:..."
    }
]
```

Logging configuration

Experiment logging can be enabled in the `logConfiguration` section of the template. For logging to CloudWatch similar to the previous section would look like this (with the region and account number filled in appropriately):

```
"logConfiguration": {
    "cloudWatchLogsConfiguration": {
        "logGroupArn":
"arn:aws:logs:YOUR_REGION_HERE:YOUR_ACCOUNT_NUMBER_HERE:log-group:/fis-
workshop/fis-logs:*"
    },
    "logSchemaVersion": 1
},
```

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 created in the **Configuring permissions** section and appropriately set the region and account number for the log group ARN.

```
{
    "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": "FisWorkshopAsg-50Percent"
            }
        },
        "Wait": {
            "actionId": "aws:fis:wait",
            "parameters": {
                "duration": "PT3M"
            }
        }
    },
    "targets": {
        "FisWorkshopAsg-50Percent": {
            "resourceType": "aws:ec2:instance",
            "resourceTags": {
                "Name": "FisStackAsg/ASG"
            },
            "selectionMode": "PERCENT(50)"
        }
    },
    "stopConditions": [
        {
            "source": "none"
        }
    ],
    "roleArn":
"arn:aws:iam::YOUR_ACCOUNT_NUMBER_HERE:role/FisWorkshopServiceRole",
    "logConfiguration": {
        "cloudWatchLogsConfiguration": {
            "logGroupArn":
"arn:aws:logs:YOUR_REGION_HERE:YOUR_ACCOUNT_NUMBER_HERE:log-group:/fis-
workshop/fis-logs:*"
        },
        "logSchemaVersion": 1
    },
}
```

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 installed **AWS CLI** and configured AWS credentials for the CLI.

Creating templates

To create an experiment template, copy the above "Finished template" JSON into a file named `fis.json` and ensure you have changed the `roleArn` entry to the ARN of the role you created earlier. To find this role ARN, navigate to the **IAM Roles** page, search for the role `FisWorkshopServiceRole`, click on it and copy the value in **Role ARN**. 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, and while the content of the template at execution time is saved with the experiment data, this may make it harder 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)'
```

Validation procedure

Like in the previous section we will be using the AWS CloudWatch dashboard from the previous sections for validation, no additional setup required.

Run FIS experiment

Like in the previous section we will generate some load:

```
# Please ensure that LAMBDA_ARN, URL_HOME, and FIX_CLI_PARAM are still set from
previous section

# Run load for 5min, 3x in parallel because max per lambda is 1000
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
      }" \
    $FIX_CLI_PARAM \
    --invocation-type Event \
    /dev/null
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.

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

Additionally, the benefit of using AWS CLI to create and run experiments, is to allow you to document and automate the process for consistency. The best practice is to work with version controlled (e.g. in a Git repository)scripts, as shown here, or CloudFormation templates, as shown in the next section. With that you can setup peer review processes as well as the ability to run experiments continuously via a CI/CD pipeline.

From here you can explore how to set up experiments using **AWS CloudFormation** or move on exploring **more fault types** to inject.

EXPERIMENT (CLOUDFORMATION)

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

Note

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

CFN template format

The AWS CloudFormation template uses the same format as the API but capitalizes the first letter of section names. As such the AWS 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": "FisWorkshopAsg-50Percent"
            }
        },
        "Wait": {
            "ActionId": "aws:fis:wait",
            "Parameters": {
                "duration": "PT3M"
```

```
            }
        }
    },
    "Targets": {
        "FisWorkshopAsg-50Percent": {
            "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` and `AWS::Region` parameters 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": "FisWorkshopAsg-50Percent"
                        }
                    },
                    "Wait": {
                        "ActionId": "aws:fis:wait",
                        "Parameters": {
                            "duration": "PT3M"
```

```
                    }
                }
            },
            "Targets": {
                "FisWorkshopAsg-50Percent": {
                    "ResourceType": "aws:ec2:instance",
                    "ResourceTags": {
                        "Name": "FisStackAsg/ASG"
                    },
                    "SelectionMode": "PERCENT(50)"
                }
            },
            "StopConditions": [
                {
                    "Source": "none"
                }
            ],
            "RoleArn": {
                "Fn::Sub":
"arn:aws:iam::${AWS::AccountId}:role/FisWorkshopServiceRole"
            },
            "LogConfiguration": {
                "CloudWatchLogsConfiguration": {
                    "LogGroupArn": {
                        "Fn::Sub":
"arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/fis-workshop/fis-
logs:*"
                    }
                },
                "LogSchemaVersion": 1
            }
        }
    }
  }
}
```

Using the CFN template

A deep dive into **AWS 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.

Validation and running FIS experiment

The steps so far created an experiment template to run an experiment and validate outcomes you can follow the procedures outlined in the previous **Experiment (Console)** or **Experiment (CLI)** sections.

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

As mentioned in the previous section, it is valuable to version control the contents of experiment templates for consistency and automation by using AWS CLI scripting. Using CloudFormation goes one step further and allows you to version control the creation of experiment templates in addition to the template content.

In the next section we will explore more fault injection options.

AWS SYSTEMS MANAGER INTEGRATION

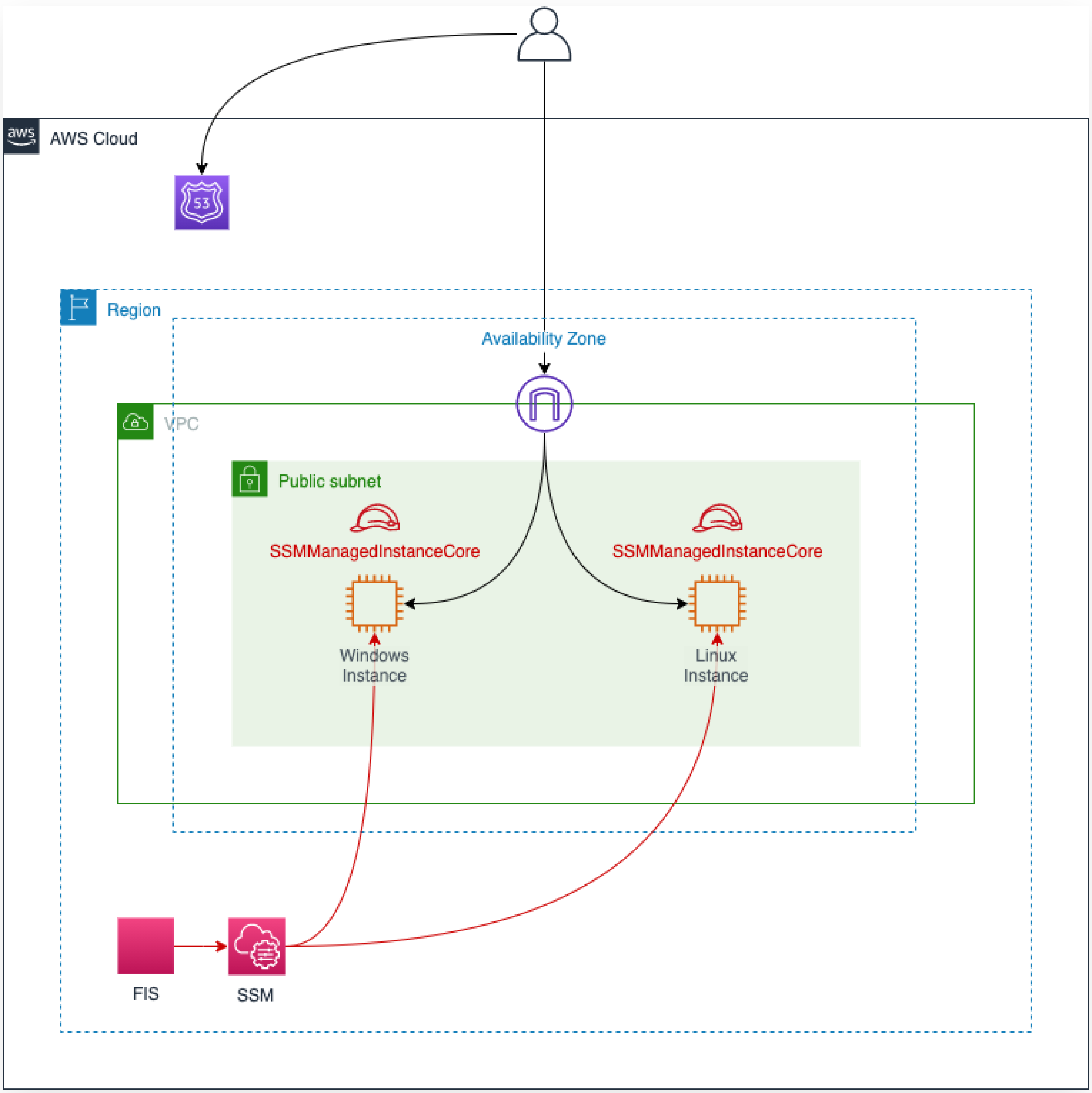
In this section, we will demonstrate how you can use **AWS Systems Manager (SSM)** along with AWS Fault Injection Simulator (FIS) to emulate faults within an EC2 Instance.

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 stopping instances or failing over Amazon Relational Database Service (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 account setup or in the **Start the workshop** section. If you would like to examine how these resources were defined you can examine the **AWS Cloud Formation template**.

LINUX CPU STRESS EXPERIMENT

Experiment idea

In this section we are exploring tooling so we will start without a hypothesis. However, we will provide some learnings and next steps at the end.

Specifically, 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

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.

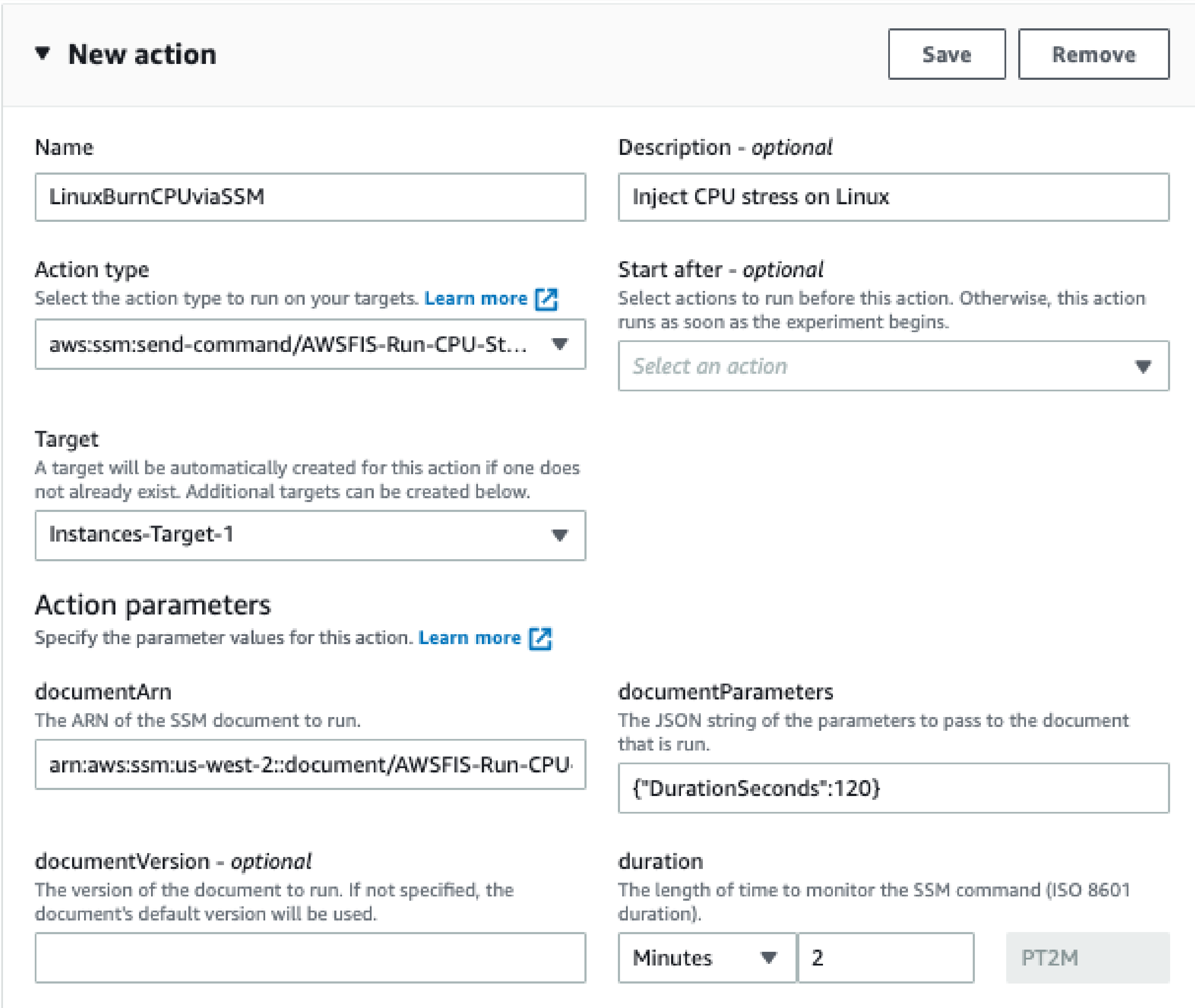
General template setup

- Create a new experiment template
  - Add a name for the template using a Tag with key as `Name` and value as `LinuxBurnCPUviaSSM` (located at bottom of page)
  - Add `Description` of `Inject CPU stress on Linux`
  - Select `FisCpuStress-FISRole` as execution role

Action definition

In the "Actions" section select the **"Add Action"** button.

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` minutes which tells FIS how long to wait for a result. Leave the default “Target” `Instances-Target-1` and select **“Save”**.

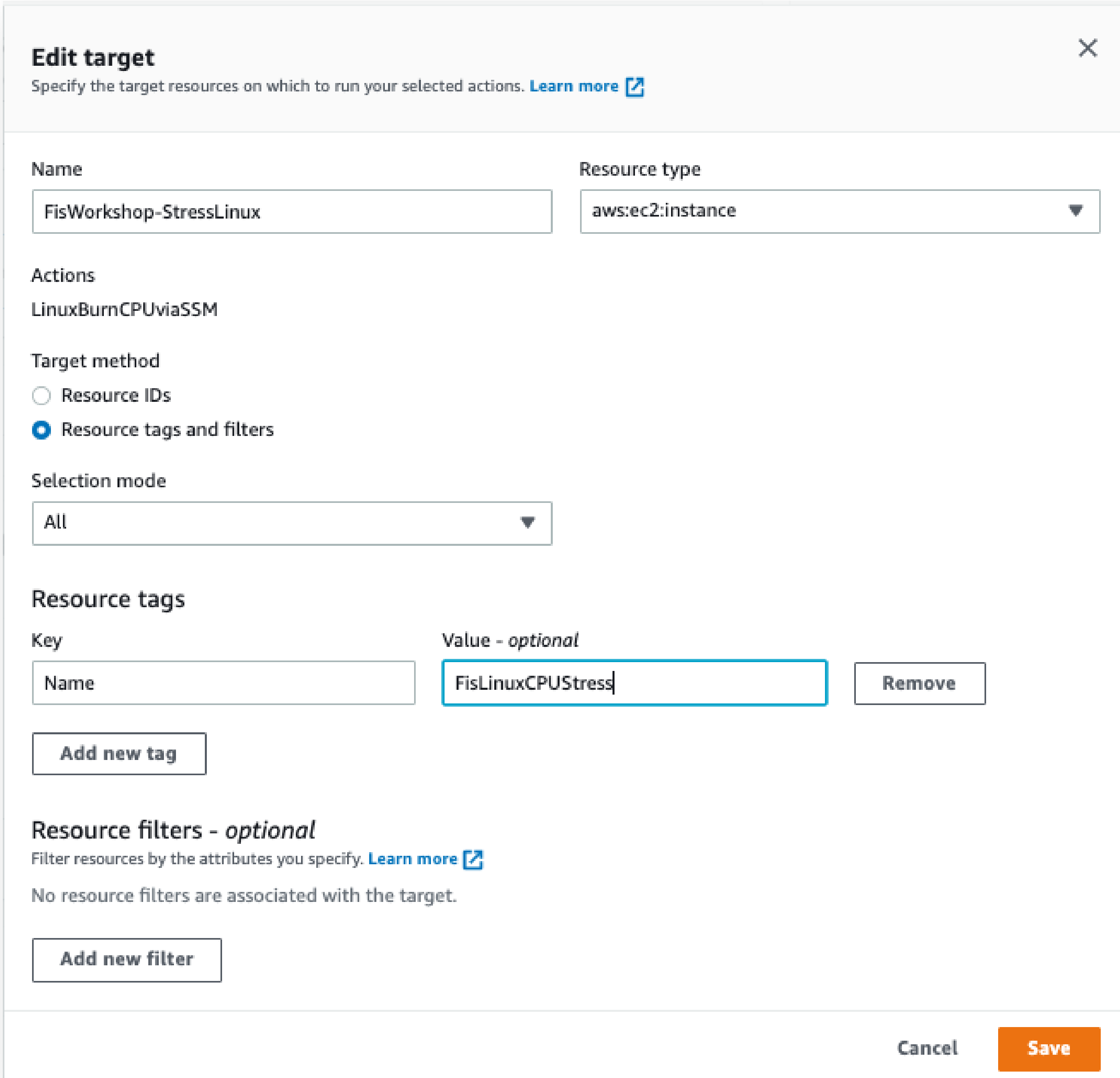


This action will use **AWS Systems Manager Run Command** to run the `AWSFIS-Run-Cpu-Stress` command document against our targets for two minutes.

Target selection

For this action we need to designate EC2 instance targets on which to run the commands. Go to the “Targets” section, select the `Instances-Target-1` section, and select **“Edit”**.

You may leave the default name `Instances-Target-1` but for maintainability we rcommend using descriptive target names. Change the name to `FisWorkshop-StressLinux` (this will automatically update the name in the action as well) and make sure "Resource type" is set to `aws:ec2:instances`. To select our target instances by tag select "Resource tags and filters" and keep selection mode `ALL`. Select **"Add new tag"** and enter a "Key" of `Name` and a "Value" of `FisLinuxCPUStress`. Finally select **"Save"**.



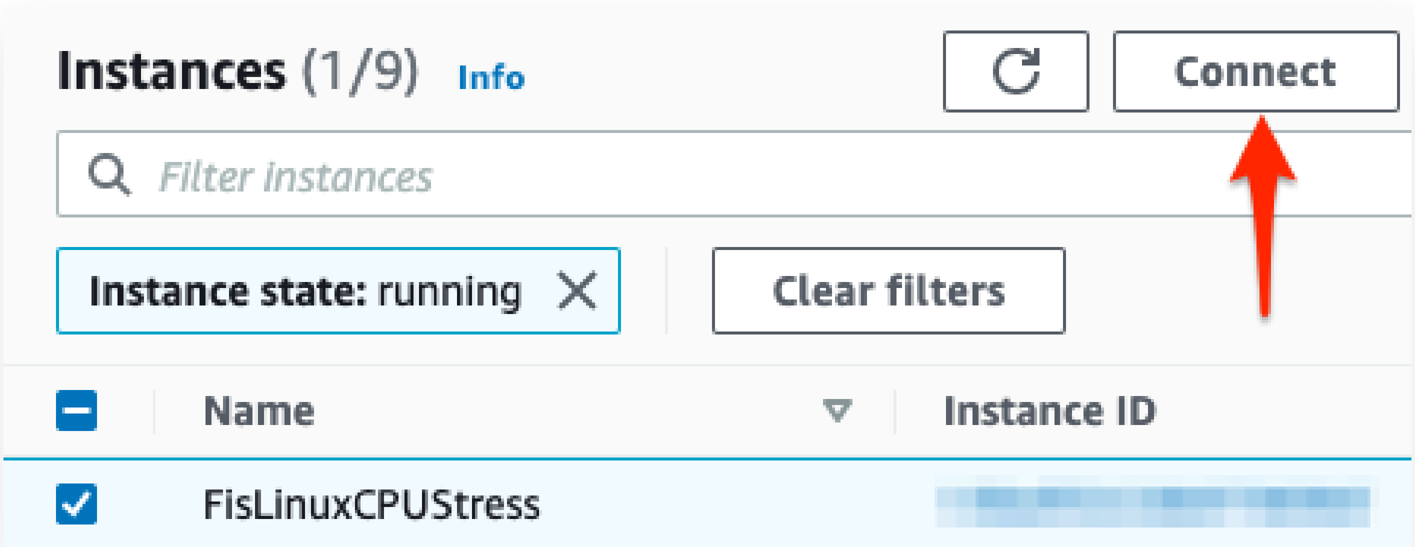
Creating template without stop conditions

Select **"Create experiment template"** and confirm that you wish to create a template without stop conditions.

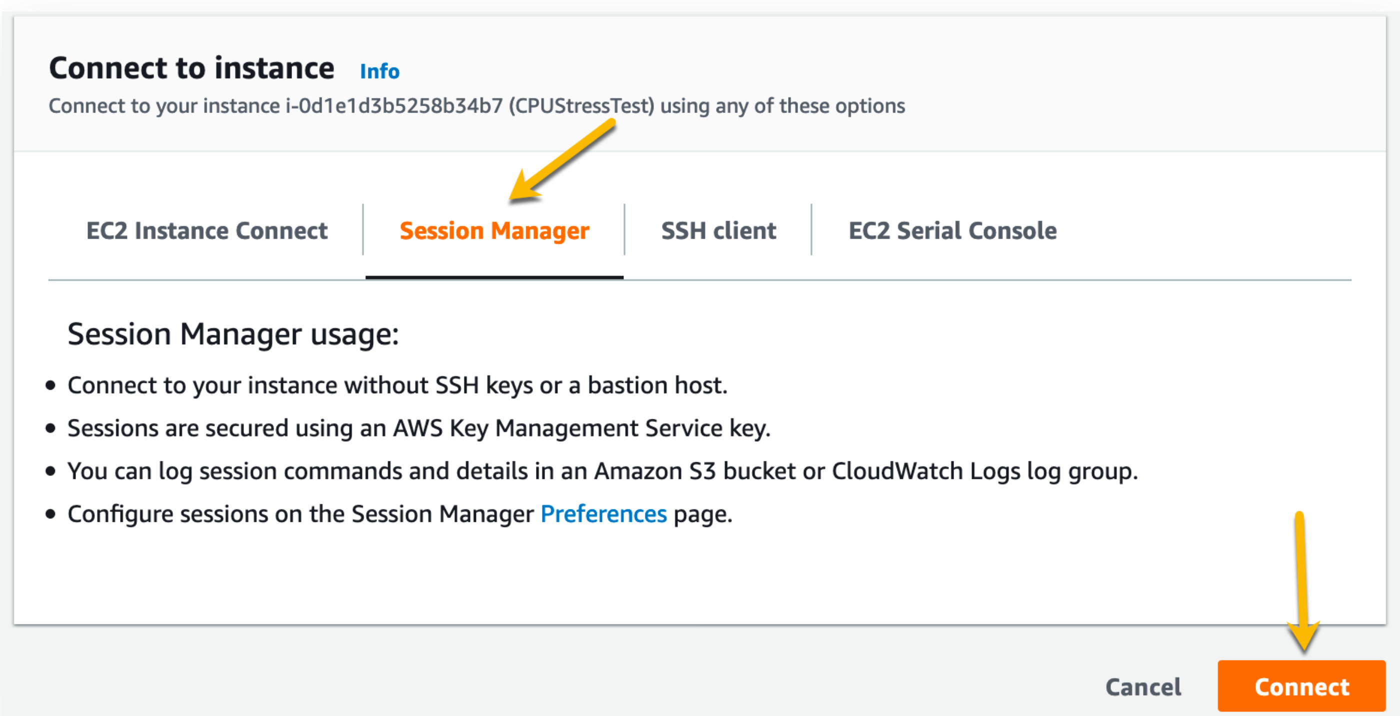
Validation procedure

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.



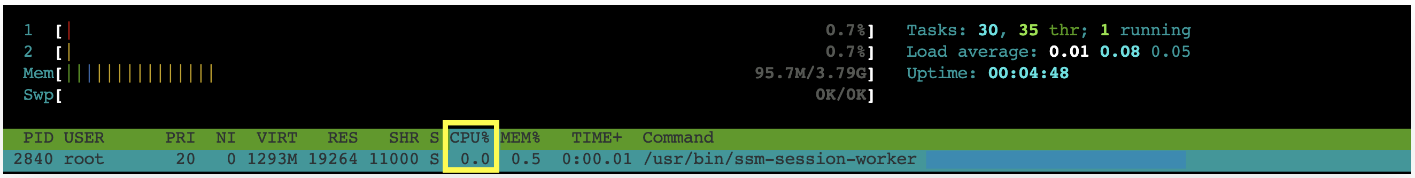
2. Select **"Session Manager"** and select **"Connect"**.



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

```
htop
```

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.



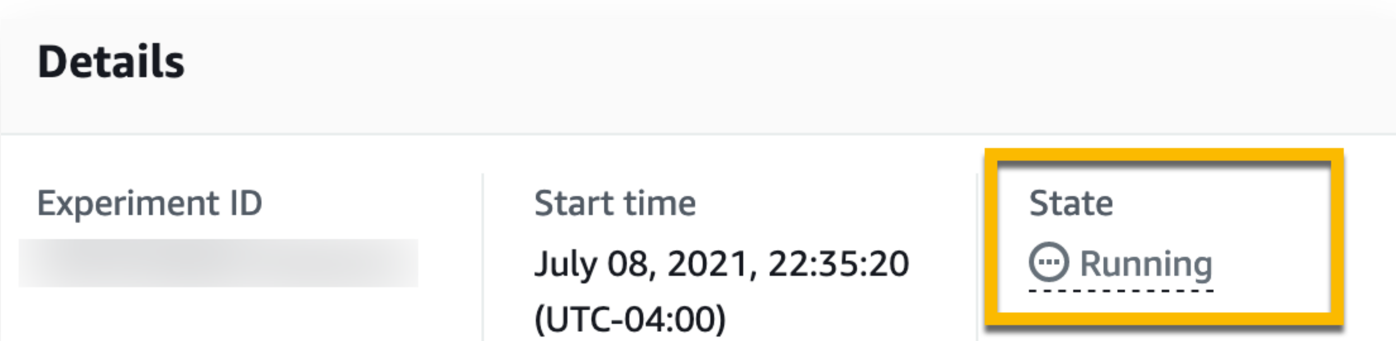
Run CPU Stress Experiment

Note

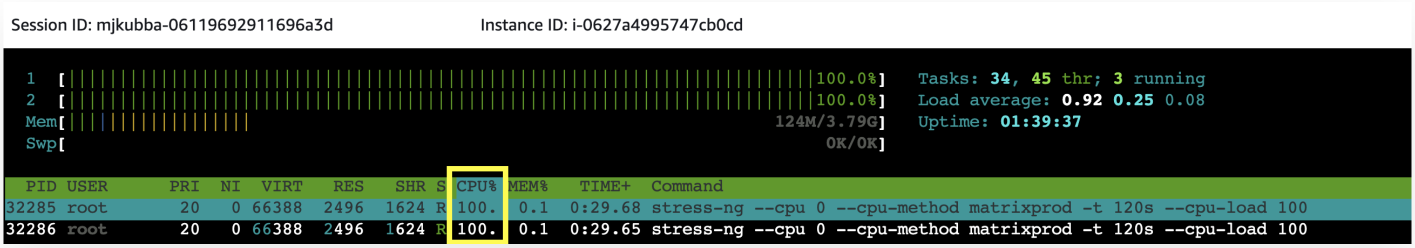
We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

Keep the EC2 instance session with `top` running. In a new browser window navigate to the **AWS Fault Injection Simulator Console** and start the experiment:

- use the `LinuxBurnCPUviaSSM`
- add a `Name` tag of `FisWorkshopLinuxStress1`
- confirm that you want to start the experiment
- ensure that the "State" is `Running`



In the EC2 terminal window watch the CPU percentage displayed by `top`: 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.



Congratulations 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

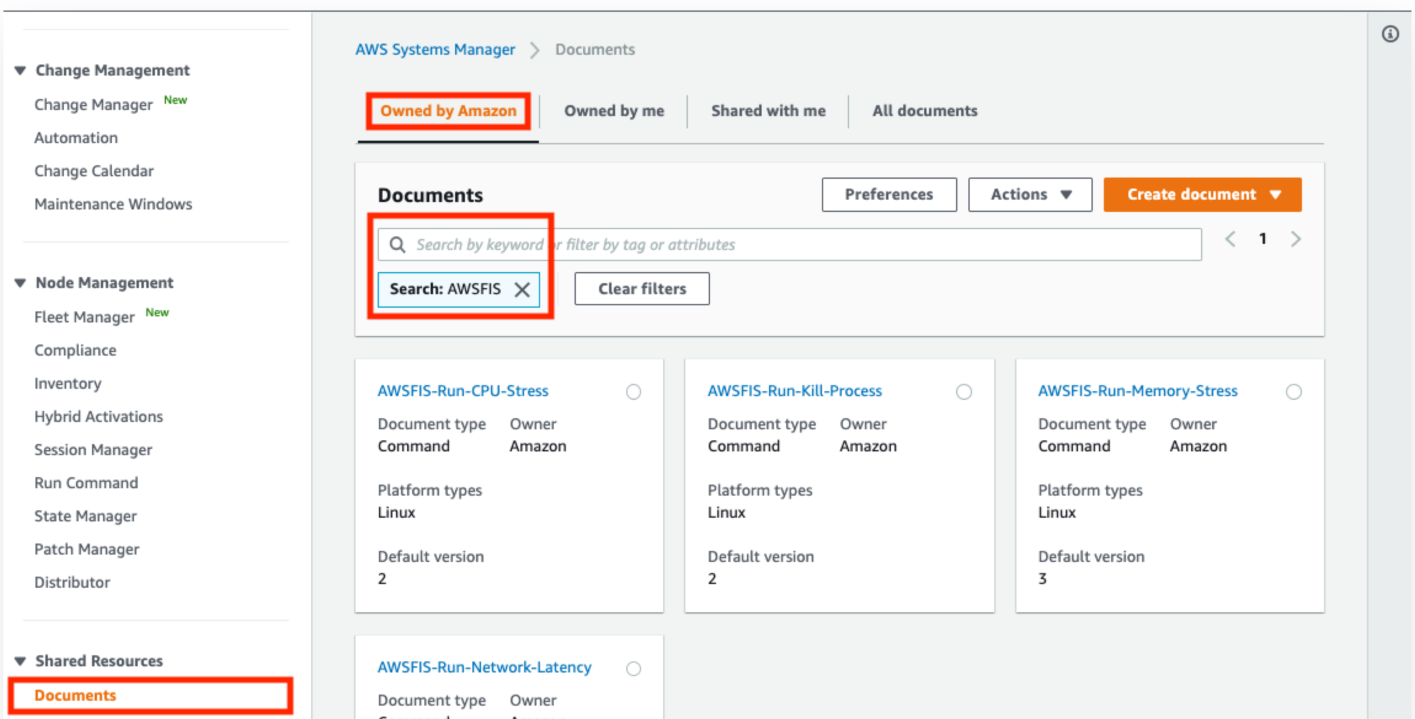
Since this instance wasn't doing anything, there aren't any actions. To think about how to use this to test a hypothesis and make an improvement, consider running the same experiment against the ASG instances from the **First Experiment** section. Maybe you could use this to tune the optimal CPU levels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

WORKING WITH SSM DOCUMENTS

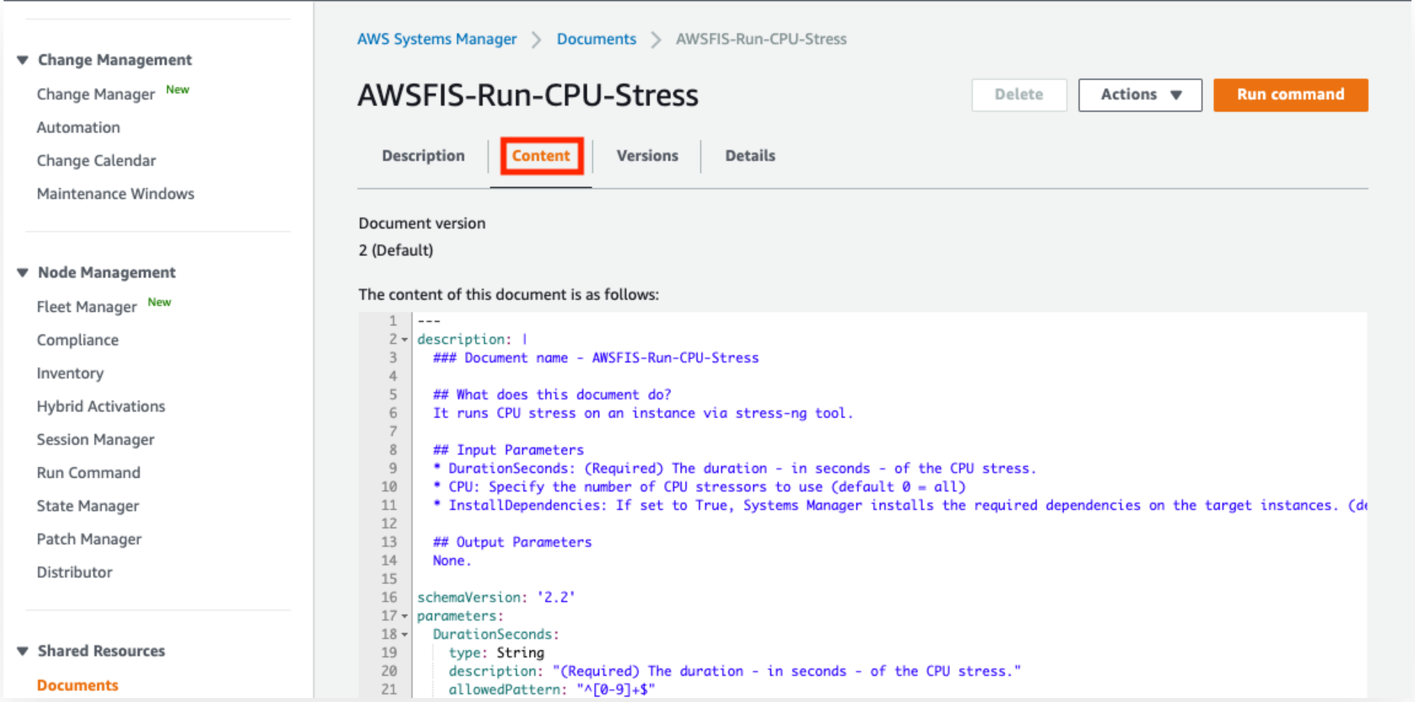
Pre-configured SSM documents

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 as shown in the next section. 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. The ARN can easily be constructed from the document name (and the owner account ID if the template is shared with you) using 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 a Remote Desktop Protocol (RDP) client on your local machine. This section cannot be performed from a Cloud9 instance.

Warning

The shell syntax in this section is written for bash. If you are on a Mac with zsh as default shell please switch to bash to execute the commands in this section.

Experiment idea

In this section we are exploring tooling so we will start without a hypothesis. However, we will provide some learnings and next steps at the end.

Specifically, 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

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 a name for the template using a Tag with key as `Name` and value as `WindowsBurnCPUviaSSM` (located at bottom of page)
  - Add `Description` of `Inject CPU stress on Windows`
  - Select `FisCpuStress-FISRole` as execution role

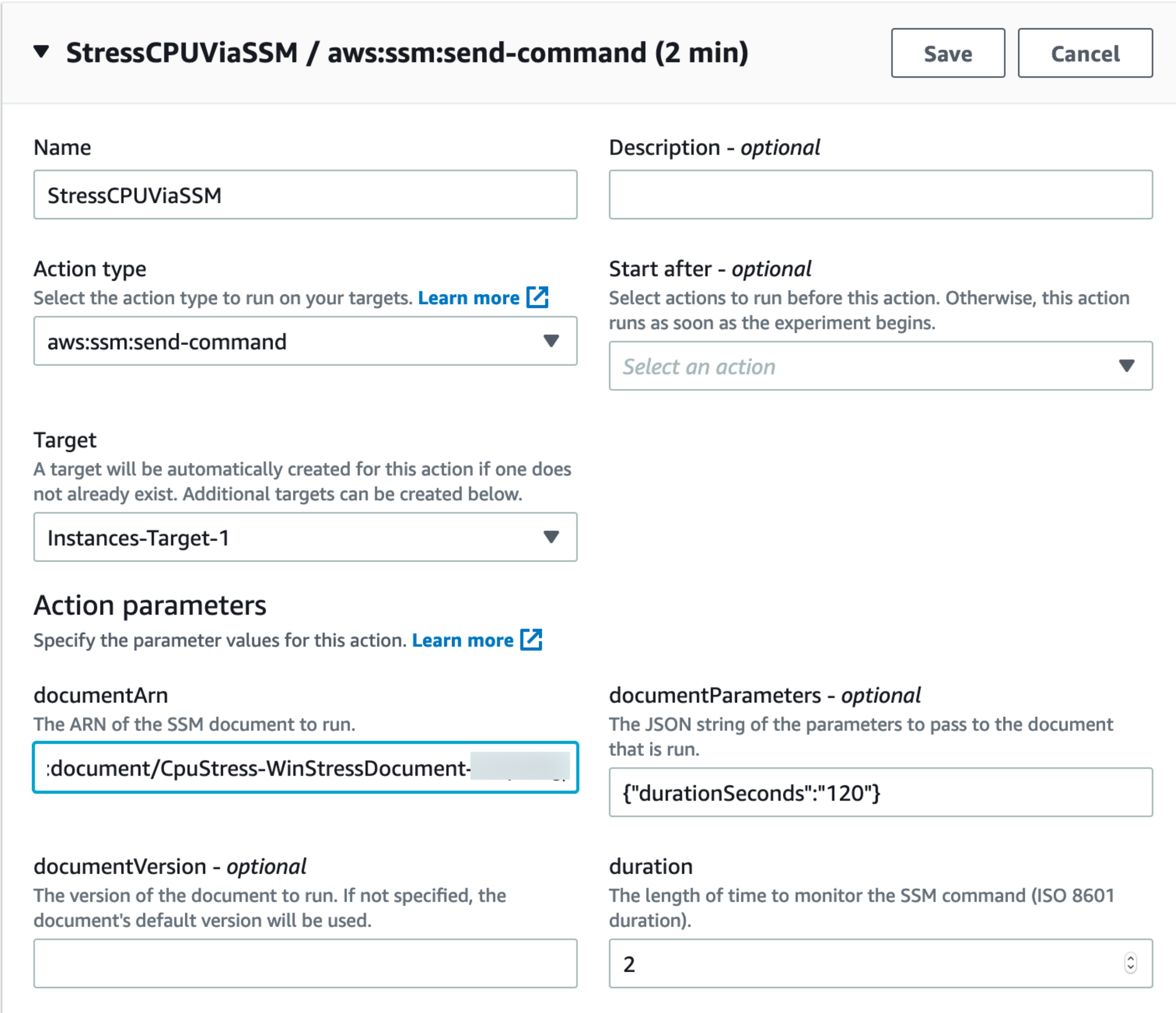
Action definition

In the "Actions" section select the **"Add Action"** button.

"Name" the action as `StressCPUViaSSM`, and under "Action Type" select the `aws:ssm:send-command` action. Currently there is no 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"**, select 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` minutes which tells FIS how long to wait for a result. Leave the default "Target" `Instances-Target-1` and select **"Save"**.

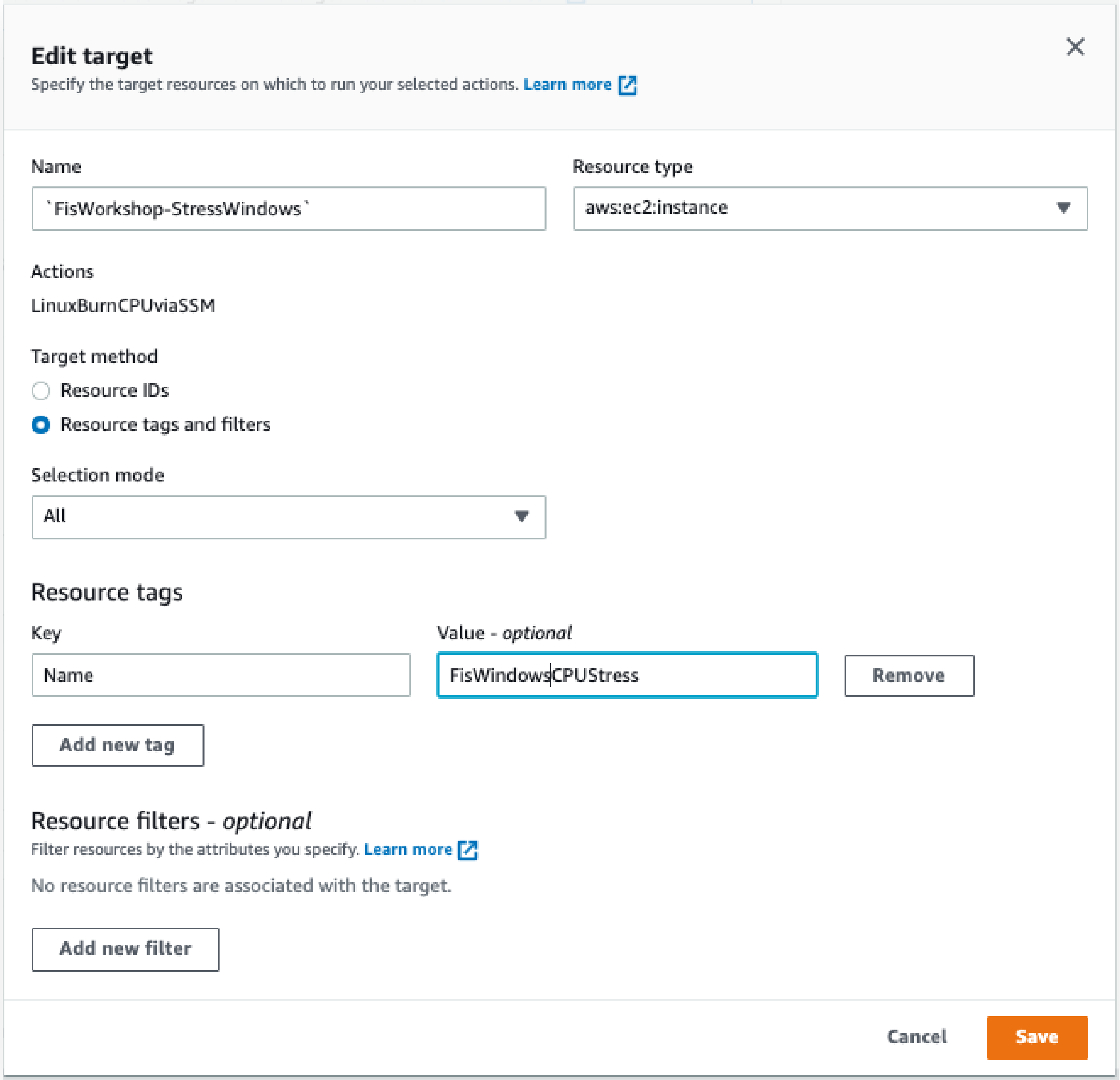


This action will use **AWS Systems Manager Run Command** to run the `FisCpuStress-WinStressDocument` document against our targets for two minutes.

Target selection

For this action we need to designate EC2 instance targets on which to run the commands. Go to the "Targets" section, select the `Instances-Target-1` section, and select **"Edit"**.

You may leave the default name `Instances-Target-1` but for maintainability we rcommend using descriptive target names. Change the name to `FisWorkshop-StressWindows` (this will automatically update the name in the action as well) and make sure "Resource type" is set to `aws:ec2:instances`. To select our target instances by tag select "Resource tags and filters" and keep selection mode `ALL`. Select **"Add new tag"** and enter a "Key" of `Name` and a "Value" of `FisWindowsCPUStress`. Finally select **"Save"**.



Creating template without stop conditions

Select **"Create experiment template"** and confirm that you wish to create a template without stop conditions.

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.

> **Warning**
>
> The password reset command will report "success" even if a trivial password is picked but will not reset the password in that case. Please pick a password with sufficient complexity (uppercase, lowercase, numbers, symbols) to ensure successfull password reset.

```
# For readability - passing passwords this way is not secure
# Pick complex password
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.

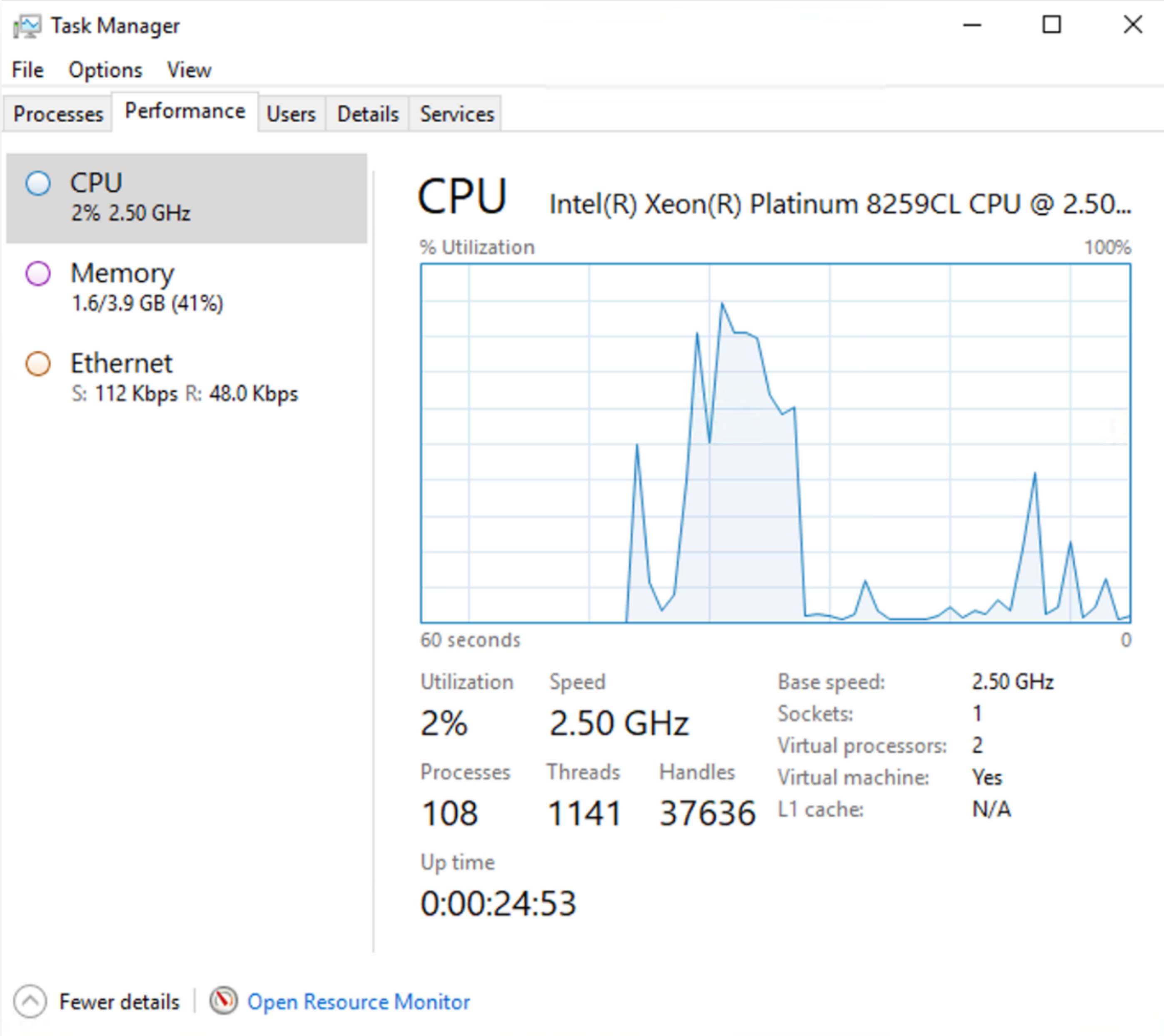
2. Run the following command first, this will securely forward local port 56788 to port 3389 on the Windows EC2 Instance. Note that we are targeting a specific instance by passing the `TMP_INSTANCE` variable from above.

```
# 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” (or by using the SHIFT-CTRL-ESC keyboard sortcut). 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

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.

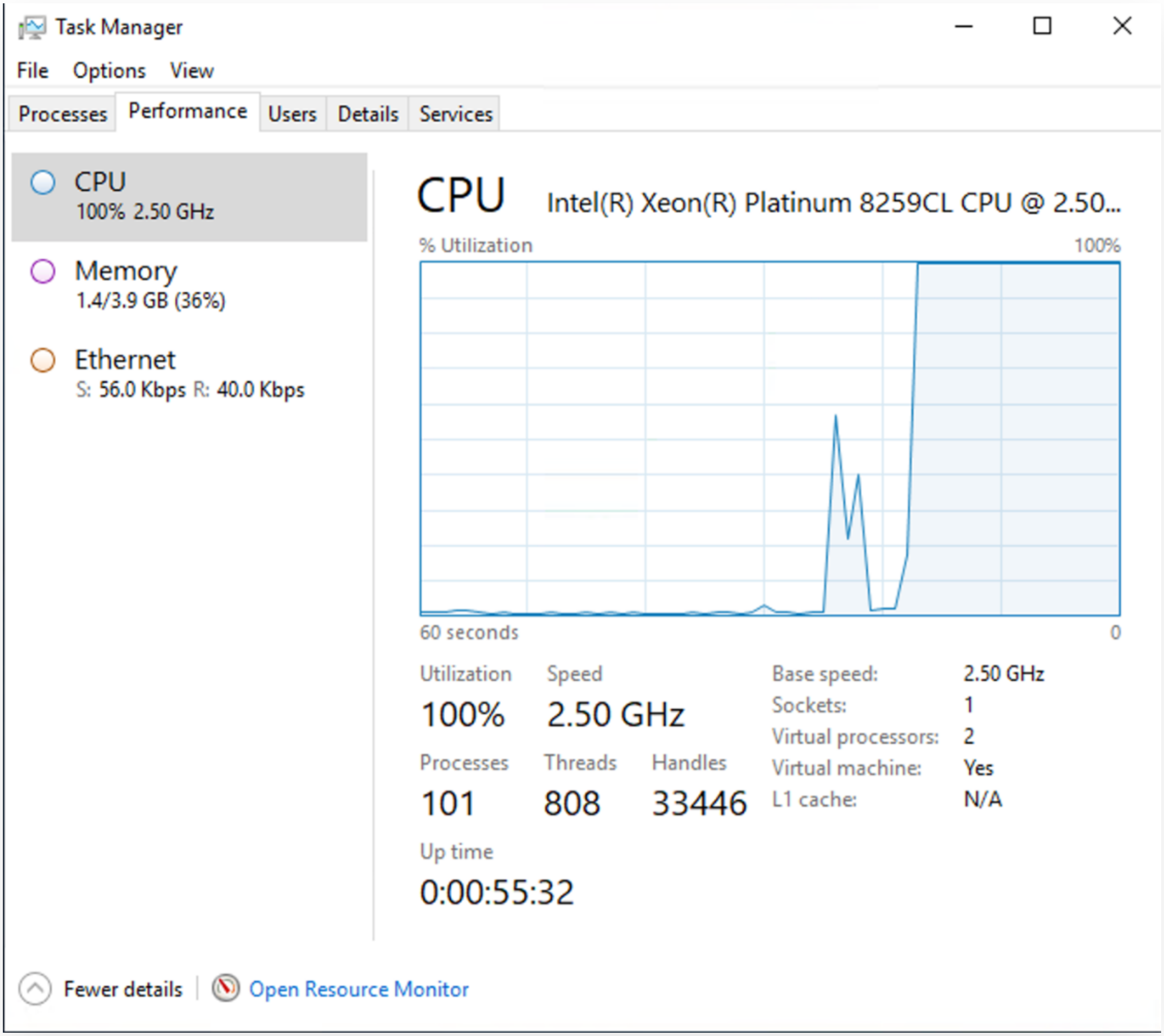
Keep the RDP session with "Task Manager" running. In a new browser window navigate to the **AWS Fault Injection Simulator Console** and start the experiment:

- use the `WindowsBurnCPUviaSSM`
- add a `Name` tag of `FisWorkshopWindowsStress1`
- confirm that you want to start the experiment

- ensure that the "State" is `Running`

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.



Congratulations 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.

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 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.

