Chaos Engineering

Final Review

Aashish Waikar

Capital One

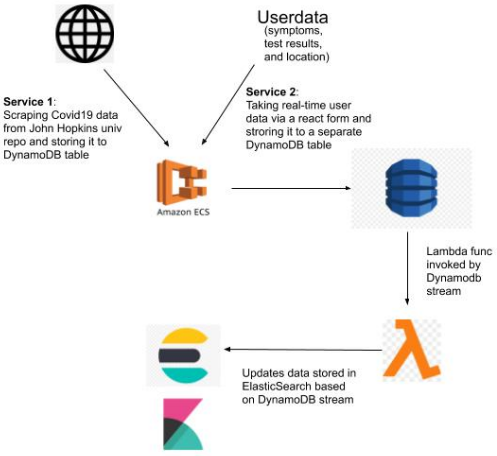
June 23, 2020

Table of Contents

Revisiting the Objectives

- Chaos Engineering is the field of testing which deals with performing experiments to build confidence in the cloud application's robustness and resilience to unexpected conditions (Eg. large network variations, unexpected shutdowns of cluster instances, etc).
- Our main objectives are to demonstrate how Chaos Engineering expts can be incorporated in cloud-based architectures, the various types of tests that can be performed and how they might help in figuring out loose ends in our application.

Architecture



Description

My application has the following services which run on ECS:-

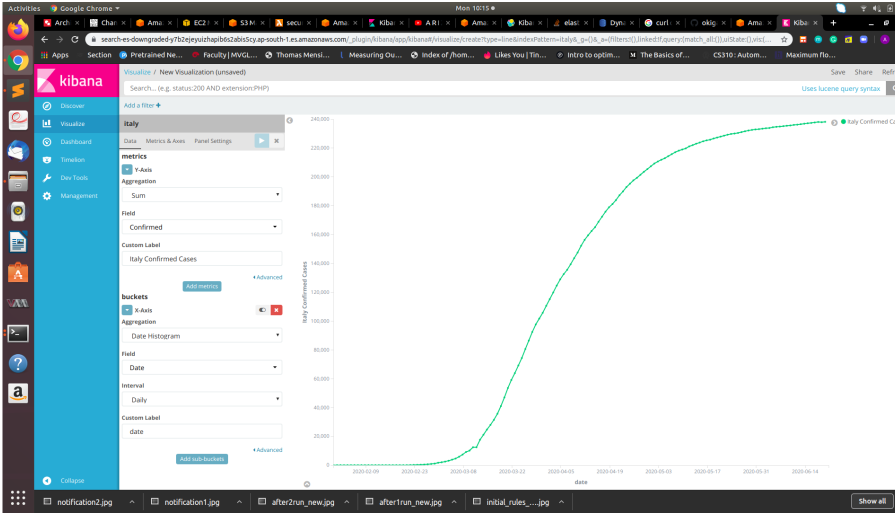
1. **Scraping Service:** This service consists of a task which scrapes world-wide and date-wise COVID19 data from the github repo maintained by John Hopkins university, makes some transformations and stores it in a DynamoDB table named "CoronaVirus".
2. **React Service:** This service consists of a task which hosts a React form through which users can submit real-time information regarding whether they have COVID symptoms, their test results (if been tested), and their location. On submitting this data is feeded into a DynamoDB table named "RealTimeUpdates".

Description Contd.

In addition, I have created an AWS Lambda function which gets triggered by DynamoDB streams, and based on the changes in the DynamoDB table, the same changes are made to the data present in the ElasticSearch cluster. The ElasticSearch cluster is also connected to Kibana for viewing visualisations.

Visualisations

Datewise increase of cases in Italy. (Almost saturated)



Visualize / New Visualization (unsaved)

italy

metrics

Y-Axis

Aggregation: Sum

Field: Confirmed

Custom Label: Italy Confirmed Cases

buckets

X-Axis

Aggregation: Date Histogram