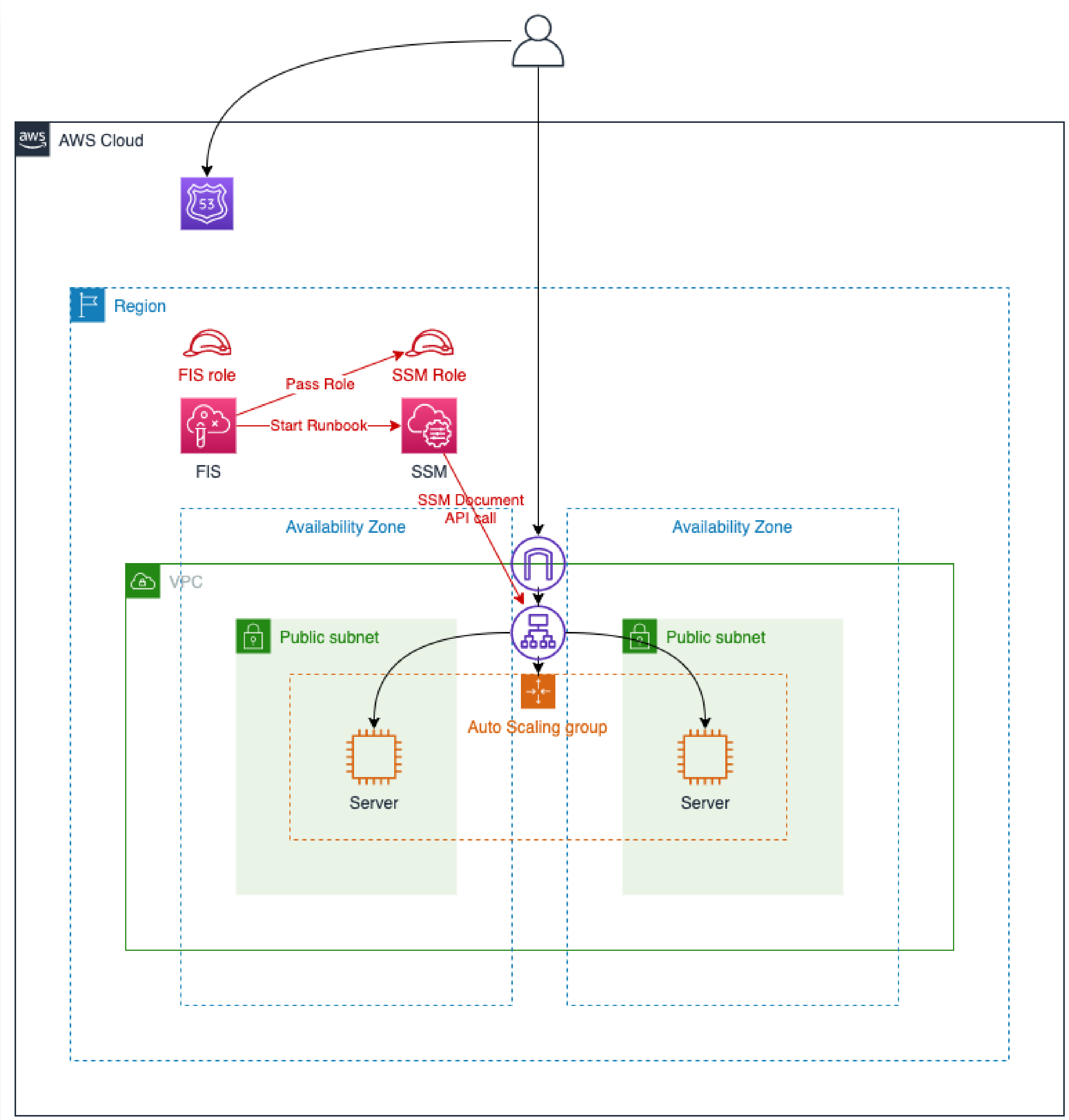
FIS SSM START AUTOMATION SETUP

Warning

The automation in this section creates and modifies IAM roles. With the current workshop description this will not work in Cloud9. Please either perform the role creation on the console or follow the instructions in **Configure AWS CloudShell** to use **AWS CloudShell**. If you use CloudShell, you will need to check out the GitHub repository in CloudShell as described in **Provision AWS resources**.

In the previous sections we used AWS 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 (see **EC2 Actions** and **Autoscaling Actions**):

```
{
    "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 blocks 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`

```
cd ~/environment/aws-fault-injection-simulator-workshop
cd workshop/content/030_basic_content/040_ssm/050_direct_automation
```

```
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

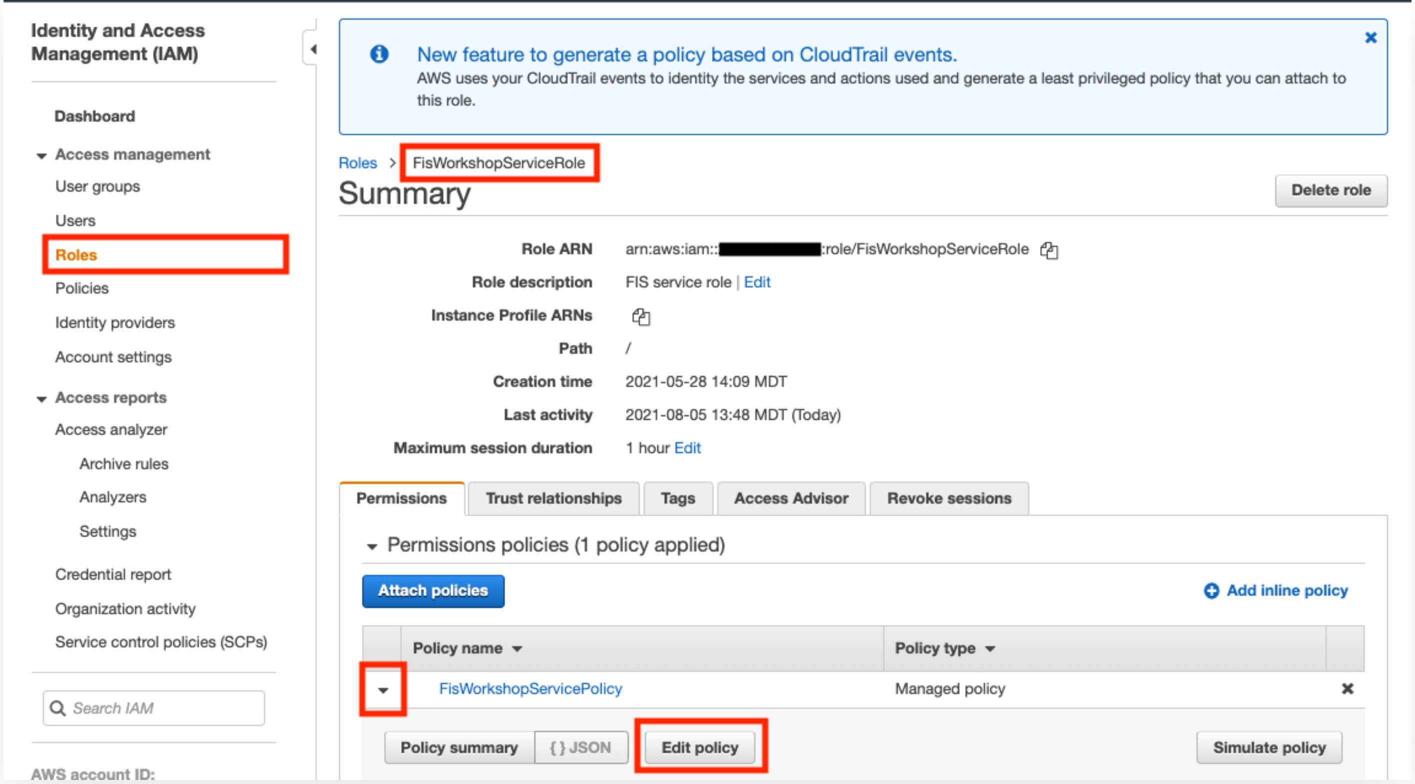
Update FIS service role

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

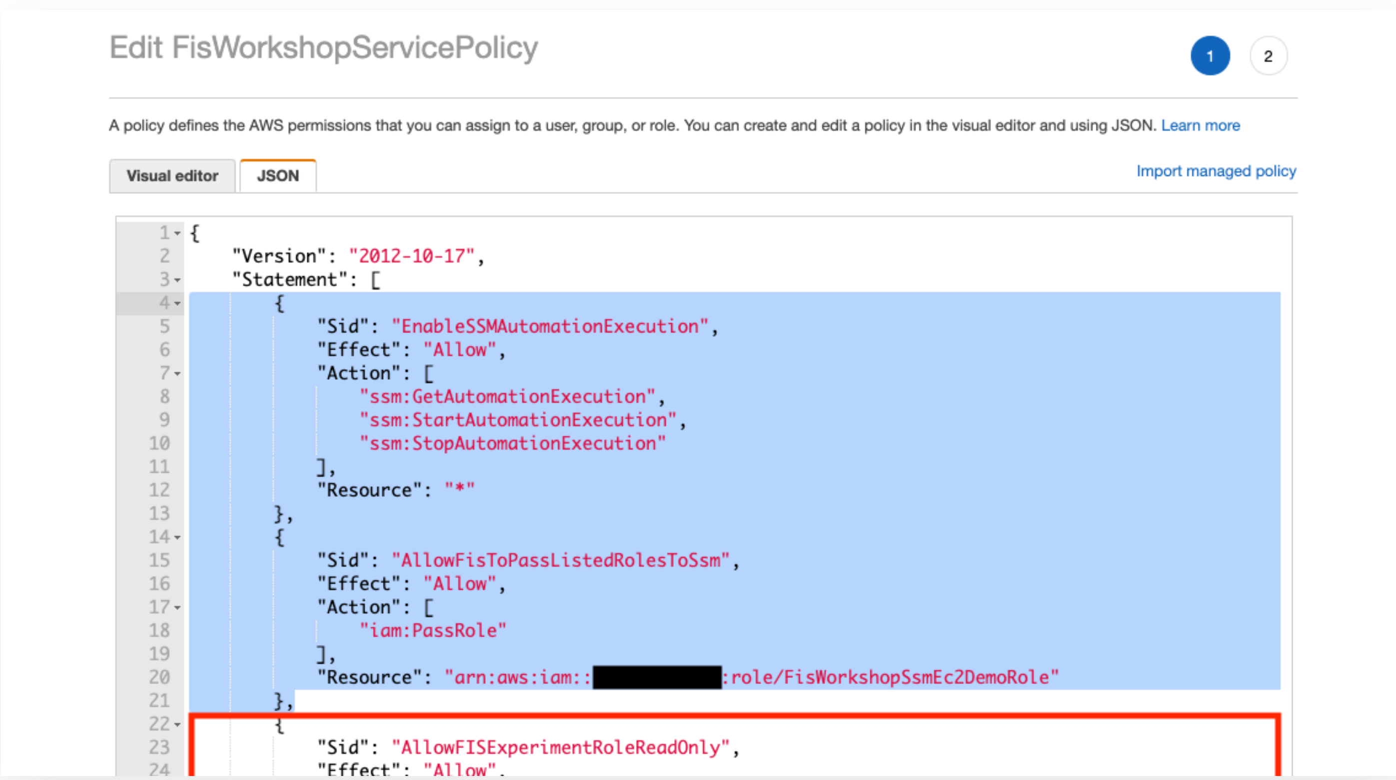
```
        {
            "Sid": "EnableSSMAutomationExecution",
            "Effect": "Allow",
            "Action": [
                "ssm:GetAutomationExecution",
                "ssm:StartAutomationExecution",
                "ssm:StopAutomationExecution"
            ],
            "Resource": "*"
        },
        {
            "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 `FisWorkshopSsmEc2DemoRole` 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. This has no real value in and of itself but is a starting point for the advanced SSM documents in the **Common Scenarios** section. Copy the YAML below into a file named `ssm-terminate-instances-asg-az.yaml`

```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
```

```yaml
    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:

```
cd ~/environment/aws-fault-injection-simulator-workshop
cd workshop/content/030_basic_content/040_ssm/050_direct_automation

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 constructed in the previous step. See explanation at the end of the **Working with SSM documents** section.
- `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 Auto Scaling group (ASG) starting with `FisStackAsg`, then copy the full name of the ASG, e.g. `FisStackAsg-ASG46ED3070-1RAQ30VBKLWE1`
- `SSM_ROLE_ARN` - use the role ARN of the `FisWorkshopSsmEc2DemoRole` created in 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 of the `FisWorkshopServiceRole` that you updated in 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\"}",
                "maxDuration": "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
```

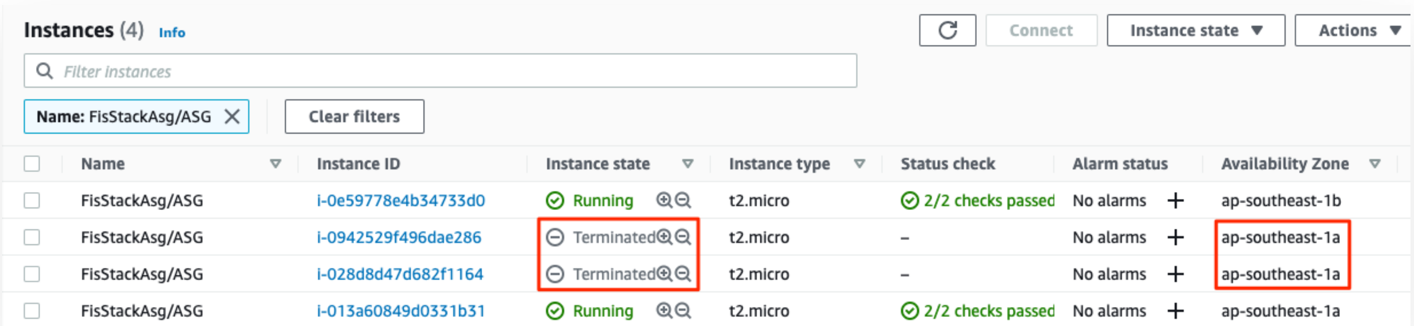
Note the experiment template ID as we will use this to start the experiment next.

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:

```
TEMPLATE_ID=[PASTE_ID_HERE]
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 the AZ we selected. If the experiment runs successfully, all of our instances in that 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 **AWS Systems Manager console** and selecting **"Automation"** in the burger menu 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 the `onCancel` action on the SSM document (see the end of the **Working with SSM documents** section). Edit the FIS template and ensure that you allow enough time in FIS for the SSM document to finish.

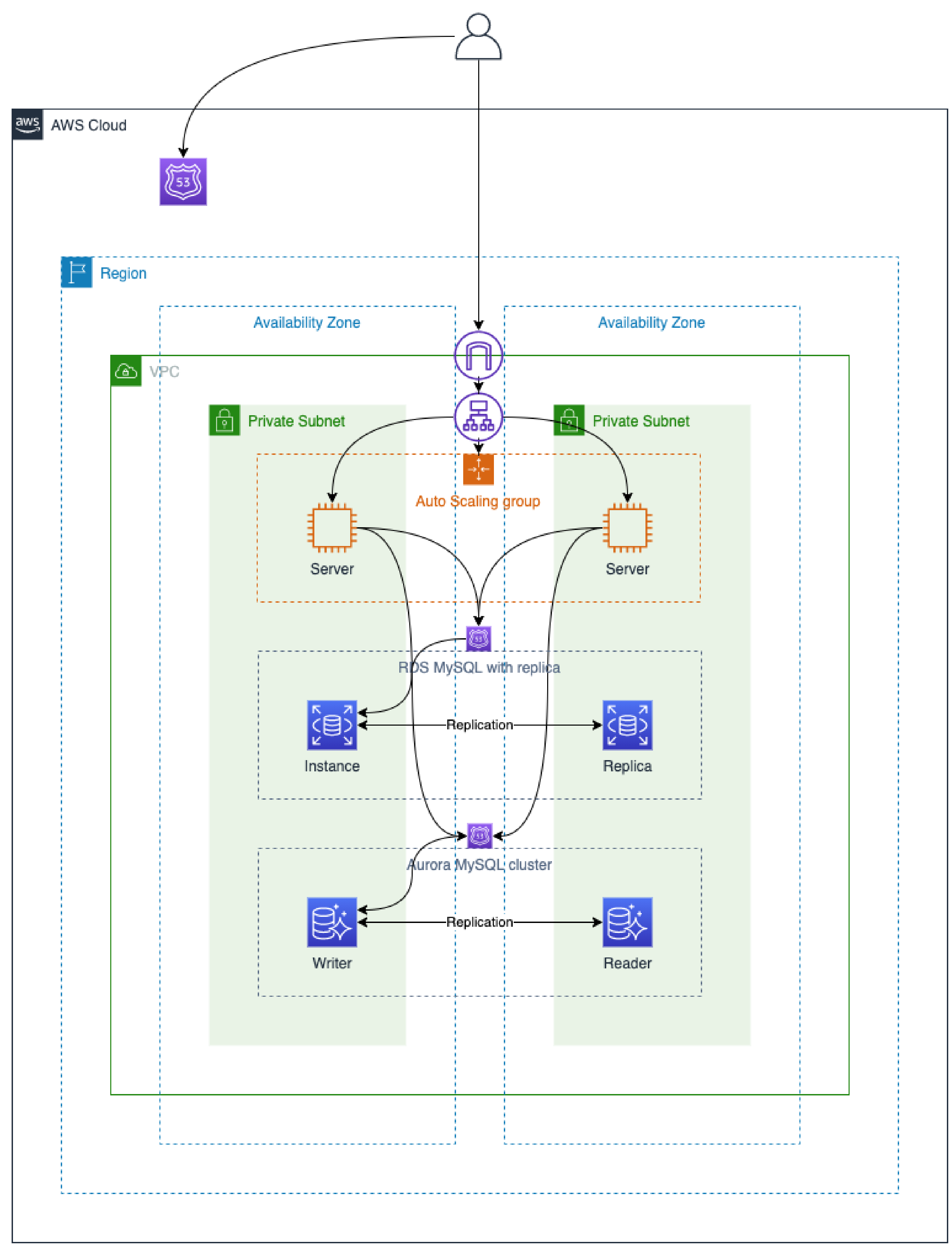
SSM ADDITIONAL RESOURCES

For additional AWS Systems Manager (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 Amazon Relational Database Service (RDS) MySQL and Amazon Aurora (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 Availability Zones (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.

Note

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell:

`source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

General template setup

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

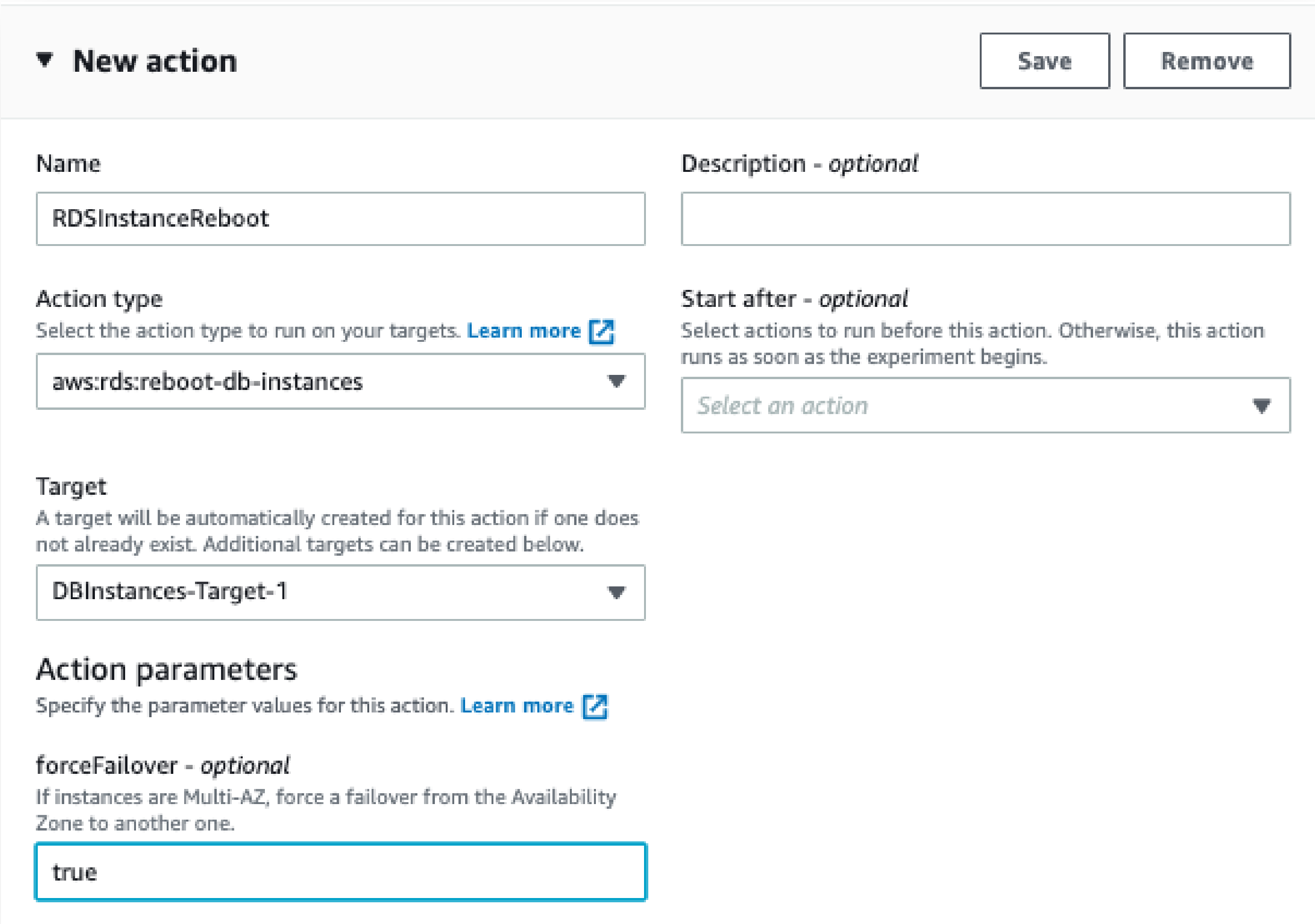
Action definition

In the “Actions” section select the **“Add Action”** button.

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`.

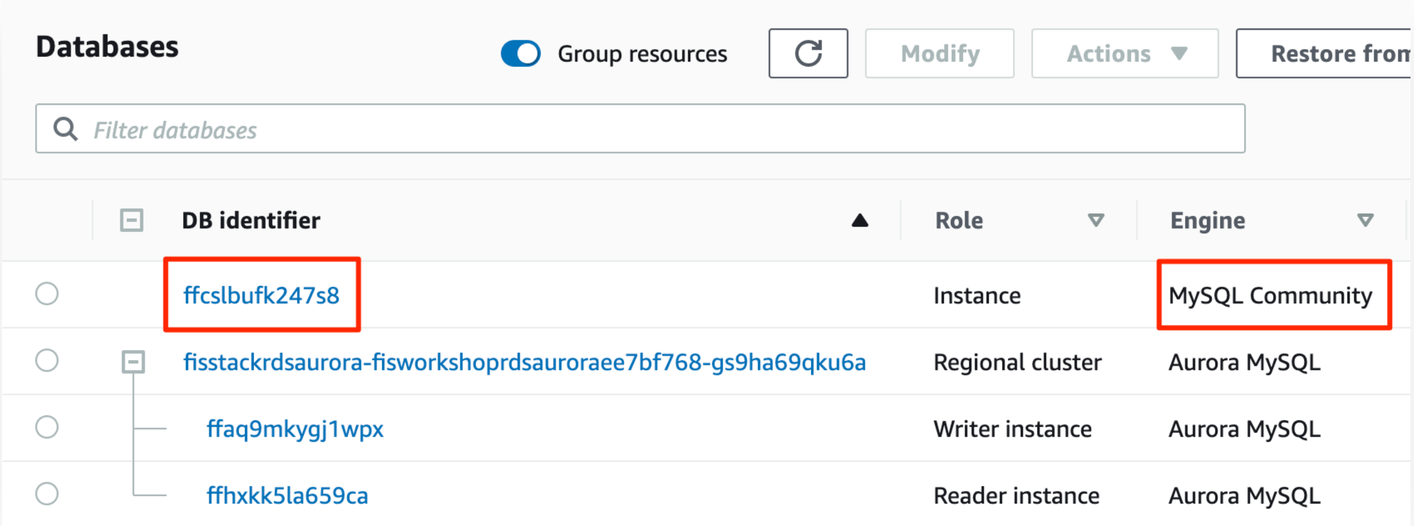
Leave the default “Target” `DBInstances-Target-1` and select **“Save”**.



This action will reboot the main database instance and, due to the “forceFailover” setting, promote the previous replica and update the information associated with the connection string (see below).

Target selection

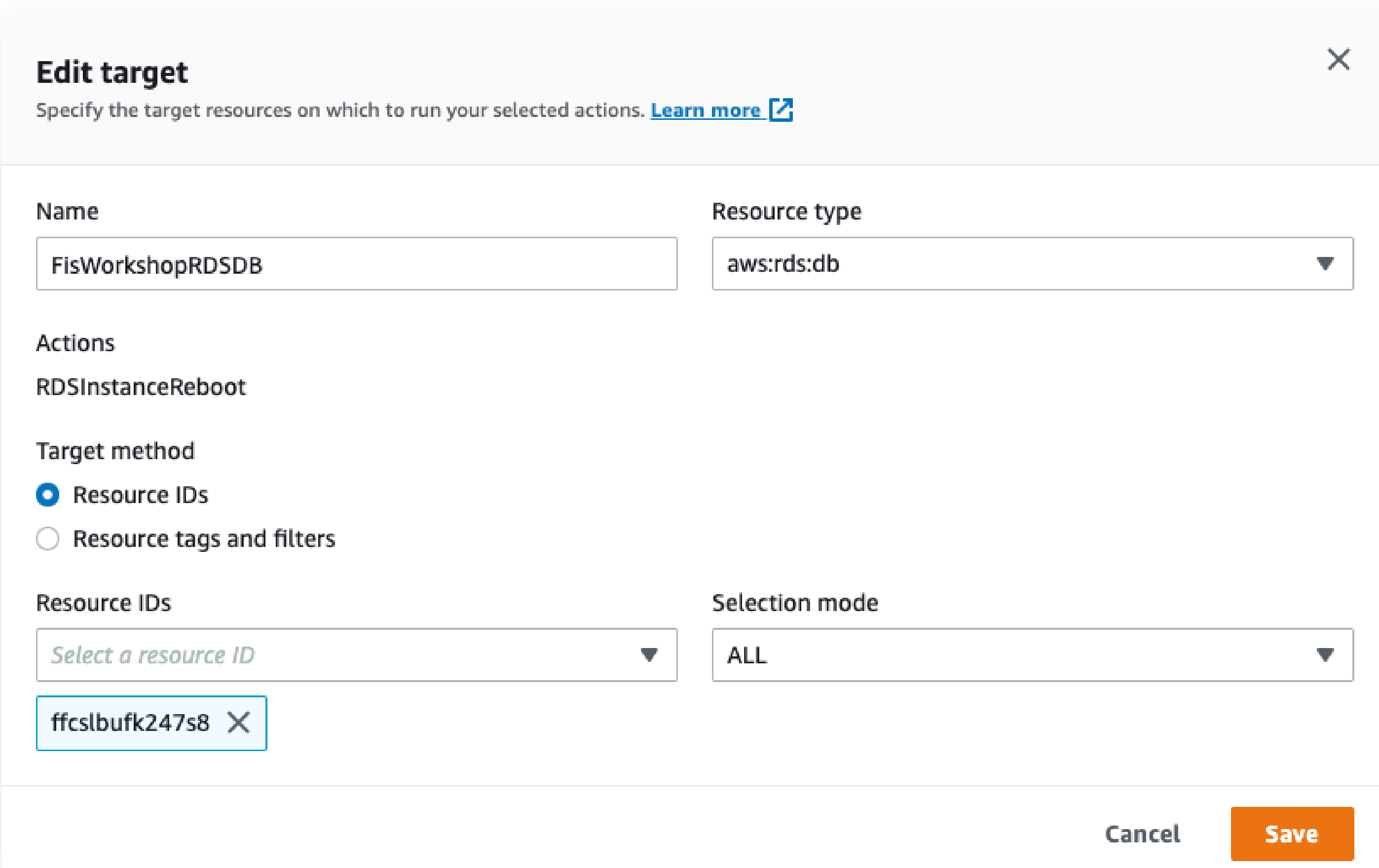
For this action we need to select our RDS MySQL “Instance”. For this we will need to know the instance resource ID. To find this ID open a new browser window and navigate to the **RDS console**). Note the “DB identifier” for the target DB instance, the one with “Engine” type “MySQL Community”.



Return to the FIS experiment setup, scroll to the "Targets" section, select `DBInstances-Target-1` and select **"Edit"**.

You may leave the default name `Instances-Target-1` but for maintainability we rcommend using descriptive target names. Change "Name" to `FisWorkshopRDSDB` for name (this will automatically update the name in the action as well) and make sure "Resource type" is set to `aws:rds:db`.

For "Target method" we will select resources based on the ID. Select the "Resource IDs" checkbox. Under "Resource IDs" pick the target DB instance matching the "DB Identifier" you noted above, then select `All` from "Selection mode". Select **"Save"**.



Creating template without stop conditions

Select **"Create experiment template"** and confirm that you wish to create a template without stop conditions.

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 (see **full code in GitHub**):

```
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))
```

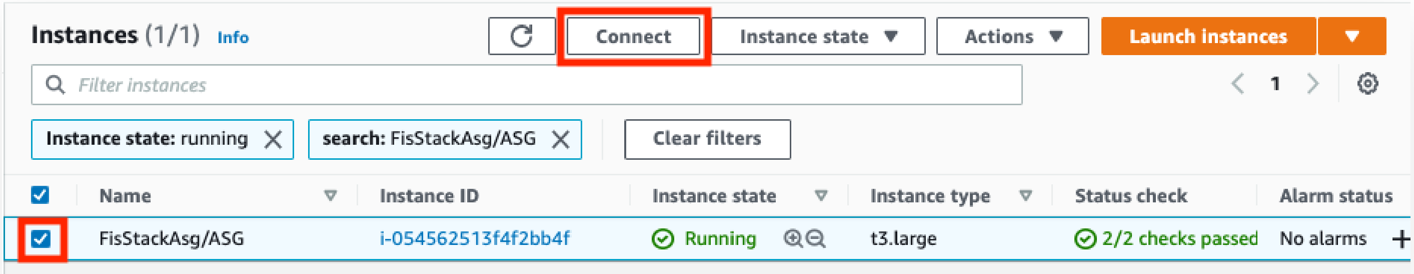
We would expect that this would keep writing output while the DB is available, stop while it's failing over and restart when the DB has successfully failed over.

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:

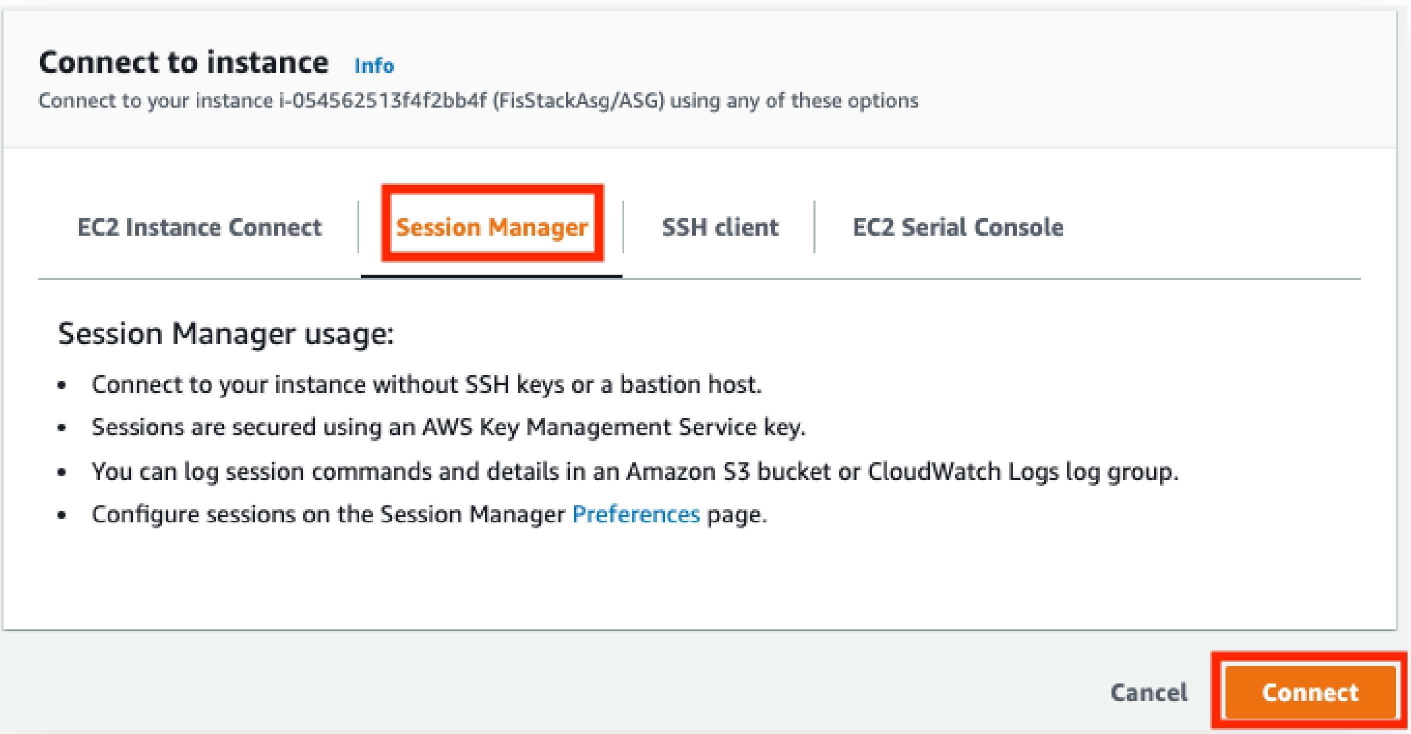
```
AURORA                          RDS
10.0.89.224                     10.0.95.247
(7711, 2282)                    (5419, 15189)
(7710, 5964)                    (5418, 15841)
(7709, 10634)                   (5417, 8071)
(7708, 4834)                    (5416, 21948)
(7707, 20291)                   (5415, 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 select 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 (review **code in GitHub**) 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
```

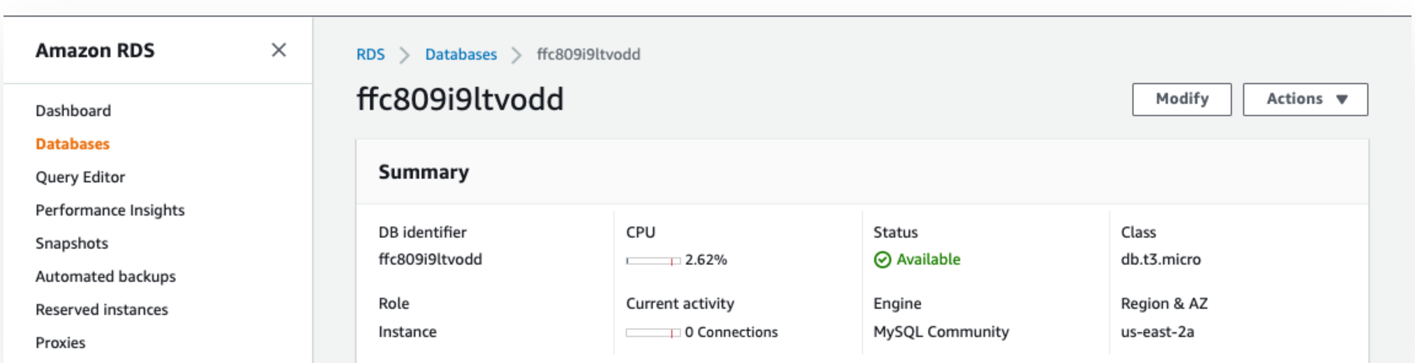
Run FIS experiment

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.

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":

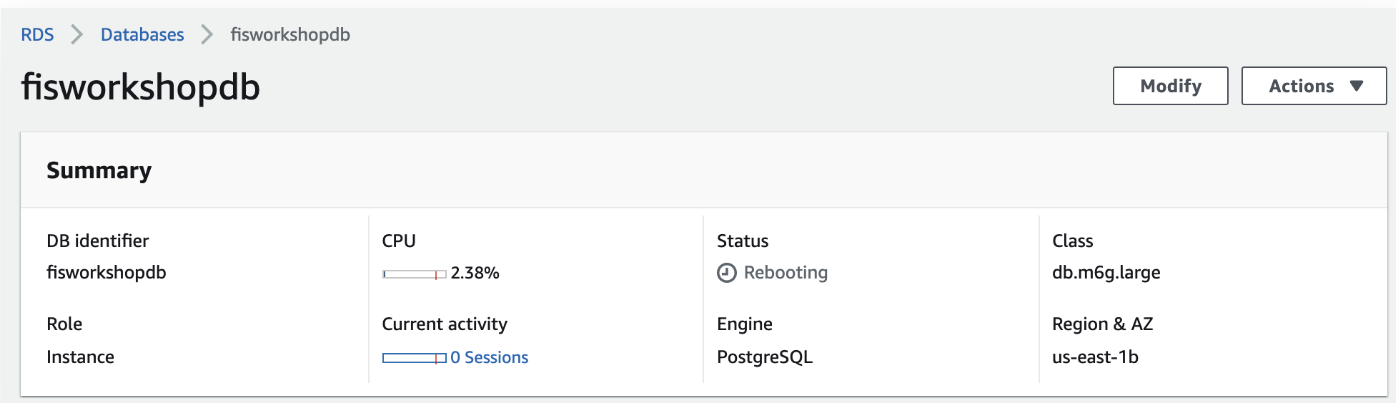


Start the experiment

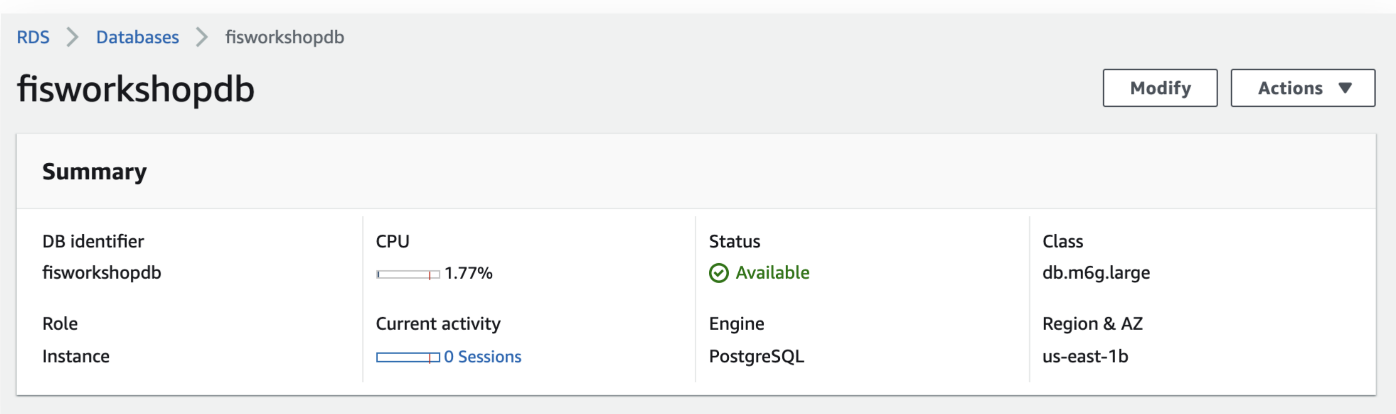
- Select the `FisWorkshopRds1` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopMysql1Run1`
- Confirm that you want to start an experiment
- Watch the output of your test script
- Check the state of your database in the **RDS console**

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.

Great! 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 (review **code in GitHub**). 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.

Note

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell:
`source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

General template setup

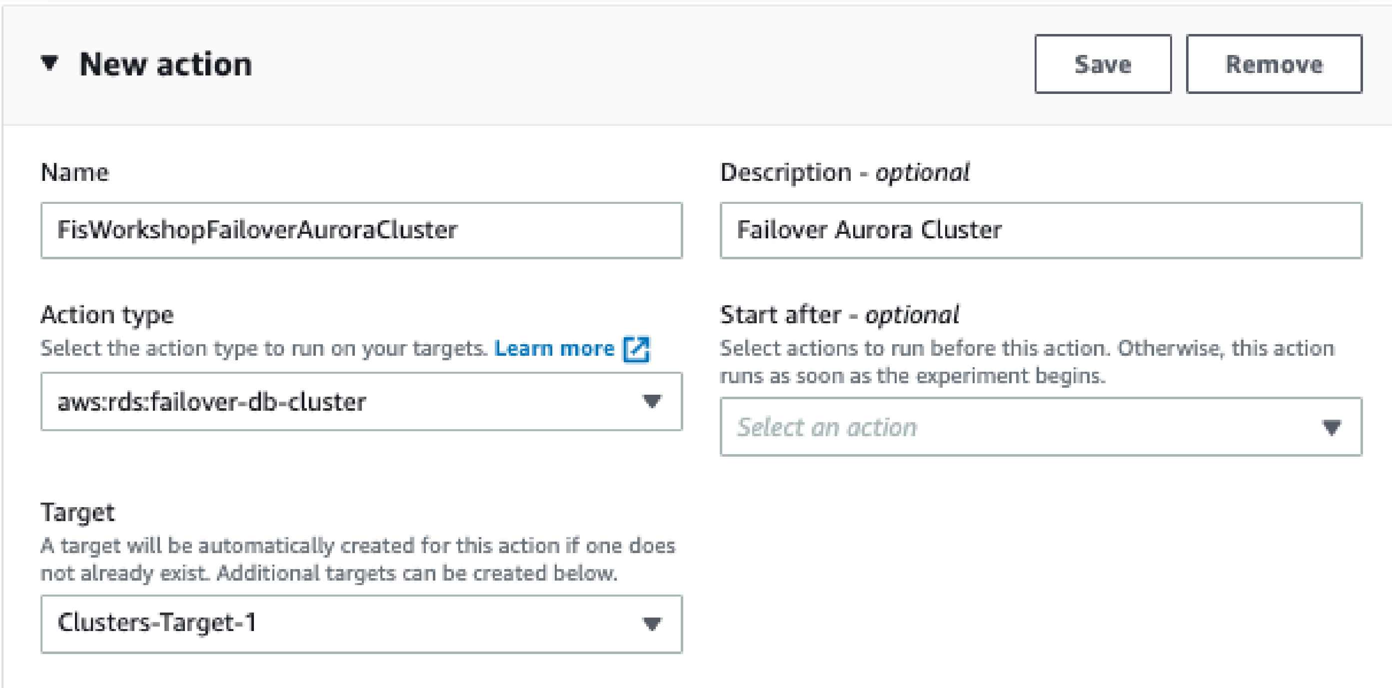
- Create a new experiment template
  - Add `Name` tag of `FisWorkshopAurora1`
  - Add `Description` of `FailoverAuroraCluster`
  - Select `FisWorkshopServiceRole` as execution role

Action definition

In the "Actions" section select the **"Add Action"** button.

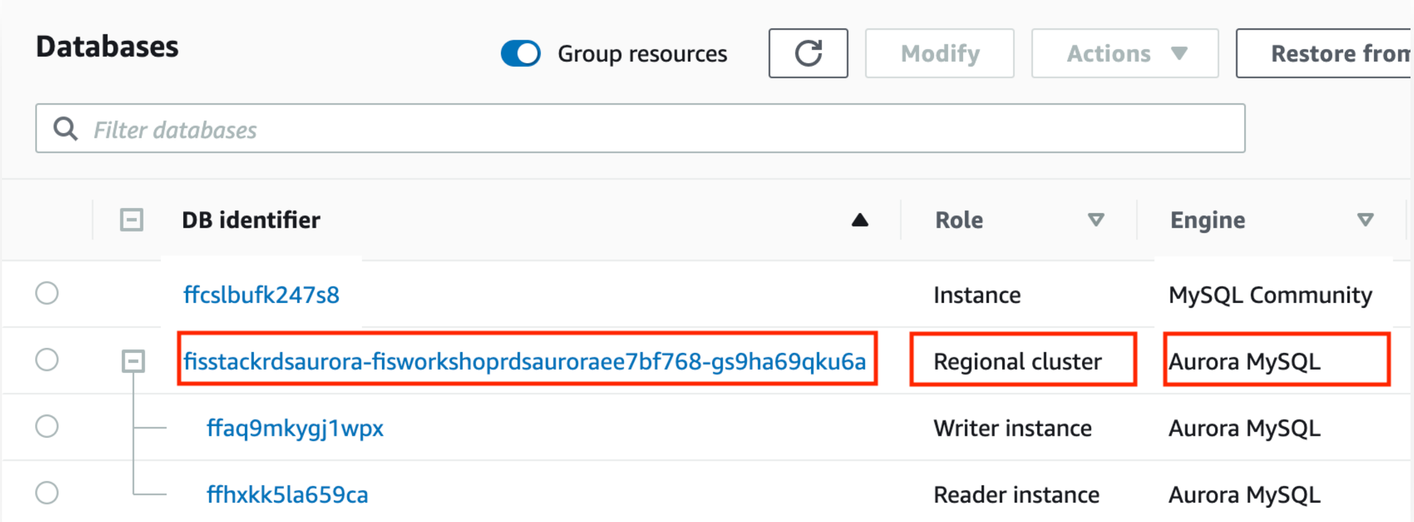
For "Name" enter `FisWorkshopFailoverAuroraCluster` and add a "Description" like `Failover Aurora Cluster`. For "Action type" select `aws:rds:failover-db-cluster`.

Leave the default "Target" `Clusters-Target-1` and select **"Save"**.



Target selection

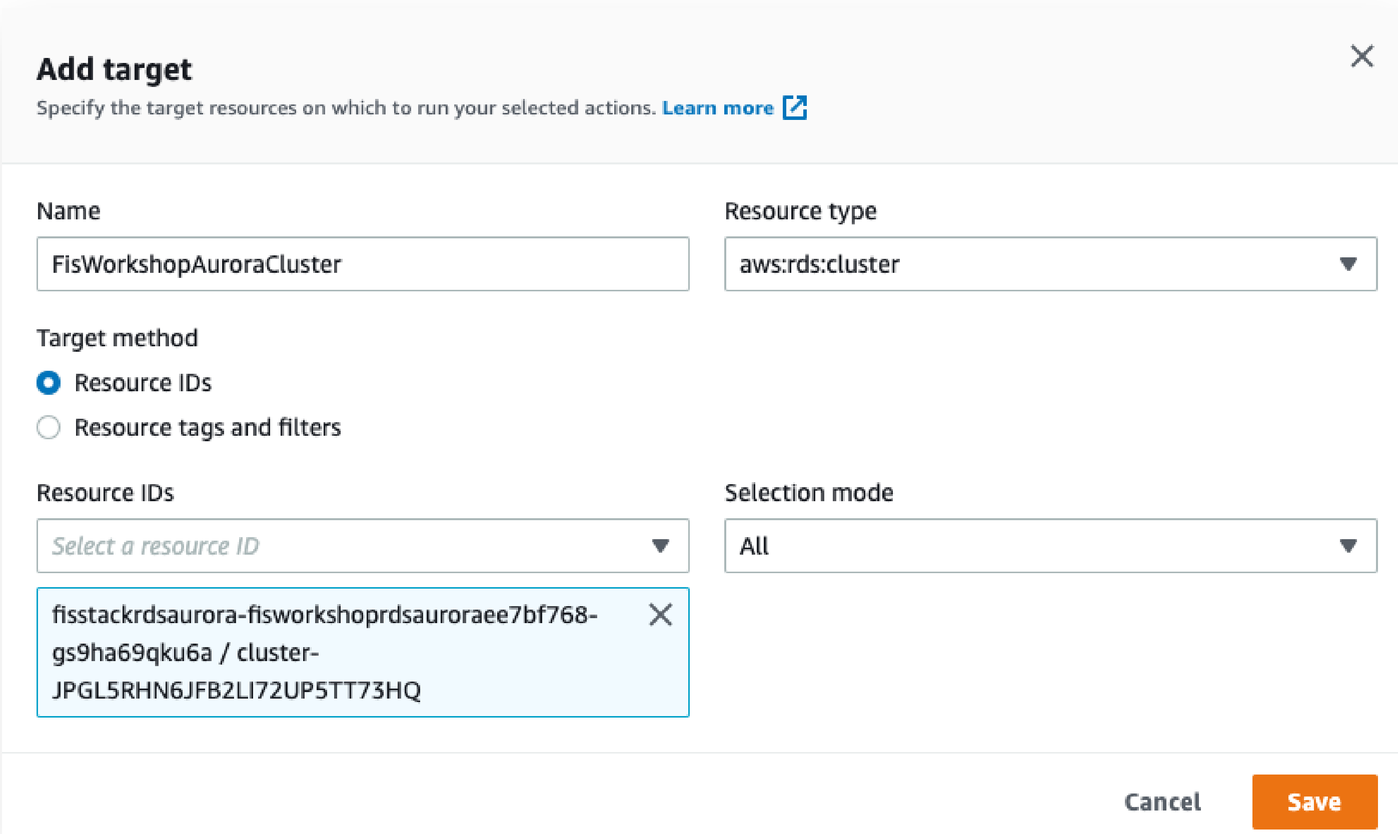
For this action we need to select our Amazon Aurora "Cluster". For this we will need to know the instance resource ID. To find this ID open a new browser window and navigate to the **RDS console**). Note the "DB identifier" for the target cluster, the one with "Engine" type "Aurora MySQL" and "Role" "Regional Cluster".



Return to the FIS experiment setup, scroll to the "Targets" section, select `Clusters-Target-1` and select **"Edit"**.

You may leave the default name `Clusters-Target-1` but for maintainability we rcommend using descriptive target names. Change "Name" to `FisWorkshopAuroraCluster` for name (this will automatically update the name in the action as well) and make sure "Resource type" is set to `aws:rds:cluster`.

For "Target method" we will select resources based on the ID. Select the "Resource IDs" checkbox. Under "Resource IDs" pick the target DB instance matching the "DB Identifier" you noted above, then select `All` from "Selection mode". Select **"Save"**.



Creating template without stop conditions

Select **“Create experiment template”** and confirm that you wish to create a template without stop conditions.

Validation procedure

The validation procedure is identical to what we did in the **RDS DB Instance Reboot** section. If you have not explored that section before, perform the steps as described there under the “Validation Procedure” heading and return here when you reach the “Run FIS experiment” heading.

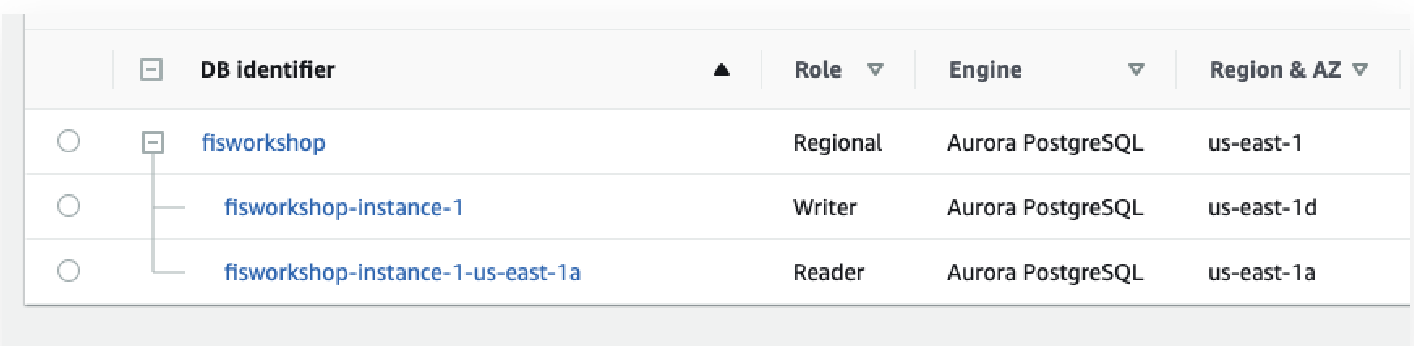
Run FIS experiment

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.

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

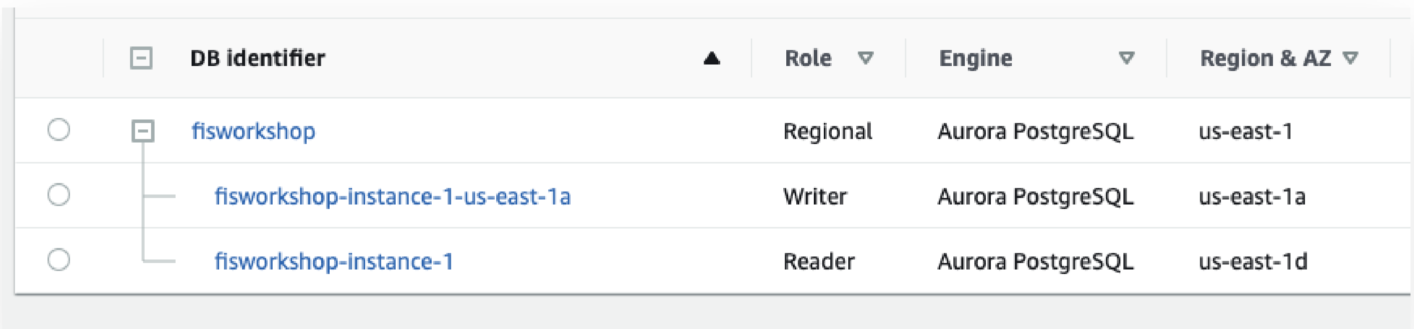
- Select the `FisWorkshopAurora1` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopAurora1Run1`
- Confirm that you want to start an experiment

- 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 (see **code in GitHub**).

Learning and improving

As this was essentially the same as the previous **RDS DB Instance Reboot** section there are no new learnings here.

However, you may want to experiment further with built-in Aurora fault injection queries for **MySQL** and **PostgreSQL**.

E.g. for the Aurora MySQL database provisioned in this workshop, 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:

› Expand to see scripted version

```
# hostname  / username / dbname from secret
export DB_HOST_NAME=[host from secret]
export DB_USER_NAME=[username from secret]
export DB_NAME=[dbname from secret]
```

Note

The code below will not work from CloudShell because the database is in a private VPC. Make sure to run this from an EC2 instances with access to the VPC"

```
mysql -h $DB_HOST_NAME -u $DB_USER_NAME -p $DB_NAME
```

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;
```

Note that in contrast to the FIS actions these actions will only affect the connection making the queries. All other connections to the database will be unaffected by this simulation.

ADVANCED EXPERIMENTS

In this section we will cover more advanced experiment configurations

- Access controls
- Access control tags
- Tags: update vs. create
- Template sharing

ACCESS CONTROLS

In the **Configuring Permissions** section we showed how to limit the access of a running FIS experiment. In this section we will demonstrate how to control user access to AWS Fault Injection Simulator (FIS).

Controlling user access to FIS

AWS Identity and Access Management (IAM) provides you fine-grained controls for to the use of FIS. As part of the provisioned infrastructure we have created three roles that can be assumed from within your account:

- `FisAccessControlAdmin` - This Role extends the `ReadOnlyAccess` AWS managed policy by adding all **FIS actions**. Note that this role does not have permission to perform impacting actions outside of FIS such as terminating EC2 instances. Those permissions have to be granted by the FIS execution role. Navigate to the **IAM Console** and expand the `AllowFisFullAccess` policy to see permissions granted.
- `FisAccessControlUser` - This Role extends the `ReadOnlyAccess` AWS managed policy by adding the ability to start/stop experiments and to tag Experiments (required to add the “Name” tag when starting an experiment). Note that this role does not have permission to perform impacting actions outside of FIS such as terminating EC2 instances. Those permissions have to be granted by the FIS execution role. Navigate to the **IAM Console** and expand the `AllowFisUsageAccess` policy to see permissions granted.
- `FisAccessControlNonUser` - This Role extends the `ReadOnlyAccess` AWS managed policy by explicitly denying all FIS actions. Navigate to the **IAM Console** and expand the `DenyFisAccess` policy to see permissions granted.

Exploring FIS with assumed roles

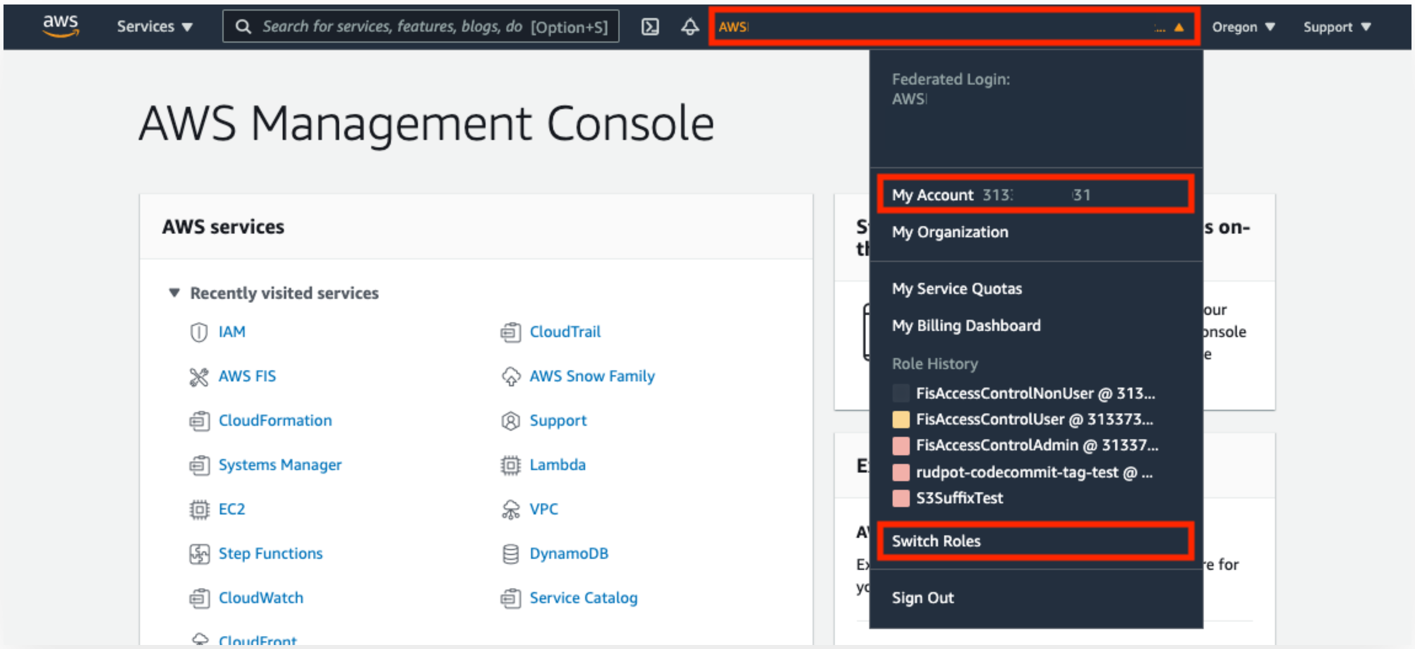
To see the effect of the above roles we will assume each role on the AWS console and explore its effect on the use of FIS.

Warning

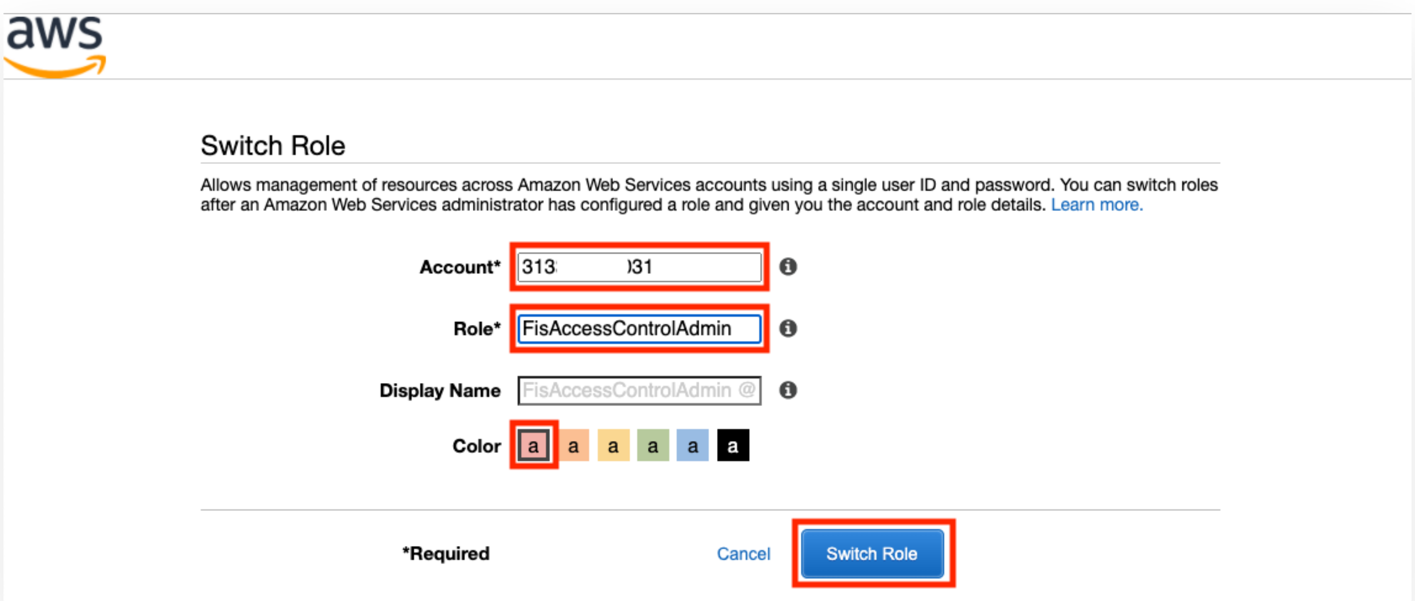
All tabs in a browser profile will share the same AWS identity. As such, assuming roles will expire all other active AWS console tabs and you will have to reload those tabs. Reloading the tabs will navigate to the same URL as before but with the new IAM Role .

Full access via FisAccessControlAdmin

To assume the `FisAccessControlAdmin` role navigate to the **AWS console** and click on the user identity at the top to get an info drop down. From the drop-down copy the account ID (12 digit number). Finally select “Switch Roles”.



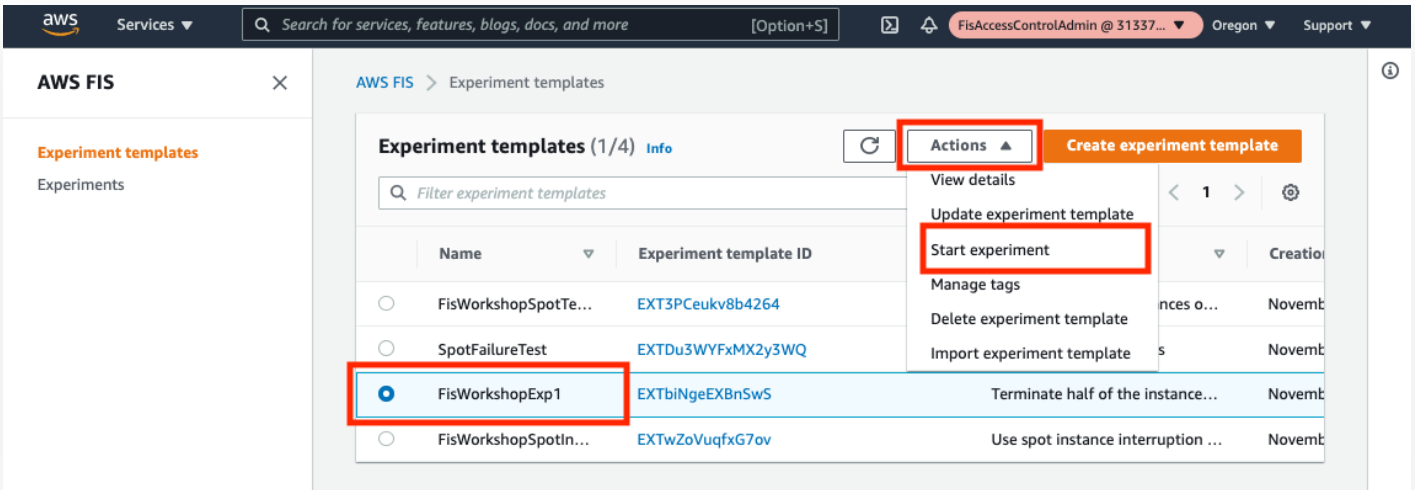
To define the role we would like to assume enter the account number you just copied and use the role name `FisAccessControlAdmin`. Pick a color to identify the role in the dropdown later. Since this is a privileged role we are using “red”. Finally select “Switch Role”.



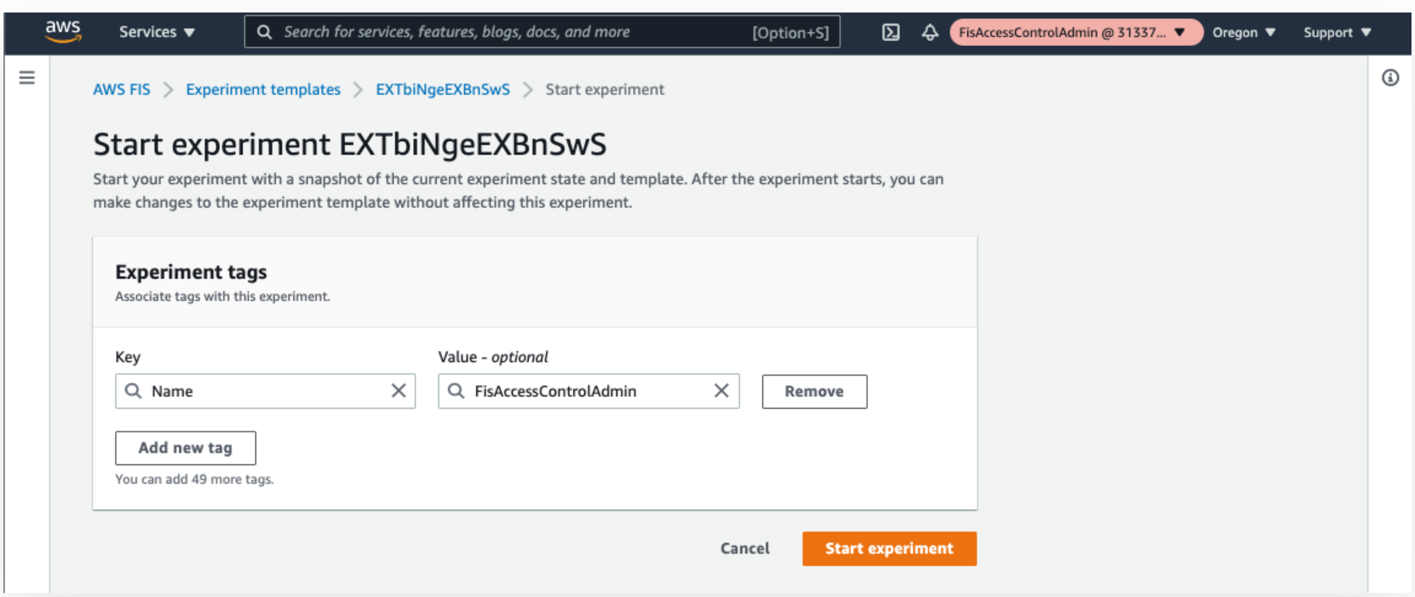
Note

We will assume that you have previously created the `FisWorkshopExp1` Experiment template from the **First Experiment** section and will use that template for the examples below but this should work with other templates as well.

With the assumed role (visible at the top) navigate to the **FIS console**, select the `FisWorkshopExp1` template, and from the "Actions" drop down select "Start Experiment".



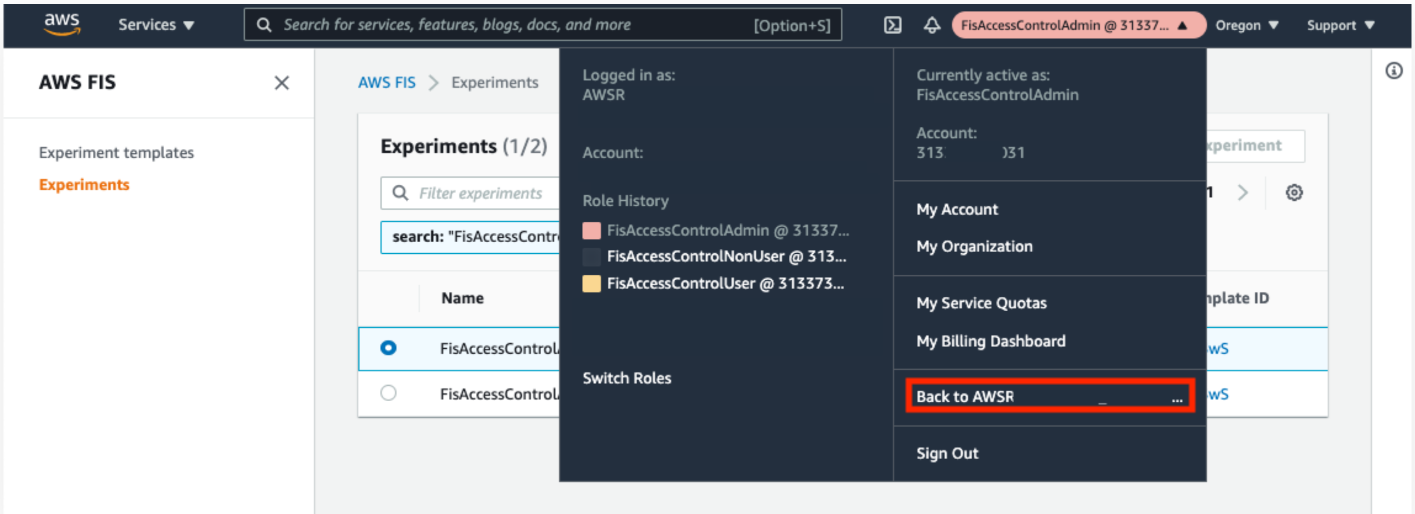
Add a new tag with "Key" `Name` and "Value" `FisAccessControlAdmin`, then select "Start Experiment" and confirm you wish to start the experiment.



Even though the `FisAccessControlAdmin` role itself does not have `ec2:TerminateInstances` privileges, the experiment will run and you will get a "Completed" or "Failed" result depending on how many instances were in the auto-scaling group, just as observed in the **First Experiment** section.

Just as in the First Experiment section you can also update the template as needed.

Before the next step, return to the normal workshop role by using the same dropdown you used to assume the role, then selecting “Back to …”.

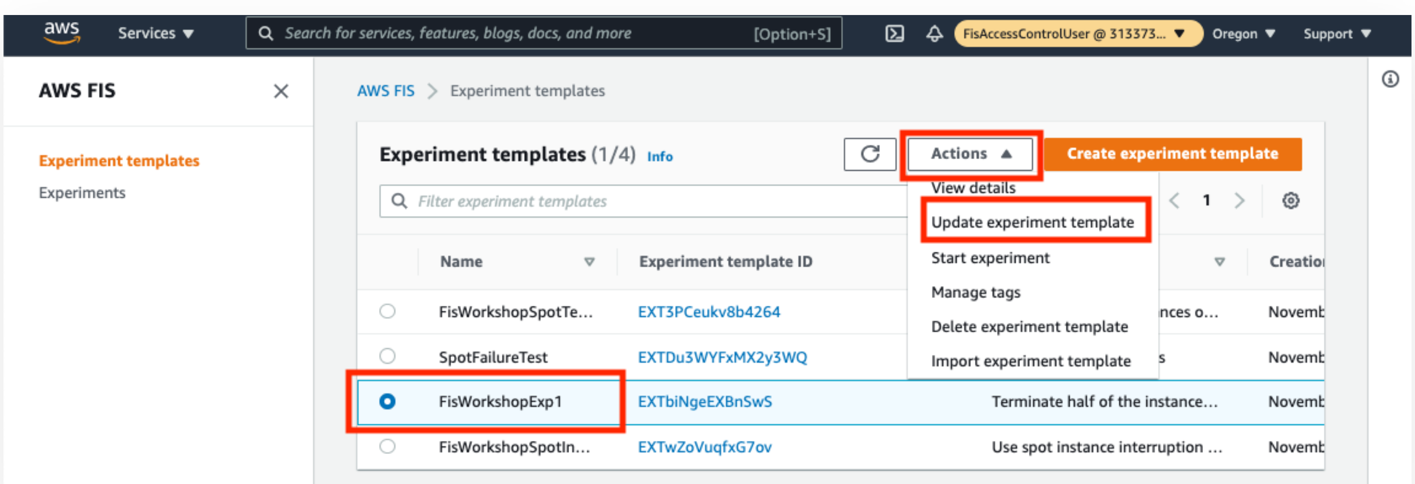


Execution access via FisAccessControlUser

Repeat the assume role steps above with the `FisAccessControlUser` role. You may pick a different color, e.g. orange, to signify a less privileged user.

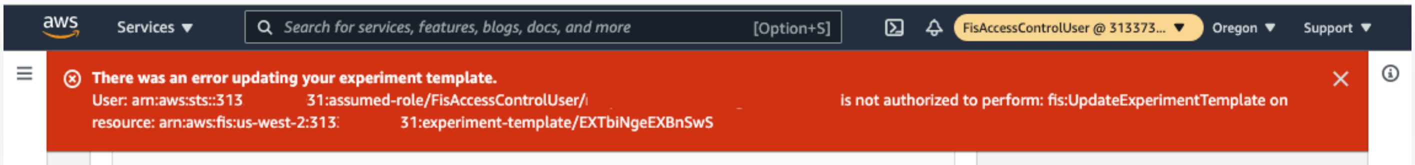
With this role you can list experiments and experiment templates and run an experiment. However, this role is not allowed to edit an experiment template.

To test this, navigate to the **FIS Console**, select “Experiment Templates”, select the `FisWorkshopExp1` template, and under the “Actions” drop down select “Update experiment template”.



Edit the `FisWorkshopAsg-50Percent1` "Target", set "Selection mode" to `COUNT` and "Mumber of resources" to `1`, and select "Save" on the edit modal.

Select "Update experiment template" and confirm the intent to update. This will result in a failure banner informing you that the assumed role lacks the required edit/update privileges.



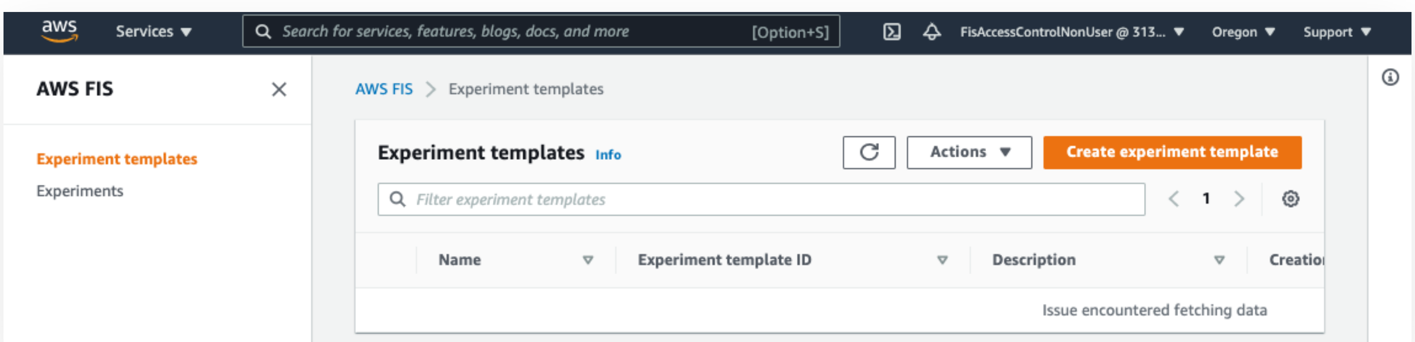
Before the next step, return to the normal workshop role by using the same dropdown you used to assume the role, then selecting "Back to …".

No access via FisAccessControlNonUser

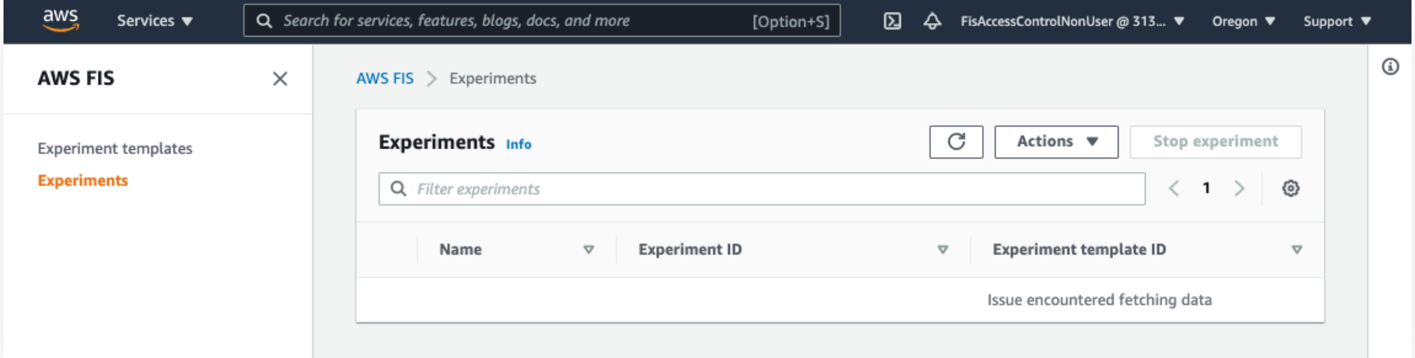
Repeat the assume role steps above with the `FisAccessControlNonUser` role. You may pick a different color, e.g. black, to signify an unprivileged user.

Even though this role is based on the AWS managed `ReadOnlyAccess` policy, access to FIS has been explicitly denied.

Navigate to the **FIS Console** and select "Experiment Templates". You will notice that no templates are listed because the user is not sufficiently privileged.



Similarly if you select "Experiments" you will notice that no experiments are listed because the user is not sufficiently privileged.


aws
Services
Search for services, features, blogs, docs, and more
[Option+S]
FisAccessControlNonUser @ 313...
Oregon
Support
AWS FIS
Experiment templates
Experiments
AWS FIS > Experiments
Experiments Info
Actions
Stop experiment
Filter experiments
1
Name
Experiment ID
Experiment template ID
Issue encountered fetching data

ACCESS CONTROL TAGS

In the previous section we saw how to use IAM roles and policies to control access to experiments and templates. In addition to fixed IAM policies it is also possible to use **resource tags** to add more granular access control.

Configuring CloudShell

Note

To simplify the assume role functionality for the workshop, this section will use AWS CloudShell. If you want to use the same approach from other environments, review this link for other ways to configure your credentials provider.

Warning

To protect your exisiting AWS CLI config file, we will use a custom AWS CLI config file. We will reference this file by setting the `AWS_CONFIG_FILE` environment variable. If you need to return to using your default config file either unset the environment variable or open a new CloudShell tab.

Navigate to **CloudShell** and wait for your CloudShell instance to start up.

Once ready, set up a directory in which to work and create a custom AWS CLI config file. This file defines profiles for the three roles we used in the previous secton plus an additional `FisAccessControlSecurityAdmin` role:

```
# Create same path as used by GitHub repository
mkdir -p ~/environment/aws-fault-injection-simulator-
workshop/resources/templates/access-controls/
cd ~/environment/aws-fault-injection-simulator-
workshop/resources/templates/access-controls/


ACCOUNT_ID=$(aws sts get-caller-identity --output text --query 'Account')
cat > aws_test_config <<EOT
[profile FisAccessControlSecurityAdmin]
```

```
    role_arn = arn:aws:iam::${ACCOUNT_ID}:role/FisAccessControlSecurityAdmin
    credential_source = EcsContainer

[profile FisAccessControlAdmin]
    role_arn = arn:aws:iam::${ACCOUNT_ID}:role/FisAccessControlAdmin
    credential_source = EcsContainer

[profile FisAccessControlUser]
    role_arn = arn:aws:iam::${ACCOUNT_ID}:role/FisAccessControlUser
    credential_source = EcsContainer

[profile FisAccessControlNonUser]
    role_arn = arn:aws:iam::${ACCOUNT_ID}:role/FisAccessControlNonUser
    credential_source = EcsContainer

EOT

export AWS_CONFIG_FILE=${PWD}/aws_test_config
export AWS_PAGER=""
```

Let’s test the setup:

```
# Validate that we can assume the role
aws --profile FisAccessControlAdmin sts get-caller-identity

# List experiment templates
aws --profile FisAccessControlAdmin fis list-experiment-templates
```

Restricting update permissions with tags

For this demonstration we will export the the first experiment template into two separate experiment templates files. In the first template file we will change the `Name` tag to `TagAccessTest1Dev` and add a new `Environment` tag with value `dev`. For the second template file we will change the `Name` tag to `TagAccessTest1Prod` and add a new `Environment` tag with value `prod`. Later in this section we will show how to use these tags for access control.

```
# Get experiment template ID
EXPERIMENT_TEMPLATE_ID=$( aws fis list-experiment-templates --query
"experimentTemplates[?tags.Name=='FisWorkshopExp1'].id" --output text )

# Save template with a "dev" environment tag
aws fis get-experiment-template --id $EXPERIMENT_TEMPLATE_ID \
| jq '.experimentTemplate' \
| jq 'del( .id) | del(.creationTime) | del(.lastUpdateTime)' \
```

```
| jq '.tags.Name="TagAccessTest1Dev"' \
| jq '.tags.Environment="dev"' \
> tag-test-template-dev.json

# Save template with a "prod" environment tag
aws fis get-experiment-template --id $EXPERIMENT_TEMPLATE_ID \
| jq '.experimentTemplate' \
| jq 'del( .id) | del(.creationTime) | del(.lastUpdateTime)' \
| jq '.tags.Name="TagAccessTest1Prod"' \
| jq '.tags.Environment="prod"' \
> tag-test-template-prod.json
```

Privileged user experience without prod constraint

Our newly created security admin user has no restrictions on their ability to use FIS. In particular this role is able to create experiment templates and experiments with any attached tags.

As such it can create both the dev and prod experiment templates

```
# Privileged admin user can create dev templates
DEV_TEMPLATE_1=$(
  aws fis create-experiment-template \
    --profile FisAccessControlSecurityAdmin \
    --cli-input-json file://tag-test-template-dev.json \
    --query 'experimentTemplate.id' \
    --output text
)
echo $DEV_TEMPLATE_1

# Privileged admin user can create prod templates
PROD_TEMPLATE_1=$(
  aws fis create-experiment-template \
    --profile FisAccessControlSecurityAdmin \
    --cli-input-json file://tag-test-template-prod.json \
    --query 'experimentTemplate.id' \
    --output text
)
echo $PROD_TEMPLATE_1
```

It can start experiments from both dev and prod templates and can tag the resulting experiments with dev and prod tags

```
# Privileged admin can user start experiments from dev templates
DEV_EXPERIMENT_1=$(
  aws fis start-experiment \
```

```
    --profile FisAccessControlSecurityAdmin \
    --experiment-template-id ${DEV_TEMPLATE_1} \
    --tags \
        Name=FisWorkshop-TagLimit-Dev-
FisAccessControlSecurityAdmin,Environment=dev \
    --query 'experiment.id' \
    --output text
)
echo $DEV_EXPERIMENT_1

# Privileged admin can user start and tag experiments from prod templates
PROD_EXPERIMENT_1=$(
  aws fis start-experiment \
    --profile FisAccessControlSecurityAdmin \
    --experiment-template-id ${PROD_TEMPLATE_1} \
    --tags \
        Name=FisWorkshop-TagLimit-Prod-
FisAccessControlSecurityAdmin,Environment=prod \
    --query 'experiment.id' \
    --output text
)
echo $PROD_EXPERIMENT_1
```

It can retrieve the content of both dev and prod tagged experiment templates

```
# Privileged admin can retrieve dev experiment templates
aws fis get-experiment-template \
  --profile FisAccessControlSecurityAdmin \
  --id ${DEV_TEMPLATE_1}

# Privileged admin can retrieve prod experiment templates
aws fis get-experiment-template \
  --profile FisAccessControlSecurityAdmin \
  --id ${PROD_TEMPLATE_1}
```

It can retrieve the content of both dev and prod tagged experiments

```
# Privileged admin can retrieve dev experiments
aws fis get-experiment \
  --profile FisAccessControlSecurityAdmin \
  --id ${DEV_EXPERIMENT_1}

# Privileged admin can retrieve prod experiments
aws fis get-experiment \
  --profile FisAccessControlSecurityAdmin \
  --id ${PROD_EXPERIMENT_1}
```

Privileged user experience with prod constraint

Now lets look at a user that can perform any FIS actions *unless* the resource created or used has an attached `Environment` tag with value `prod`.

Repeating the previous steps with the less privileged role / profile we can see that this user can create dev templates but cannot create templates with an attached `Environment` tag with value `prod`

```
# Admin user can create dev templates
DEV_TEMPLATE_2=$(
  aws fis create-experiment-template \
    --profile FisAccessControlAdmin \
    --cli-input-json file://tag-test-template-dev.json \
    --query 'experimentTemplate.id' \
    --output text
)
echo $DEV_TEMPLATE_2

# Admin user cannot create prod templates
PROD_TEMPLATE_2=$(
  aws fis create-experiment-template \
    --profile FisAccessControlAdmin \
    --cli-input-json file://tag-test-template-prod.json \
    --query 'experimentTemplate.id' \
    --output text
)
echo $PROD_TEMPLATE_2
```

The constrained admin role can start experiments from templates tagged with an `Environment` tag with value `dev` but not with value `prod`. The constrained role also cannot start experiments from `dev` templates and tag the result as `prod`.

```
# Admin can user start experiments from dev templates
DEV_EXPERIMENT_2=$(
  aws fis start-experiment \
    --profile FisAccessControlAdmin \
    --experiment-template-id ${DEV_TEMPLATE_1} \
    --tags \
        Name=FisWorkshop-TagLimit-Dev-FisAccessControlAdmin,Environment=dev \
    --query 'experiment.id' \
    --output text
)
echo $DEV_EXPERIMENT_2

# Admin cannot user start experiments from prod templates
DEV_EXPERIMENT_3=$(
  aws fis start-experiment \
```

```
    --profile FisAccessControlAdmin \
    --experiment-template-id ${PROD_TEMPLATE_1} \
    --tags \
      Name=FisWorkshop-TagLimit-Dev-FisAccessControlAdmin,Environment=dev \
    --query 'experiment.id' \
    --output text
)
echo $DEV_EXPERIMENT_3

# Admin cannot user tag experiments with prod tag
PROD_EXPERIMENT_2=$(
  aws fis start-experiment \
    --profile FisAccessControlAdmin \
    --experiment-template-id ${DEV_TEMPLATE_1} \
    --tags \
      Name=FisWorkshop-TagLimit-Prod-FisAccessControlAdmin,Environment=prod \
    --query 'experiment.id' \
    --output text
)
echo $PROD_EXPERIMENT_2
```

The constrained role can retrieve the content of experiment templates with an attached `Environment` tag with value `dev` but not `prod`

```
# Admin can retrieve dev experiment templates
aws fis get-experiment-template \
  --profile FisAccessControlAdmin \
  --id ${DEV_TEMPLATE_1}

# Admin can retrieve prod experiment templates
aws fis get-experiment-template \
  --profile FisAccessControlAdmin \
  --id ${PROD_TEMPLATE_1}
```

The constrained role can retrieve the content of experiments with an attached `Environment` tag with value `dev` but not `prod`

```
# Admin can retrieve dev experiments
aws fis get-experiment \
  --profile FisAccessControlAdmin \
  --id ${DEV_EXPERIMENT_1}

# Admin can retrieve prod experiments
aws fis get-experiment \
  --profile FisAccessControlAdmin \
  --id ${PROD_EXPERIMENT_1}
```

Note that list operations are not constrained by tags so this user can still see the list of all prod experiments that have been performed.

Unprivileged user experience

Repeating the above commands with the `FisAccessControlUser` role will demonstrate the additional constraint of not being able to create experiment templates. Like the constrained admin user, this user can see the list of all prod experiments that have been performed.

Repeating the above commands with the `FisAccessControlNonUser` will show no access to FIS resources. Because this role's access to FIS has been constrained by an explicit deny it also cannot list experiment templates or experiments even though the AWS managed `ReadOnlyAccess` policy would have allowed the list actions.

TAGS: UPDATE VS. CREATE

Note

This section is aimed at large, distributed, and *extremely* security conscious teams. If that's not a high concern to you, feel free to skip this section.

As we saw in the previous section, tags can be used as part of access control policies. To enable update workflows, FIS provides separate API calls for tagging resources (**CLI** / **API**) and for updating template content (**CLI** / **API**).

Because tags and experiment templates are managed by independent services it is not possible to atomically update tags *and* experiment template content *at the same time*.

If you have use cases where you need prevent template execution while performing updates on both tags and template content, we recommend that you update the templates and tags with the following steps:

1. Update tags to prevent all execution of the template. The exact approach will depend on the relevant IAM policies in your account.
2. Update template content.
3. Update tags to desired target state.

TEMPLATE SHARING

AWS Fault Injection Simulator is a regional service that allows targeting resources by availability zones or even affect all resources in a region to simulate whole region outages.

However, there are two scenarios where you might want to manage experiment templates across multiple regions and multiple accounts:

- **Users from one account accessing FIS in another account**, e.g. because you are using a **multi-account strategy**
- **Template replication**, e.g. because you are running identical stacks in multiple regions and want to run identical experiments in all regions

Cross-account access

Warning

This workshop only provisions *one* account. If you wish to test this you will need *another* account. If you use one of your corporate accounts to test this as part of the workshop please make sure that (1) your corporate account is the one *assuming* the role ("client") and (2) you remove any role changes you've made in your corporate ("client") account to access the workshop ("server") account.

We will assume that you have a firm grasp of the assume role procedure from the **Access controls** section. If not we suggest you revisit that section and consult the **AWS documentation**.

Enabling access from the workshop "server" account

Follow these steps

- Note the account ID for your workshop account - we will refer to this as `111122223333` or "server" for the remainder of this section.
- Note the account ID for your other account - we will refer to this as `444455556666` or "client" for the remainder of this section.

- In your "server" account navigate to the **IAM console** and locate the `FisAccessControlSecurityAdmin` role.
- Select the role and select the "Trust relationships" tab. This tab should currently show a single entry under "Trusted entities", the "server" account `111122223333`
- Select "Edit trust relationship"
- Update the JSON to read (replace the account IDs appropriately):

```
{
  "Version": "2012-10-17",
  "Statement": [
      {
      "Effect": "Allow",
      "Principal": {
          "AWS": [
          "arn:aws:iam::111122223333:root",
          "arn:aws:iam::444455556666:root"
          ]
      },
      "Action": "sts:AssumeRole"
      }
  ]
}
```

- Select "Update Trust Policy"

Accessing from the client account

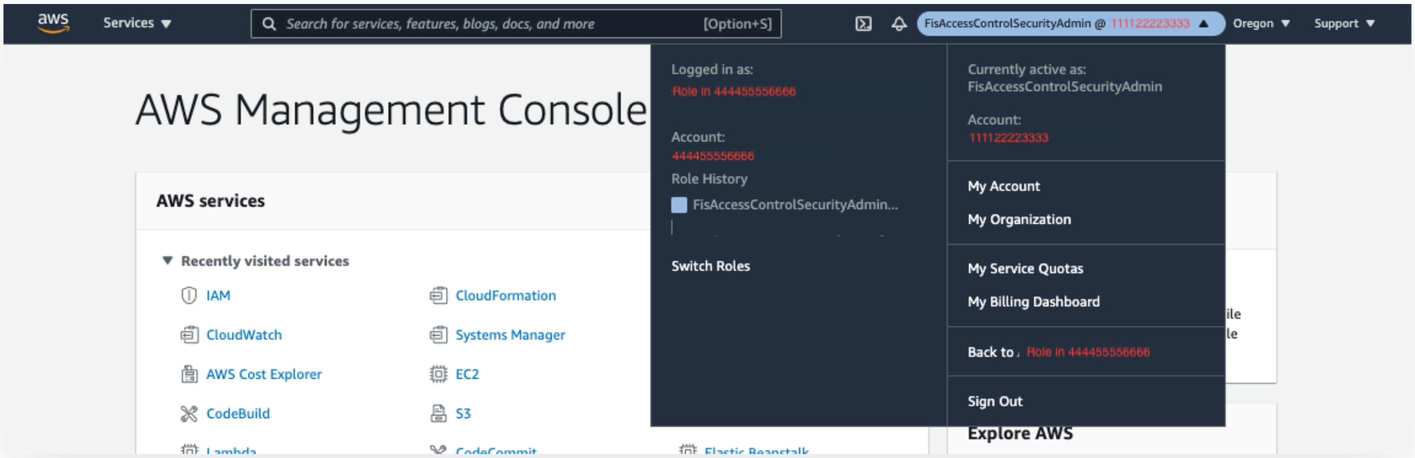
Warning

All tabs in a browser profile will share the same AWS identity. As such, logging into another AWS account or assuming roles will expire all other active AWS console tabs and you will have to reload those tabs. For this section we suggest that you use an browser profiles (**Chrome**, **Firefox**) or use an incognito window to avoid confusion about which account you are logged into.

In a new browser / profile / incognito window log into your "client" AWS account, which should be distinct from your workshop account.

In the "client" account window follow the same procedure outlined in the **Access controls** section. For "Account" enter the workshop / "server" account number `111122223333`. For "Role" enter the name (not the ARN) of the role you want to assume, in this case the role that we modified above to allow access: e.g. `FisAccessControlSecurityAdmin`. Pick a color, we suggest blue to differentiate it from the other choices in this workshop, and select "Switch Role".

At this point you should see a blue indicator at the top of your console indicating that you are no longer "client" account `444455556666` but are instead logged into the "server" account `111122223333` with role `FisAccessControlSecurityAdmin`. You should also be able to see your role history in the left part of the drop down indicating your origin "client" account and role.



As this approach is based on IAM you can use instance or service roles in the "client" account or you can configure the AWS CLI to use **profiles that assume a role** or to **use AWS SSO**.

Template replication

Currently templates are static objects and in many cases need to reference targeted resources with account and region specific information. We have previously covered how to **create templates** via CLI or CloudFormation and we have discussed access controls earlier in this section so we will limit the discussion to a few points for you to consider:

- **ARNs** - where ARNs are required, e.g. execution roles, ECS or EKS clusters, and SSM documents, these must contain the account number. If you use CloudFormation you can inject this information using **Pseudo Parameters**. If you prefer using a CLI/API approach you can use a JSON templating engine such as **mustache** or **handlebars**.
- **Static resources** - while FIS allows targeting resources based on filters, sometimes it is necessary to specify a particular resource via ID or ARN. If you use CloudFormation and can define the FIS experiment template in the same CloudFormation template as the resource then you can directly reference the resource. If you opt for a CLI/API approach, most JSON templating engines allow injecting variables so you could write a small script to do the lookup in the target account and parametrize your template.
- **AZ naming** - if you need to replicate templates across accounts but wish to perform an experiment that targets the same AZ across multiple accounts you will need to determine the correct **AZ ID** as part of the templating.

Managing infrastructure across multiple accounts

In addition to sharing templates across accounts you will need to manage IAM roles, SSM documents, and other resources across accounts to ensure consistency in naming, access controls, etc. While there are many ways to achieve this we recommend reviewing **AWS CloudFormation StackSets**,

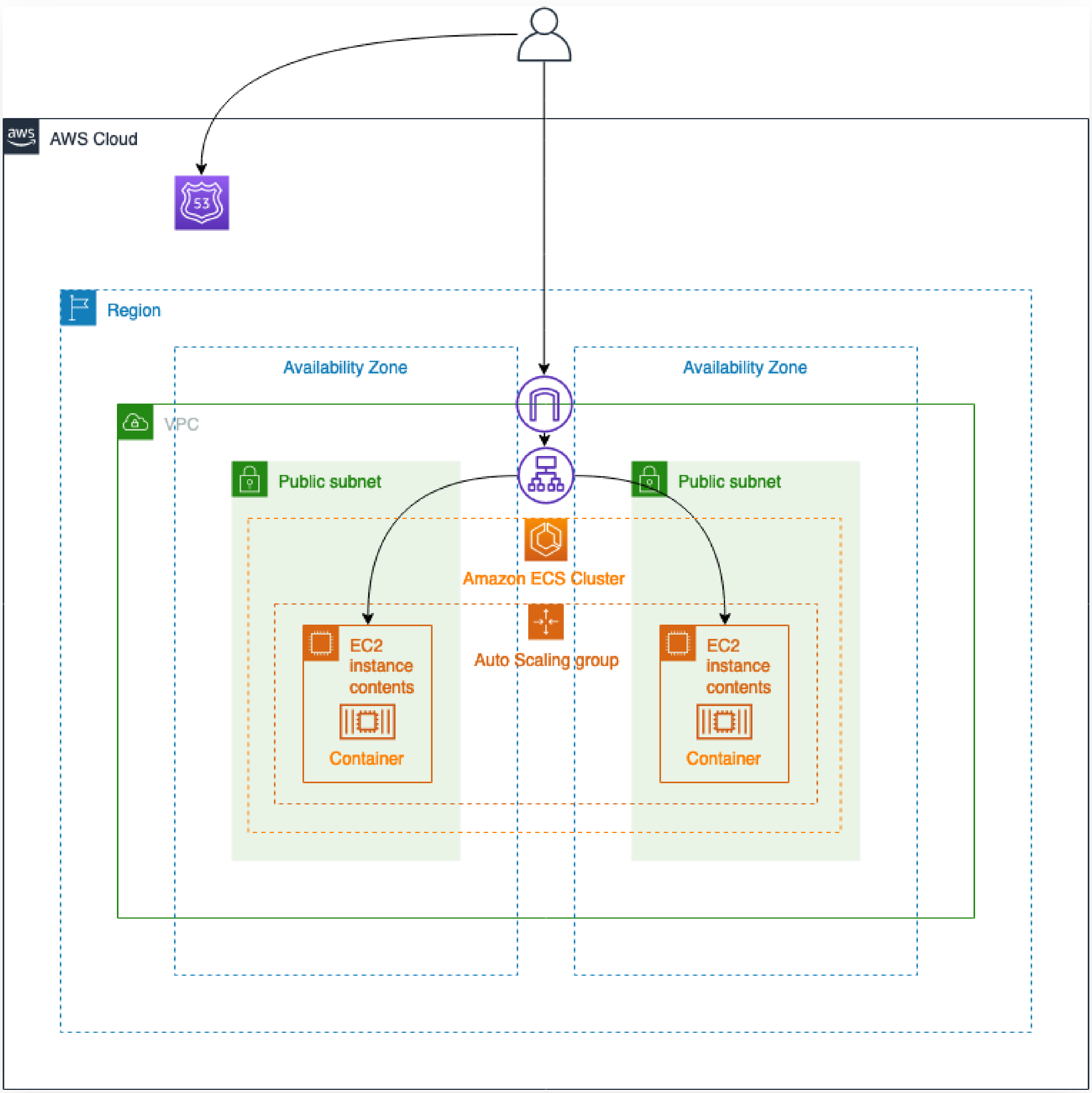
CONTAINERS

AWS provides managed services to help 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** (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

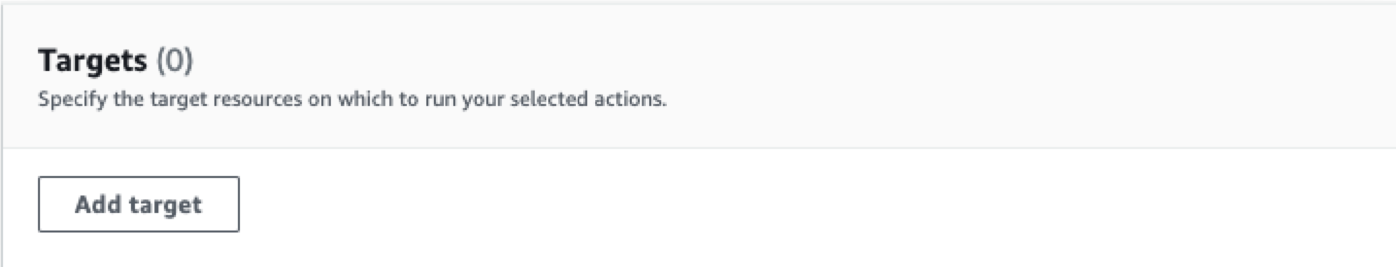
This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

Create a new experiment template:

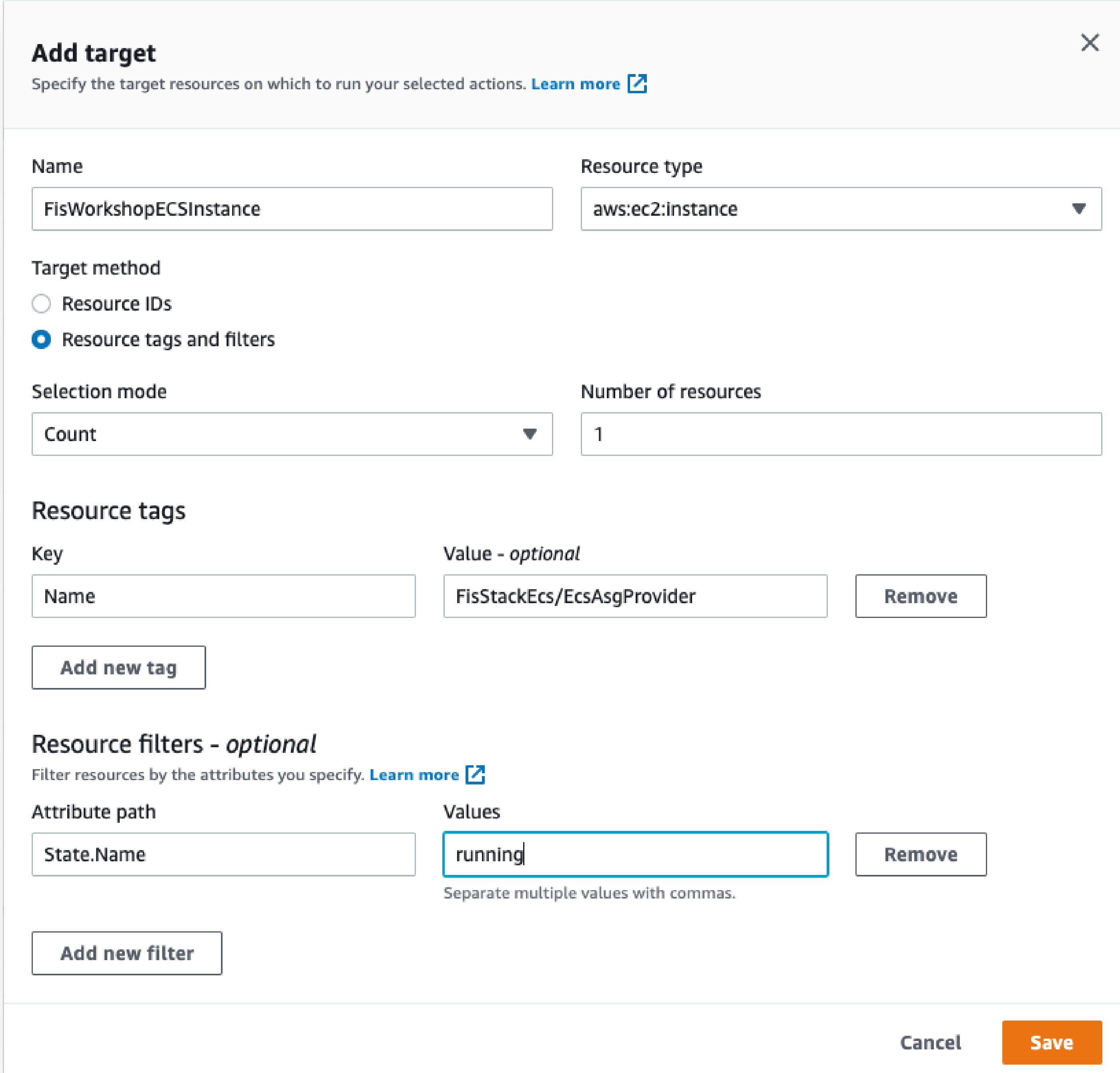
- add `Name` tag of `FisWorkshopECS`
- add `Description` of `Terminate ECS Cluster Instance`
- select `FisWorkshopServiceRole` as execution role

Target selection

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

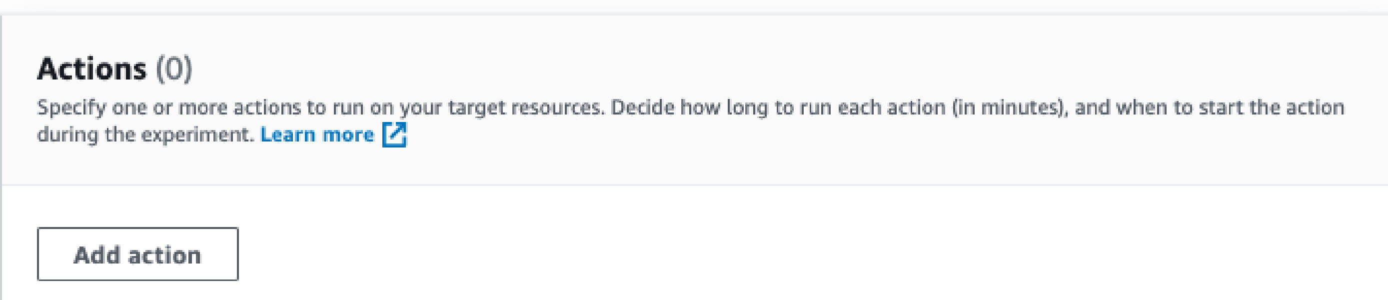


On the “Add target” popup enter `FisWorkshopECSInstance` for name and select `aws:ec2:instance` for resource type. For “Target method” we will dynamically select resources based on an associated tag. Select the `Resource tags and filters` checkbox. Pick `Count` from “Selection mode” and enter `1`. Under “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

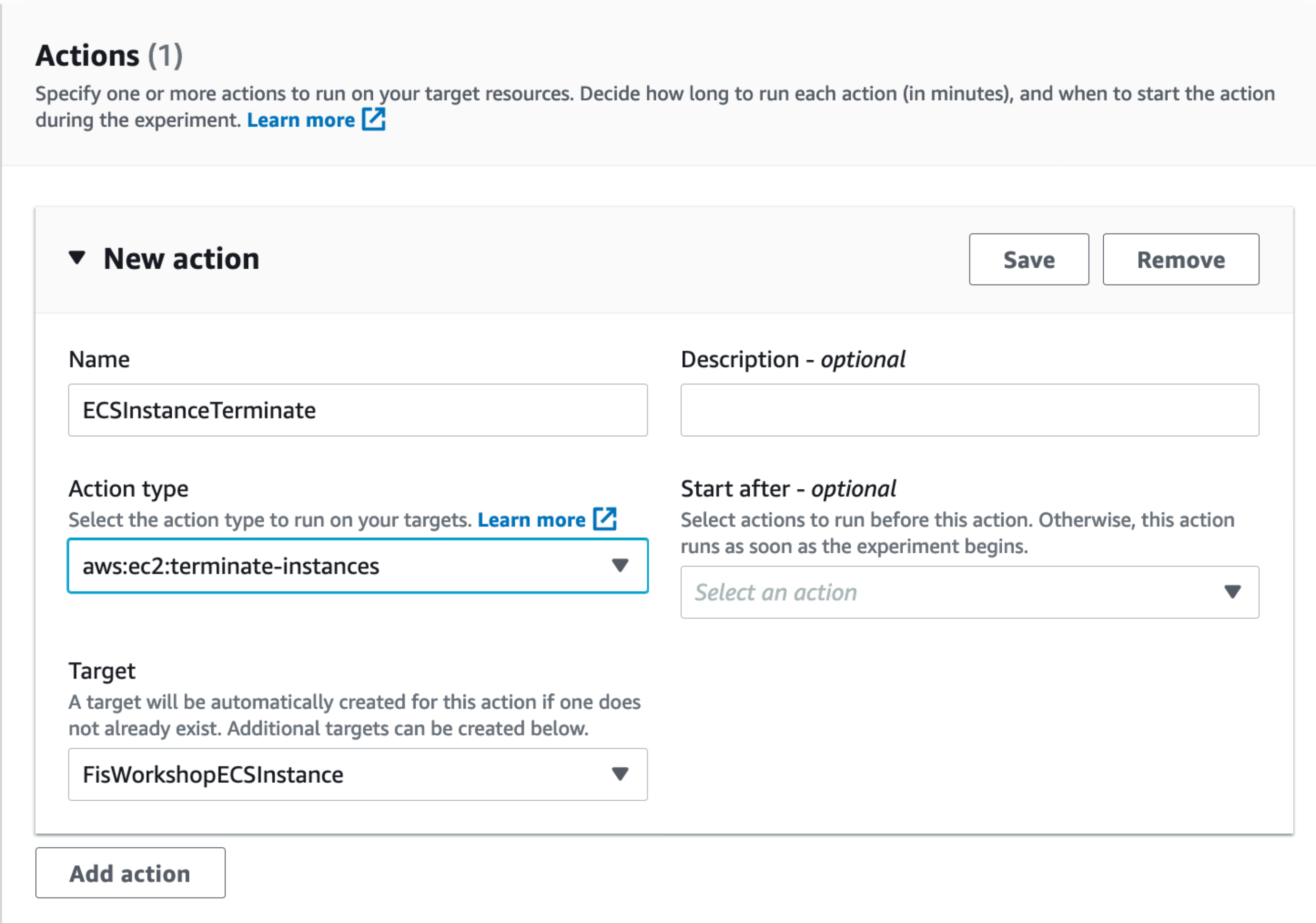
With targets defined we define the action to take. Scroll to the “Actions” section" and select “Add Action”



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

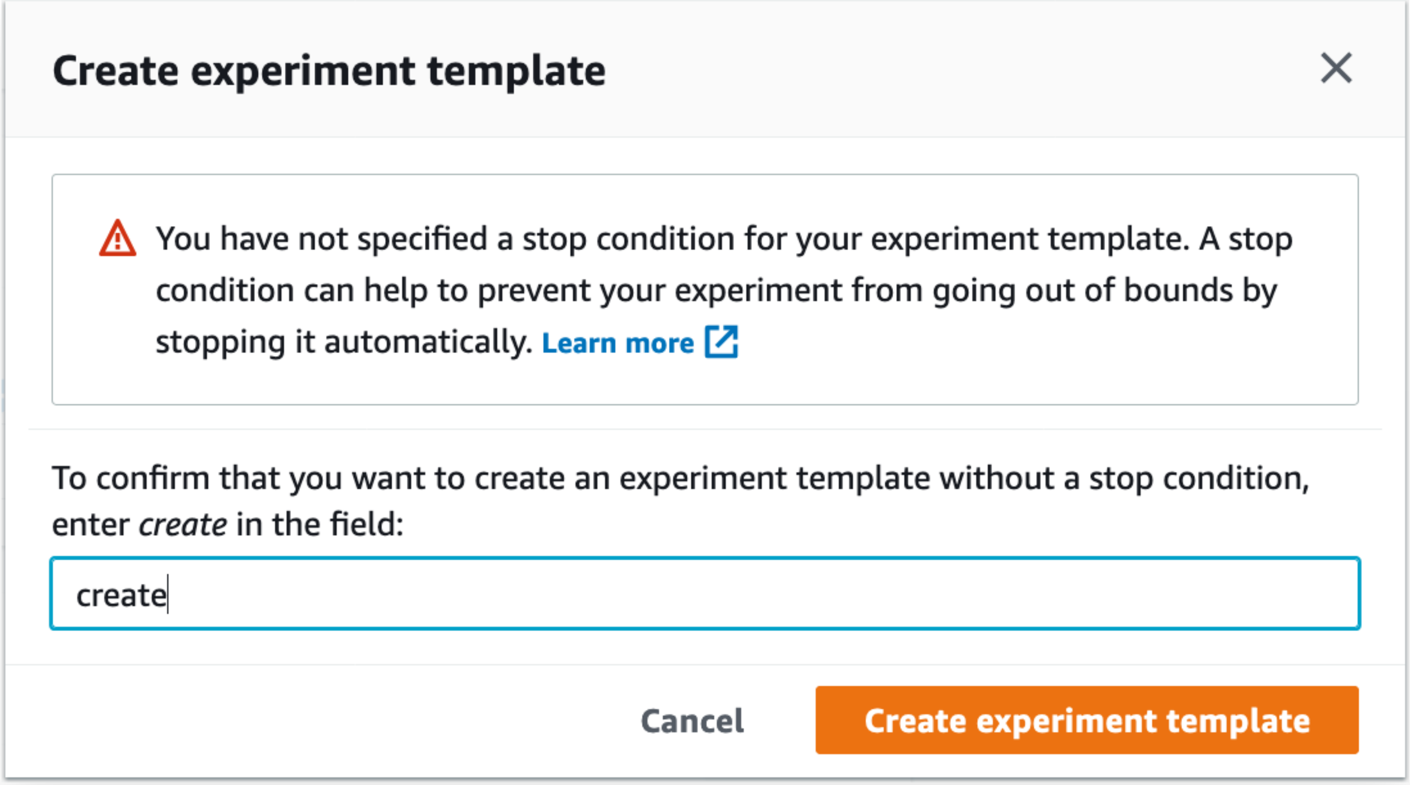
We will leave the “Start after” section blank since the instances we are terminating are part of an auto scaling group and we can let the auto scaling group create new instances to replace the terminated ones.

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



Creating template without stop conditions

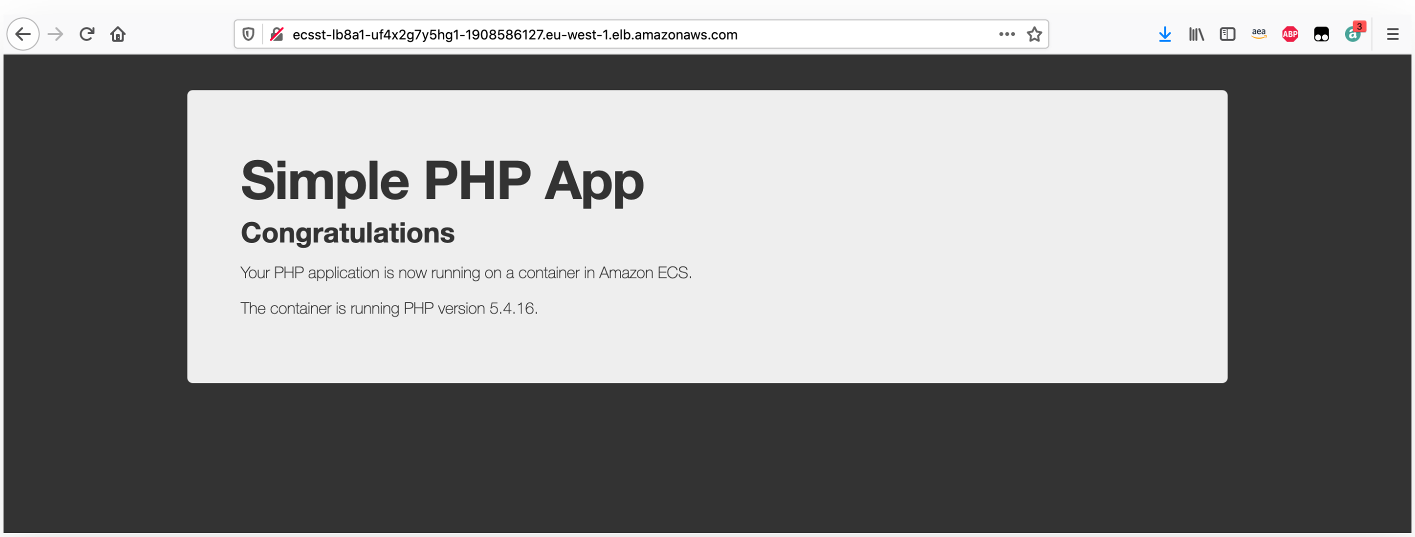
Confirm that you wish to create the template without stop condition.



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:

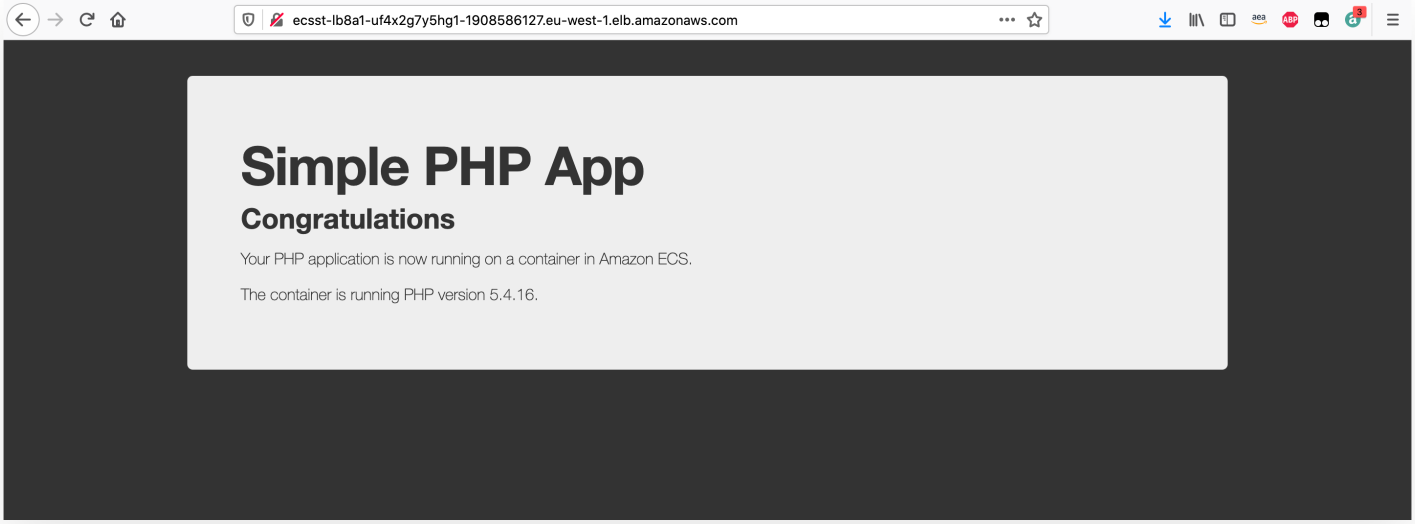
```
Code 200 Duration 0.140314
Code 200 Duration 0.086206
Code 200 Duration 0.085946
Code 200 Duration 0.084102
Code 200 Duration 0.085972
```

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:

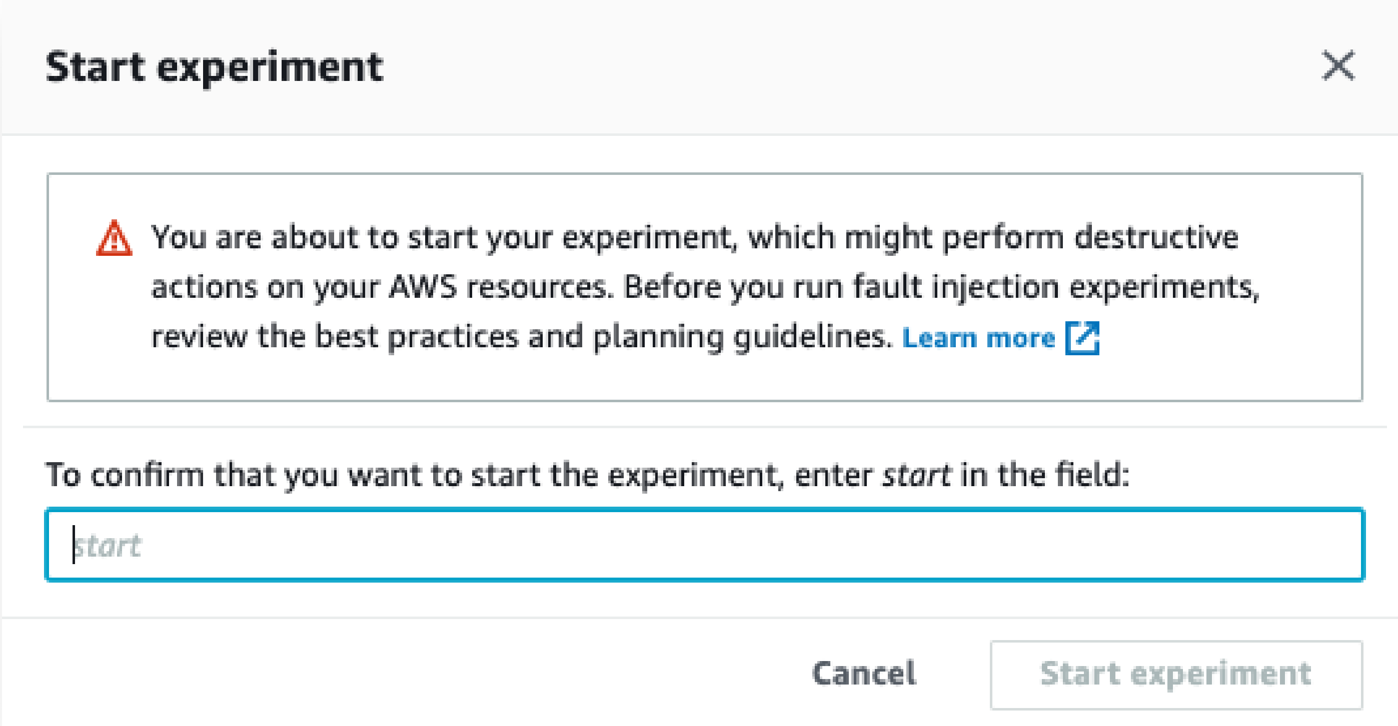


You can also verify the HTTP return code using this command, replacing `REPLACE_WITH_ECS_SERVICE_ALB_URL` with the load balancer DNS name you copied earlier:

```
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



Note

If you are working in CloudShell you terminal may expire throughout this workshop. To save your environment variables from this section so they re-populate when you restart your terminal, paste this into your shell:

```
source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.
```

OBSERVE THE SYSTEM

Review results

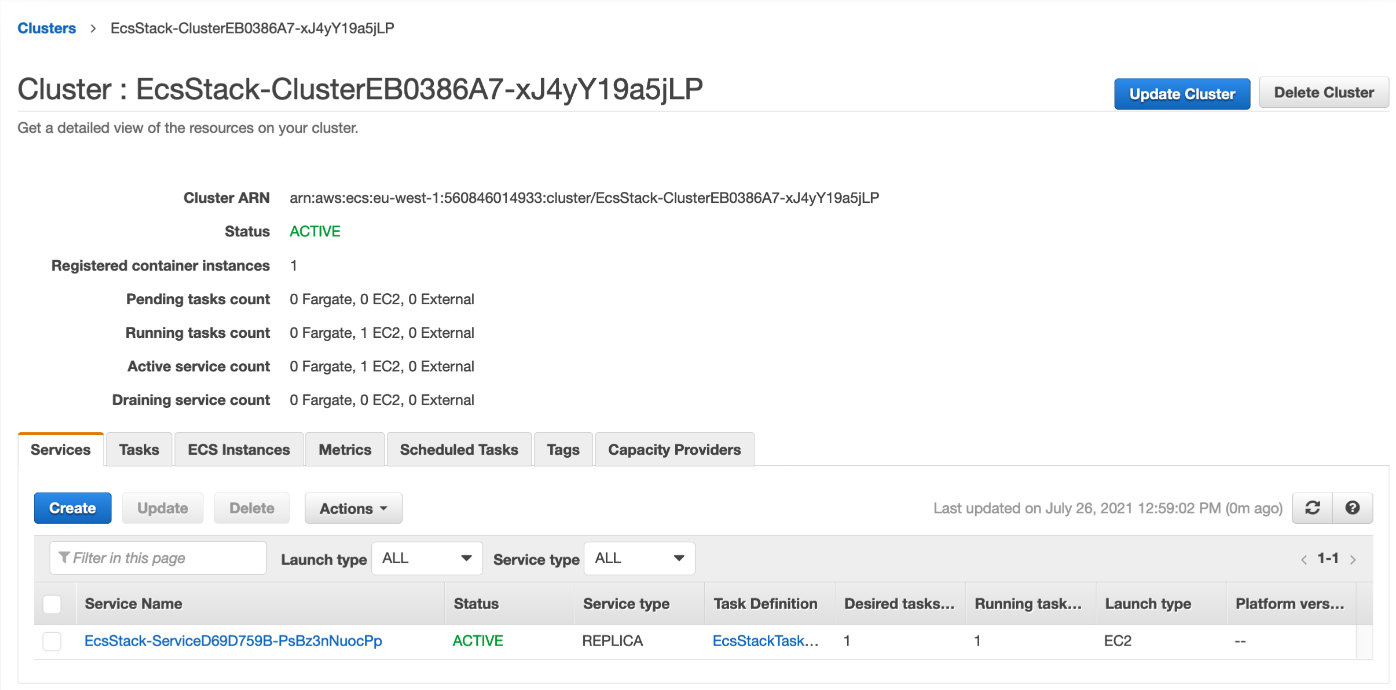
Let's take a look at the output in the terminal window where your Bash script is running:

```
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
```

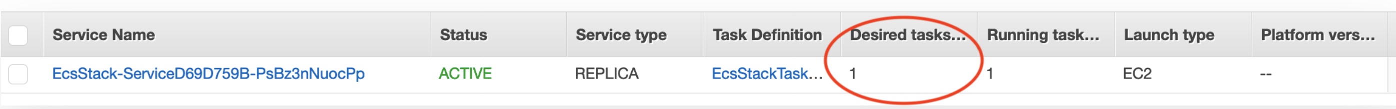
You'll notice that as not all the requests were successful. As the FIS experiment starts you should see some HTTP 502 "Bad Gateway" and HTTP 503 "Service Unavailable" return codes. This means our application was not available for a period of time. This is not what we were expecting, so let's dive a bit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

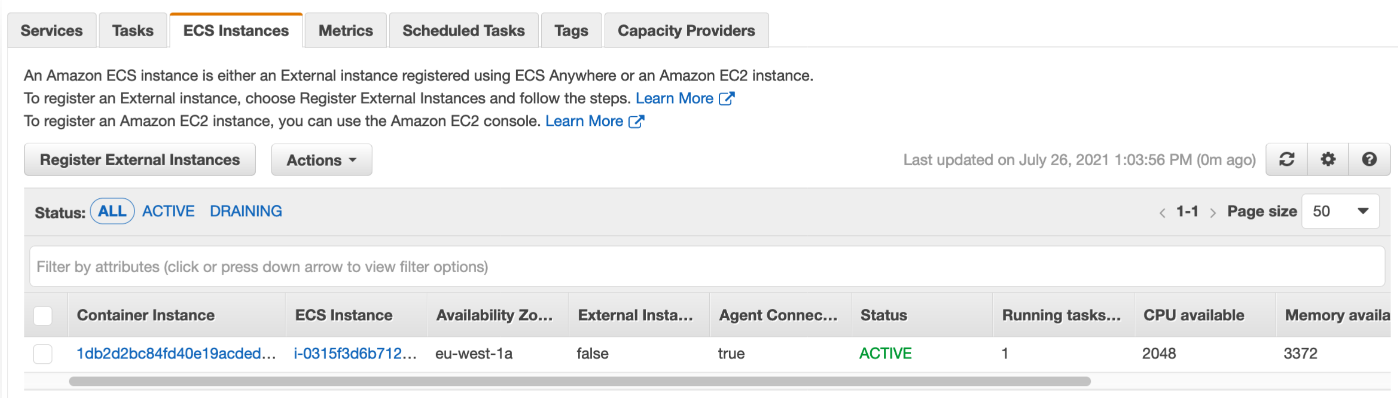


You'll notice that the service named `FisStackEcs-SampleAppService...`, e.g. `FisStackEcs-SampleAppServiceD69D759B-PsBz3nNuocPp` - i.e. our application - only has **one** desired task, meaning that only one copy of our containerized application will be running at any time.



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.

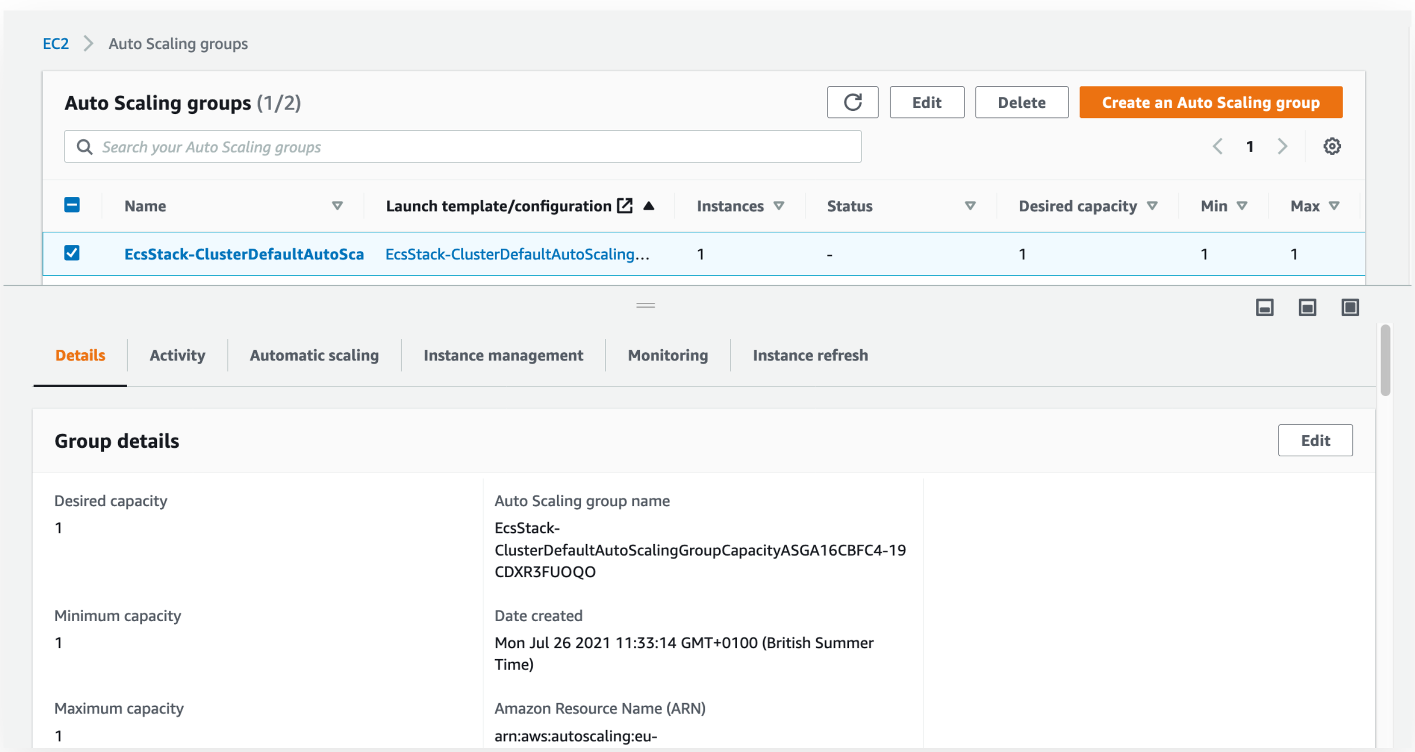
IMPROVE & REPEAT

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.

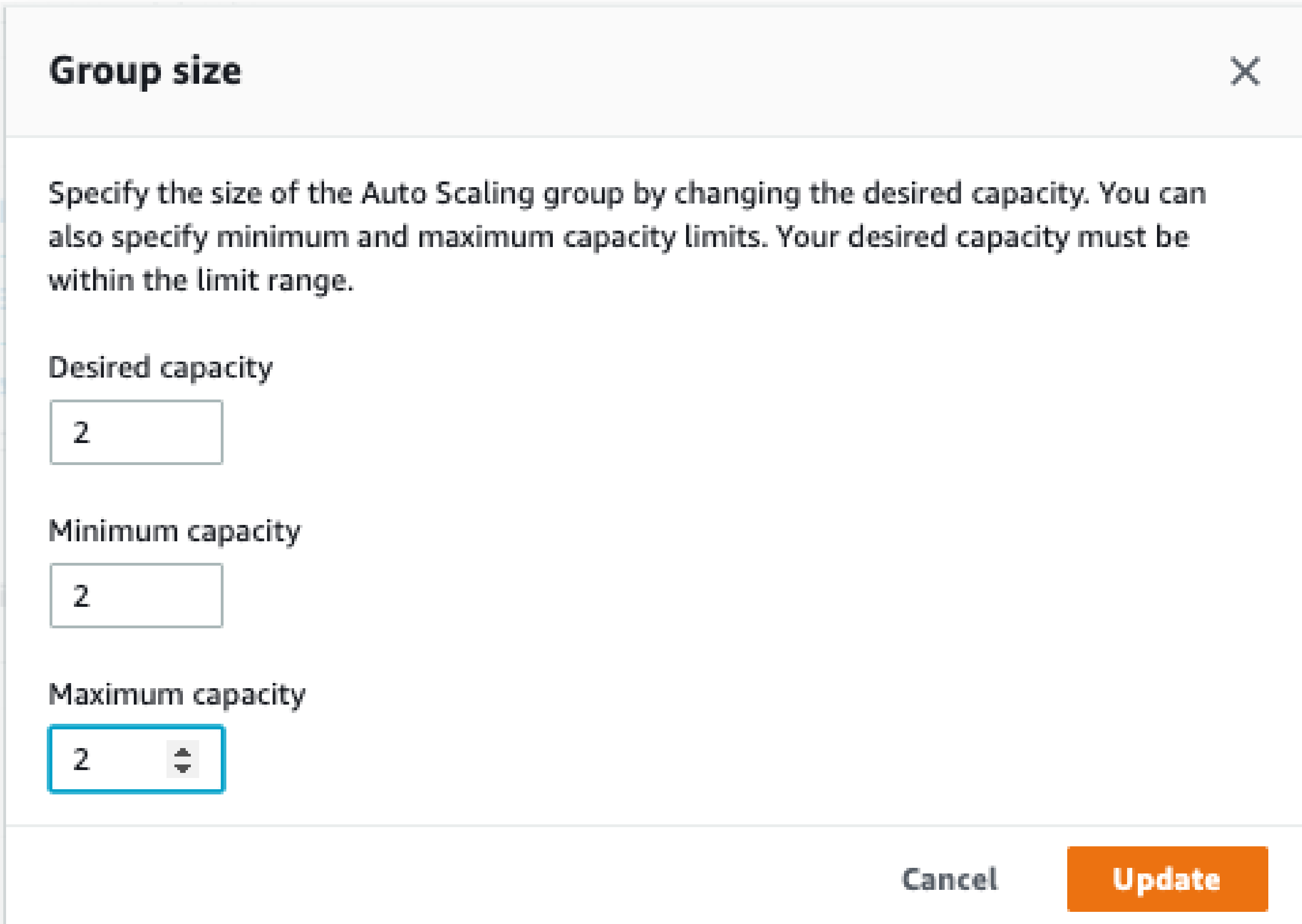
Increase the number of instances

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.



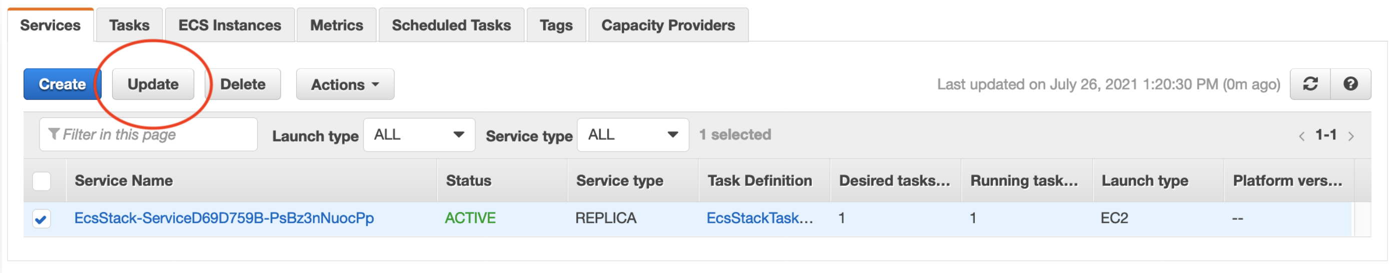
In the lower pane, in the **Details** tab and under **Group details** section, click the **Edit** button.

- 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:

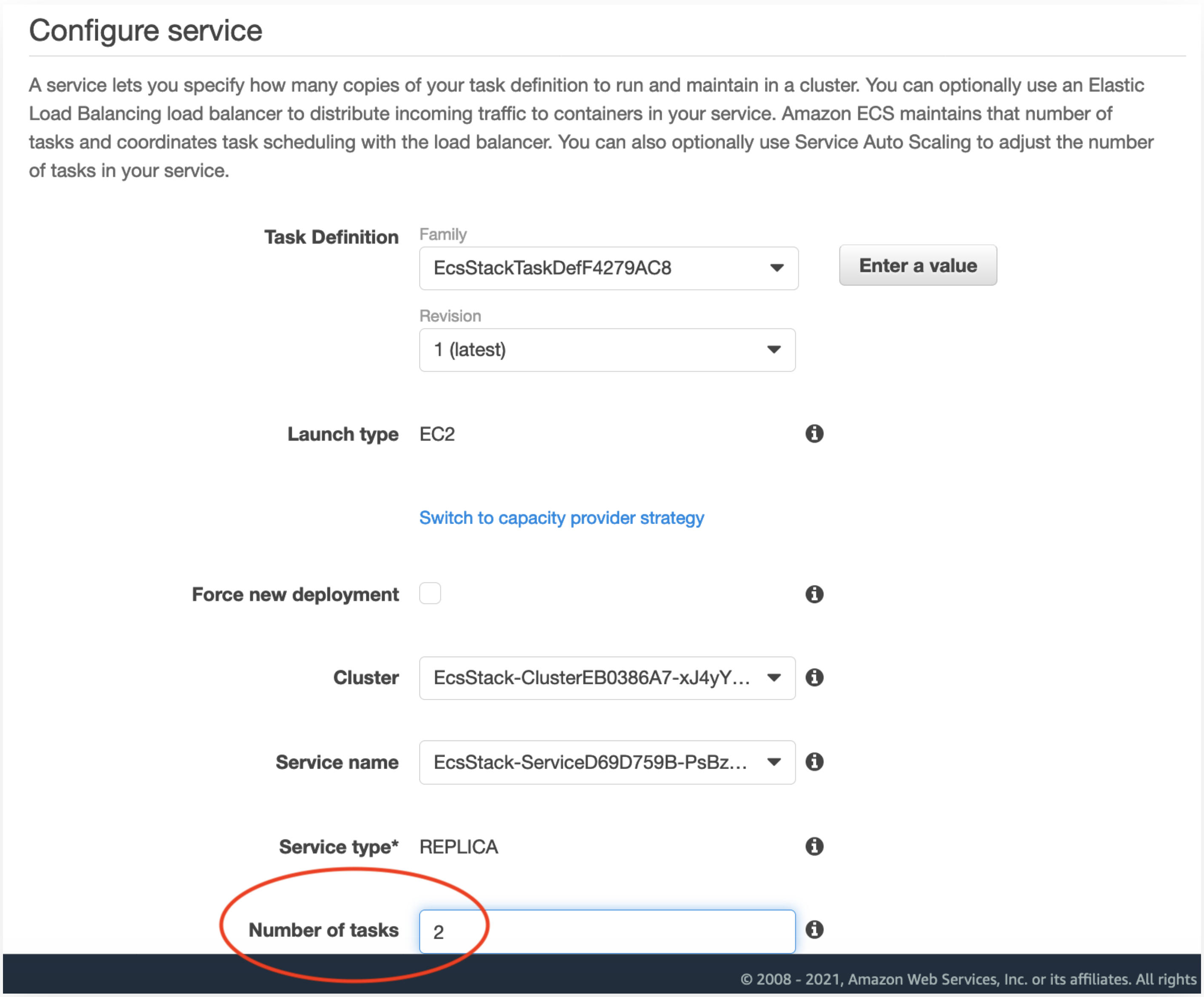


Increase the number of tasks

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`.

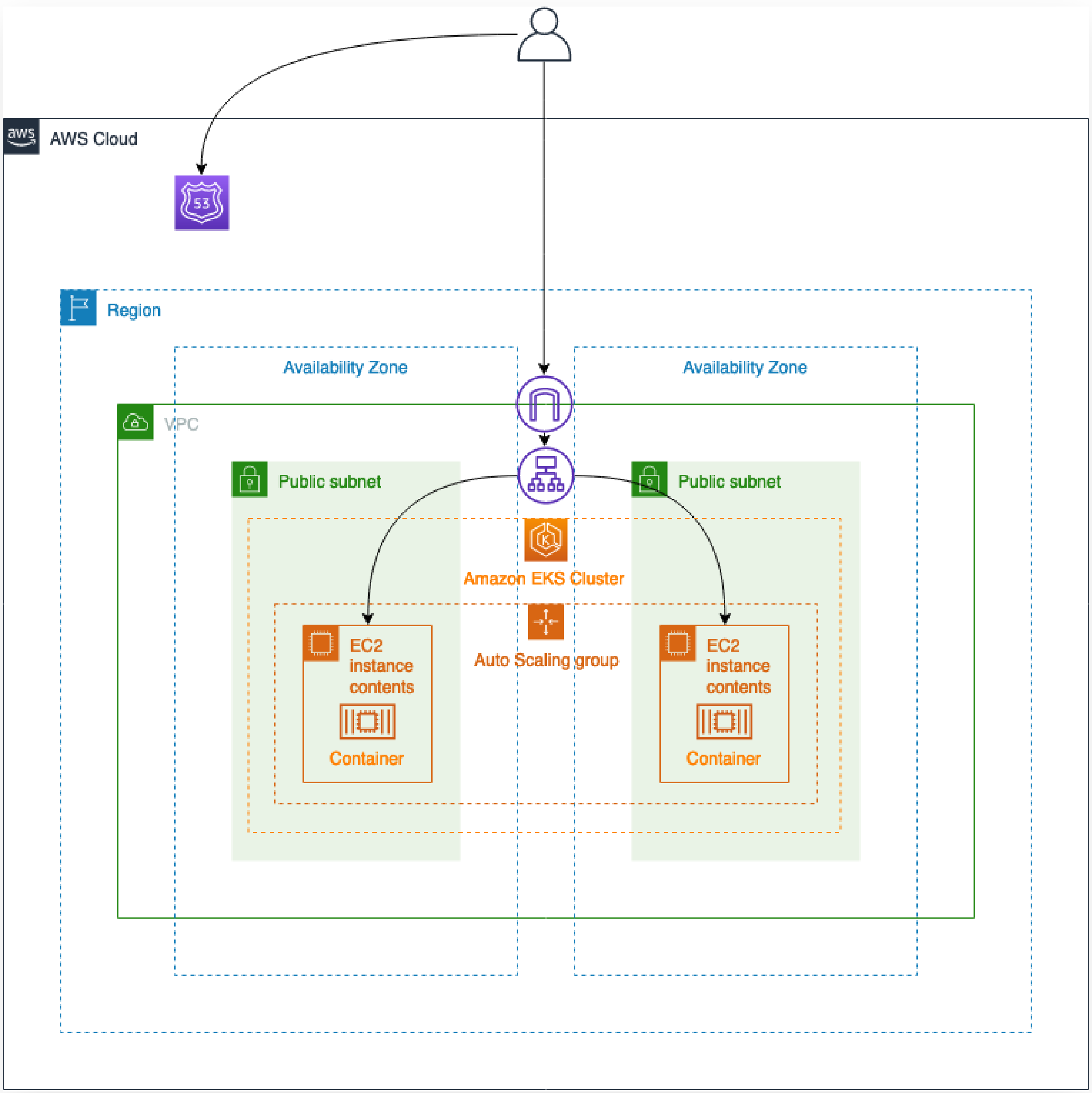
This time we should observe that, even when one of the container instances gets terminated, our application is still available and successfully serving requests. In the output of the Bash script there we should no longer see the HTTP 503 “Service Unavailable” return codes.

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 (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

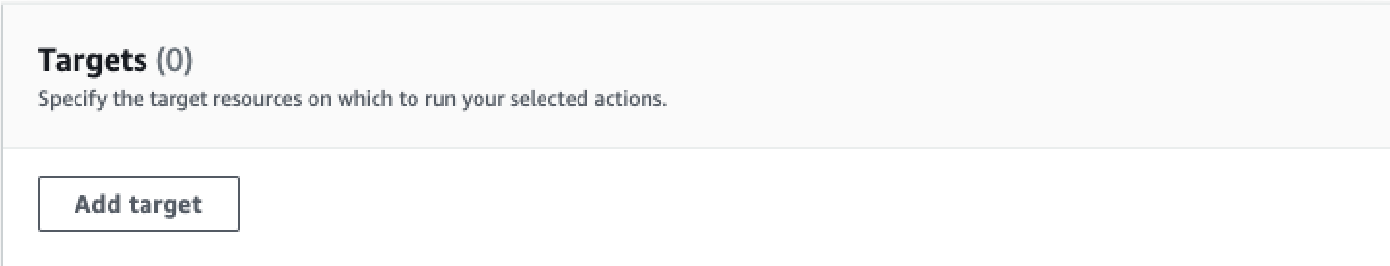
This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

Create a new experiment template:

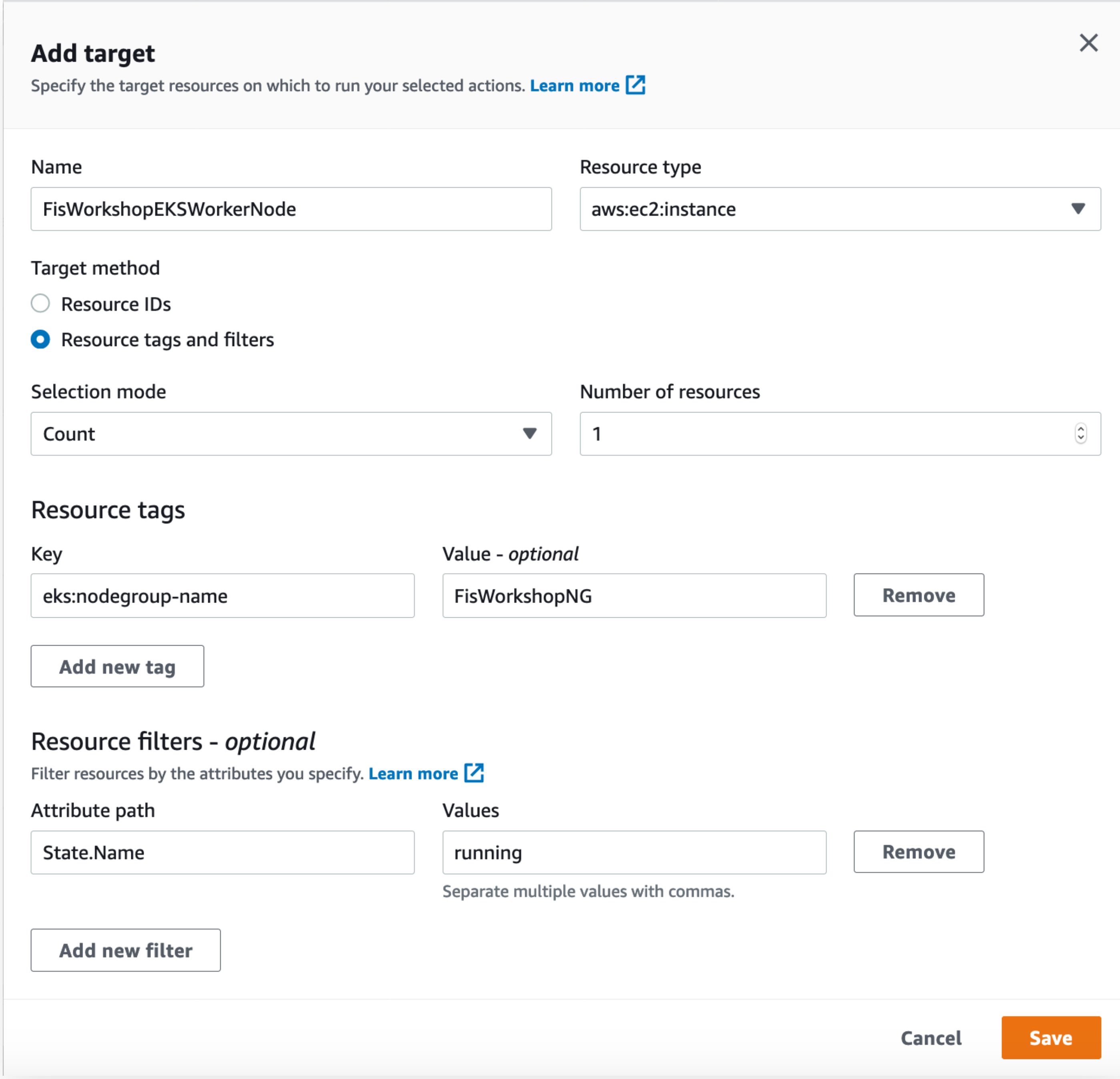
- add `Name` tag of `FisWorkshopEKS`
- add `Description` of `Terminate EKS Worker Node`
- select `FisWorkshopServiceRole` as execution role

Target selection

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



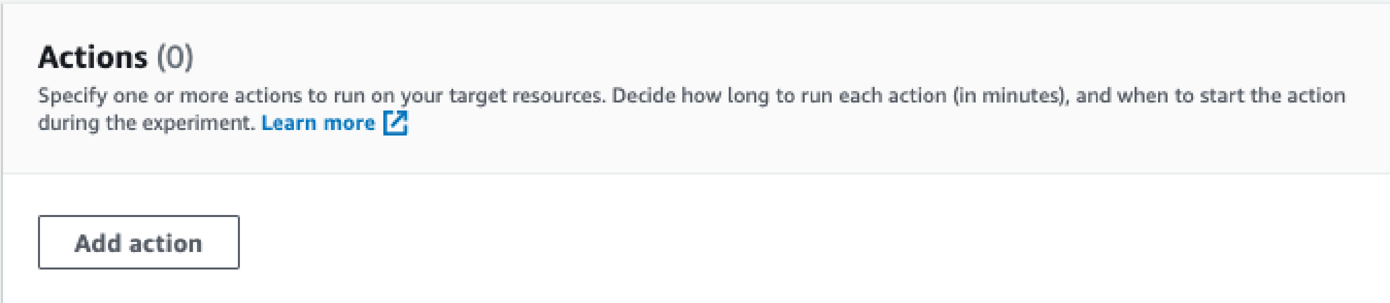
On the "Add target" popup enter `FisWorkshopEKSWorkerNode` for name and select `aws:ec2:instance`. For "Target method" we will dynamically select resources based on an associated tag. Select the `Resource tags and filters` checkbox. Pick `Count` from "Selection mode" and enter `1`. Under "Resource tags" 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

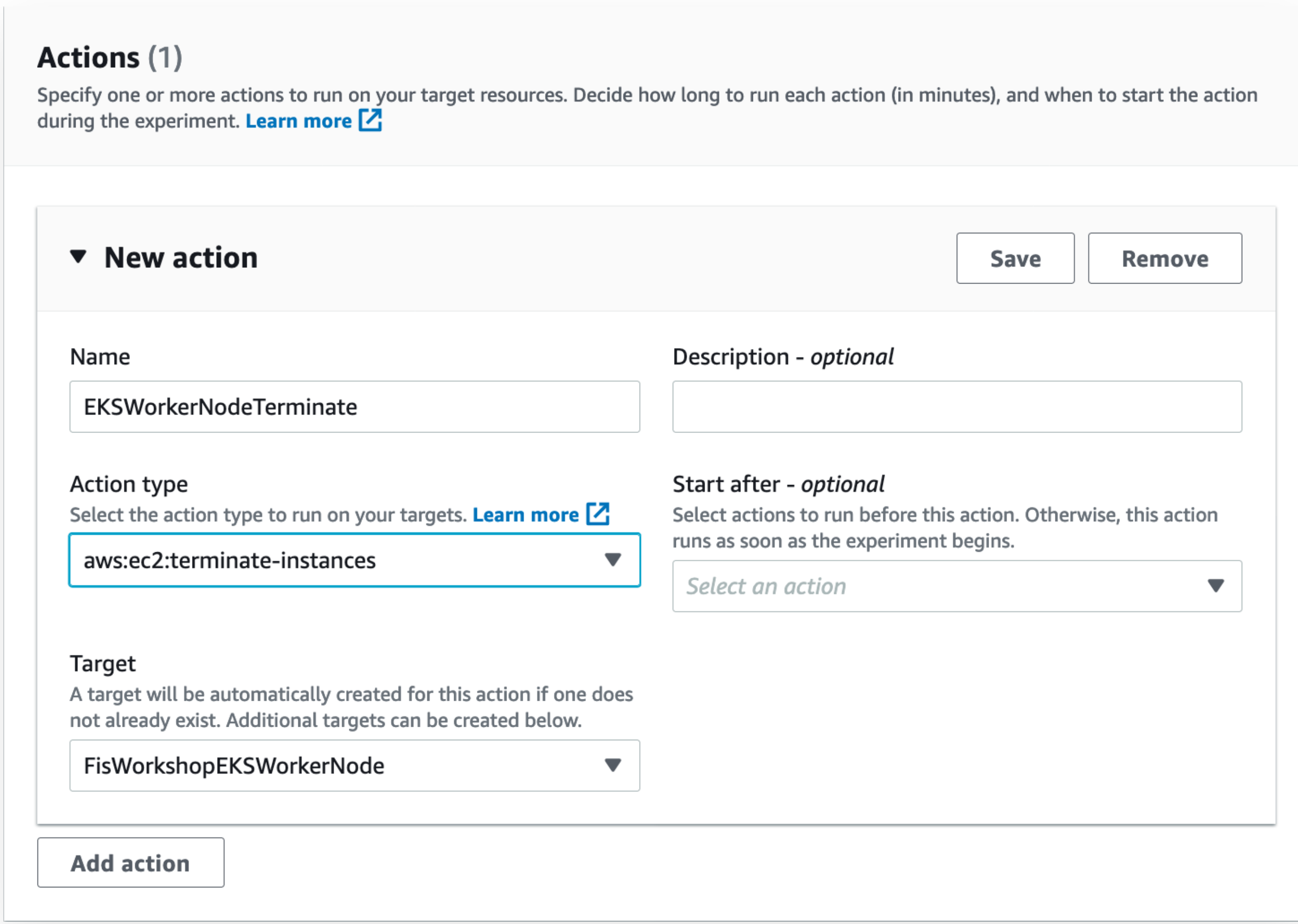
With targets defined we define the action to take. Scroll to the “Actions” section" and select “Add Action”



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

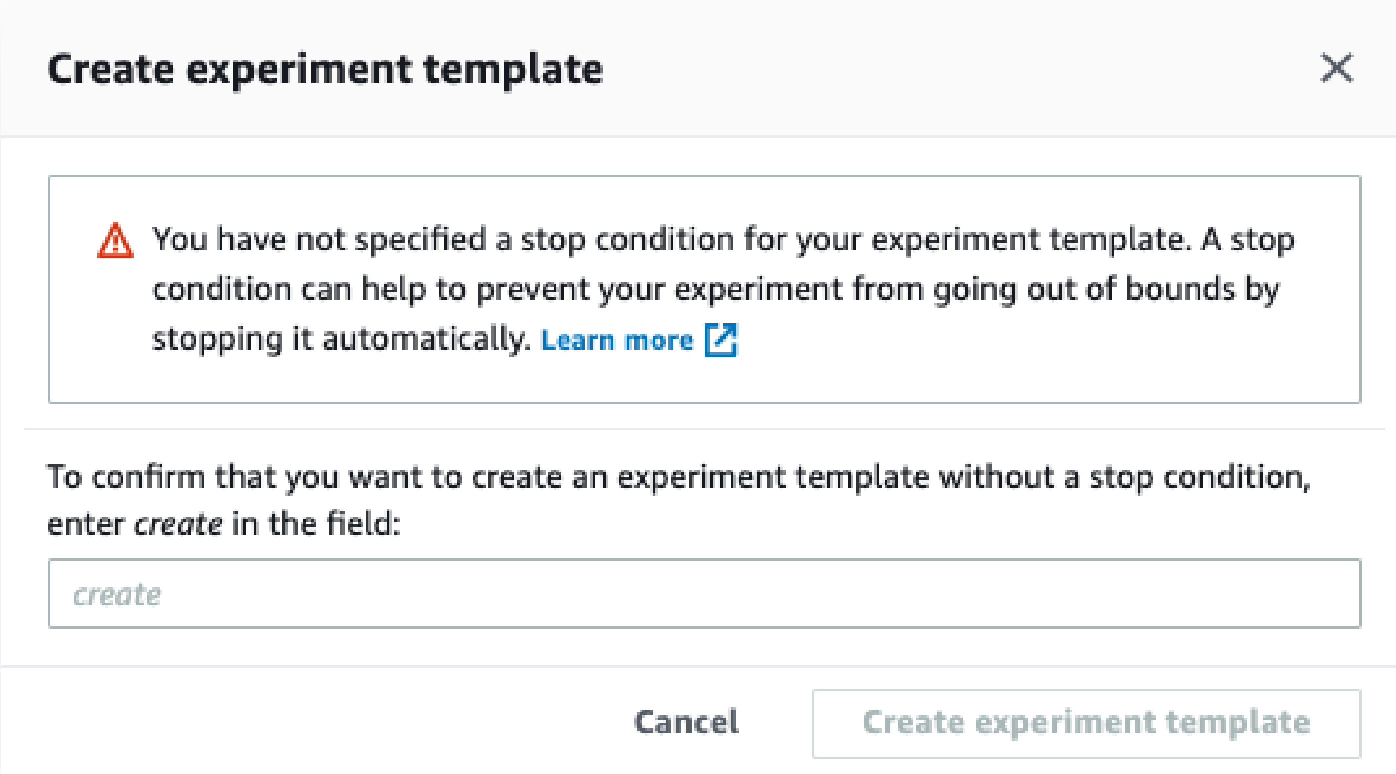
We will leave the “Start after” section blank since the instances we are terminating are part of an EKS Managed Node Group and we can let the Managed Node Group create new instances to replace the terminated ones.

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



Creating template without stop conditions

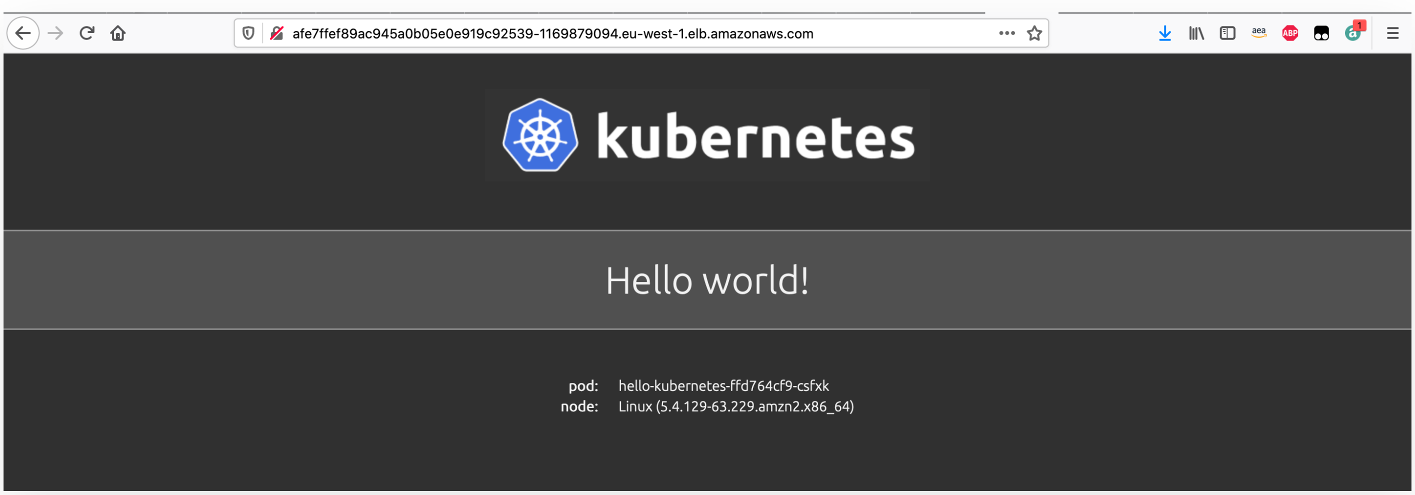
Confirm that you wish to create the template without stop condition.



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:

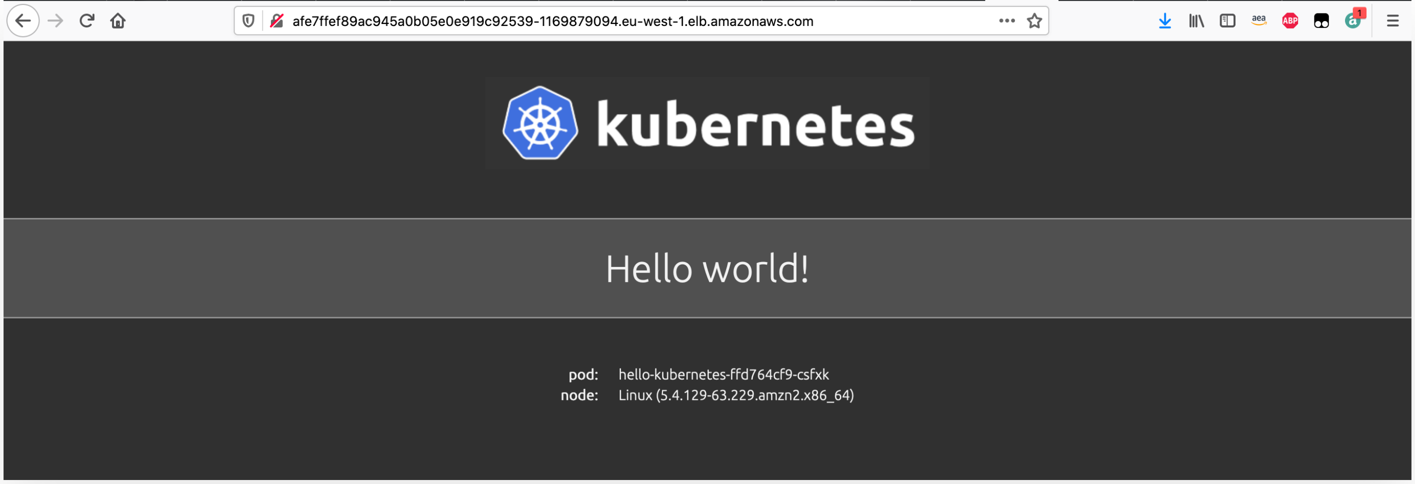
```
Code 200 Duration 0.140314
Code 200 Duration 0.086206
Code 200 Duration 0.085946
Code 200 Duration 0.084102
Code 200 Duration 0.085972
```

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:

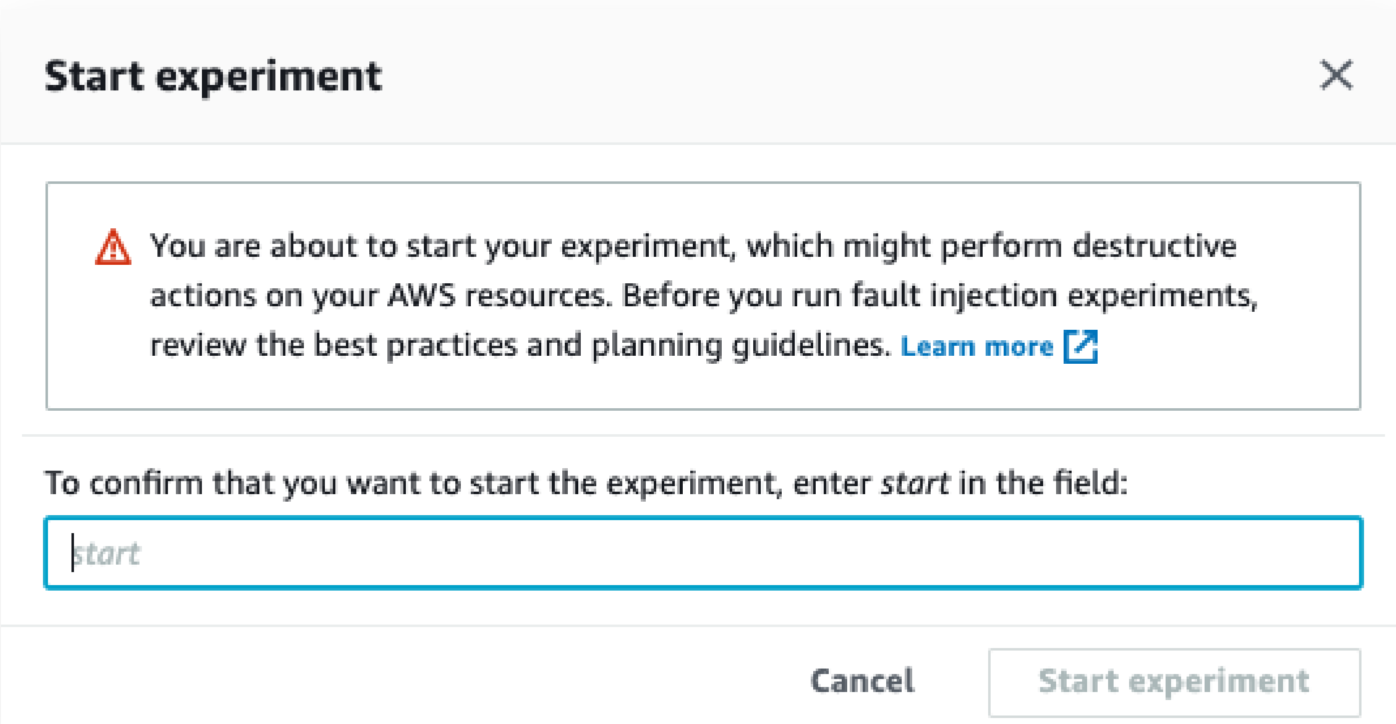


You can also verify the HTTP return code using this command, replacing `REPLACE_WITH_EKS_SERVICE_ALB_URL` with the load balancer DNS name you copied earlier:

```
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



Note

If you are working in CloudShell you terminal may expire throughout this workshop. To save your environment variables from this section so they re-populate when you restart your terminal, paste this into your shell:

```
source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.
```

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 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.

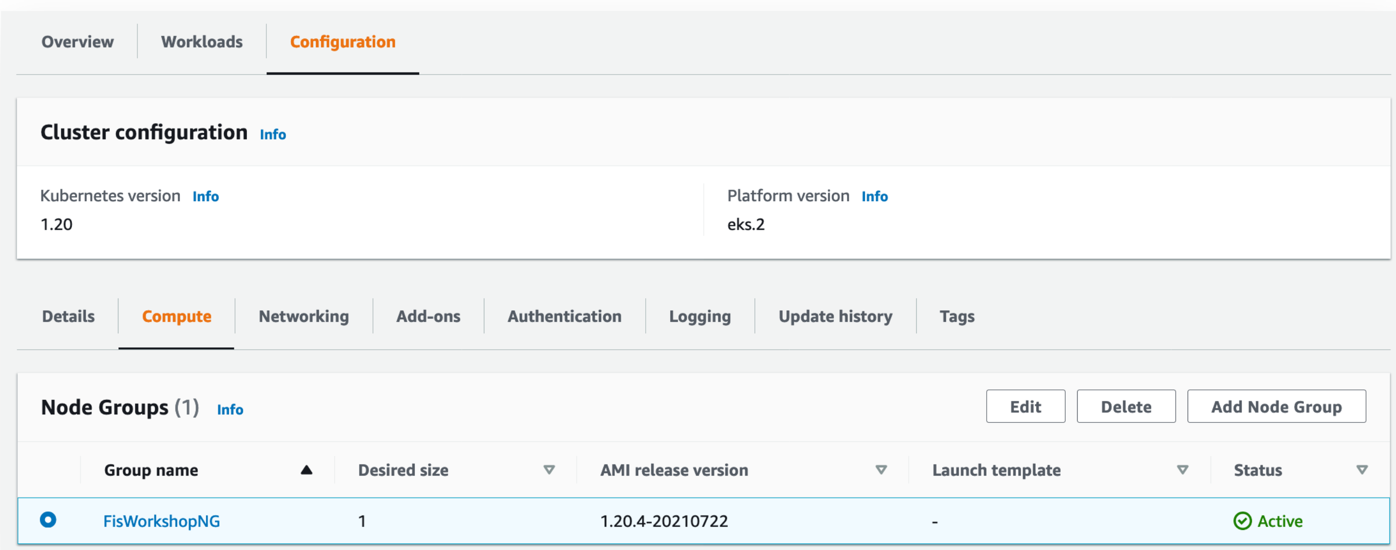
IMPROVE & REPEAT

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.

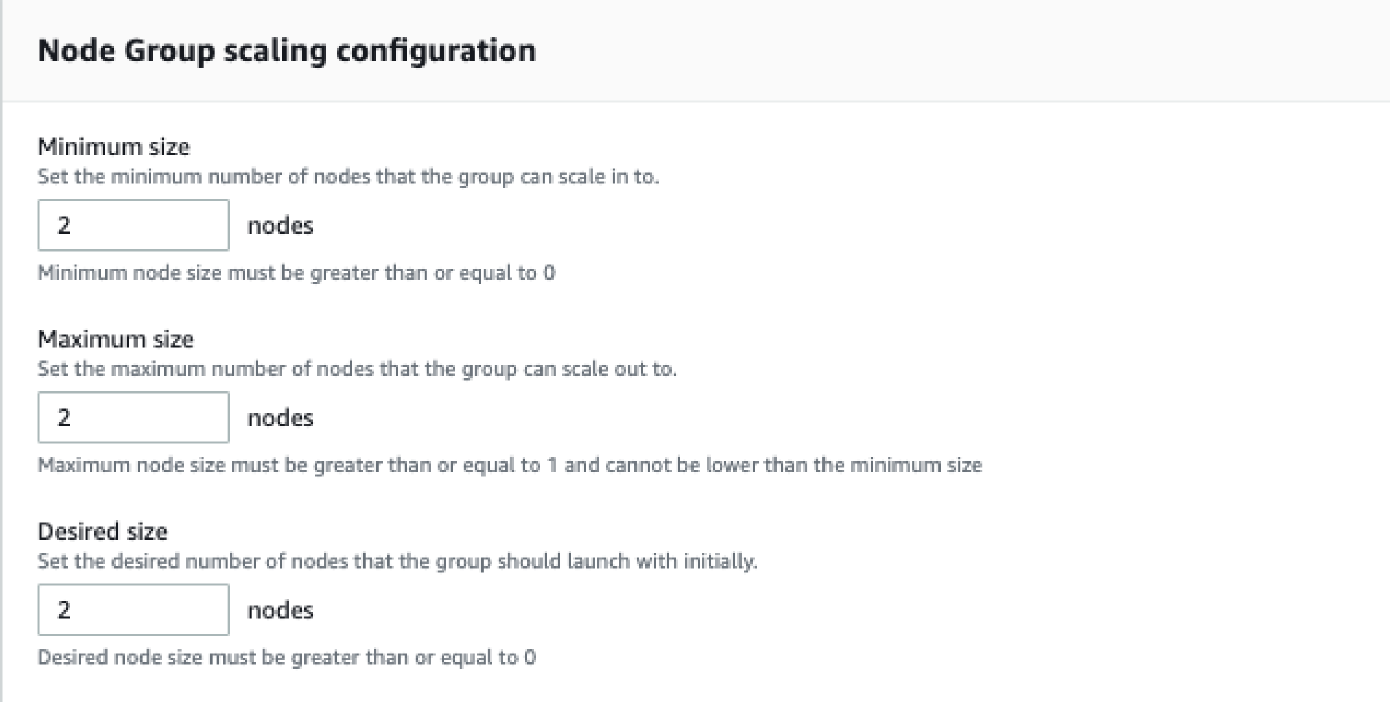
Increase the number of instances

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.

EKS further learning

For more on EKS configurations see the EKS workshop.

EC2 SPOT INSTANCES

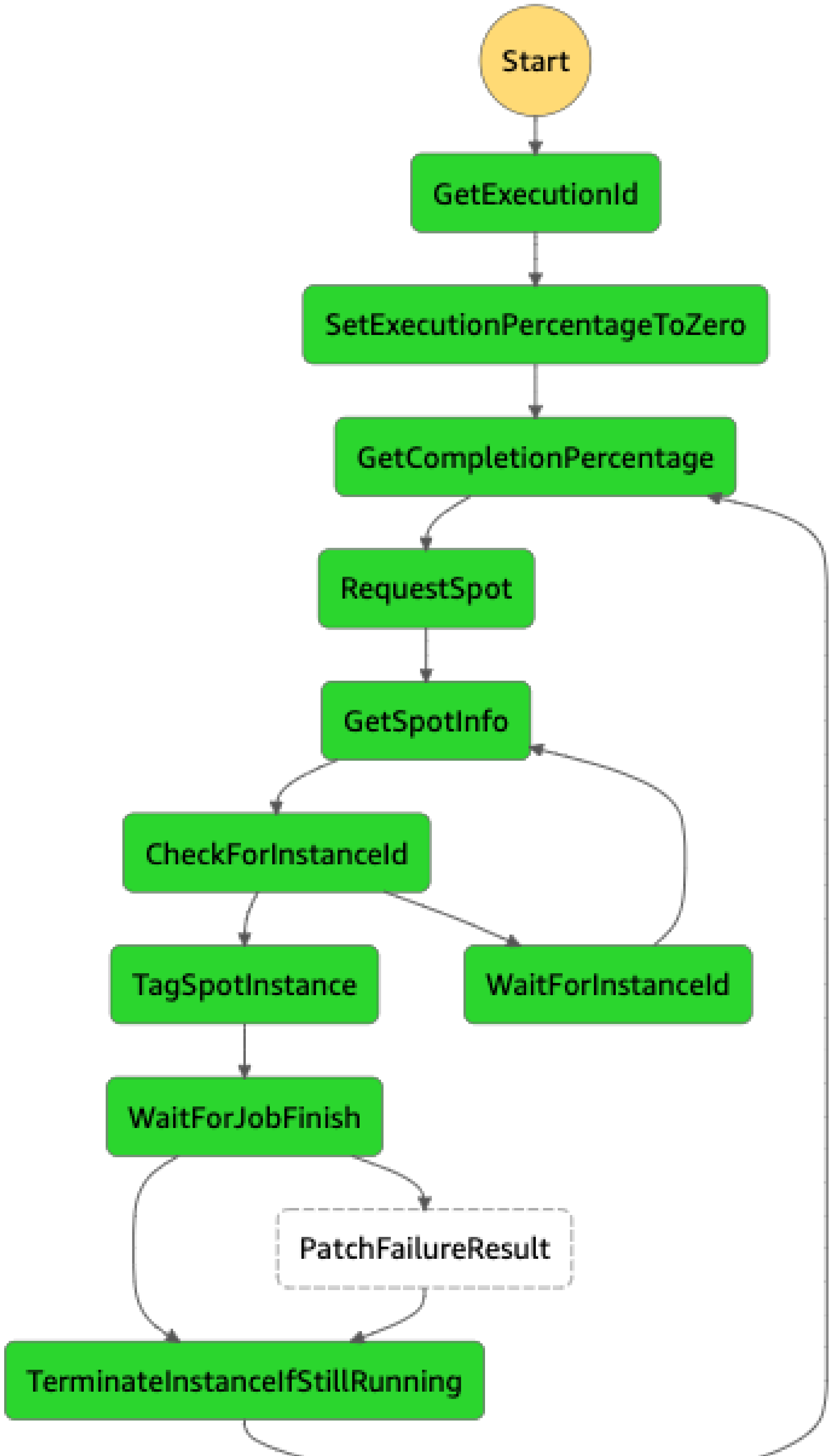
In this section we will cover how to validate AWS EC2 Spot Instance Interruption behavior.

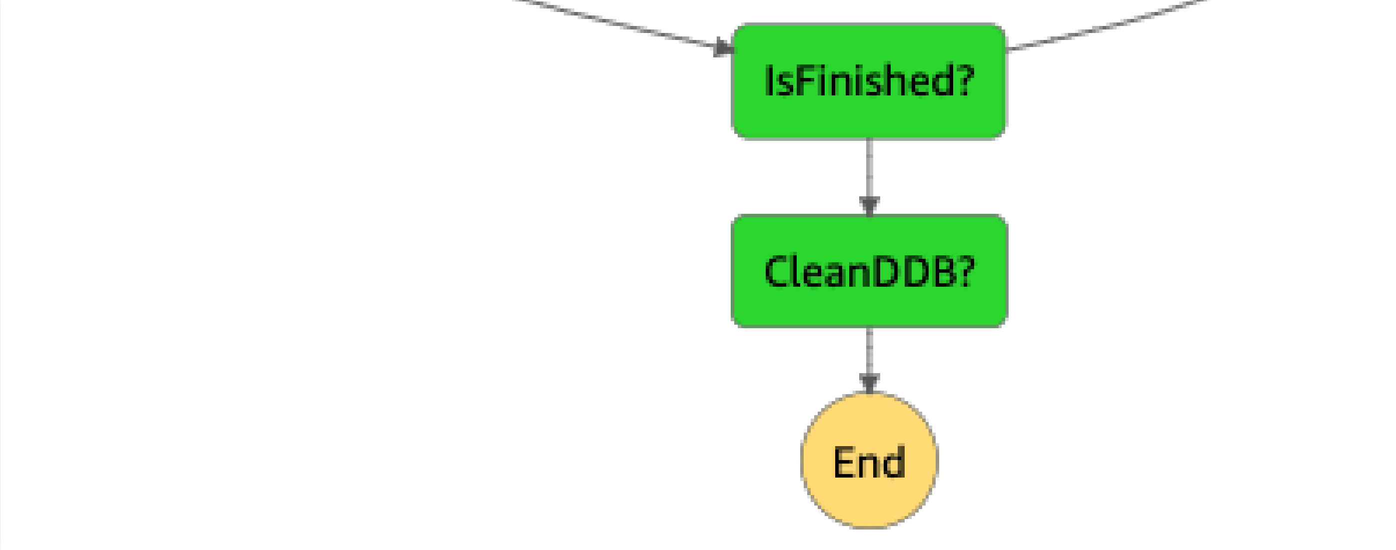
EC2 Spot Instances make spare EC2 capacity available for steep discounts in exchange for returning them when Amazon EC2 needs the capacity back. Because demand for Spot Instances can vary significantly over time, it is always possible that your Spot Instance might be interrupted.

To help you gracefully handle interruptions, AWS will send **Spot Instance Interruption notices** two minutes before Amazon EC2 stops or terminates your Spot Instance. While it is not always possible to predict demand, AWS may occasionally send an **EC2 rebalance recommendation** signal before sending the Instance interruption notice.

EC2 Spot instances can be used with Auto Scaling groups or as worker nodes for various forms of batch processing. Because nodes in Auto Scaling groups are usually stateless while batch processes usually generate stateful data we will demonstrate fault injection on a batch compute example with **checkpointing**.

In this section we will use **AWS Step Functions** to orchestrate a hypothetical batch workload:

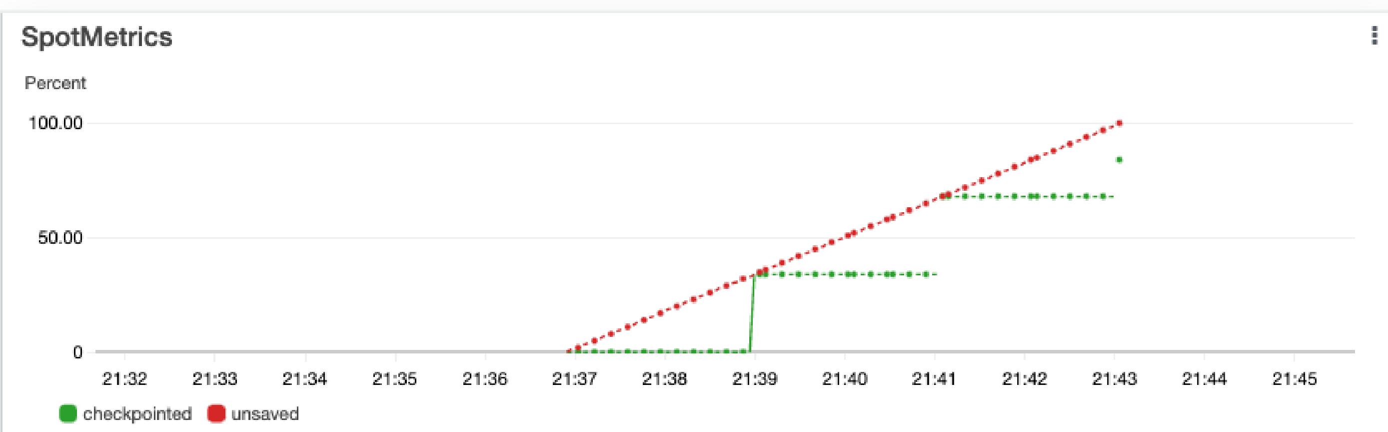

Start
GetExecutionId
SetExecutionPercentageToZero
GetCompletionPercentage
RequestSpot
GetSpotInfo
CheckForInstanceId
TagSpotInstance
WaitForInstanceId
WaitForJobFinish
PatchFailureResult
TerminateInstanceIfStillRunning



The workflow will:

- initialize a workload parameterized with total duration and checkpoint duration
- request a spot instance to run the workload
- wait for the spot instance run to finish
- repeat the request-and-wait cycle until 100% of the job is done

The workload is a python script, passed as **user data**, that writes metrics to CloudWatch:



More details in the next section.

BASELINING

Experiment idea

In this section we explore the effect of regular EC2 instance termination on an experiment with checkpoints enabled:

- **Given**: we have am AWS Step Functions workflow that will restart spot instances until the job is 100% finished.
- **Hypothesis**: terminating an EC2 spot instance will require additional computation but the job will reach 100% completion without human intervention.

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

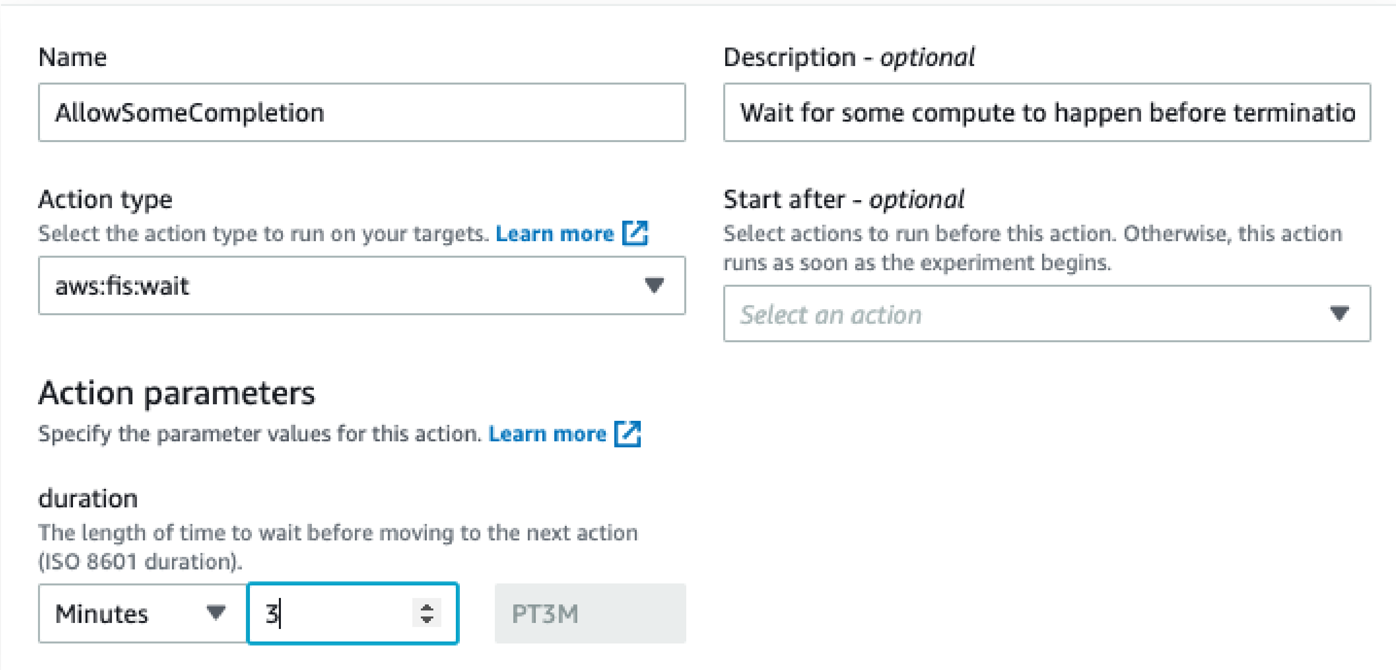
This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

- Create a new experiment template
  - Add "Name" tag of `FisWorkshopSpotTerminate`
  - Add "Description" of `Use EC2 terminate instances on spot instance`
  - Select `FisWorkshopSpotRole` as execution role

Action / Target definition 1

In this experiment we will introduce an initial wait before triggering instance termination. Go to the “Actions” section and select **“Add Action”**.

For “Name” enter `AllowSomeCompletion` and add a “Description” like `Wait for some compute to happen before termination`. For “Action type” select `aws:fis:wait` and for “Action parameters” / “duration” select `3` minutes. Select **“Save”**.

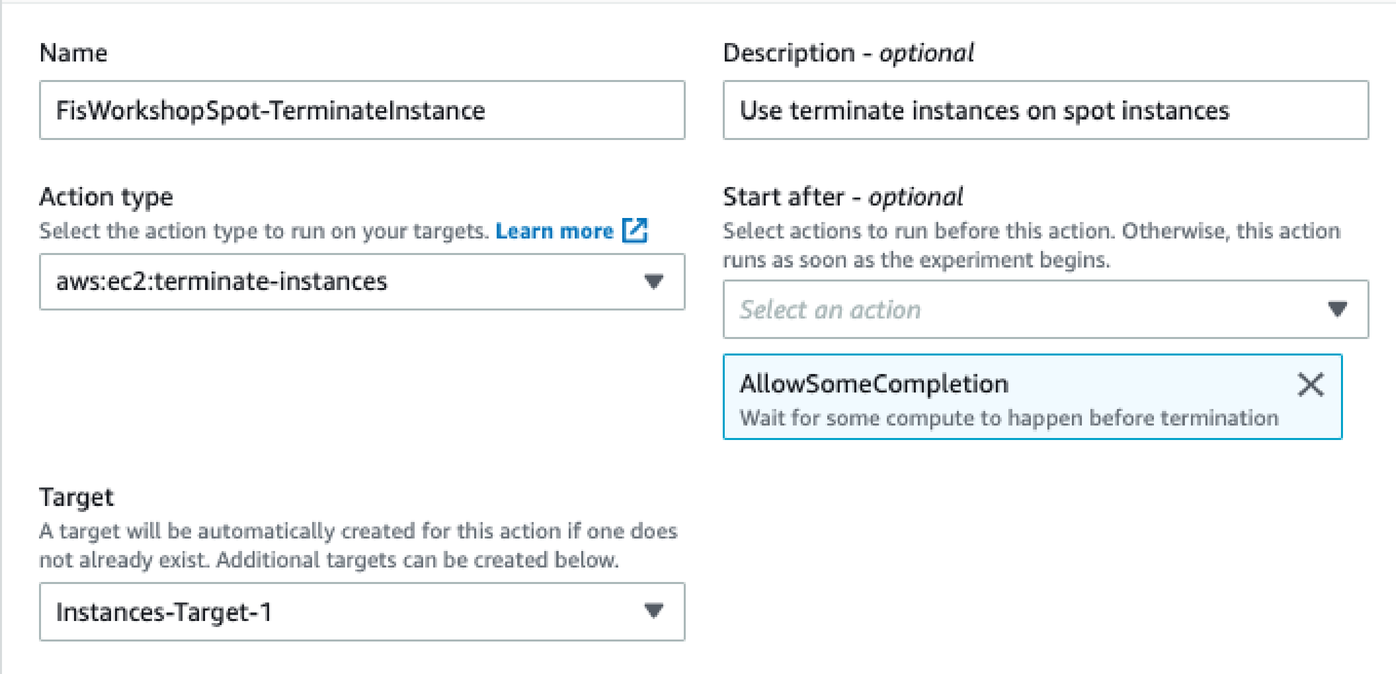


Action / Target definition 2

Following the same process as described in **First Experiment** define actions:

- “Name”: `FisWorkshopSpot-TerminateInstance`
- “Description”: `Use terminate instances on spot instances`
- “Action Type”: `aws:ec2:terminate-instances`

Since we want this action to execute after an initial wait, select the `AllowSomeCompletion` action from the “Start after” drop down.



Define targets by editing the auto-generated `Instances-Target-1` using:

- "Name": `FisWorkshopSpot-SpotInstance`
- "Resource type": `aws:ec2:instance`
- "Target method": "Resource tags and filters
- "Selection mode": "All"
- "Resource tags":
  - "Key": `Name`
  - "Value": `Fis/Spot`
- "Resource filters":
  - "Attribute path": `State.Name`
  - "Values": `running`

Validation procedure

Similar to the first experiment we will use a CloudWatch dashboard created as part of resource creation. Navigate to the **CloudWatch console** and select a dashboard named "FisSpot-REGION", e.g. `FisSpot-us-west-2`.

Run FIS experiment

First we need to start the StepFunctions workflow. For demonstration purposes we will run this with a total duration of 6 minutes and a checkpoint duration of 2 minutes but if you have the time you may want to explore what happens if you set checkpoint duration to >= total duration.

```
STATE_MACHINE_ARN=$( aws stepfunctions list-state-machines --query
"stateMachines[?contains(name,'SpotChaosStateMachine')].stateMachineArn" --output
text )

aws stepfunctions start-execution \
  --state-machine-arn ${STATE_MACHINE_ARN} \
  --input '{ "JobDuration": "6", "CheckpointDuration": "2"}'
```

Warning

Currently all target resolution is performed at the beginning of the experiment run. As such it is possible that the FIS experiment will fail target resolution if the spot instance is not running yet. If that happens, wait a few seconds and restart the FIS experiment below.

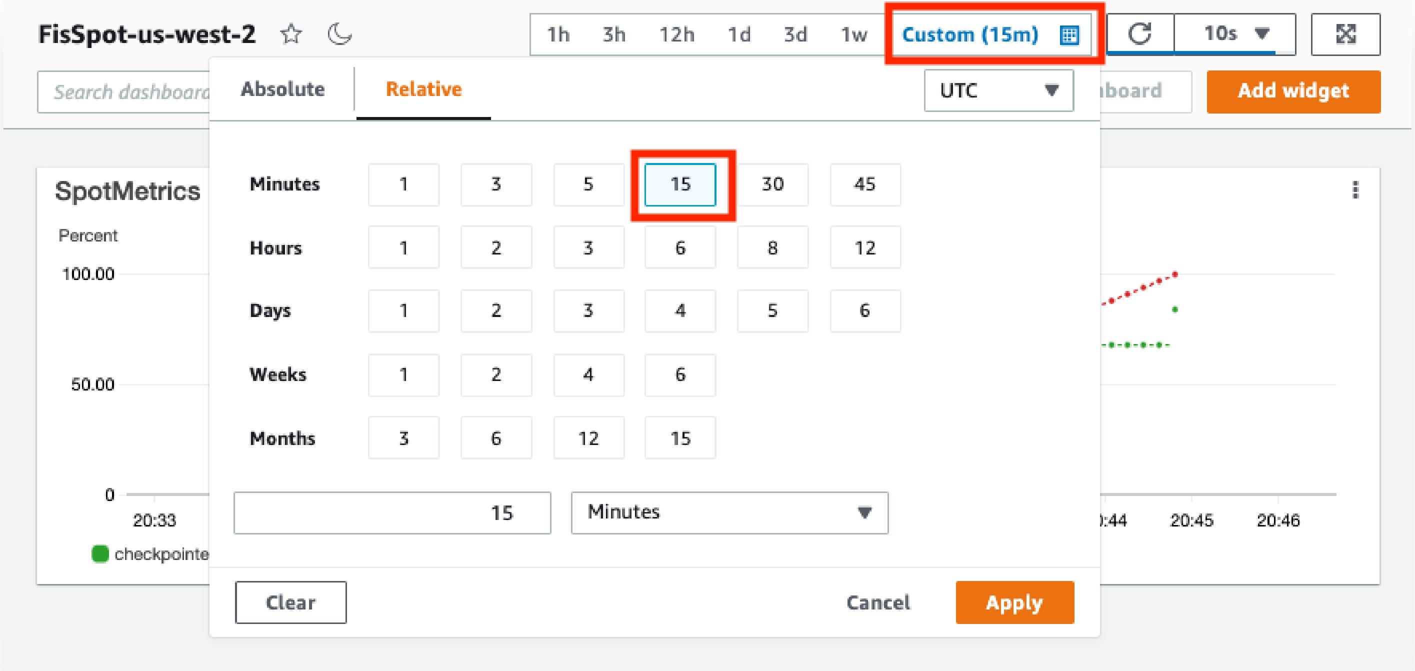
Then start the experiment. If you named the template as described above this should work, otherwise adjust `SPOT_EXPERIMENT_TEMPLATE_ID` as needed:

```
SPOT_EXPERIMENT_TEMPLATE_ID=$( aws fis list-experiment-templates --query
"experimentTemplates[?tags.Name=='FisWorkshopSpotTerminate'].id" --output text )

aws fis start-experiment \
  --experiment-template-id $SPOT_EXPERIMENT_TEMPLATE_ID \
  --tags Name=FisWorkshopSpotTerminateTest \
| jq -rc '.experiment.id'
```

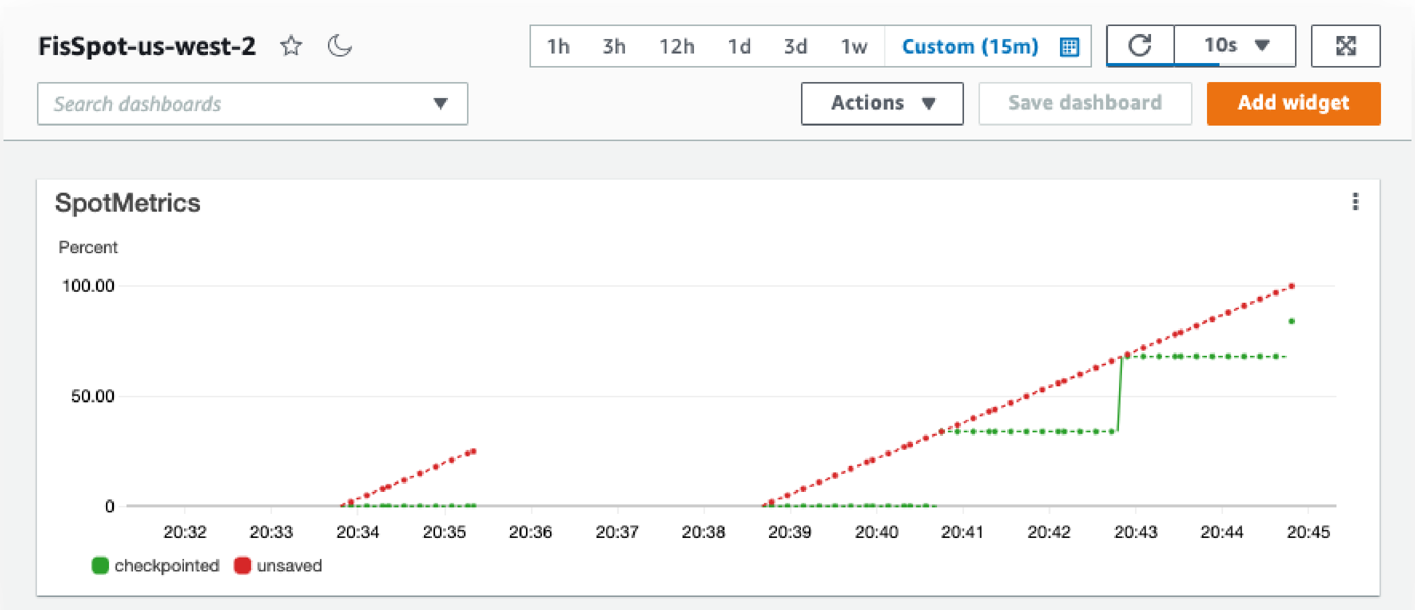
Copy the experiment ID and navigate to the **FIS console**. Search for the experiment ID and check that the state is "running". If the experiment failed because of empty target lookup, run the start experiment command again.

If the experiment keeps failing, navigate to the **StepFunctions console**, select the "SpotChaosStateMachine" and examine the most recent execution to ensure a spot instance has been created.

Finally navigate to the **CloudWatch console**, select the FisSpot dashboard and set a custom duration of 15min:



You may have to wait for a few minutes for data to become available. You should then see data like this (no checkpoint happened before interruption):



Learning and improving

From the graphs we can see that the workflow will successfully restart from the last checkpoint. However, we can also see that a substantial amount of progress has to be re-calculated and it would be better if we could save progress closer to the actual interruption of the instance. In the next section we will repeat the same experiment but using the `aws:ec2:send-spot-instance-interruptions` action which will replicate normal spot instance interruption behavior by sending a notification before terminating the instance.

SIMULATING INTERRUPTS

Experiment idea

In this section we explore how to mitigate the effect of instance interruption by reacting to the spot instance interrupt notification:

- **Given**: we have am AWS Step Functions workflow that will restart spot instances until the job is 100% finished.
- **Hypothesis**: capturing the spot instance interrupt request and checkpointing when it is received will better utilize EC2 spot instances and the job will still reach 100% completion without human intervention.

Experiment setup

Note

We will follow the exact same steps as in the **previous section**. We will only change the action type from `aws:ec2:instance` to `aws:ec2:send-spot-instance-interruptions`.

Warning

Even though the target selection looks the same as before, spot instance target selection is distinct from EC2 instance target selection. For this reason it is recommended that you create a completely new experiment template here instead of just editing the previous one.

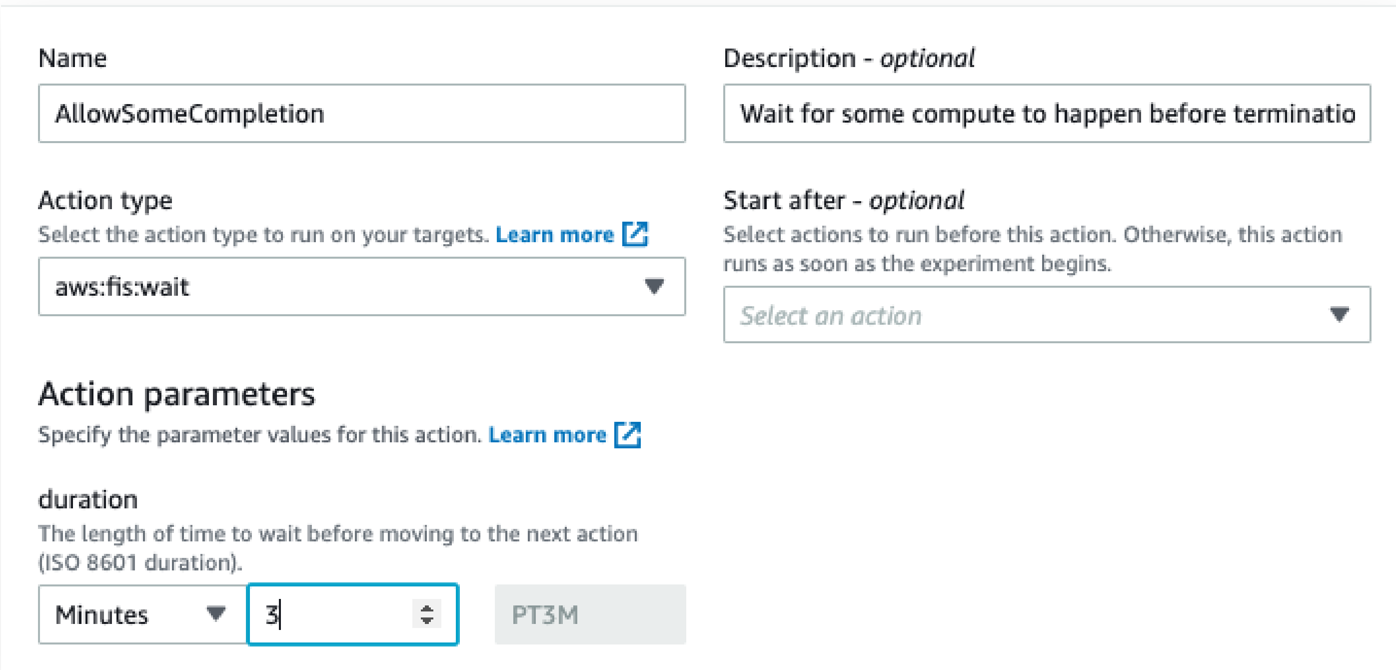
General template setup

- Create a new experiment template
  - Add "Name" tag of `FisWorkshopSpotInterrupt`
  - Add "Description" of `Use spot instance interruption on spot instance`
  - Select `FisWorkshopSpotRole` as execution role

Action / Target definition 1

Define action:

- “Name”: `AllowSomeCompletion`
- “Description”: `Wait for some compute to happen before termination`
- “Action Type”: `aws:fis:wait`
- “Action parameters” / “duration”: `3` minutes

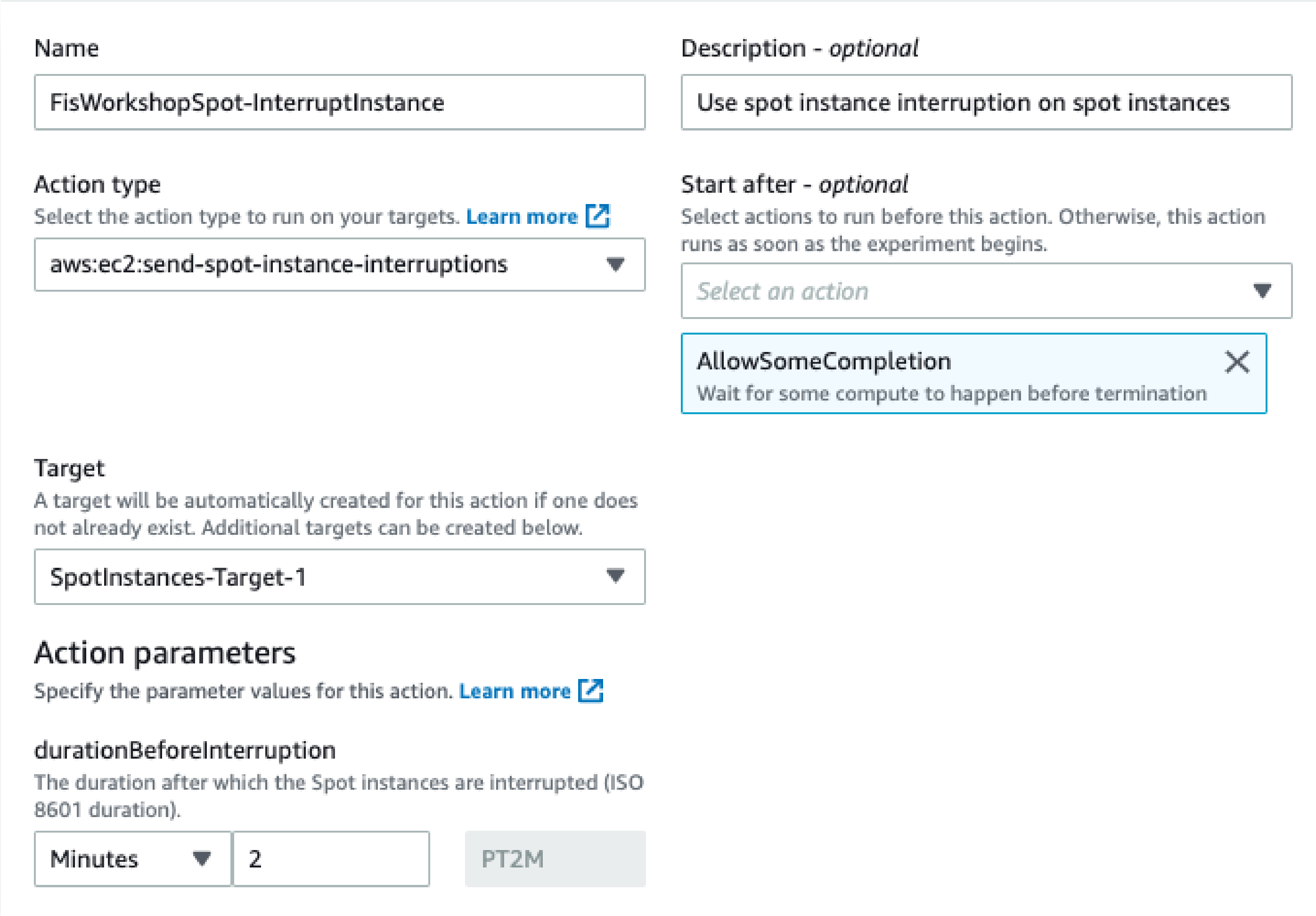


Action / Target definition 2

Define action:

- “Name”: `FisWorkshopSpot-InterruptInstance`
- “Description”: `Use spot instance interruption on spot instances`
- “Action Type”: `aws:ec2:send-spot-instance-interruptions`
- “Start after”: `AllowSomeCompletion`

We also need to set an amount of time to pass between the notification and the actual instance termination. We will set this to the minimum allowed value of `2` minutes:



Define targets by editing the auto-generated `SpotInstances-Target-1` using:

- "Name": `FisWorkshopSpot-SpotInstance`
- "Resource type": `aws:ec2:spot-instance`
- "Target method": "Resource tags and filters
- "Selection mode": "All"
- "Resource tags":
  - "Key": `Name`
  - "Value": `Fis/Spot`
- "Resource filters":
  - "Attribute path": `State.Name`
  - "Values": `running`

Validation procedure

Similar to the first experiment we will use a CloudWatch dashboard created as part of resource creation. Navigate to the **CloudWatch console** and select a dashboard named "FisSpot-REGION", e.g. `FisSpot-us-west-2`.

Run FIS experiment

First we need to start the StepFunctions workflow. For demonstration purposes we will run this with a total duration of 6 minutes and a checkpoint duration of 2 minutes but if you have the time you may want to explore what happens if you set checkpoint duration to >= total duration.

```
STATE_MACHINE_ARN=$( aws stepfunctions list-state-machines --query
"stateMachines[?contains(name,'SpotChaosStateMachine')].stateMachineArn" --output
text )

aws stepfunctions start-execution \
  --state-machine-arn ${STATE_MACHINE_ARN} \
  --input '{ "JobDuration": "6", "CheckpointDuration": "2"}'
```

Warning

Currently all target resolution is performed at the beginning of the experiment run. As such it is possible that the FIS experiment will fail target resolution if the spot instance is not running yet. If that happens, wait a few seconds and restart the FIS experiment below.

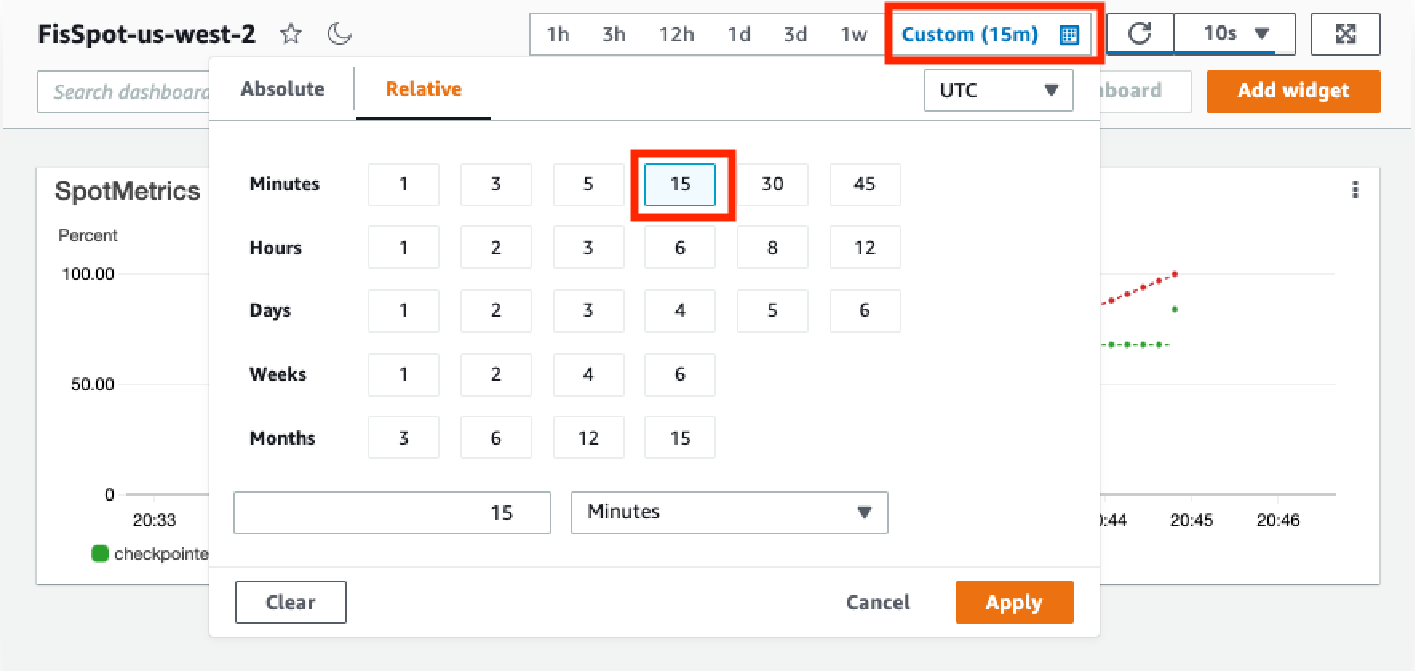
Then start the experiment. If you named the template as described above this should work, otherwise adjust `SPOT_EXPERIMENT_TEMPLATE_ID` as needed:

```
SPOT_EXPERIMENT_TEMPLATE_ID=$( aws fis list-experiment-templates --query
"experimentTemplates[?tags.Name=='FisWorkshopSpotInterrupt'].id" --output text )

aws fis start-experiment \
  --experiment-template-id $SPOT_EXPERIMENT_TEMPLATE_ID \
  --tags Name=FisWorkshopSpotInterruptTest \
| jq -rc '.experiment.id'
```

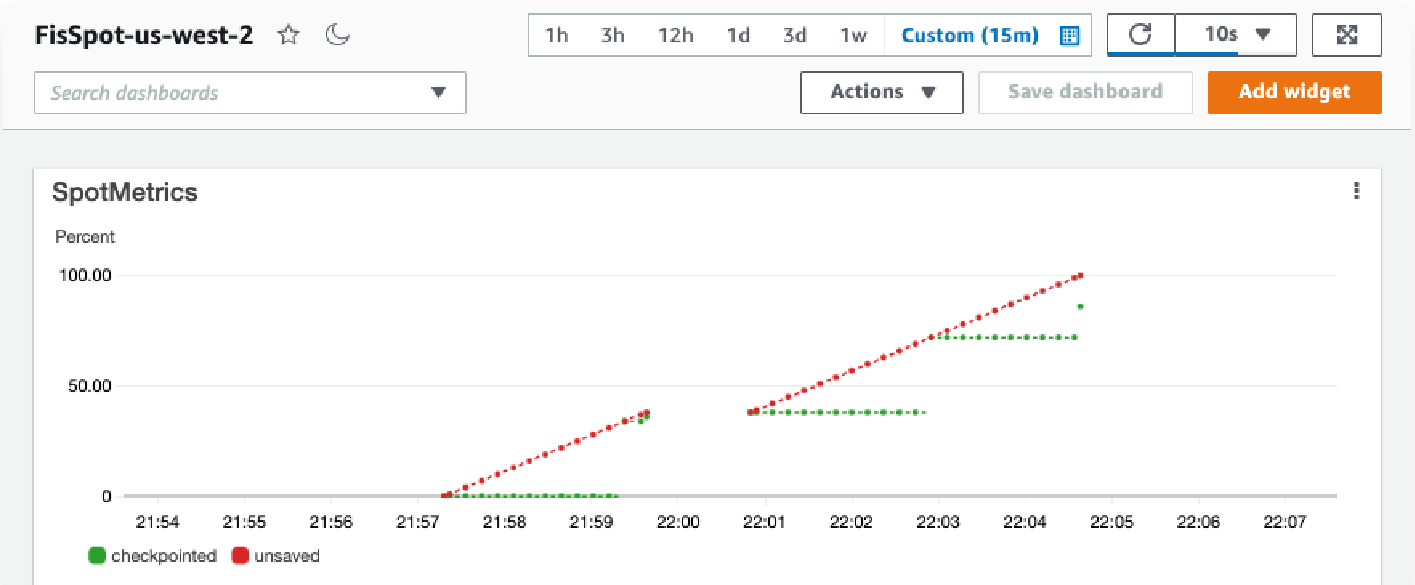
Copy the experiment ID and navigate to the **FIS console**. Search for the experiment ID and check that the state is “running”. If the experiment failed because of empty target lookup, run the start experiment command again.

If the experiment keeps failing, navigate to the **StepFunctions console**, select the “SpotChaosStateMachine” and examine the most recent execution to ensure a spot instance has been created.

Finally navigate to the **CloudWatch console**, select the FisSpot dashboard and set a custom duration of 15min:



You may have to wait for a few minutes for data to become available. You should then see data like this. In this graph a checkpoint happened at the 2minute mark and another checkpoint immediately after that resulting from the instance interruption. Notably the newly created spot instance did not have to re-do any of the work:



Learning and improving

Capturing the spot instance interruption notice and acting on it can substantially decrease the amount of repeated calculations.

In our example the checkpointing is instantaneous whereas in the real world checkpointing might require substantial amounts of time for data offloading, writing to databases etc. With the ability to simulate interruption behavior you can now tune your interrupt behavior to make the most of your spot resources.

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.

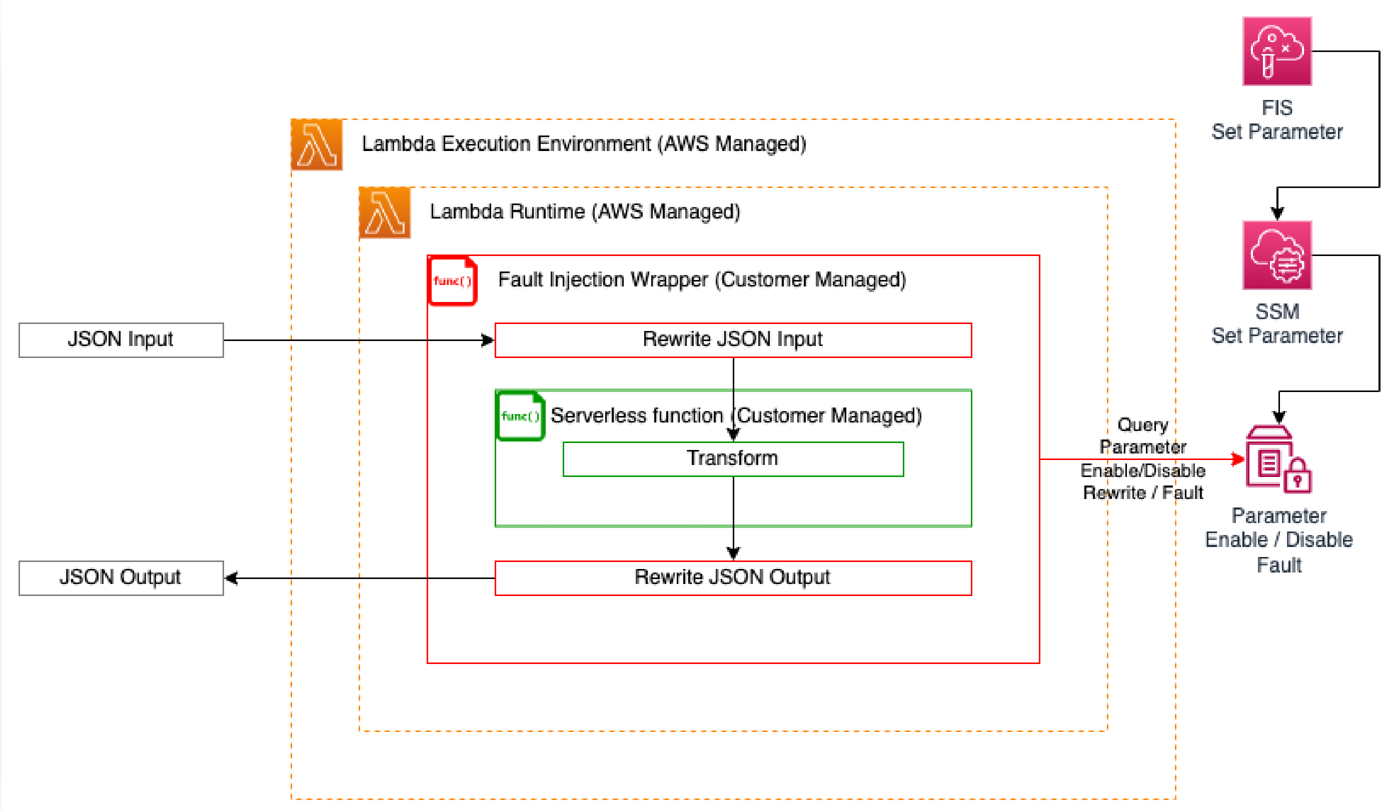
In this section we use the open source **chaos_lambda** python library to demonstrate how to

- inject latency into serverless calls,
- change the response code of the serverless function, and
- simulate exceptions in code execution.

A similar JS library, **failure-lambda** is described in the **Serverless Chaos workshop** and the understanding of the principles should allow the reader to build their own in their preferred language.

Architecture

In AWS Lambda, serverless functions expose a "handler" function that receives a JSON object and returns a JSON object. We can keep this handler function unchanged by inserting a new wrapper function around the original handler function, e.g. using a **decorator pattern**. This wrapper can cause exceptions in lambda function execution before or after customer code is called, can inject latency before or after customer code is called, can modify output thus simulating a failure result, and can even modify input thus triggering failures in the customer function code:



In chaos-lambda, to allow injecting failures on-demand, the wrapper function will query an **SSM Parameter Store parameter ** to check whether failures should be injected at all and, if so, what failures.

To inject failures in the context of an FIS experiment, we will use an **SSM Automation** document to change the value of the SSM Parameter Store parameter and turn on different types of failures.

Experiment idea

In this section we are focusing on tooling rather than presenting a full experiment, with some guidance on how to expand on the tooling at the end.

Specifically in this section we will inject failures in an API backed by AWS Lambda instrumented with chaos_lambda. We will run an FIS experiment that will, in order:

- inject latency
- inject an error code response
- inject a runtime exception

We will observe these changes by continuously checking response time, response code, and response body.

Experiment setup

Note

The experiment setup section is for reference only. All required components have been created as part of the workshop setup. If you just want to see the serverless fault injection you can skip ahead to "Validation Procedure" below.

Note

In earlier sections we have described how configure to service roles, create FIS experiment templates, and create SSM automation documents. For this section, we have created all the required resources as part of the infrastructure setup, and we will only outline the configuration process on the console.

Prerequisites

We will be using an SSM document to call the SSM PutParameter API. As such, we will require an IAM role that allows `ssm:PutParameter` - see **template definition in GitHub**. Name this role `FisWorkshopLambdaSsmRole`.

We will also need an IAM role that allows FIS to call SSM Automation and pass the above role to SSM - see **template definition in GitHub**. Name this role `FisWorkshopLambdaServiceRole`

Finally, we will need an SSM automation document to put a parameter value - see **template definition in GitHub**. Note that this document will create or overwrite the parameter with a value that disables fault injection. If you create this document manually you will have to construct the ARN as described in the **Working with SSM documents** section.

General template setup

- Add "Description" of `Inject Lambda Failures`
- Add a "Name" of `FisWorkshopLambdaFailure`
- Select `FisWorkshopLambdaServiceRole` as execution role

Action definition

We will define multiple actions that we want to run in sequence. This follows the same procedure as before except that we will populate the optional "Start after" selection to sequence action execution. Create the following actions:

- "Name": `S01_EnableLatency`

- "Action type": `aws:ssm:start-automation-execution`
- "Start after": leave this empty as the first step starts at the beginning of the experiment
- "documentArn": the ARN found in the "Outputs" tab of the FisStackServerless **CloudFormation**.
- "documentParameters": Reformatted here for legibility. For the `AutomationAssumeRole` you will need to insert the ARN of the `FisWorkshopLambdaSsmRole` either from the "Outputs" of the cloudformation stack or from the "Prerequisites" section.

```
{
    "AutomationAssumeRole":
"arn:aws:iam::ACCOUNT_ID:role/FisStackServerless-
FisWorkshopLambdaSsmRole-xxxxyyyyzzzz",
    "FaultParameterValue": "{
        \"is_enabled\":true,
        \"fault_type\":\"latency\",
        \"delay\":400,
        \"error_code\":404,
        \"exception_msg\":\"Fault injected by chaos-lambda\",
        \"rate\":1
    }"
}
```

- "maxDuration": `1` "Minutes"

- "Name": `S02_Wait1`
  - "Action type": `aws:fis:wait`
  - "Start after": `S01_EnableLatency`
  - duration: `1` "Minutes"

- "Name": `S03_EnableStatusCode`
  - "Action type": `aws:ssm:start-automation-execution`
  - "Start after": `S02_Wait1`
  - "documentArn": the ARN found in the "Outputs" tab of the FisStackServerless **CloudFormation**.
  - "documentParameters": Reformatted here for legibility. For the `AutomationAssumeRole` you will need to insert the ARN of the `FisWorkshopLambdaSsmRole` either from the "Outputs" of the cloudformation stack or from the "Prerequisites" section.

```
{
    "AutomationAssumeRole":
"arn:aws:iam::ACCOUNT_ID:role/FisStackServerless-
FisWorkshopLambdaSsmRole-xxxxyyyyzzzz",
    "FaultParameterValue": "{
        \"is_enabled\":true,
```

```
            \"fault_type\":\"status_code\",
            \"delay\":400,
            \"error_code\":404,
            \"exception_msg\":\"Fault injected by chaos-lambda\",
            \"rate\":1
        }"
    }
```

   - "maxDuration": `1` "Minutes"

- "Name": `S04_Wait2`

   - "Action type": `aws:fis:wait`
   - "Start after": `S03_EnableStatusCode`
   - duration: `1` "Minutes"

- "Name": `S05_EnableException`

   - "Action type": `aws:ssm:start-automation-execution`
   - "Start after": `S04_Wait2`
   - "documentArn": the ARN found in the "Outputs" tab of the FisStackServerless **CloudFormation**.
   - "documentParameters": Reformatted here for legibility. For the `AutomationAssumeRole` you will need to insert the ARN of the `FisWorkshopLambdaSsmRole` either from the "Outputs" of the cloudformation stack or from the "Prerequisites" section.

```
    {
        "AutomationAssumeRole":
    "arn:aws:iam::ACCOUNT_ID:role/FisStackServerless-
    FisWorkshopLambdaSsmRole-xxxxyyyyzzzz",
        "FaultParameterValue": "{
            \"is_enabled\":true,
            \"fault_type\":\"exception\",
            \"delay\":400,
            \"error_code\":404,
            \"exception_msg\":\"Fault injected by chaos-lambda\",
            \"rate\":1
        }"
    }
```

   - "maxDuration": `1` "Minutes"

- "Name": `S06_Wait3`

   - "Action type": `aws:fis:wait`
   - "Start after": `S05_EnableException`

- duration: `1` "Minutes"

- "Name": `S07_DisableFaults`

  - "Action type": `aws:ssm:start-automation-execution`
  - "Start after": `S06_Wait3`
  - "documentArn": the ARN found in the "Outputs" tab of the FisStackServerless **CloudFormation**.
  - "documentParameters": Reformatted here for legibility. For the `AutomationAssumeRole` you will need to insert the ARN of the `FisWorkshopLambdaSsmRole` either from the "Outputs" of the cloudformation stack or from the "Prerequisites" section.

```
{
    "AutomationAssumeRole":
"arn:aws:iam::ACCOUNT_ID:role/FisStackServerless-
FisWorkshopLambdaSsmRole-xxxxyyyyzzzz",
    "FaultParameterValue": "{
        \"is_enabled\":false,
        \"fault_type\":\"exception\",
        \"delay\":400,
        \"error_code\":404,
        \"exception_msg\":\"Fault injected by chaos-lambda\",
        \"rate\":1
    }"
}
```

  - "maxDuration": `1` "Minutes"

Target selection

Because we are exclusively using SSM Automation documents, we don't need to specify any targets.

Creating template without stop conditions

Select **"Create experiment template"** and confirm that you wish to create a template without stop conditions.

Validation procedure

As part of the workshop setup, we've created a "Hello World" lambda function instrumented with chaos_lambda - see in **GitHub**.

We will validate our experiment by using curl in **CloudShell**. To help us focus on only the response message, status code, and duration, we have created a convenient test script that will run in a loop querying the API:

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

cd ~/environment/aws-fault-injection-simulator-
workshop/resources/templates/serverless
./test.sh ${SERVERLESS_URL}
```

We should see output similar to this updating about once per second:

```
...
Hello from Lambda! - 200 - 0.134764
Hello from Lambda! - 200 - 0.135114
Hello from Lambda! - 200 - 0.105795
...
```

The output shows us the response from the Lambda function `Hello from Lambda!`, the status code `200` and the response time. Note the average response time as we will inject about 400ms of additional latency as part of the experiment.

Run serverless failure injection experiment

Note

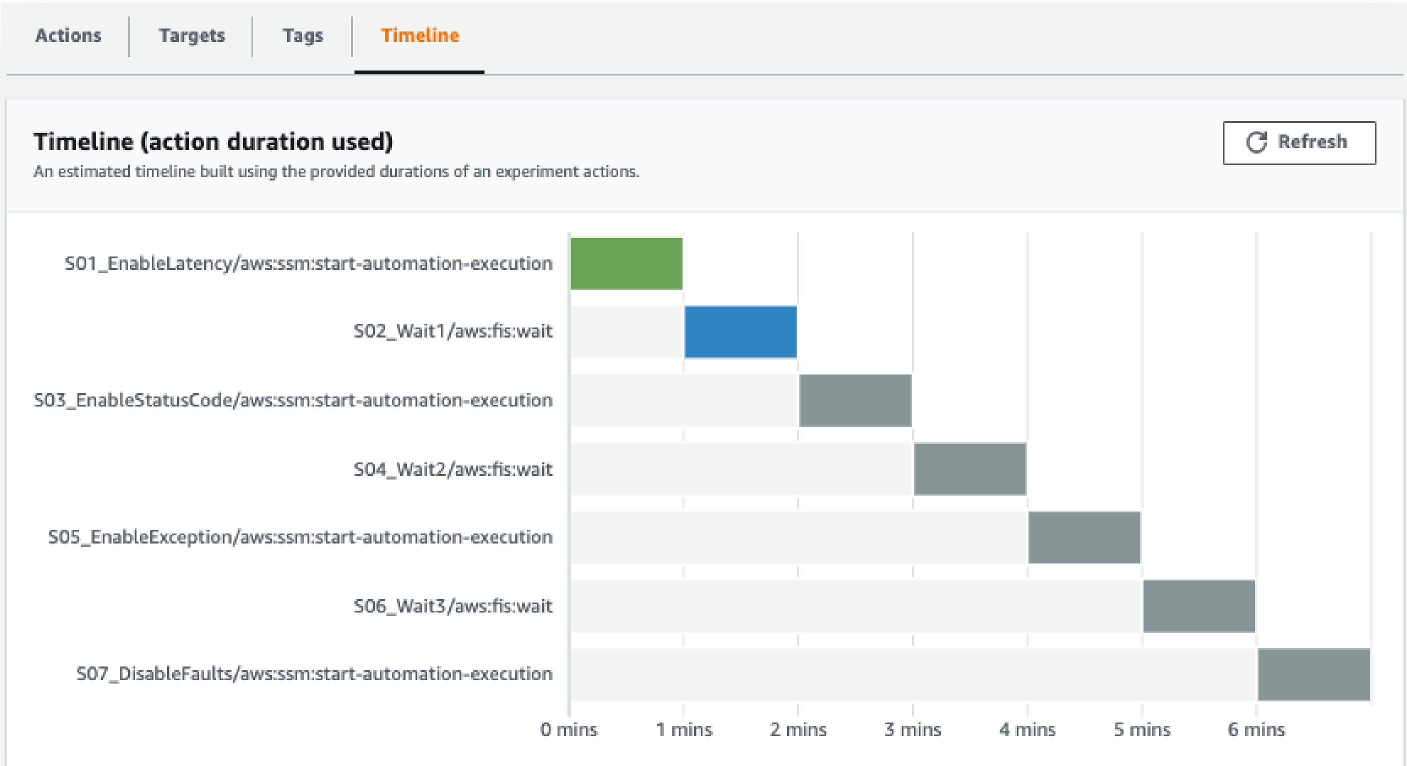
We are assuming that you know how to set up a basic FIS experiment and will focus on things specific to this experiment. If you need a refresher see the previous **First Experiment** section.

Keep the CloudShell session running with the curl generating new information about once per second. In a new browser window navigate to the **AWS Fault Injection Simulator Console** and start the experiment:

- use the `FisWorkshopLambdaFailure` template
- add a `Name` tag of `FisWorkshopLambdaFailure1`
- confirm that you want to start the experiment
- ensure that the “State” is `Running`

› Troubleshooting

In the FIS window select the "Timeline" tab and hit refresh every minute or so. You should see the experiment progressing through the individual states with green indicating finished steps, blue indicating in-progress steps, and grey signifying steps yet to be started:



At the same time, watch the curl output in the CloudShell window. As the experiment transitions from one step to the next you should see the output change, first to increase the latency:

```
...
Hello from Lambda! - 200 - 0.541716
Hello from Lambda! - 200 - 0.513623
Hello from Lambda! - 200 - 0.546924
...
```

then for the latency to return to normal but the response code changing to 404:

```
...
Hello from Lambda! - 404 - 0.113380
Hello from Lambda! - 404 - 0.274236
Hello from Lambda! - 404 - 0.136305
...
```

and finally changing to an error message indicating an exception has occurred during code execution:

```
...
{"message": "Internal server error"} - 502 - 0.215665
{"message": "Internal server error"} - 502 - 0.113820
{"message": "Internal server error"} - 502 - 0.163391
...
```

before returning to normal at the end of the experiment.

Congratulations for completing this lab! In this lab, you walked through running a multi-step experiment, changed an SSM Parameter Store parameter, and injected faults into a Lambda function.

Learning and improving

The setup we've shown here provides failure modes similar to those available for instances and containers. For teaching purposes it also has various problems that you can experiment with and use for ideation on how to customize your own serverless fault injection libraries:

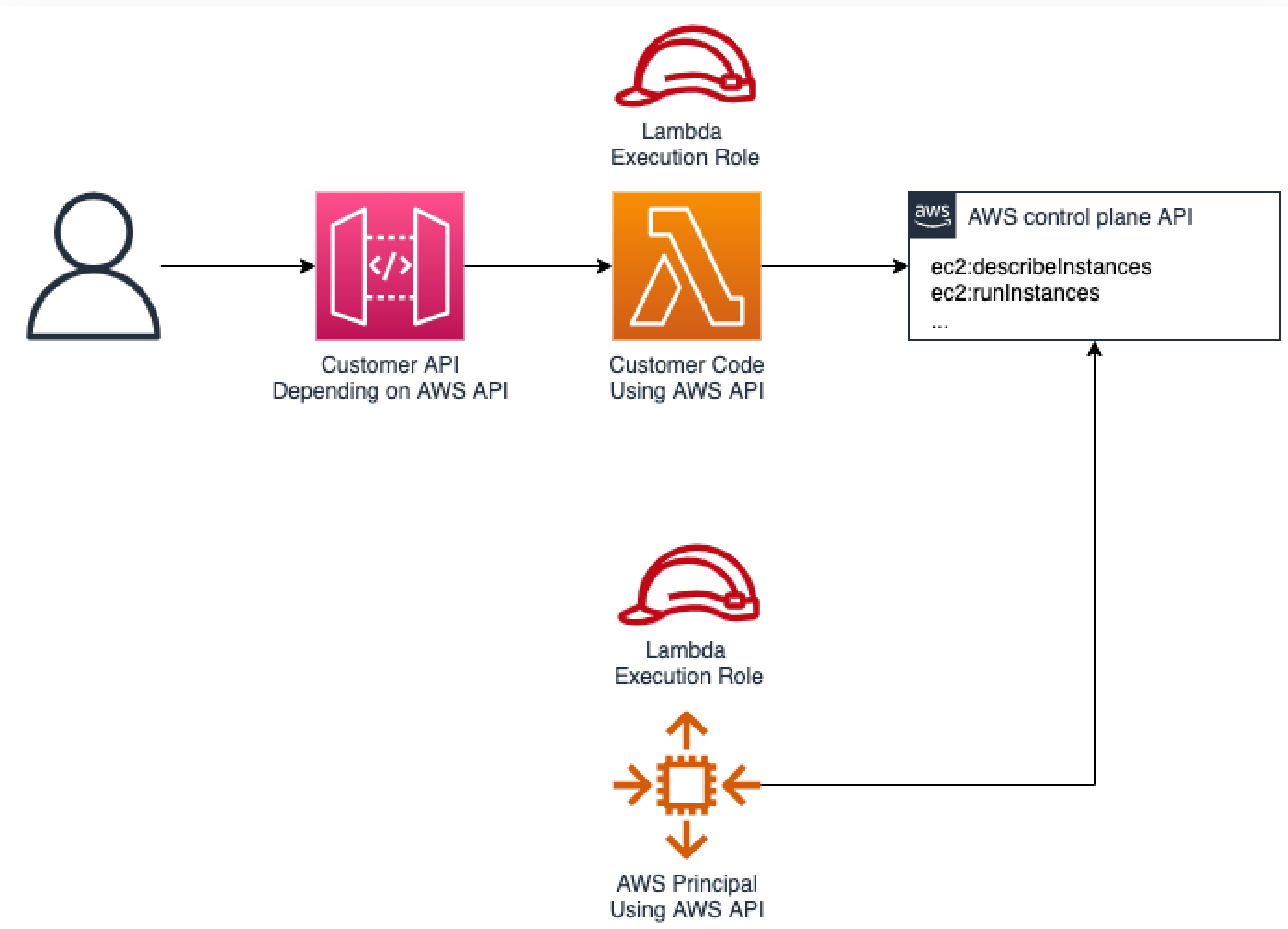
- **Parameter resets** - In the example above, we are using FIS to control the parameter value in two separate steps rather than setting / un-setting the parameter using a single long-running SSM document. Therefore, if you stop the FIS experiment prematurely, the parameter will not be reset to a non-impacting configuration. To address this you could add a `RollbackValue` parameter to the SSM document / FIS template and add an `onError` / `onCancel` path to the SSM document as shown in the aws-fis-templates-cdk GitHub examples here and here. You could even read the parameter at the start of the SSM document run, but please consider concurrency implications if another experiment is also changing the parameter.
- **Order of events** - If you are simulating a failure, do you still want the Lambda code to run or not? There is no single correct answer to this question, as it may depend on your business logic. At the time of writing, if you use the `status-code` error, the Lambda code still executes but reports a failure when no failure occurred. Similarly, in the current implementation an exception occurs before executing user code but could be moved to occur after user code. As you create your own versions, ask yourself: what impact would the mismatch between code execution and error reporting have on error handling in downstream systems?
- **Rate limiting** - As we saw in the **First Experiment**, small differences like terminating 50% vs. terminating 1 of an assumed 2 instances can lead to substantially different outcomes. Similarly the pattern of failures in consecutive invocations may matter to your experiment. E.g., sometimes you may want to affect all invocations for the duration of the fault, sometimes you may want to affect up to a certain number of

invocations per time unit, and sometimes you may want to affect a certain fraction of invocations. Sometimes you may prefer deterministic outcomes, sometimes you may prefer heuristic outcomes. As you create your own scenarios, you can review the heuristic implementation in **chaos_lambda**.

API FAILURES

Cloud infrastructure is controlled by “control plane” APIs. These APIs can be used to query existing infrastructure, e.g. to list all the EC2 instances running in a region. These APIs can also be used to create new infrastructure or modify infrastructure configurations, e.g. an autoscaling group adding or removing instances in response to load.

AWS achieves very high availability for control plane APIs but as Dr. Werner Vogels reminds us “Everything fails all the time” and our code needs to engineer for resilience against possible failures. In order to ensure that our resilience measures are effective, AWS Fault Injection Service (FIS) allows simulating failures by narrowly targeting individual execution roles. For this module we will be deploying an Amazon API Gateway integrated with a Lambda function. Within the Lambda function, we will be using the DescribeInstances action for the EC2 service API to demonstrate how API failures can impact integrated applications.



FIS provides three error scenarios:

- API is partially unavailable (intermittent failures due to throttling)
- API is fully unavailable (all API calls fail)
- API returns an error message on invocation

In this section we will demonstrate API throttling and unavailability by using FIS to to inject failures into AWS API calls by targeting an IAM role and the associated resources that leverage that role for permissions.

Note

Only EC2 Service Actions are supported at this time.

API THROTTLING

Experiment idea

AWS **throttles** API requests for each AWS account on a per-region basis. Amazon does this to help ensure the performance of all services, and to ensure fair usage for all AWS customers.

As an AWS account grows in resources and usage, API usage is likely to grow as well, potentially exceeding quotas. As such, handling API throttling events is an important design consideration as you build applications that rely on the availablility of AWS APIs.

AWS developers considered this when building their SDKs. Each AWS SDK implements automatic **retry logic**. Our experiment looks as follows:

- **Given**: We are using AWS SDKs in a serverless application
- **Hypothesis**: The SDK will manage AWS API retries during API throttling conditions and eventually (in time for dependent services) return a successful response.

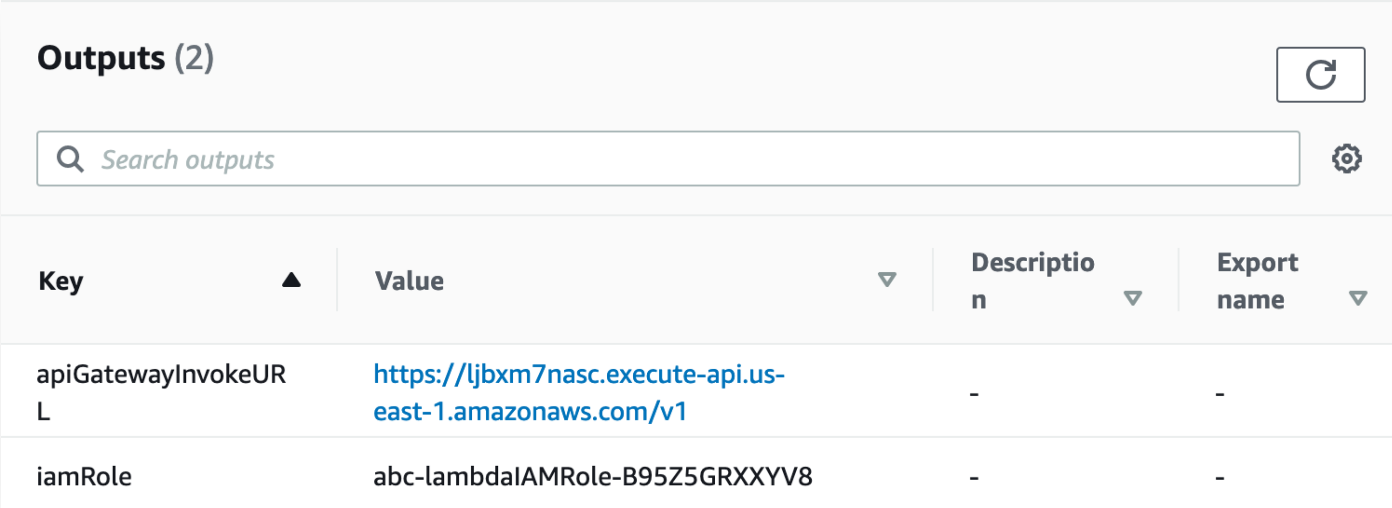
Environment setup

We will be using a simple serverless application that returns the response of an `ec2:DescribeInstances` API call. A Lambda function will run our serverless application and an API Gateway will be used to proxy the request from the client to the Lambda function.

CloudFormation resources

As part of resource setup this workshop created the required resources using the `api-throttling.yaml` file in the **GitHub repo**.

Navigate to the **CloudFormation console** and in the stack outputs note the values of `apiGatewayInvokeUrl` and `iamRole`.


Outputs (2)

Search outputs

Key	Value	Description	Export name
apiGatewayInvokeURL	https://ljbxm7nasc.execute-api.us-east-1.amazonaws.com/v1	-	-
iamRole	abc-lambdaIAMRole-B95Z5GRXXYV8	-	-

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

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

We will be creating a new experiment template in FIS

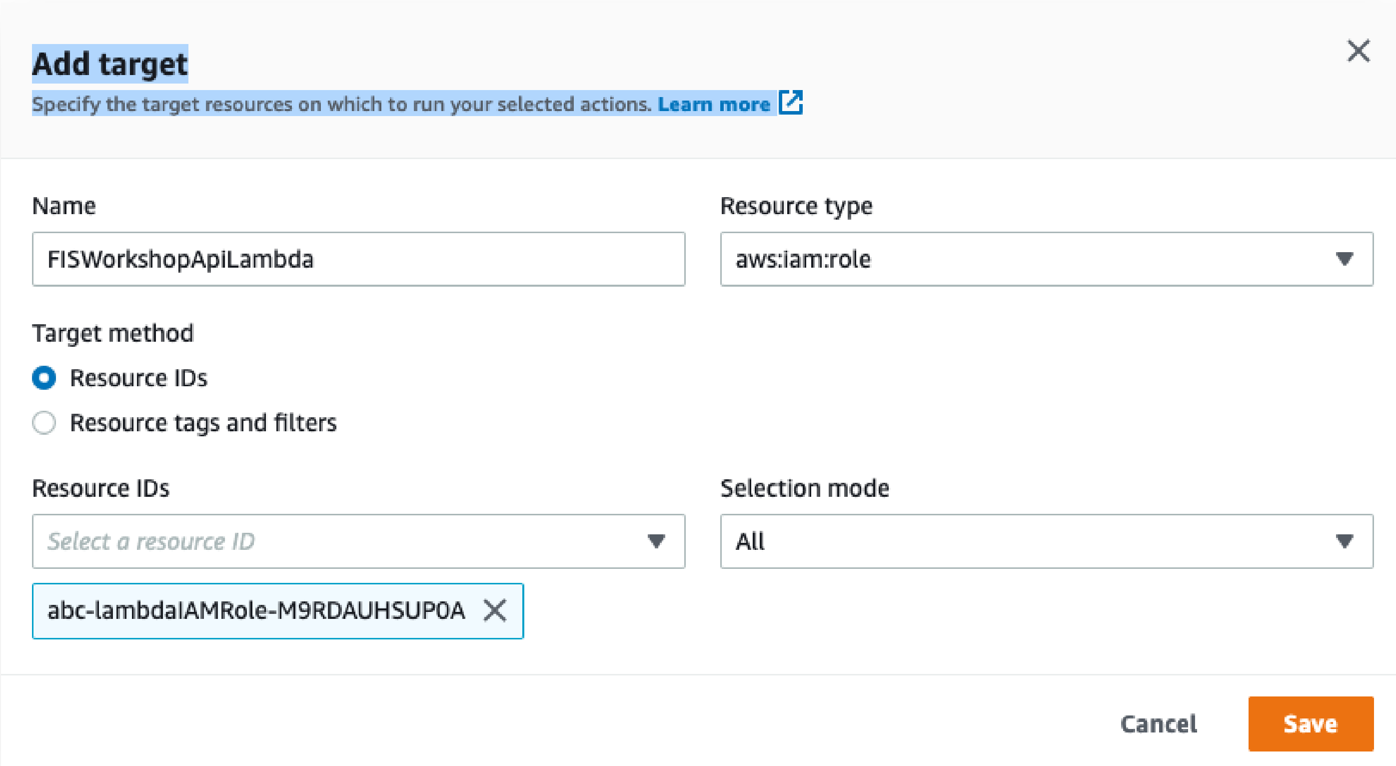
- Create a new experiment template
  - Add “Name” tag of `FisWorkshopApiThrottle`
  - Add “Description” of `ApiThrottling`
  - Select `FisWorkshopServiceRole` as “execution role”

Target Selection

In the target selection, click add target.

Inside the target modal, enter `FISWorkshopApiLambda` for “Name” and select `aws:iam:role` for “Resource type”. The “Target method” should be left as `Resource IDs` and then enter the role value that you obtained from the Cloudformation stack output.

The selection mode should read as “All”. When done select “Save”.



Action definition

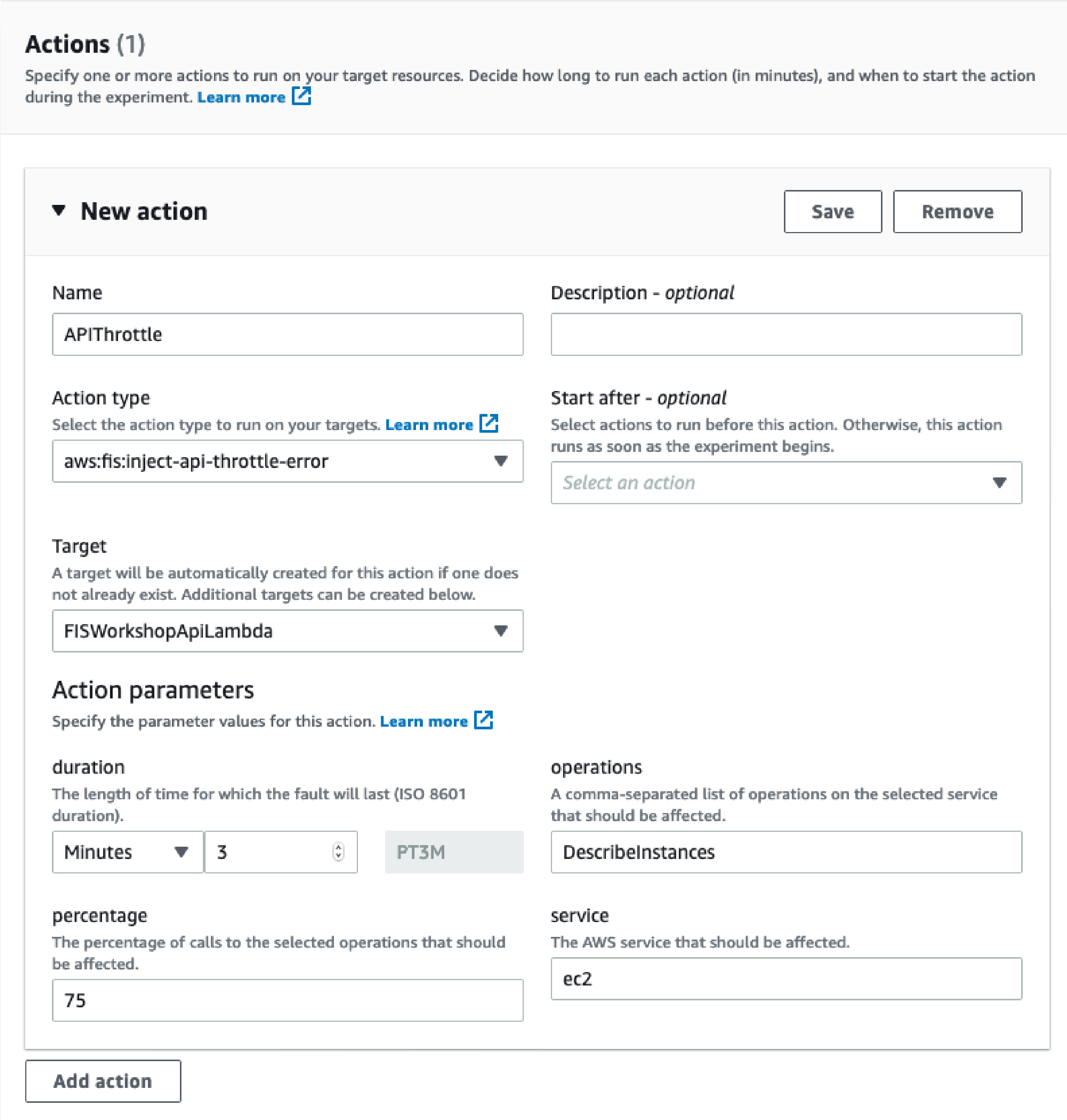
With a target defined we need define the action to take. Scroll to the “Actions” section and select “Add Action”

Type `APIThrottle` for" Name". For “Action type”, select `aws:fis:inject-api-throttle-error`. Select the target you created in the previous section. It should read `FISWorkshopApiLambda`.

In the Action parameters section set the following fields:

- duration: `Minutes 3`
- operations: `DescribeInstances`
- percentage: `75`
- service: `ec2`

Hit “Save” and then select “Create experiment template”.



Creating template without stop conditions

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

Validation procedure

Before we validate our hypothesis, we need to understand what normal state is. We have read that the SDK automatically handles retries, but what impact will adding throttling to our environment have and how will we be able to measure that impact?

For that, we will be using Curl to make a request to the url endpoint that was created during the environment setup. Use the `apiGatewayInvokeUrl` you noted from the CloudFormation stack outputs earlier, e.g.:

```
THROTTLE_URL=[replace with apiGatewayInvokeUrl]
curl ${THROTTLE_URL}
```

You will see a result that looks similar to:

`{'InstanceIds': ['i-036173389128de59b'], 'RetryAttempts': 0}`

Note

The list of Ids will be different in your environment depending on how many ec2 instances are running.

`RetryAttempts` are hopefully at 0 in your test. Any number above 0 indicates that the API call received an error response.

For reference, the relavant part of our application that we are testing reads:

```
import boto3

ec2 = boto3.client('ec2')

def describe_instances():
  resp = ec2.describe_instances(
    Filters=[{
              'Name': 'instance-state-name',
              'Values': ['running']
    }]
  )

  instance_ids = [ i['Instances'][0].get('InstanceId') for i in
resp['Reservations']]

  return {
    "InstanceIds": instance_ids,
    "RetryAttempts": resp['ResponseMetadata'].get('RetryAttempts')
  }
```

Run FIS experiment

Start the experiment

Within FIS

- Select the `FisWorkshopApiThrottle` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopThrottleRun1`
- Confirm that you want to start an experiment

Going back to the curl command, lets go ahead and fire off another request. Is the `RetryAttempts` value still at 0? Remember that we set throttling to *75%* in our experiment template so it is possible that the response was the same as the previous attempt. Lets run several more requests to see if we notice any difference when we increase the volume of traffic.

```
for i in {1..10}
do
  curl ${THROTTLE_URL}
done
```

Did you see a failure message or an increase in retries? Did you notice any difference in response times?

Learning and Improving

In this scenario, a Lambda function is using the AWS Boto3 SDK to integrate with the EC2 DescribeInstances API. By default, it will retry an API call 5 times before raising the error. You can reference Boto3 documentantion for complete details.

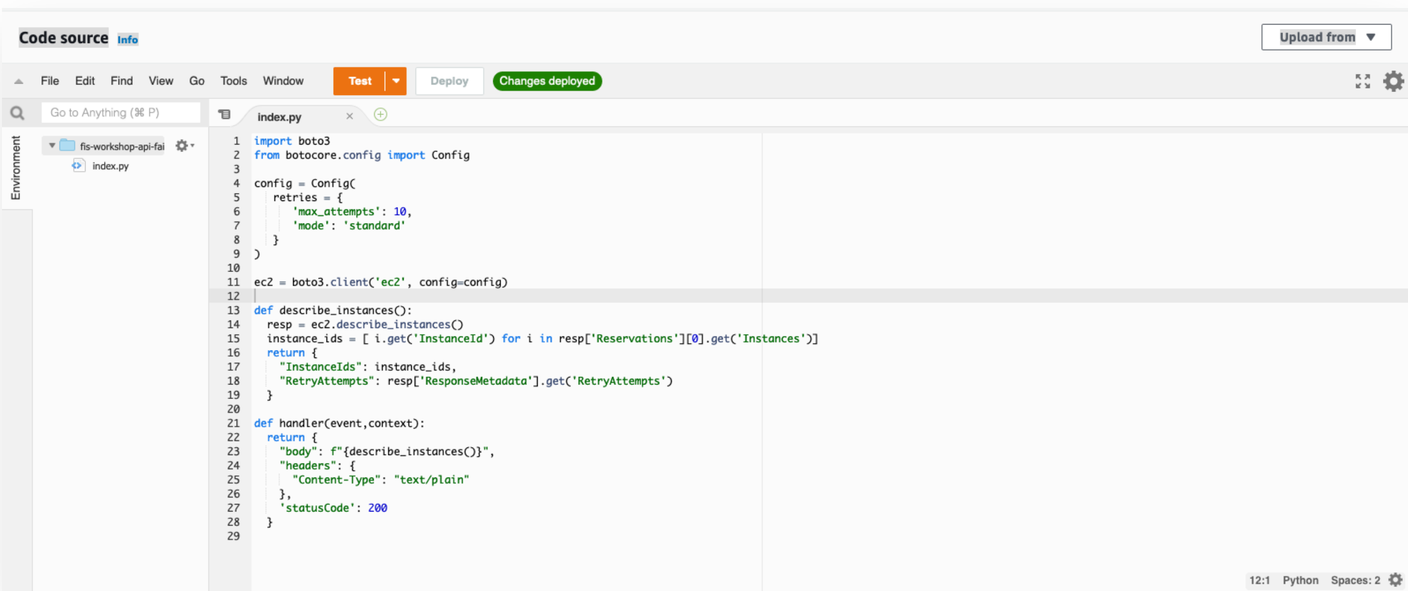
Remember we set our experiment to throttle at a rate of `75%`? From our curl calls, not all requests failed, but its likely you had at least 1 error. During times of high volume many more requests would have failed. To address these failures, we are going to increase the amount of retries to raise our chance of success.

Open up the **Lambda console**. Navigate to the `fis-workshop-api-errors-throttling` function and browse to the “Code source” section. We will use the embedded editor to update our code.

Add the following block under the import boto3 line. Be sure to remove the existing ec2 variable declaration.

```python
from botocore.config import Config

config = Config(
   retries = {
      'max_attempts': 10,
      'mode': 'standard'
   }
)

ec2 = boto3.client('ec2', config=config)
```

Your final function should look like:



Click the “Deploy” button above the editor

Re-run experiment

Back in the FIS console, start a new experiment from the same template as earlier. Tag this experiment with “Name” `FisWorkshopThrottleRun2` and start the experiment.

Re-run the same loop curl command. Do you see retry counts >= 5? Did you receive any errors or timeouts?

Conclusion

Even after updating our configuration to retry up to 10 times, we likely still saw at least 1 error from our multiple requests to our endpoint. Why did this happen? Extending our retries in our library only considers the integration between our application code and the AWS EC2 api. It does not account for the other pieces of our architecture that may be impacted up increasing the retry account. In this example, we also need to consider the maximum time our Lambda function is configured to run (30 seconds), and the hard limit our API Gateway requires a response from our Lambda function (30 seconds). By increasing the retry count, we also increased the time it would take for the AWS SDK to complete the call or return a response.

This is a great example of the tradeoff of increasing retries. Sometimes it makes sense to increase this value to ensure completion of a certain action. For example, in background batch jobs where response times are not as critical, increasing retries might provide a mechanism that results in less failures during high throttling rates. In contrast, in applications that benefit from faster responses, such as synchrounous web application integrations, it might make more sense to reduce the retry count to handle failures sooner.

API UNAVAILABLE

Experiment idea

In the last module we discussed handling AWS API throtting. In that module our example showed an application that `read` data. What about a scenario that includes `writing, updating, or deleting` data. Does increasing retries apply here as well? In this module we are going to simulate unavailability of an AWS API and how that relates to mutating calls.

- **Given**: We are using AWS SDKs in a serverless application
- **Hypothesis**: The SDK will manage AWS API retries during high rates of API errors and eventually (in time for dependent services) return a successful response.

Environment setup

The same serverless application approach will be used in this module to return the same `ec2:DescribeInstances` API call. In this module, we are also adding a capability to destroy instances which represents our mutation call. We will be updating our existing CloudFormation stack to deploy additional code to our lambda function as well as updates to our Amazon API Gateway (API Gateway). We will also be creating an SQS queue and a t3.micro ec2 instance.

Note

This workshop will not impact any ec2 instances outside of this module. An IAM role is used to only allow terminate instances against the instance ID deployed by the CloudFormation stack. An additional safeguard is also included in the application's logic.

CloudFormation resources

As part of resource setup this workshop created the required resources using the `api-unavailable.yaml` file in the **GitHub repo**.

Navigate to the **CloudFormation console** and in the stack outputs note the values of `apiGatewayInvokeUrl`, `iamRole` and `InstancdeId`. The additional `InstanceId` value is the ID of an EC2 instance that was created specifically for the instance termination flow described below.

Outputs (3)

Search outputs

Key	Value	Description	Export name
apiGatewayInvokeURL	https://0hyfzy2ou1.execute-api.us-east-1.amazonaws.com/v1	-	-
iamRole	abc-lambdaIAMRole-M9RDAUHSUP0A	-	-
instanceId	i-0cd9f3d21f59e6f3d	-	-

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

This section relies on the `FisWorkshopServiceRole` role created in the **Configuring Permissions** section. You can create this role by pasting this into CloudShell: `source ~/environment/aws-fault-injection-simulator-workshop/resources/code/scripts/cheat.`

General template setup

We will be creating a new experiment template in FIS

- Create a new experiment template
  - Add "Name" tag of `FisWorkshopUnavailable`
  - Add "Description" of `ApiUnavailable`
  - Select `FisWorkshopServiceRole` as "execution role"

Target Selection

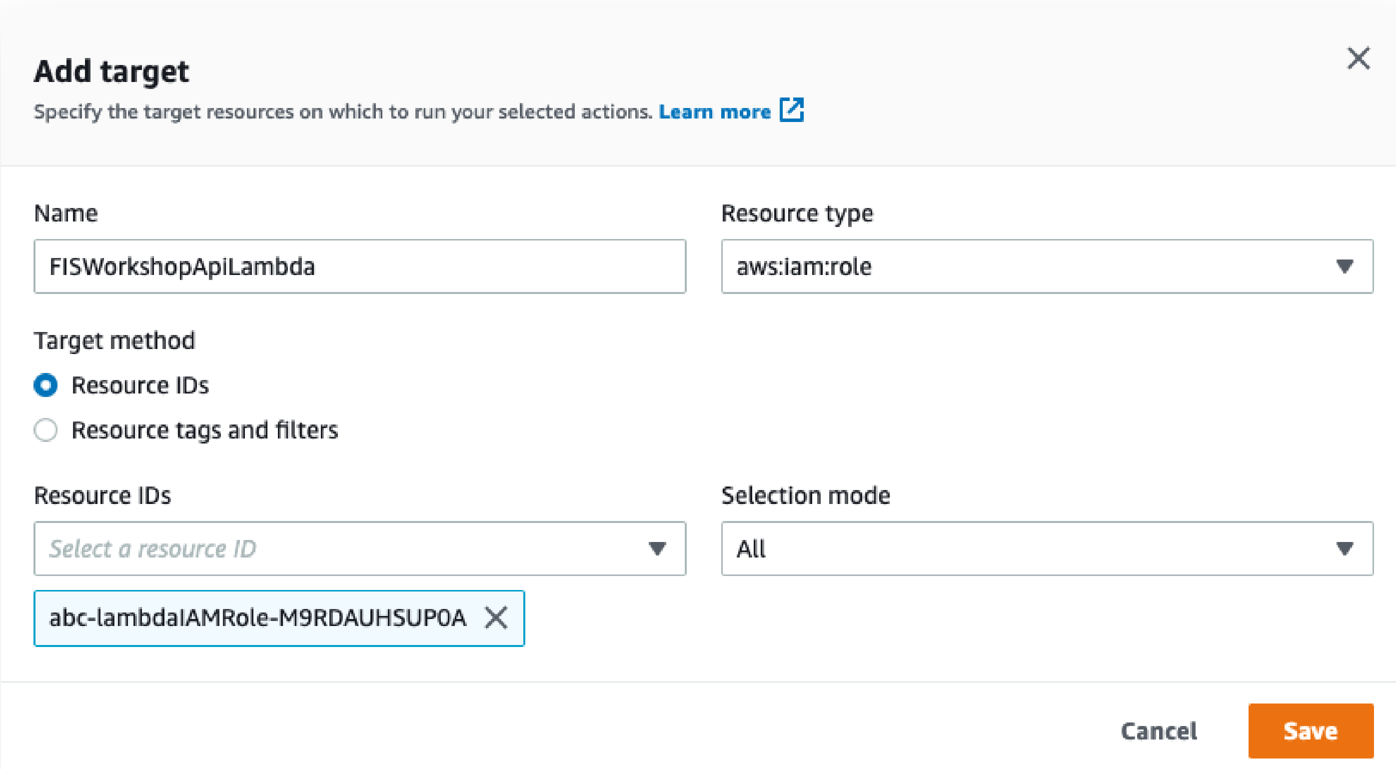
In the target selection, click add target.

Inside the target modal, enter `FISWorkshopApiLambda` for "Name" and select `aws:iam:role` for "Resource type". The "Target method" should be left as `Resource IDs` and then enter the role value that you obtained from the CloudFormation stack output.

Note

When selecting the IAM role, ensure you only add the role that includes lambda in the name

The selection mode should read as "All". When done hit "Save".



Action definition

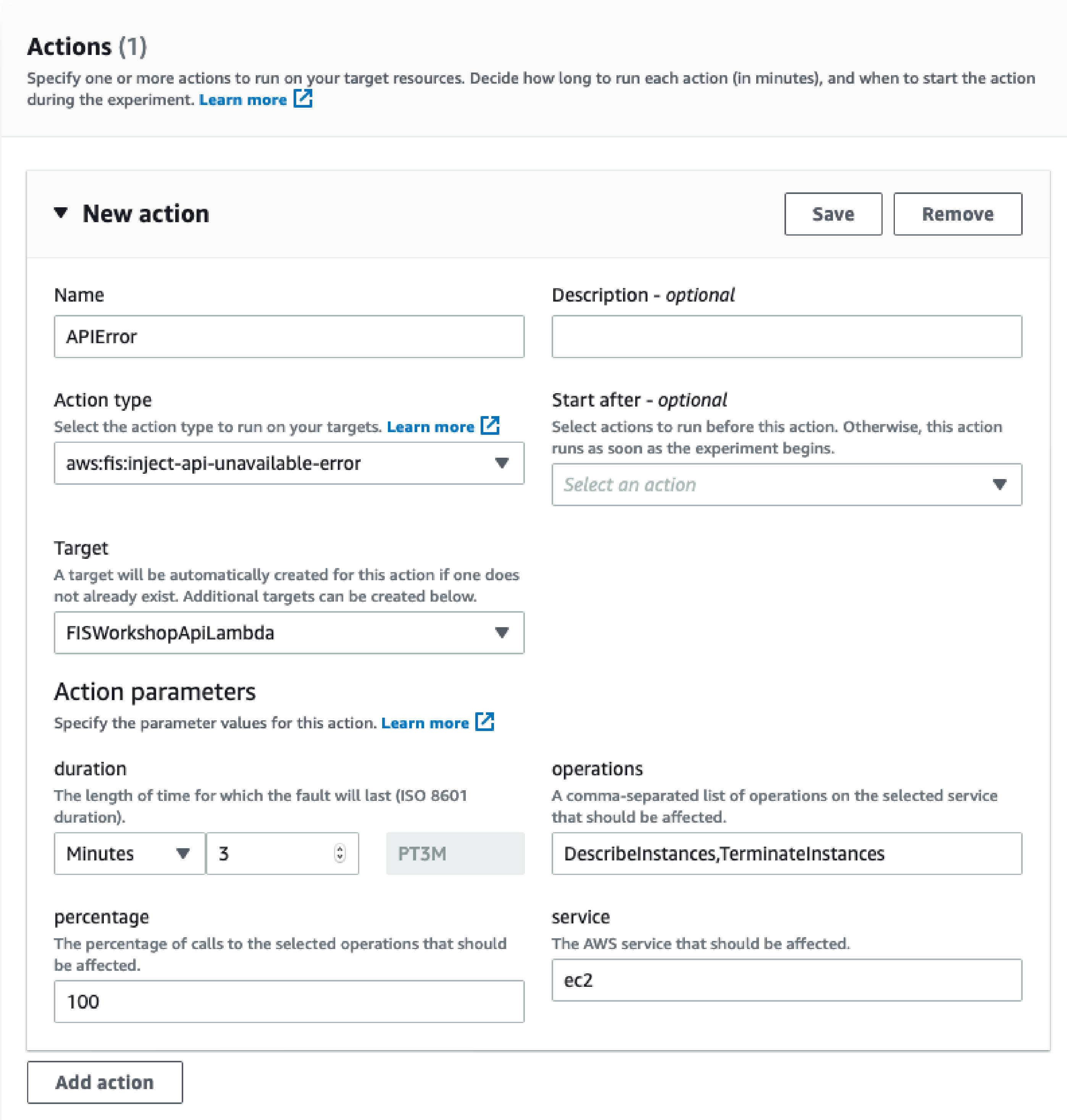
With a target defined we need define the action to take. Scroll to the "Actions" section and select "Add Action"

Type `APIError` for "Name". For "Action type", select `aws:fis:inject-api-unavailble-error`. Select the target you created in the previous section. It should read `FISWorkshopApiLambda`.

In the Action parameters section set the following fields:

- duration: `Minutes 3`
- operations: `DescribeInstances,TerminateInstances`
- percentage: `100`
- service: `ec2`

Hit “Save” and then select “Create expiriment template”.



Creating template without stop conditions

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

Validation Procedure

Just as we did in the last module, we will use curl to validate our environment prior to starting the experiment. Run the curl command with the new URL you noted from the CloudFormation stack:

```
UNAVAILABLE_URL=[replace with apiGatewayInvokeUrl]
curl ${UNAVAILABLE_URL}
```

This should result in the same response as the previous module

```
{'InstanceIds': ['i-0823fd3823e25afd3', 'i-036173389128de59b'], 'RetryAttempts':
0}
```

In the list of IDs displayed you should see the ID you noted from the outputs section of the CloudFormation stack. When we run the experiment, we will use a different endpoint that will result in a mutation call and issue a termination of this instance.

Run FIS experiment

Start the experiment

Within FIS

- Select the `FisWorkshopApiUnavailable` experiment template you created above
- Select start experiment
- Add a `Name` tag of `FisWorkshopUnavailableRun1`
- Confirm that you want to start an experiment

Instead of issuing the same request, we are going to add the **/terminate** path to our API Gateway url. This path is configured to `mock` an endpoint that will terminate the ec2 instance that was created for this experiment.

```
TERMINATION_URL=${UNAVAILABLE_URL}/terminate
curl ${TERMINATION_URL}
```

With the experiment running, we should receive an error:

`{"message": "Internal server error"}`

While the experiment runs and we continue to call the terminate endpoint, we will continue to receive this error. During service outages that result in 100% unavailability errors, all calls will fail to complete.

Learning and Improving

In situations where APIs are unreliable or you want to minimize the scope of the impact during API unavailability, you may want to consider using asynchronous patterns to process incoming requests. So far in this module, all of the testing has been using synchronous call patterns.

Asynchronous Design Patterns allow for faster client responses and the ability to limit the impact of call failures. Implementing queues and asynchronous processing of requests seperates the processing of those requests from the injestion process.

In this environment, we have added an **Amazon Simple Queueing Service** (SQS) queue to store the request for asynchronous processing. Instead of our request being sent directly to the lambda function for processing, we will have our API Gateway write directly to the SQS queue. In this pattern, the lambda function will attempt to process this request from the queue, and will continue to retry asynchronously until the mutating call completes.

In **FIS** ensure the experiment is still running. If not, start a new experiment with Name tag `FisWorkshopUnavailableRun2`.

When the experiment begins running issue a new curl request to the `/terminate` path, but this time with a `POST` action. HTTP `POST` methods are usually used for mutating actions.

```
curl -X POST curl ${TERMINATION_URL}
```

Even with the experiment running you should receive a response that looks similar to

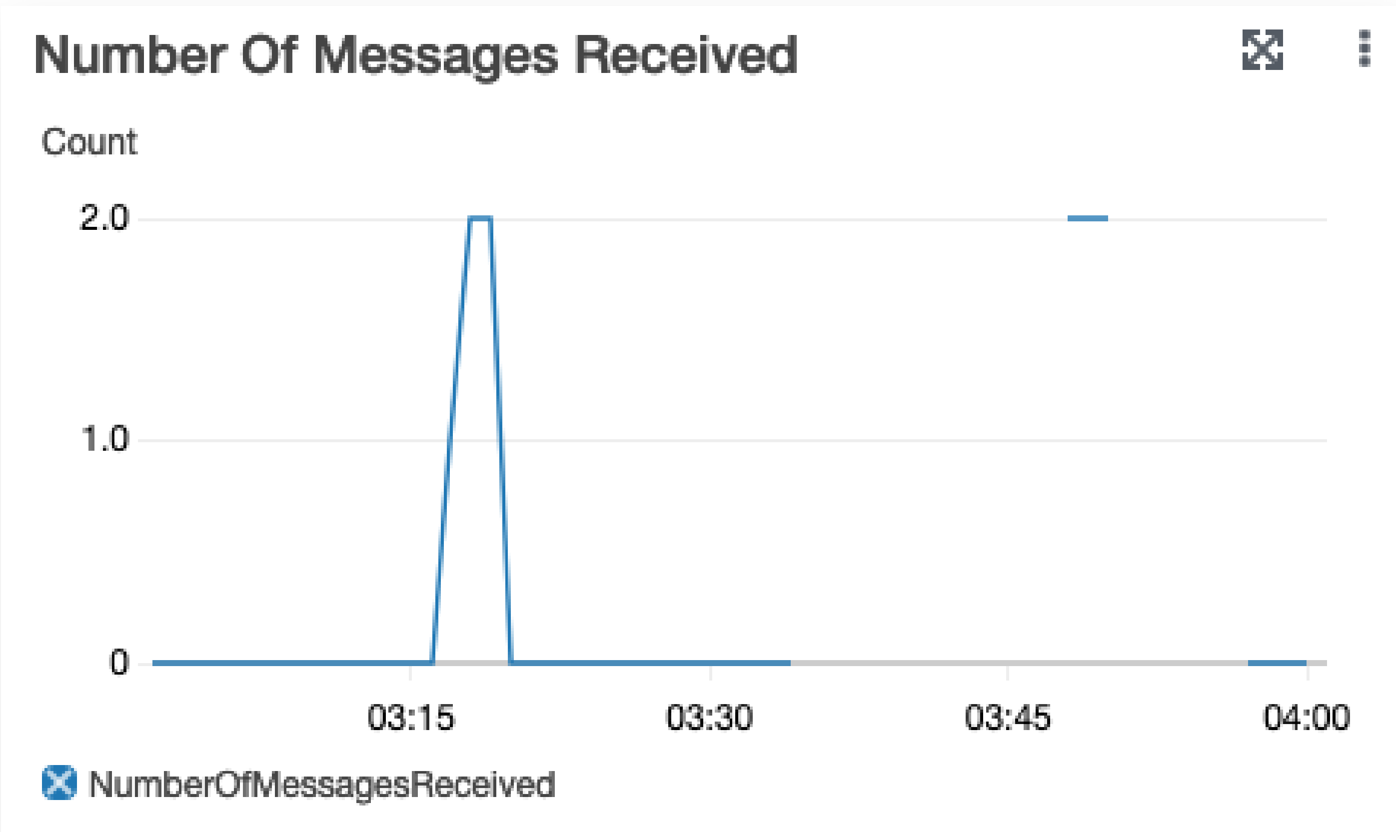
```
{"SendMessageResponse":{"ResponseMetadata":{"RequestId":"8cc4bb0a-6dbf-595b-995d-
e2ac3d9e3622"},"SendMessageResult":
```

{"MD5OfMessageAttributes":null,"MD5OfMessageBody":"74ed192a7c4e541bf34668d1e8ef0027
b5f9-48ee-8511-222144fbef01","SequenceNumber":null}}}

This response was generated from the `sqs:SendMessage` API initiated from our API Gateway and not affected by the specific EC2 throttling.

To confirm the message was sent to the queue, navigae to the sqs console and the click “Queues” > “fis-workshop-api-queue”

Under the “Monitoring” tab you should see a count in the “Number of Messages Received”



When the experiment completes after running 3 minutes, you can verify that the instance with the ID from the stack output is in the process of terminating.

Conclusion

In this module, we used an SQS queue message to ensure that the TerminateInstances API call would be retried after the fault injection to demonstrate how you can use asynchronous API patterns to mitigate API failures.

RECURRENT EXPERIMENTS - CI/CD

So far we have discussed iterating through a cycle of:

- establish baseline performance data
- develop *new* hypothesis
- run experiment
- verify hypothesis
- improve based on findings

In this section we will address use cases in which we want test and *existing* hypothesis multiple times. Common examples for this are:

- ensure the system remains resilient after changes (CI/CD)
- ensure detection and recovery continue to work (Disaster Recovery)

"Experiment" or "Test"?

A deep dive into testing terminology is outside the scope of this workshop but for readers familiar with the field we want to point out some analogies and provide some considerations:

Human-led processes

The hypothesis based cycle we've discussed up to this point is very similar to "Exploratory Testing" and "Acceptance Testing" in the sense that it steps away from purely validating that something "works as intended". Just like "Exploratory Testing" and "Acceptance Testing", the human-led fault injection process should allow for human observation to adjust the "intent".

Machine-led processes

Automating fault injection based on prior validation of a hypothesis is analogous to the wide range automated and recurrent tests such as:

- unit tests
- regression tests
- integration tests
- load tests

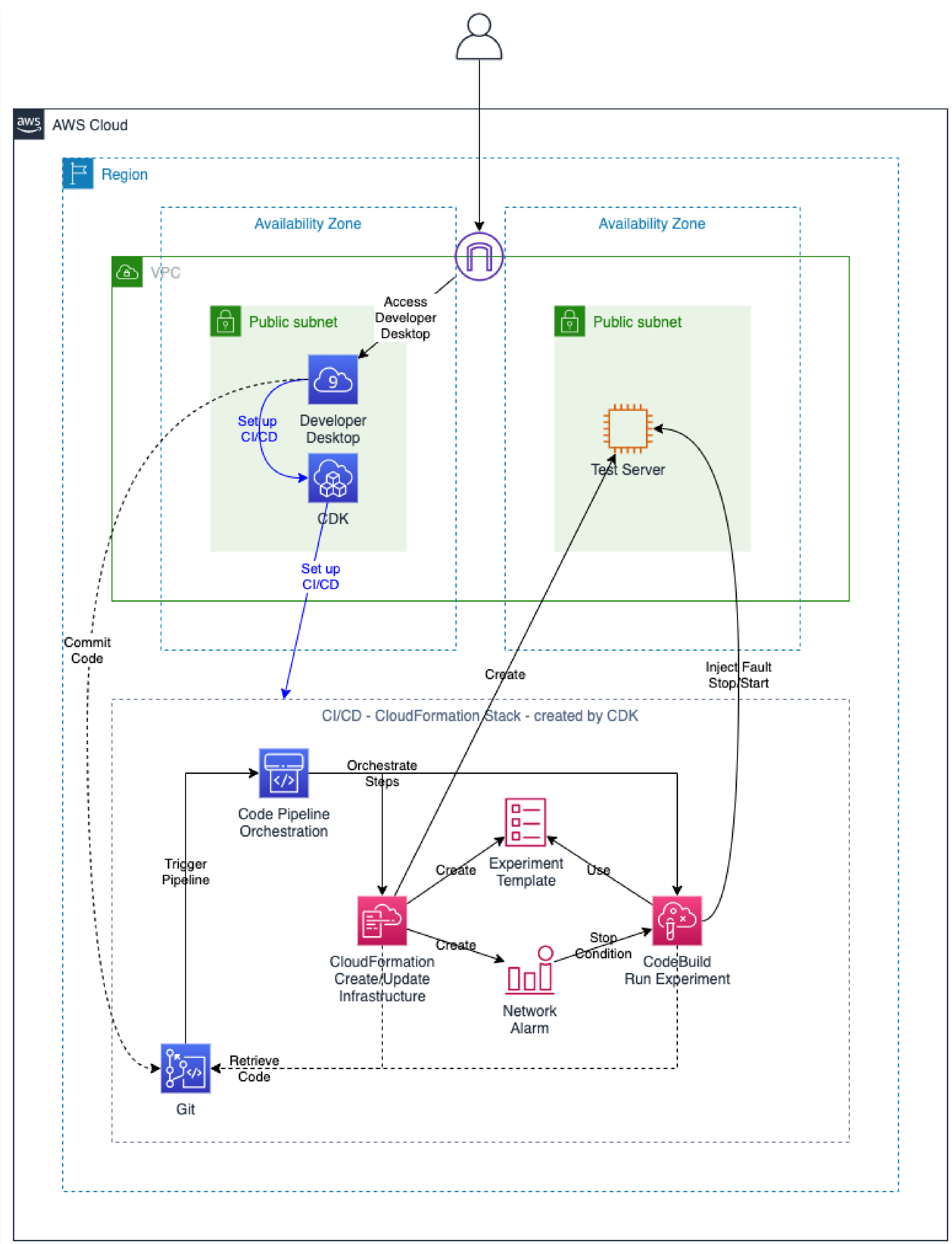
Just like for other tests, it is important to consider the scope and *duration* of recurrent fault injection experiments. Because fault injection experiments generally expose issues across a large number of linked systems they will typically require extended run times to ensure sufficient data collection. In order to not slow down developers they should be run in later stages of CI/CD pipelines.

Architecture

For demonstration purposes we have made the following choices but there are many other ways to build valuable automation:

- **CI/CD** - We focus on running experiments in a CI/CD pipeline with the argument that it's easy to slow down a pipeline to run only once a year but hard to speed up a manual process to run multiple times every day.
- **One repo** - We use a single repository to host the definition of the pipeline, the infrastructure, and fault injection template. We do this to show how one would co-version all components of a system but whether this is a good idea for you depends on your governance processes and each of the parts could easily be independent.

The setup looks like this:



In the next section we will:

- create a code repository and a pipeline using AWS CDK

- trigger the pipeline to instantiate sample infrastructure
- trigger the pipeline to update infrastructure and perform fault injection

SETUP

In this section, we will integrate an AWS Fault Injection Simulator experiment with a CI/CD pipeline.

Create The Pipeline

We will use the AWS CDK to provision our CI/CD pipeline.

If you have not done so yet, in your Cloud9 terminal clone the repository for the workshop.

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

Next change directory into the CI/CD CDK project and restore the npm packages used for the pipeline.

```
cd ~/environment/aws-fault-injection-simulator-workshop/resources/code/cdk/cicd/

# Make sure we use right npm version
sudo npm install -g npm@7

# Pull relevant npm packages
npm install
```

Finally lets deploy our stack.

```
# use local version of cdk
npx cdk deploy --require-approval never
```

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

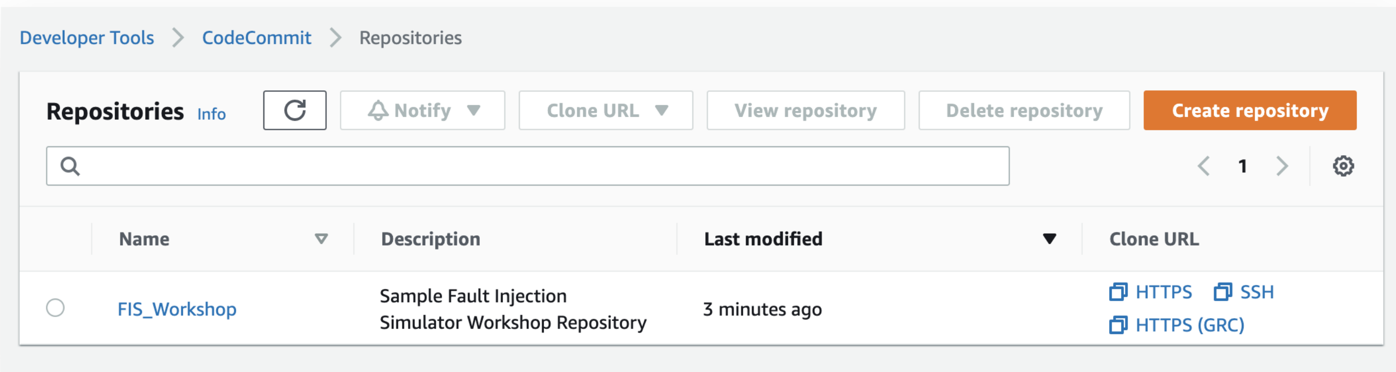
Once stack creation is complete, continue to the next section.

REVIEW THE PIPELINE

Lets review the components our previous section created.

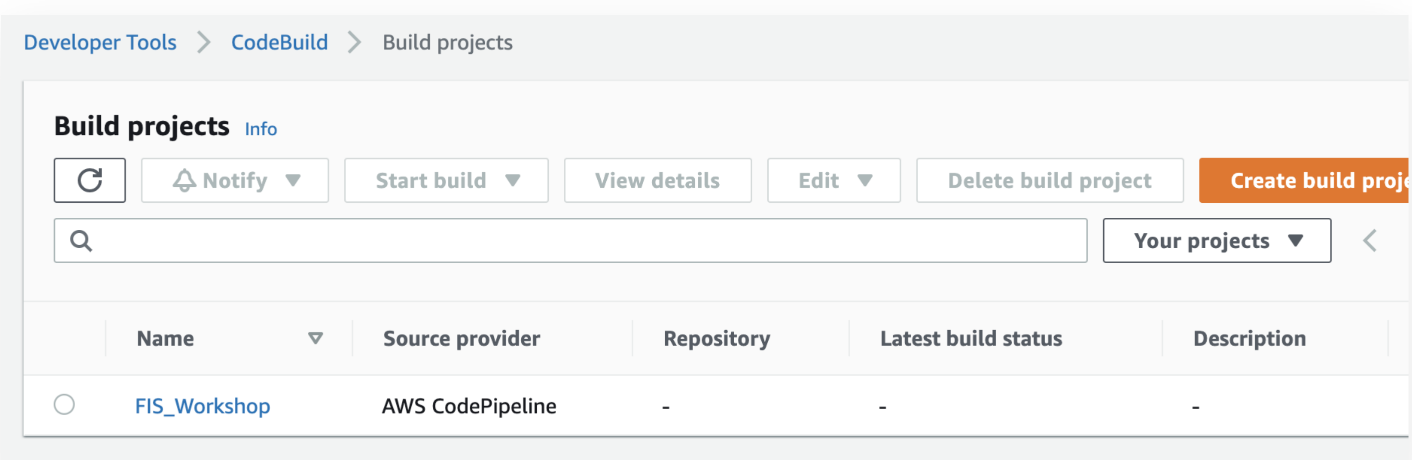
CodeCommit

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



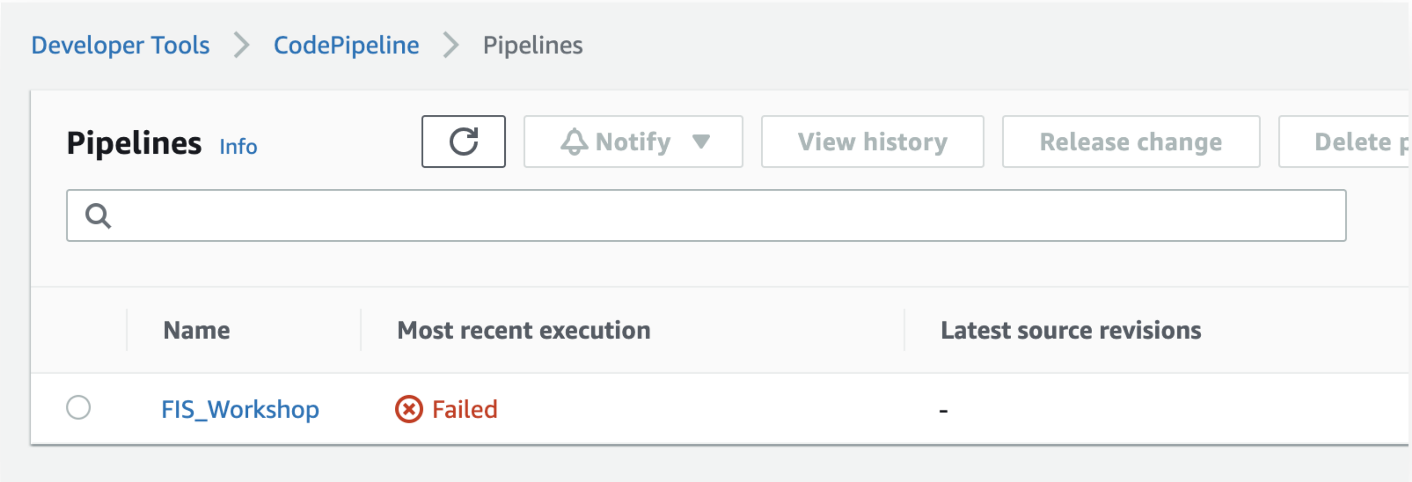
CodeBuild

Open the **AWS CodeBuild Console**. **Note:** You may have to select a region at the top right. You should see the `FIS_Workshop` build project.



CodePipeline

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

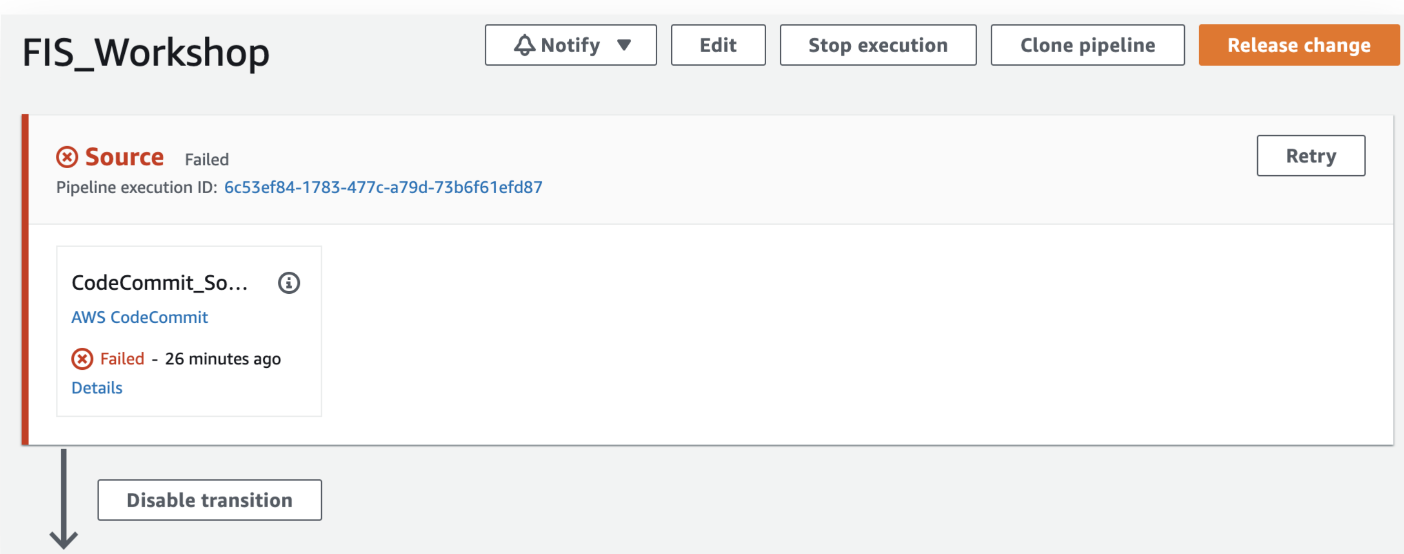


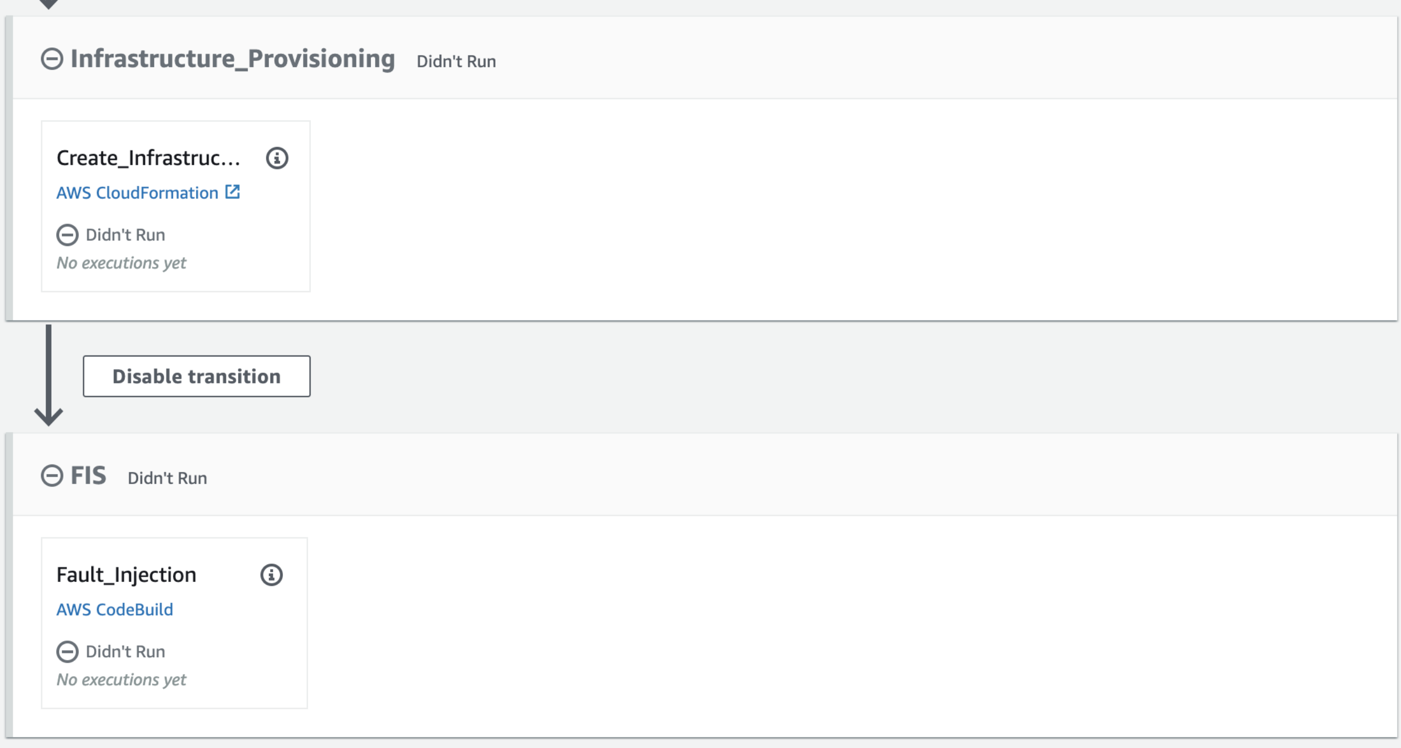
The pipeline will start in a failed state, since we have not uploaded any files to our repository.

To review, click on the pipeline name.

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 use an AWS CloudFormation template from our repo to create our test infrastructure and create our experiment templates.
3. **FIS**: This stage will use the AWS CodeBuild project to make an API call to run our experiment and monitor the results.





Continue to the next section to start the pipeline.

START THE PIPELINE

The pipeline is configured to run every time new code is committed to our AWS CodeCommit repository. To start our pipeline we need to commit files to our repository.

Update repository

Note

The instructions below are designed for use with Cloud9. On Cloud9 git access is enabled via the IAM role associated with the Cloud9 instance. If you would like to access the AWS CodeCommit repository from your local machine, review the **getting started documentation**.

Adding files to our repository is a 3-step process:

- create a local copy of our repository (`clone`)
- add or update files and save them (`add` / `commit`)
- upload them to our repo (`push`)

Clone

Open the AWS Code Commit Console. Click the `HTTPS` link next to the `FIS_Workshop` repository name to examine the clone URL.

To use IAM credentials in Cloudshell/Cloud9 we will configure the git **credential helper**:

```
git config --global credential.helper '!aws codecommit credential-helper $@'
git config --global credential.UseHttpPath true
```

In your CloudShell/Cloud9 terminal clone the repository (for convenience, the commands below show how to query the clone URL):

```
GIT_URL=$( aws codecommit get-repository --repository-name FIS_Workshop --query
"repositoryMetadata.cloneUrlHttp" --output text )
cd ~/environment
git clone ${GIT_URL}
cd FIS_Workshop
```

Add/Update

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 `add` all the files in the directory, and `commit` them as a new version with a label of `Uploading Workshop files`.

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

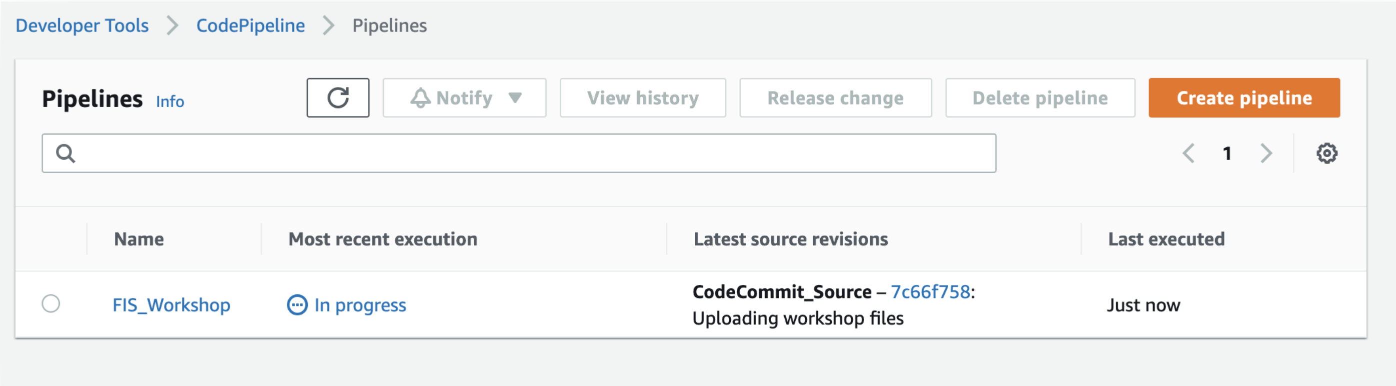
Push

Finally `push` the files to copy them to our repository and to trigger the pipeline:

```
git push -u
```

View Progress

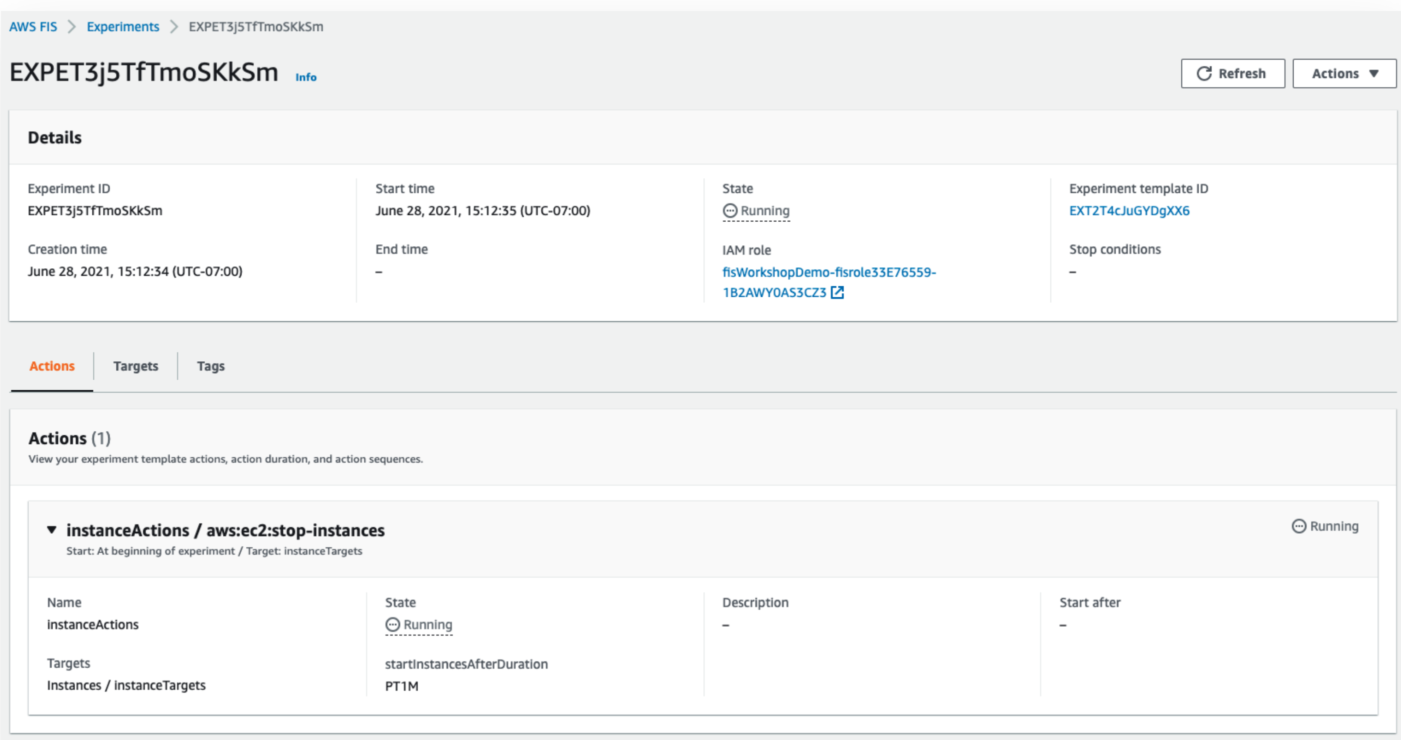
After you `push` the files, the pipeline will start. Open the **AWS CodePipeline Console**. You should now see the `FIS_Workshop` pipeline is in progress. Click on the pipeline name to view the step details.



The pipeline runs in sequence, first running the Wait for the "Infrastructure_Provisioning" step, and on success starting the "FIS" step.

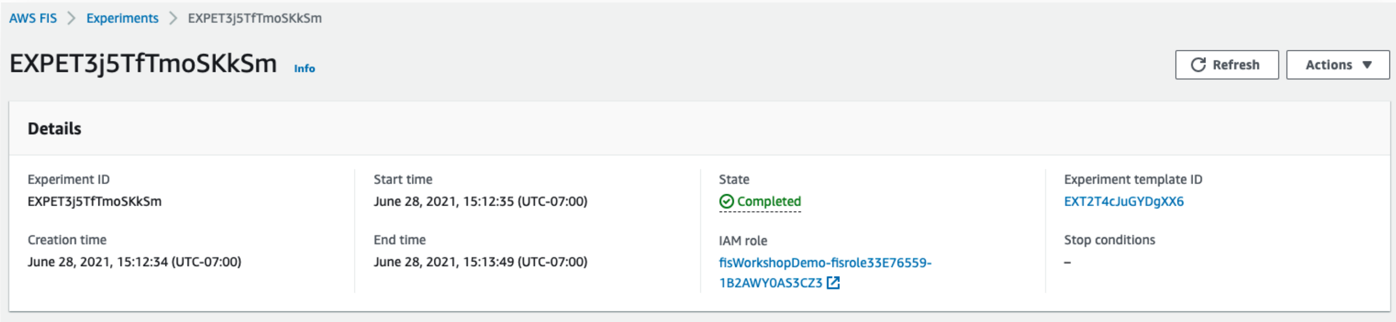
You can monitor the progress of our experiment either from the CodePipeline details page or from the AWS FIS console.

Navigate to the **FIS console**. Click on the "Experiment ID" of the running experiment. You should see the experiment in a running status:

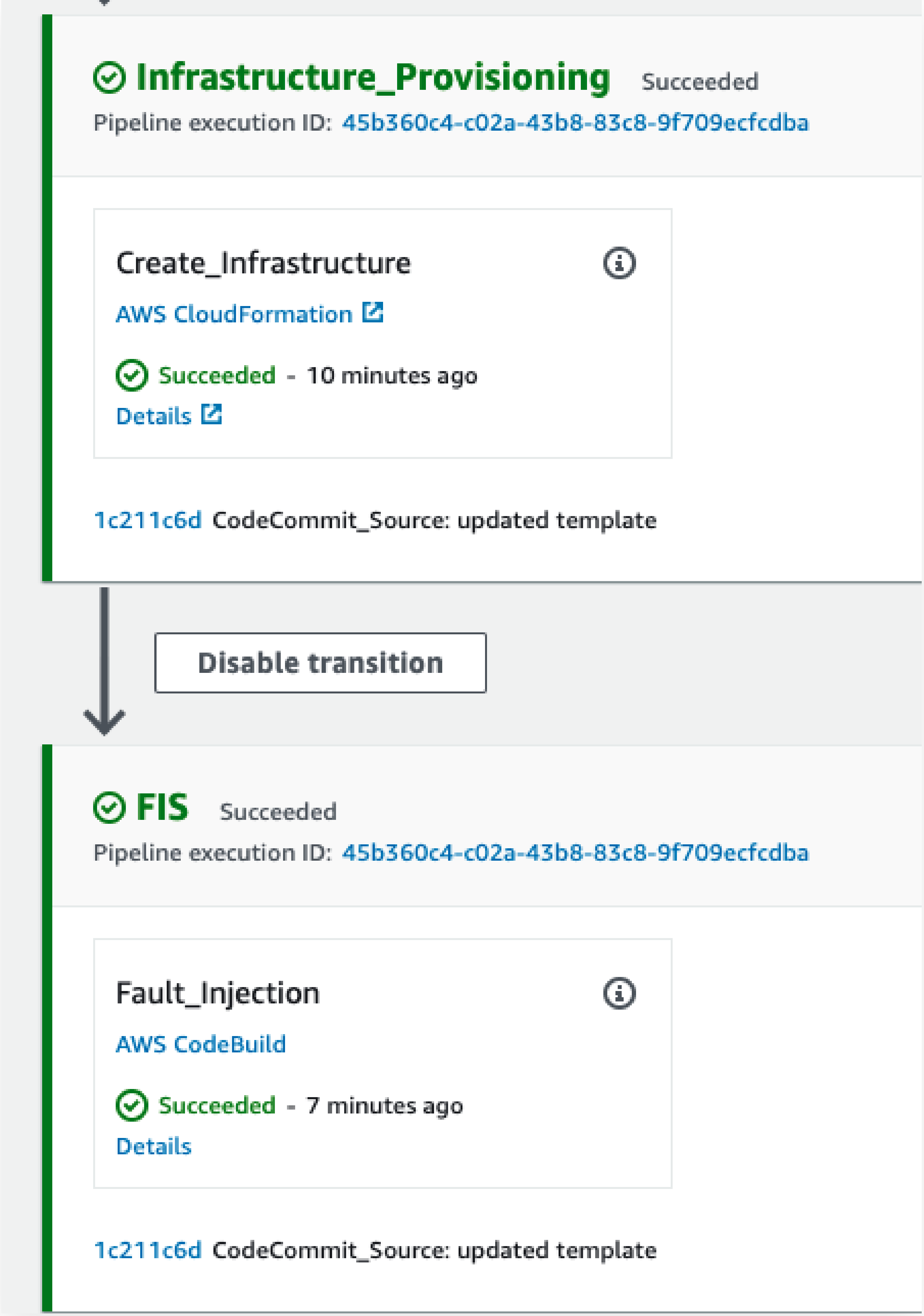


If you expand the "instanceActions / aws:ec2:stop-instance" card (as shown above) you can see that the experiment stops the test instance, waits for 1minute, then restarts the instance.

Wait a couple minutes for the instance to restart and the experiment to finish and 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) to fail.

FORCE A PIPELINE ERROR

In this section we will update the experiment template defined in our repository to contain a stop condition that will prevent or abort experiment execution if our cloudwatch alarm is in ALARM state.

We will then `push` the new revision to our repository which will trigger a new pipeline run, update our experiment template and execute our experiment template. Then while our pipeline is running, we will force an ALARM state. This will lead to a failure of the AWS FIS experiment and in turn to a failure of the pipeline.

Change the Infrastructure Template

Note

In this section we are directly updating the file in AWS CodeCommit. This is equivalent to the `add` / `commit` / `push` process that we performed in the previous section and creates a new revision. To subsequently synchronize the copy on your Cloud9 instance you would `git pull`

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 shown below to enable am AWS CloudWatch alarm as a Stop Condition.

Before:

```
      StopConditions:
        - Source: none
      #  - Source: aws:cloudwatch:alarm
      #    Value:
      #      Fn::GetAtt:
      #        - cwalarm8A77F56F
      #        - Arn
```

After:

```yaml
      StopConditions:
      #  - Source: none
        - Source: aws:cloudwatch:alarm
          Value:
            Fn::GetAtt:
              - cwalarm8A77F56F
              - Arn
```

Finally, enter your name and email at the bottom of the page and select “Commit changes”. Just like our prior `git push` this will trigger the pipeline to start.

Forcing an Error

To trigger the stop condition and simulate a 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 Cloud9 instance to force an error.

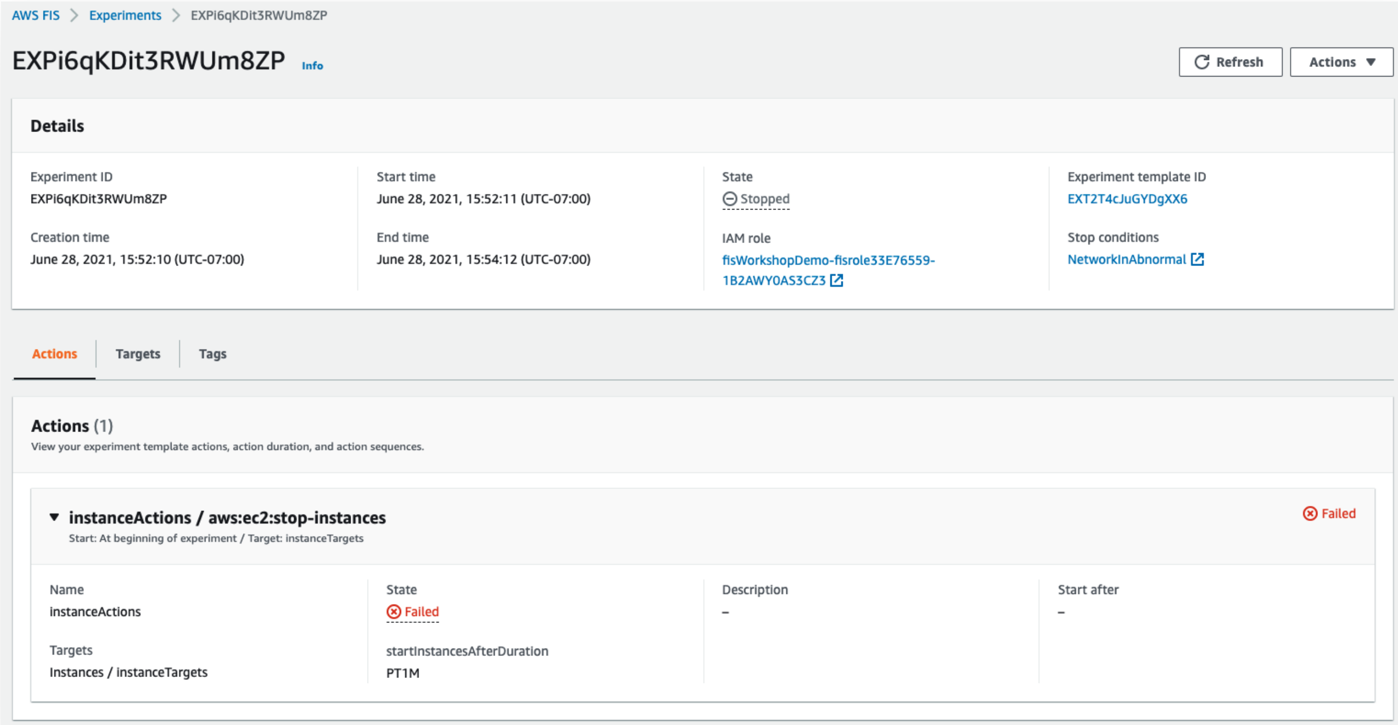
```
aws cloudwatch set-alarm-state --alarm-name "NetworkInAbnormal" --state-value
"ALARM" --state-reason "testing FIS"
```

By setting this CloudWatch alarm to an error state, this will stop a running experiment or prevent the experiment from starting.

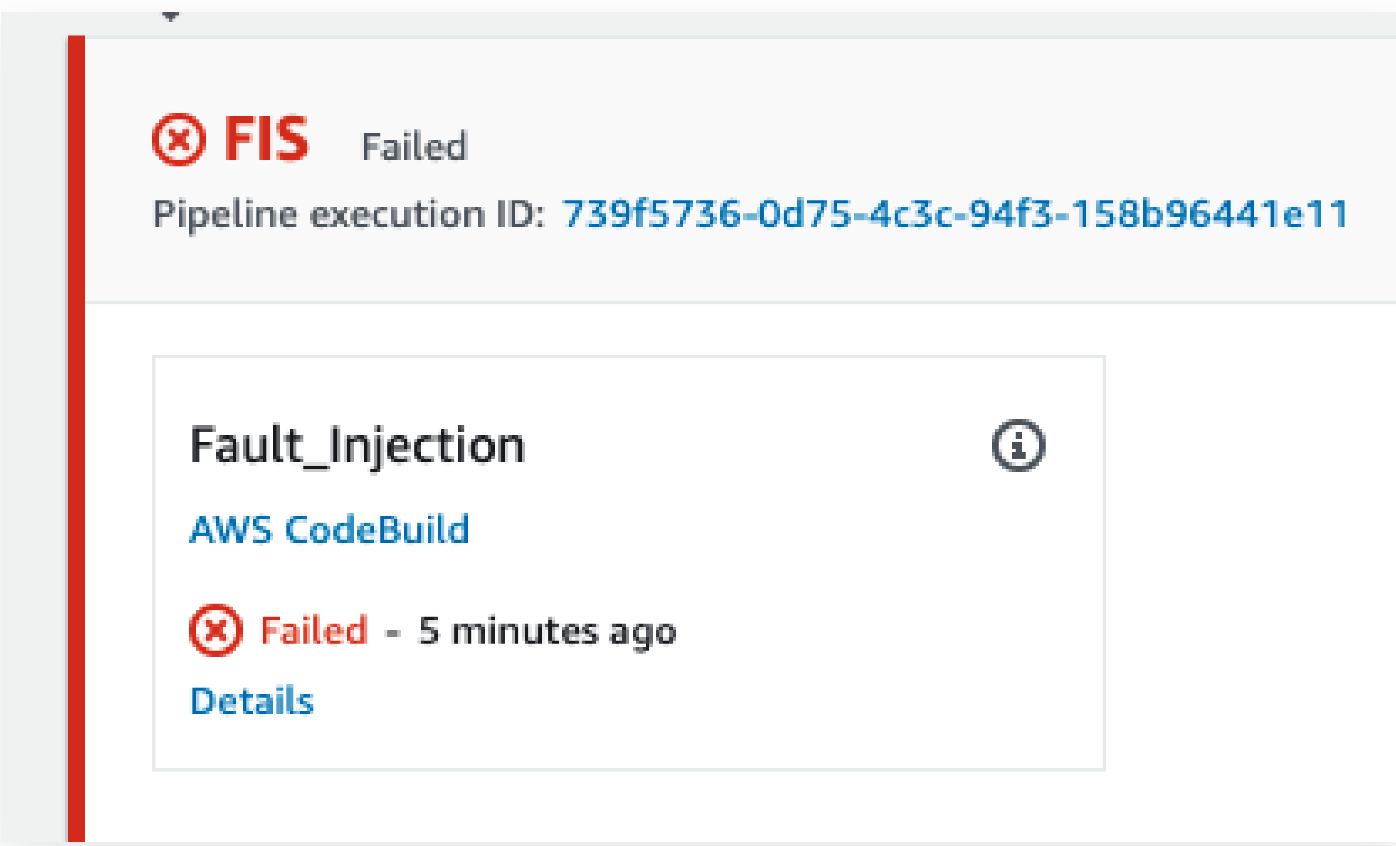
Note

We are artificially changing the alarm state. The alarm will reset to OK state after a short period of time. If you want to persist the ALARM state for longer try running the command in a loop.

To verify the Experiment was stopped, navigate to the **FIS console**. You should see that your latest experiment has failed due to the stop condition.



To verify that this resulted in a failed pipeline execution navigate back to the **AWS CodePipeline Console**. You should see that your pipeline has also failed do to the experiment stopping.



Congratulations! You have built a CI/CD pipeline, instrumented it with an AWS FIS experiment, and demonstrated both successful and failed experiment outcomes.

Next steps

From this starting point you can explore improvements like:

- **add more pipeline stages** - In our pipeline the experiment is the last step and does not gate progress. In a production scenario there might be a additional steps that would only run if the AWS FIS experiment succeeds. Try adding a pipeline stage and verify that it only runs if the experiment succeeds.
- **explore alternative ways to change the template** - in this example we are using an AWS CloudFormation template to define the experiment template as shown in the **Experiment (CloudFormation)** section. Could you store the experiment template as a separate file and update it using the CLI as show in **Experiment (CLI)** or expand the `runExperiment.py` script (see **code in GitHub**)?
- **trigger AWS CloudWatch alarm from experiment template** - AWS FIS templates allow you to run a sequence of steps, try triggering the alarm from a step in the template. Hint: you could do this via the **AWS Sytems Manager integration**/en/030_basic_content/040_ssm.html.
- **set up a real alarm**

COMMON SCENARIOS

This section covers common scenarios customers ask about.

- Targeting on-prem instances
- Simulating AZ Issues

TARGETING ON-PREM INSTANCES

Some customers use **AWS Systems Manager for hybrid environments** to manage both on-prem and cloud resources and would like to run instance-based fault injection actions against on-prem resources.

In this section we discuss how to use SSM automation (SSMA) to target on-prem instances with the same SSM runbooks used for EC2 instances.

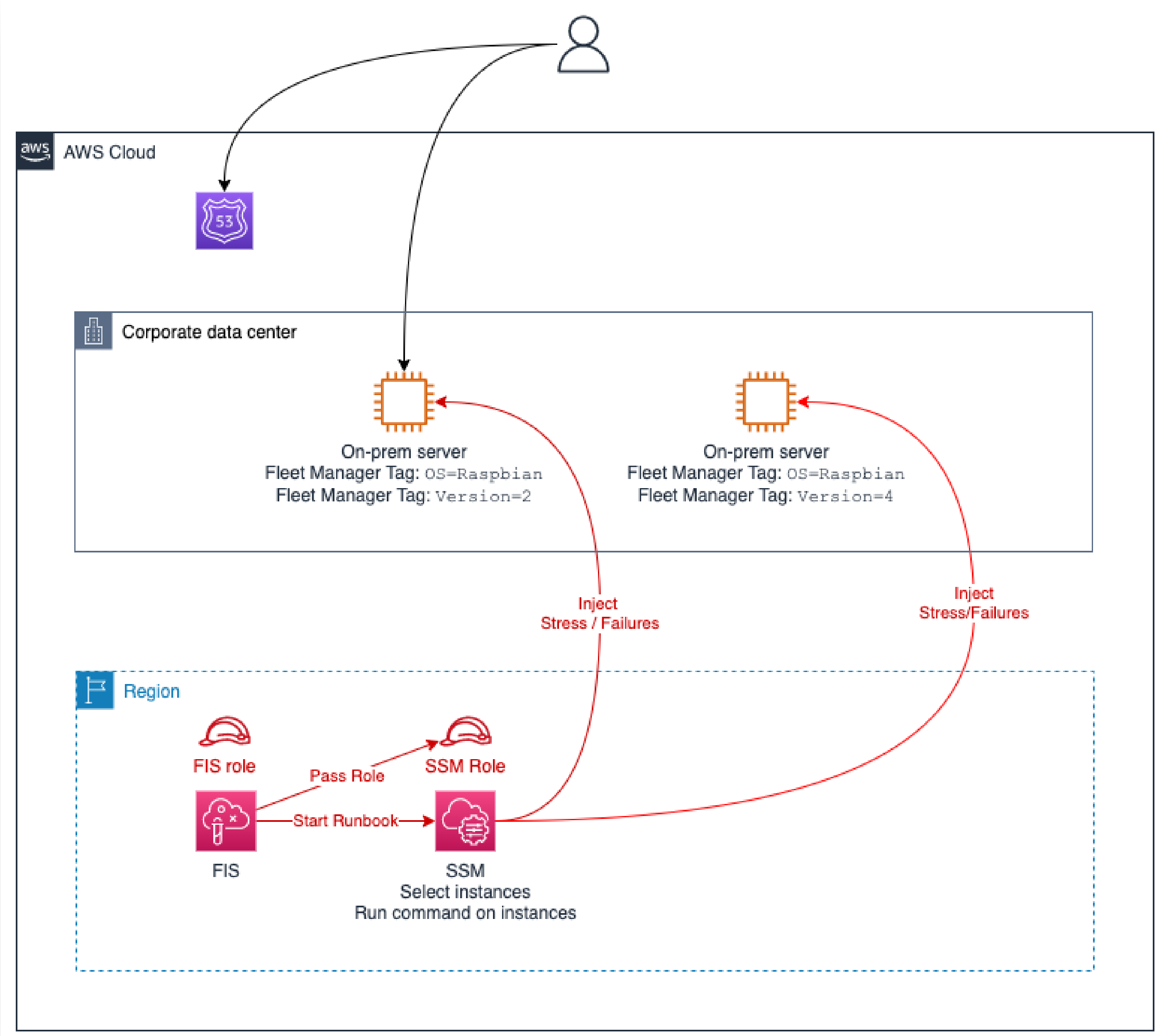
Warning

Some aspects of using hybrid instances may require activation of "advanced" tier. Please be aware that enabling advanced tier may incur substantial additional **costs**.

Note

For this section we assume that you already have a hybrid instance setup.

For illustration we will assume that you have a hybrid activation of two on-prem Raspberry Pi instances and the managed instances have been tagged in **SSM FleetManager** with tag `OS` / value `Raspbian` and tag `Version` / value `2` and `4` respectively:



Setup

To replicate the **Linux CPU Stress Experiment** on the on-prem instance we will use a variation on the **FIS SSM Start Automation Setup**.

Create SSM role

First we will need an SSM execution role to enable running the on-prem automation:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableAsgDocument",
```

```
            "Effect": "Allow",
            "Action": [
                "ssm:DescribeInstanceInformation",
                "ssm:ListCommands",
                "ssm:ListCommandInvocations",
                "ssm:SendCommand"            ],
            "Resource": "*"
        }
    ]
}
```

with an SSM assume role 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 blocks above into files named `iam-hybrid-demo-policy.json` and `iam-hybrid-demo-trust.json` and run the following CLI commands to create a role named `FisWorkshopSsmHybridDemoRole`:

```
# Set required variables
REGION=$(aws ec2 describe-availability-zones --output text --query
'AvailabilityZones[0].[RegionName]')
ACCOUNT_ID=$(aws sts get-caller-identity --output text --query 'Account')

cd ~/environment/aws-fault-injection-simulator-workshop
cd
workshop/content/030_basic_content/090_scenarios/020_targeting_hybrid_instances

HYBRID_ROLE_NAME=FisWorkshopSsmHybridDemoRole

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

aws iam put-role-policy \
  --role-name ${HYBRID_ROLE_NAME} \
  --policy-name ${HYBRID_ROLE_NAME} \
```

```
  --policy-document file://iam-hybrid-demo-policy.json

# Export ARN for later
HYBRID_ROLE_ARN=arn:aws:iam::${ACCOUNT_ID}:role/${HYBRID_ROLE_NAME}
echo ${HYBRID_ROLE_ARN}
```

Update FIS service role

Update the `FisWorkshopServiceRole` as described in the **FIS SSM Start Automation Setup** section, using the role ARN from the statement above. If you had previously performed that update note that you can add multiple role ARNs so the resulting `AllowFisToPassListedRolesToSsm` “Sid” would look like this:

```
{
            "Sid": "AllowFisToPassListedRolesToSsm",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "PREVIOUS_ROLE_ARN_HERE",
                "PLACE_NEW_ROLE_ARN_HERE"
            ]
        }
```

Create SSM document

The core of this approach is to select managed instances targets using SSM and then execute runbooks against the selected instances. The following parameters help target instances and define the fault injection to run:

- `Filters` - defines the filter parameter for the SSM **DescribeInstanceInformation** API. By default this is set to `[{"Key":"PingStatus","Values":["Online"]},{"Key":"ResourceType","Values":["ManagedInst` which will target all running managed instances. Below we will show you how to instead target instances based on FleetManager tags by adding `{"Key":"tag:OS","Values":["Raspbian"]}`.
- `DocumentName` - the name of an SSM runbook document to be called from this automation document after instance selection.
- `DocumentParameters` - Parameters to pass to the document. In our example below this will be the stress duration.

```yaml
---
description: Run SSM command on SSM hybrid instances
schemaVersion: '0.3'
assumeRole: "{{ AutomationAssumeRole }}"
parameters:
  AutomationAssumeRole:
    type: String
    description: "The ARN of the role that allows Automation to perform
      the actions on your behalf."
  DocumentName:
    type: String
    description: "SSM document name to run on hybrid instances"
  DocumentParameters:
    type: StringMap
    description: "Parameters to pass to SSM document run on hybrid instances
(string to deal with FIS serialization bug)"
  Filters:
    # Normally this would be a MapList.
    # Currently passing as string and converting to deal with some serialization
complexity.
    type: String
    description: '(Optional) Selector JSON for DescribeInstanceInformation as
described in CLI/API docs. Default [{"Key":"PingStatus","Values":["Online"]},
{"Key":"ResourceType","Values":["ManagedInstance"]}]'
    default: "[{\"Key\":\"PingStatus\",\"Values\":[\"Online\"]},
{\"Key\":\"ResourceType\",\"Values\":[\"ManagedInstance\"]}]"
mainSteps:
# -------------------------------------------------------------------
# Unpack a JSON string to JSON to deal with serialization complexity
- name: FormatConverter
  action: aws:executeScript
  onFailure: 'step:ExitHook'
  onCancel: 'step:ExitHook'
  timeoutSeconds: 60
  inputs:
    Runtime: "python3.6"
    Handler: "script_handler"
    InputPayload:
      JSONstring: "{{Filters}}"
    Script: |
      import json
      def script_handler(events, context):
          return json.loads(events.get("JSONstring","{}"))
  outputs:
    - Name: Filters
      Selector: "$.Payload"
      Type: MapList
# -------------------------------------------------------------------
# Select managed instances. Note that you can filter EITHER on tags
# OR on instance properties but not both.
- name: SelectHybridInstances
  action: aws:executeAwsApi
  onFailure: 'step:ExitHook'
  onCancel: 'step:ExitHook'
```

```yaml
    timeoutSeconds: 60
    inputs:
      Service: ssm
      Api: DescribeInstanceInformation
      Filters: "{{ FormatConverter.Filters }}"
    outputs:
      - Name: InstanceIds
        Selector: "$..InstanceId"
        Type: StringList
  # -----------------------------------------------------------------
  # Execute the DocumentName / DocumentParameters from inputs on the
  # instances selected in previous step.
  - name: DoStuff
    action: 'aws:runCommand'
    inputs:
      DocumentName: "{{ DocumentName }}"
      InstanceIds:
        - '{{SelectHybridInstances.InstanceIds}}'
      Parameters: "{{ DocumentParameters}}"
  # -----------------------------------------------------------------
  # NOOP exit point to allow skipping steps if selection fails
  - name: ExitHook
    action: aws:sleep
    inputs:
      Duration: PT1S
```

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

```bash
cd ~/environment/aws-fault-injection-simulator-workshop
cd
workshop/content/030_basic_content/090_scenarios/020_targeting_hybrid_instances

HYBRID_DOCUMENT_NAME=TargetHybridInstances

# Create SSM document
aws ssm create-document \
  --name ${HYBRID_DOCUMENT_NAME} \
  --document-format YAML \
  --document-type Automation \
  --content file://hybrid-target.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')
HYBRID_DOCUMENT_ARN=arn:aws:ssm:${REGION}:${ACCOUNT_ID}:document/${HYBRID_DOCUMENT_

echo $HYBRID_DOCUMENT_ARN
```

Assuming you have managed instances you can validate the SSM document by invoking it directly like this.

Note

Invocation on the command line requires additional square brackets around the individual parameter values independent of the parameter type defined in the SSM document. Complex parameters passed through to SSM documents may additionally require escaping quotes as show below

```
# Select all running managed instances (default with no Filters set)
aws ssm start-automation-execution \
  --document-name "TargetHybridInstances" \
  --parameters '{"AutomationAssumeRole":["'${HYBRID_ROLE_ARN}'"],"DocumentName":
["AWSFIS-Run-CPU-Stress"],"DocumentParameters":["{ \"DurationSeconds\": \"120\"
}"],"Filters":["[{\"Key\":\"PingStatus\",\"Values\":[\"Online\"]},
{\"Key\":\"ResourceType\",\"Values\":[\"ManagedInstance\"]}]"] }'
```

and

```
# Select all instances with tags OS=Raspbian and Version=4
aws ssm start-automation-execution \
  --document-name "TargetHybridInstances" \
  --parameters '{"AutomationAssumeRole":["'${HYBRID_ROLE_ARN}'"],"DocumentName":
["AWSFIS-Run-CPU-Stress"],"DocumentParameters":["{ \"DurationSeconds\": \"120\"
}"],"Filters":["[{\"Key\":\"tag:OS\",\"Values\":[\"Raspbian\"]},
{\"Key\":\"tag:Version\",\"Values\":[\"4\"]}]"] }'
```

Once started you can examine the progress by navigating to the **SSM Automation console** and selecting the execution ID from the invocation.

Create FIS template

As we saw in the **Create FIS Experiment Template** subsecton of **FIS SSM Start Automation Setup**, we need to substitute some ARN values into the FIS template. For convenience and to make the JSON string escaping easier we will do this with some shell substitutions. First we set the relevant environment variables:

```
ACCOUNT_ID=$(aws sts get-caller-identity --output text --query 'Account')
REGION=$(aws ec2 describe-availability-zones --output text --query
'AvailabilityZones[0].[RegionName]')
```

```
FIS_WORKSHOP_ROLE_ARN=arn:aws:iam::${ACCOUNT_ID}:role/FisWorkshopServiceRole
LINUX_STRESS_ARN=arn:aws:ssm:${REGION}::document/AWSFIS-Run-CPU-Stress
```

Then we use a bash trick to substitute them into our FIS template and write it to disk as `fis-hybrid-target.json`.

Note

Because we are doing an additional string evaluation we need to add extra escape characters to the source string leading to the 5 backslashes. See the final FIS template for a more human readable result with one level of escapes removed.

```
cat > fis-hybrid-target.json <<EOT
{
    "description": "Run stress on managed instance",
    "stopConditions": [
        {
            "source": "none"
        }
    ],
    "targets": {
    },
    "actions": {
        "terminateInstances": {
            "actionId": "aws:ssm:start-automation-execution",
            "description": "Managed instances run-command CPU Stress",
            "parameters": {
                "maxDuration": "PT3M",
                "documentArn": "${HYBRID_DOCUMENT_ARN}",
                "documentParameters": "{ \"AutomationAssumeRole\":
\"${HYBRID_ROLE_ARN}\", \"DocumentName\": \"${LINUX_STRESS_ARN}\",
\"DocumentParameters\": \"{ \\\\\"DurationSeconds\\\\\": \\\\\"120\\\\\" }\",
\"Filters\": \"[ {\\\\\"Key\\\\\":\\\\\"tag:OS\\\\\",\\\\\"Values\\\\\":
[\\\\\"Raspbian\\\\\"]} ]\" }"
            },
            "targets": {
            }
        }
    },
    "roleArn": "${FIS_WORKSHOP_ROLE_ARN}",
    "tags": {
        "Name": "ManagedInstanceCpuStress"
    }
}
EOT
```

Check the template content in `fis-hybrid-target.json` to confirm that the Role and Document ARNs have been filled in, then create the FIS experiment template:

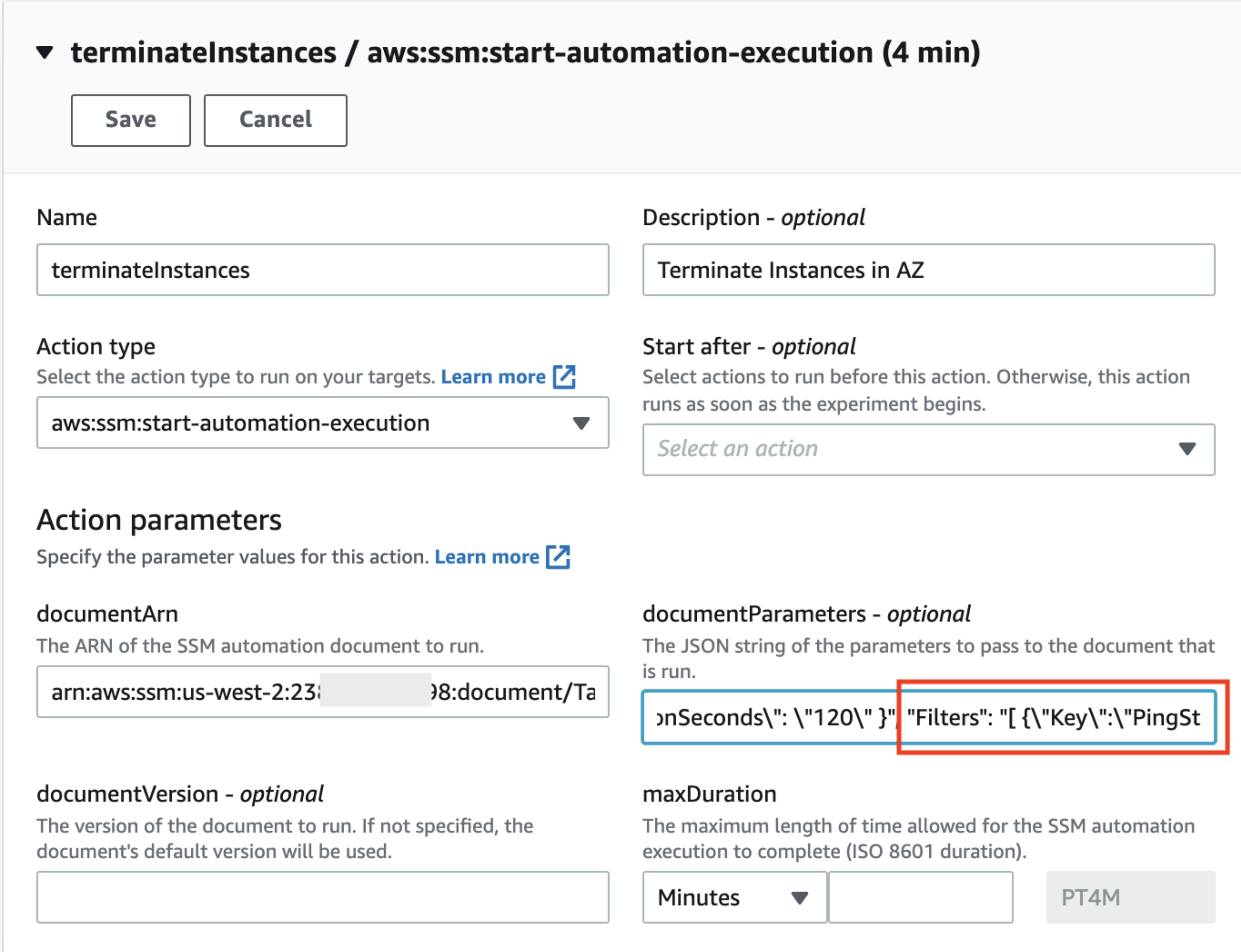
```
aws fis create-experiment-template \
   --cli-input-json file://fis-hybrid-target.json
```

Running experiments

Targeting all running hybrid instances

SSM allows targeting instances based on properties returned by the SSM **DescribeInstanceInformation** API. On prem instances are identified by a `ResourceType` of `ManagedInstance`. Additionally we might only want to include currently running instances identified by a `PingStatus` of `Online`.

Navigate to the **FIS experiment template console**, select the experiment template ID created above, and edit the `"Filters"` statement in the `documentParameters` entry:



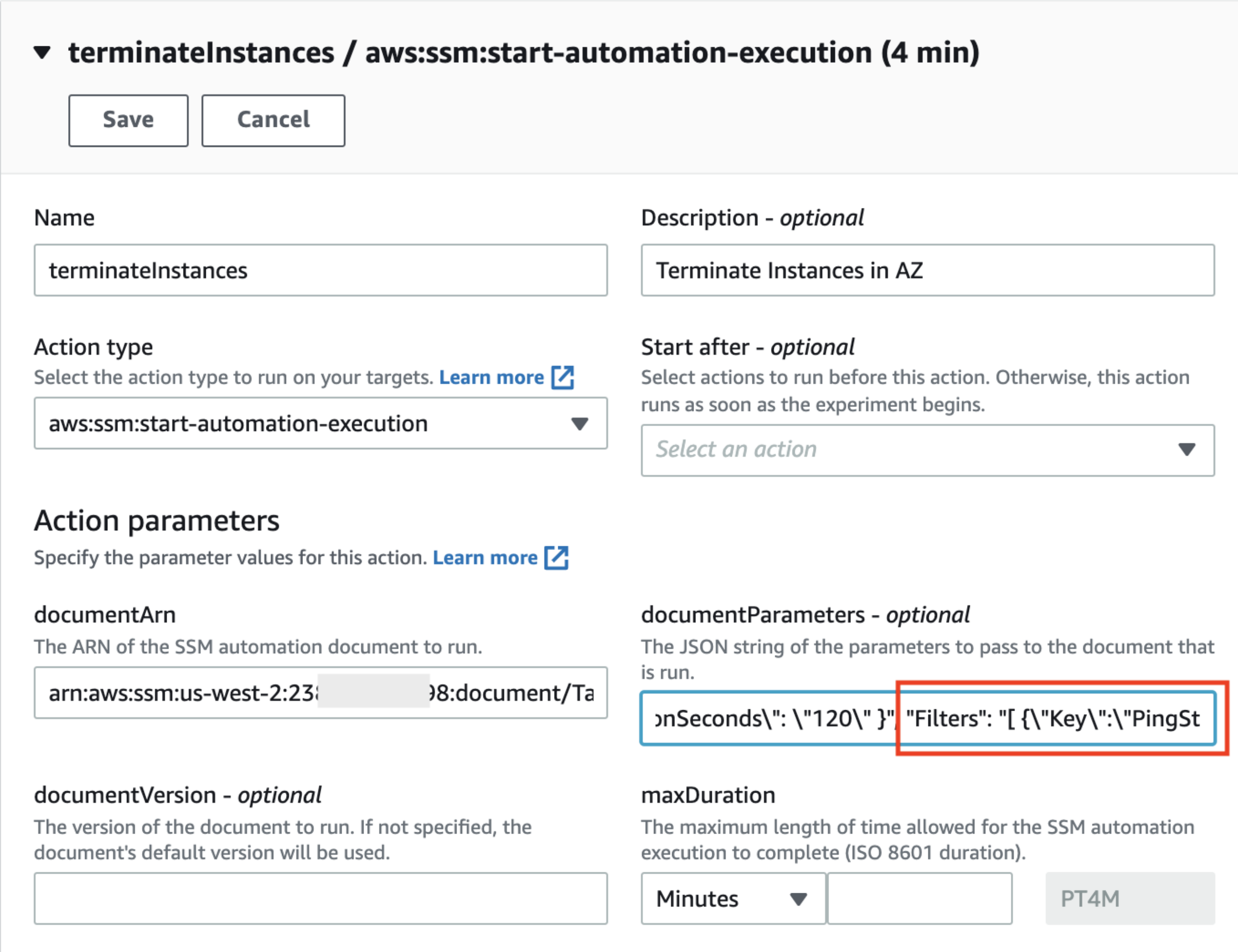
to read:

```
"Filters": "[ {\"Key\":\"PingStatus\",\"Values\":[\"Online\"]},
{\"Key\":\"ResourceType\",\"Values\":[\"ManagedInstance\"]} ]"
```

Targeting specific managed instances

SSM allows you to target instances based on tag values. The default version of the template will target all instances tagged with `OS` value `Raspbian`. We could furter refine that to only target instances with `Version` value `4`.

Navigate to the **FIS experiment template console**, select the experiment template ID created above, and edit the `"Filters"` statement in the `documentParameters` entry:



to read:

```
"Filters": "[ {\"Key\":\"tag:OS\",\"Values\":[\"Raspbian\"]},
{\"Key\":\"tag:Version\",\"Values\":[\"4\"]} ]"
```

Learnings and next steps

The approach outline above provides a generic way to run SSM documents on on-prem managed instances. You may want to expand the SSMA document to suit your needs, e.g. with custom parameters for easier targeting or with more complex selection mechanisms.

Targeting specific running instances

Because tags are stored separately from instance metadata SSM does not allow joint queries for both metadata such as `PingState` and tags such as `OS`. If you have only a small number of instances you could make two separate lookups and use the `aws:executeScript` action to merge the two result sets. For large numbers of managed instances this is potentially slow and may run into pagination issues on the API. Here we would suggest to instead manage all relevant information in tags and do a single lookup.

SIMULATING AZ ISSUES

A common ask we hear is for "Availability Zone 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 Availability Zone (AZ) failure it's worth considering what we mean by "AZ failure".

AZ vs. data center

Many of our customers phrase their idea of an AZ failure as "the whole data center 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 "data center" outage at AWS may not have the level of impact you would expect on-prem. Additionally, many AWS services use cell-based architectures to even further reduce the impact of any system failures.

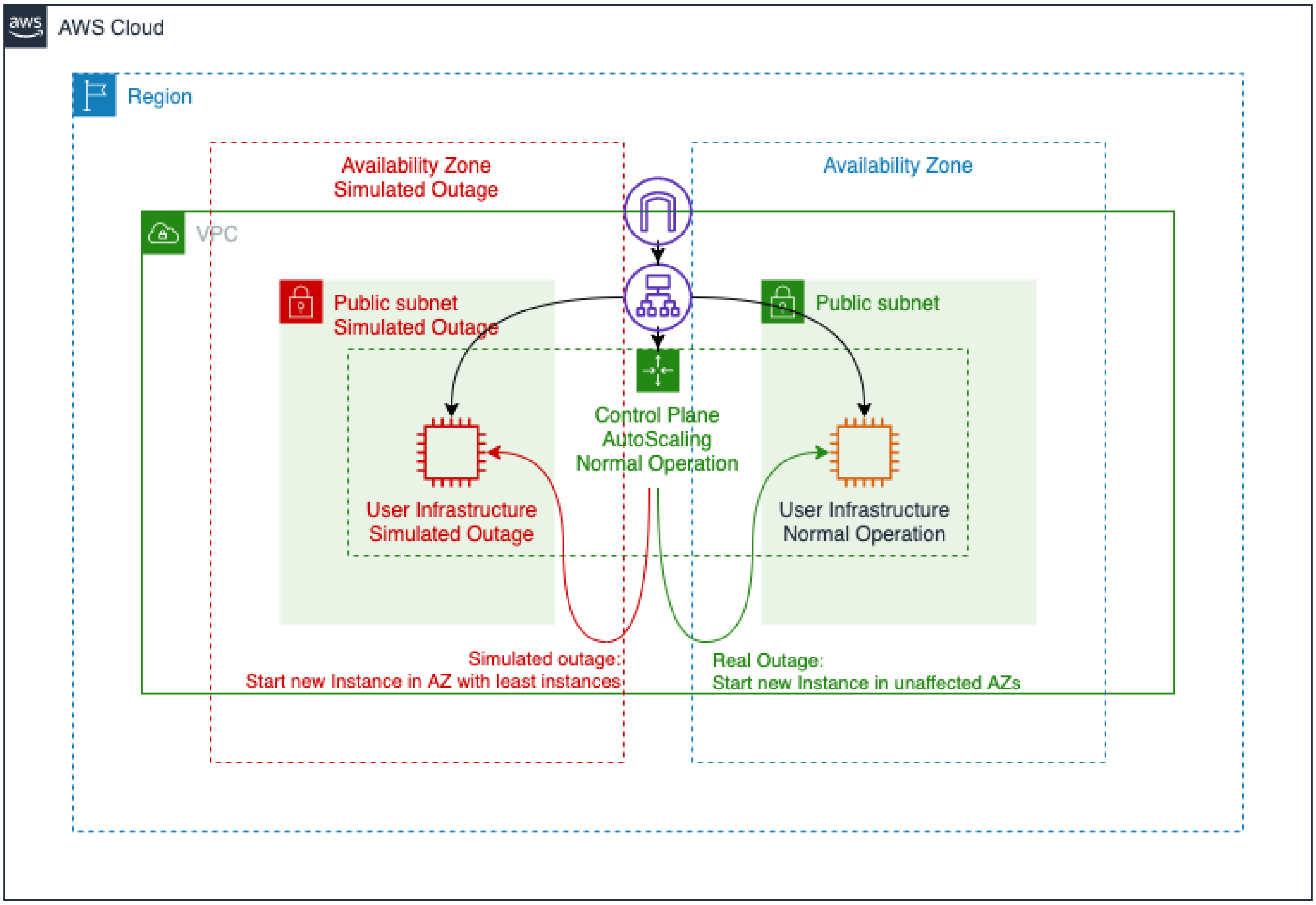
Control plane vs. data plane

When simulating AZ failure, an important thing to consider is the difference between the effects of an outage on the "control plane" vs. the "data plane" and their impact on reliability:

- Data plane is responsible for delivering service. E.g. in an AWS Auto Scaling group, the EC2 instances being started or stopped into different AZs would represent the data plane. Similarly in a user managed cluster there will typically be "worker" nodes that are involved in delivering the service.
- Control plane is used to configure an environment or service. E.g. in AWS Auto Scaling group a scheduler will constantly monitor the requirement for EC2 instances and the number of available instances and will start and stop instances according to requirements. Similarly in a user managed cluster there will typically be "master" or "control" nodes that are involved in monitoring and controlling the worker nodes.

A *real* outage, whether due to a bad cell, a full data center outage, or even a full AZ outage, would create awareness in the AWS control plane that these resources are unavailable. During the impact period the control plane would only use un-affected parts of the data plane.

In contrast a *simulated* outage will only affect the data plane, limited to just the provisioned customer resources, without affecting the control plane.



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 in a way that simulates some of the data plane awareness.

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
```

```yaml
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
```

```yaml
    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:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableAsgDocument",
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:SuspendProcesses",
                "autoscaling:ResumeProcesses",
                "autoscaling:UpdateAutoScalingGroup",
                "ec2:DescribeInstances",
                "ec2:DescribeInstanceStatus",
                "ec2:TerminateInstance",
                "ec2:DescribeSubnets"
            ],
```

```
            "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.

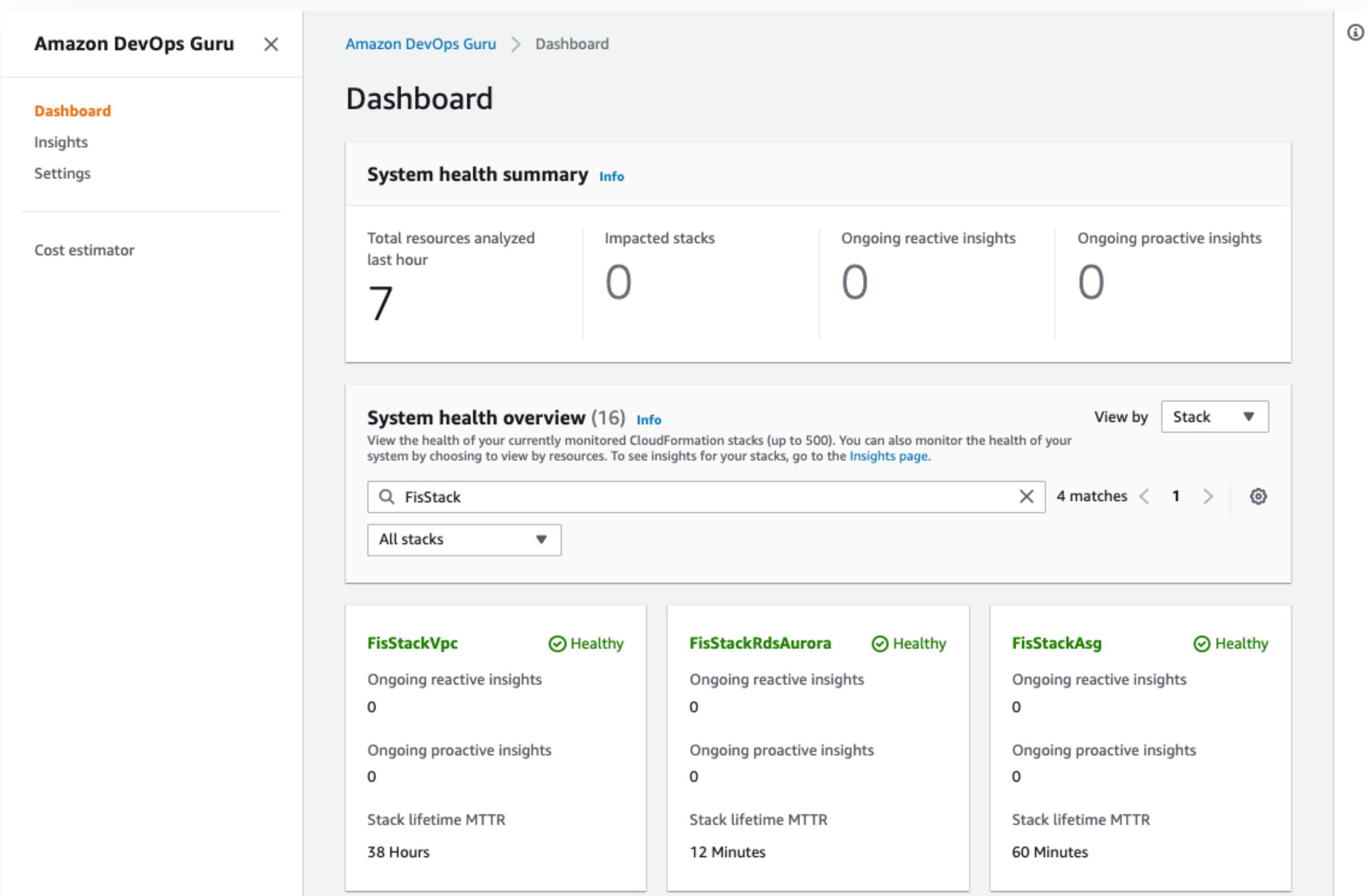
DEVOPS GURU

Warning

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

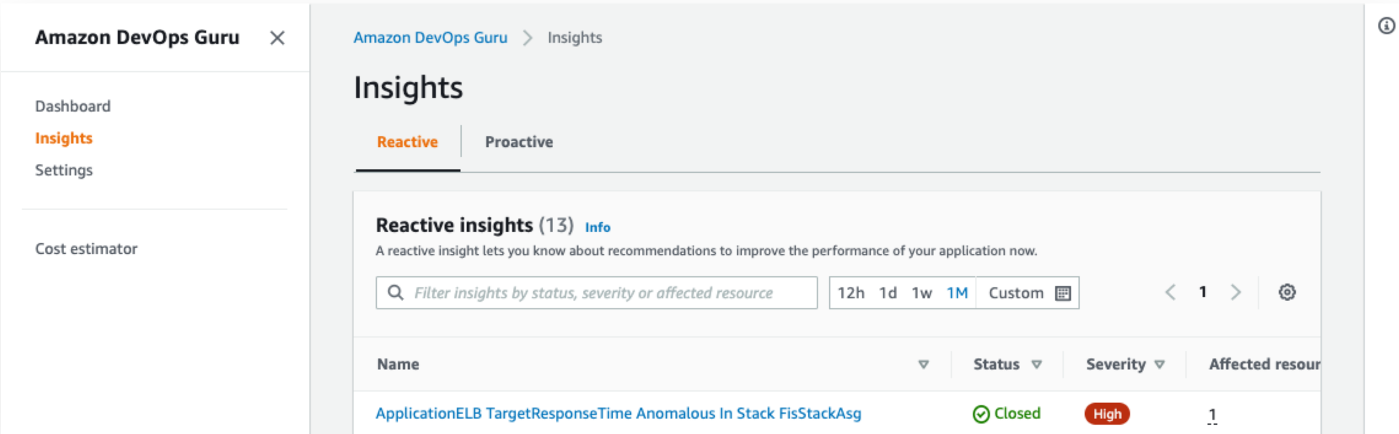
Dashboard overview

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:



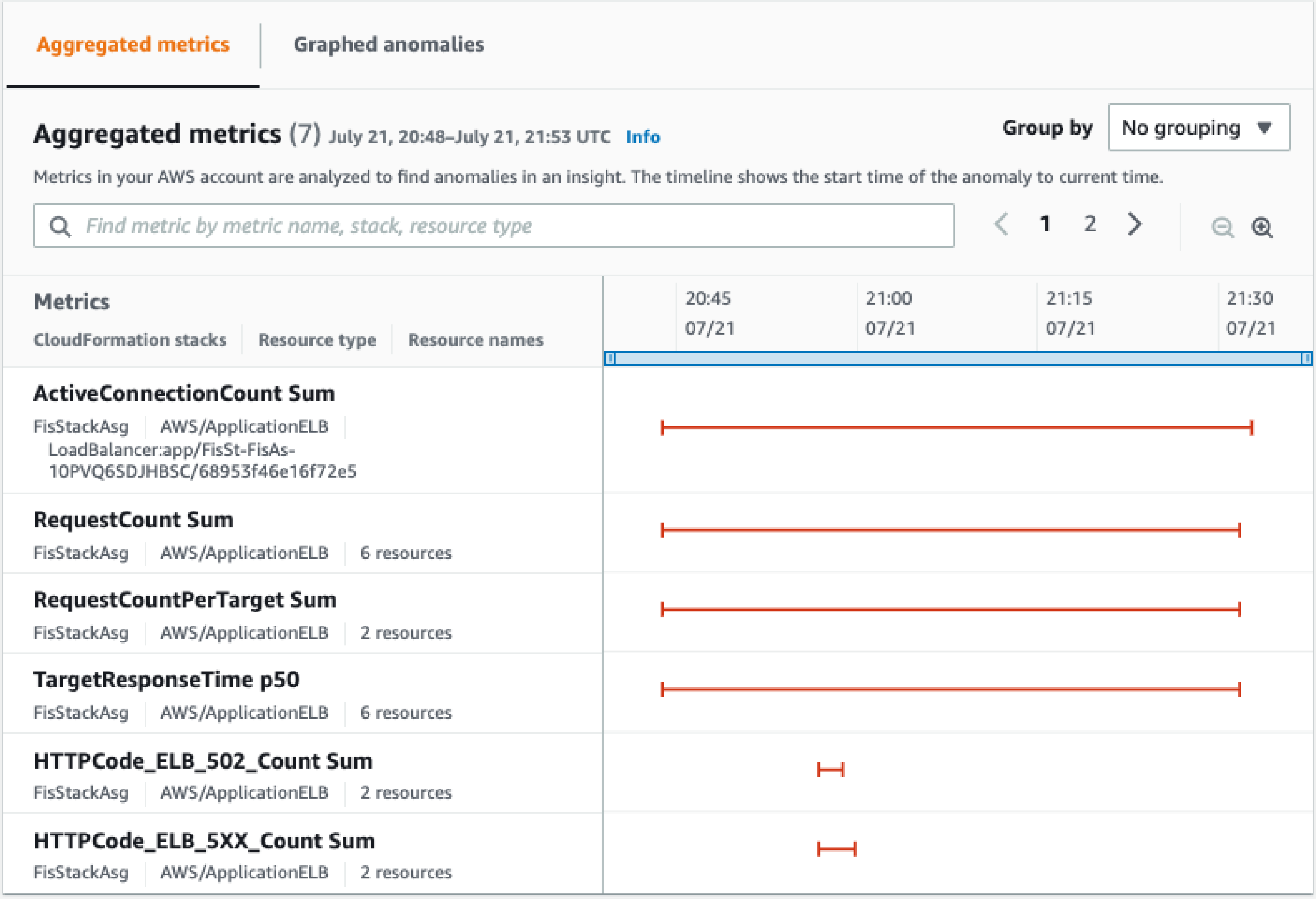
Reactive insights

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):

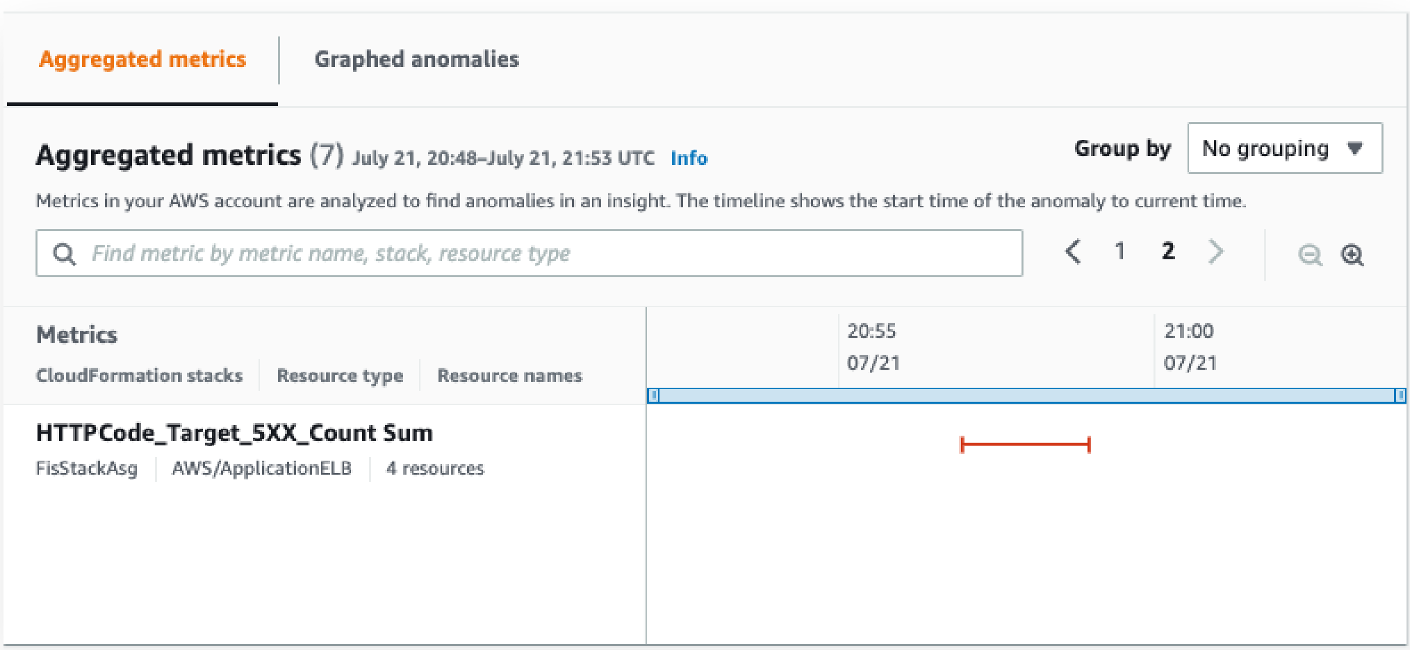


Visualizing anomalies

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

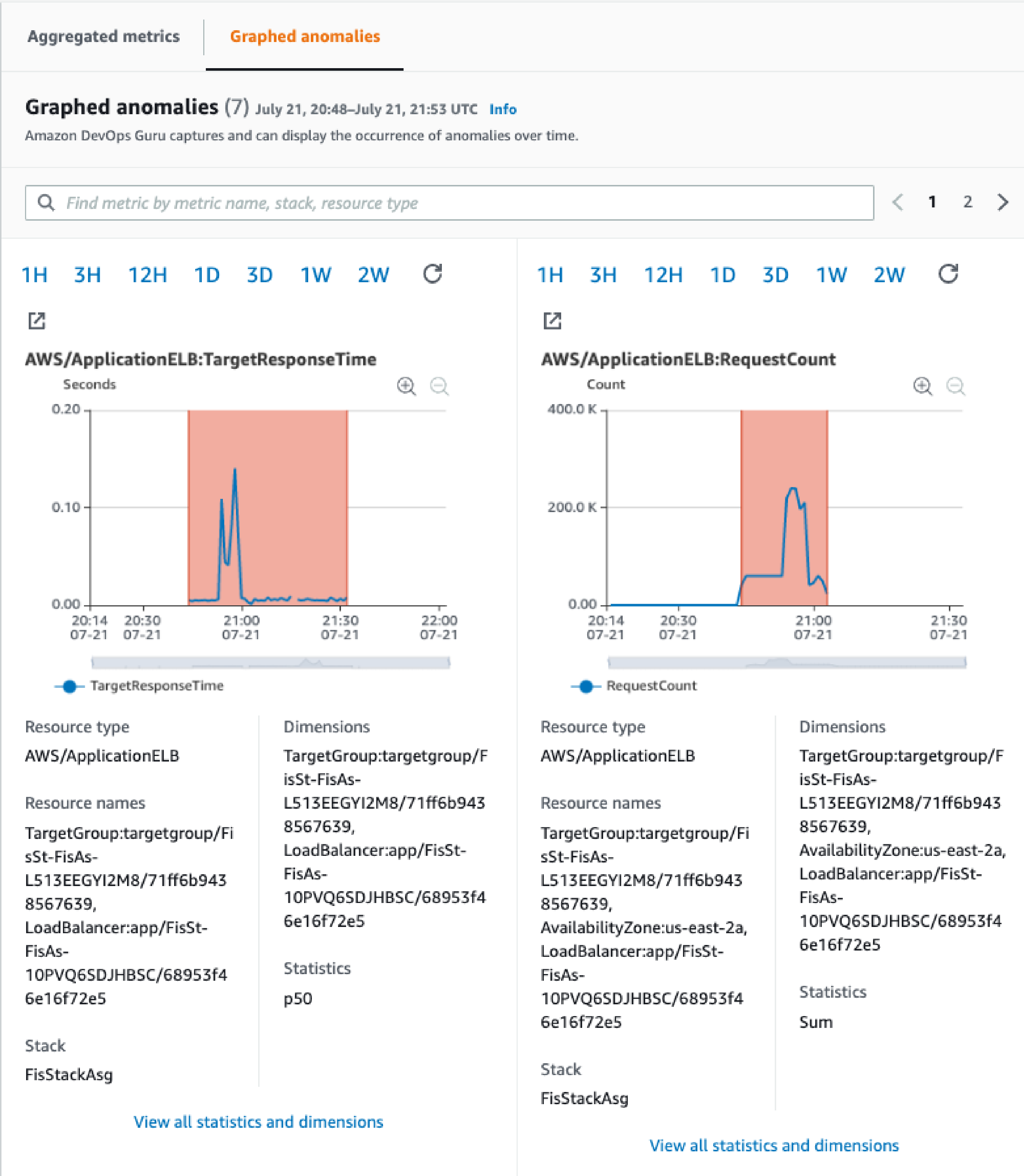
- an unusually high number of connections were made - by our external load testing tool,

- 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.

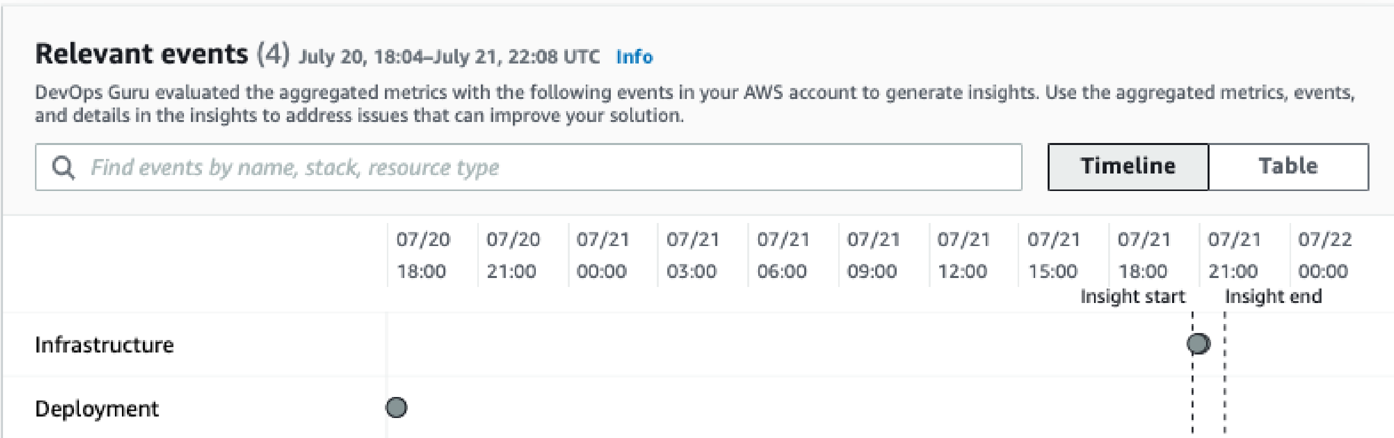
Switching to the “Graphed Anomalies” view shows the more detailed time data for each anomalous metric:



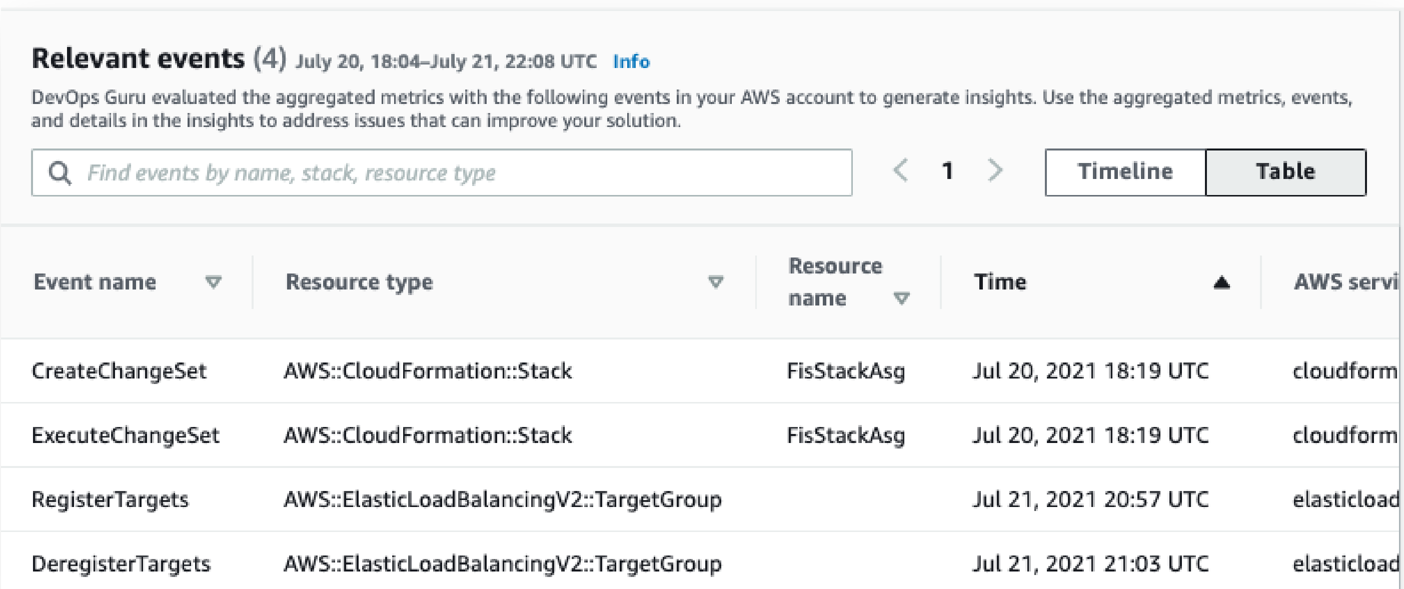
Note that in this view data outside the anomaly window are set to zero to allow focusing on the relevant details during the outage.

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:



Relevant events (4) July 20, 18:04–July 21, 22:08 UTC Info

DevOps Guru evaluated the aggregated metrics with the following events in your AWS account to generate insights. Use the aggregated metrics, events, and details in the insights to address issues that can improve your solution.

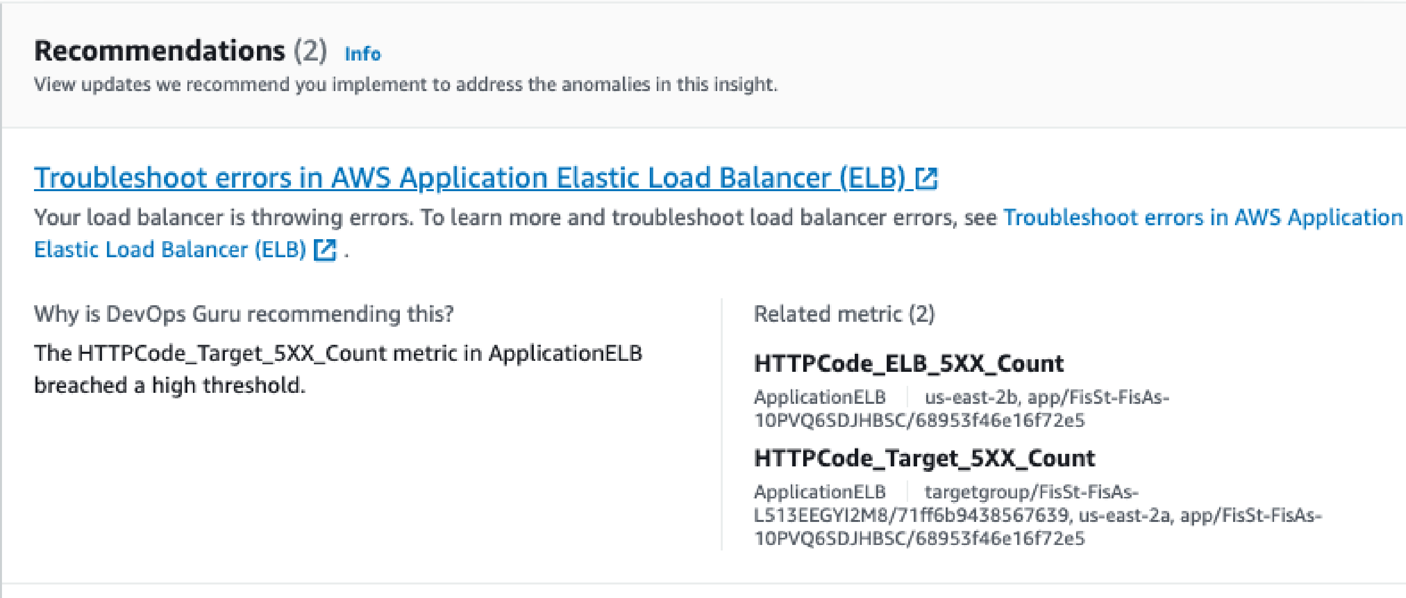
Find events by name, stack, resource type | 1 | Timeline | Table

Event name	Resource type	Resource name	Time	AWS servi
CreateChangeSet	AWS::CloudFormation::Stack	FisStackAsg	Jul 20, 2021 18:19 UTC	cloudform
ExecuteChangeSet	AWS::CloudFormation::Stack	FisStackAsg	Jul 20, 2021 18:19 UTC	cloudform
RegisterTargets	AWS::ElasticLoadBalancingV2::TargetGroup		Jul 21, 2021 20:57 UTC	elasticload
DeregisterTargets	AWS::ElasticLoadBalancingV2::TargetGroup		Jul 21, 2021 21:03 UTC	elasticload

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

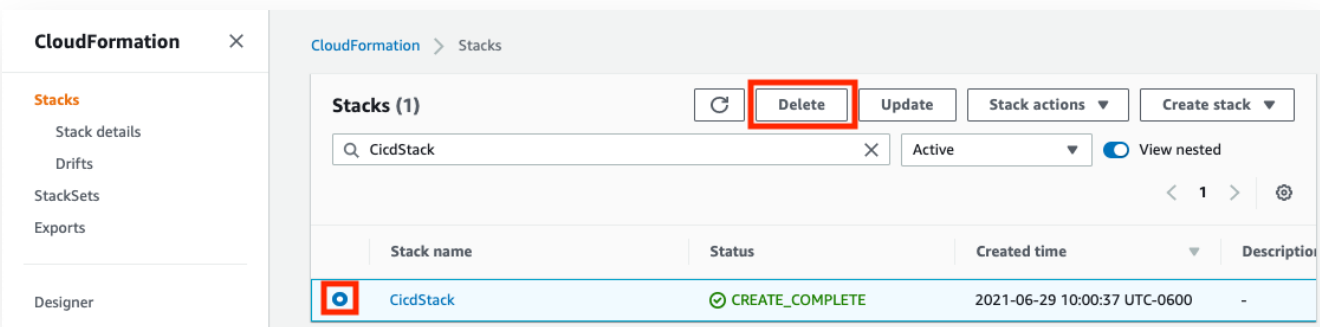
To learn more about DevOps Guru, see the **documentation**, and explore using **DevOps guru on serverless infrastructure** as well as **larger deployment strategies**.

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 `CicdStack`
  - Select the stack
  - Select "Delete"
- Once, the `CicdStack` is deleted, following the same procedure as above, delete the `CicdStack` stack



- 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-parallel.sh
```

Retained resources

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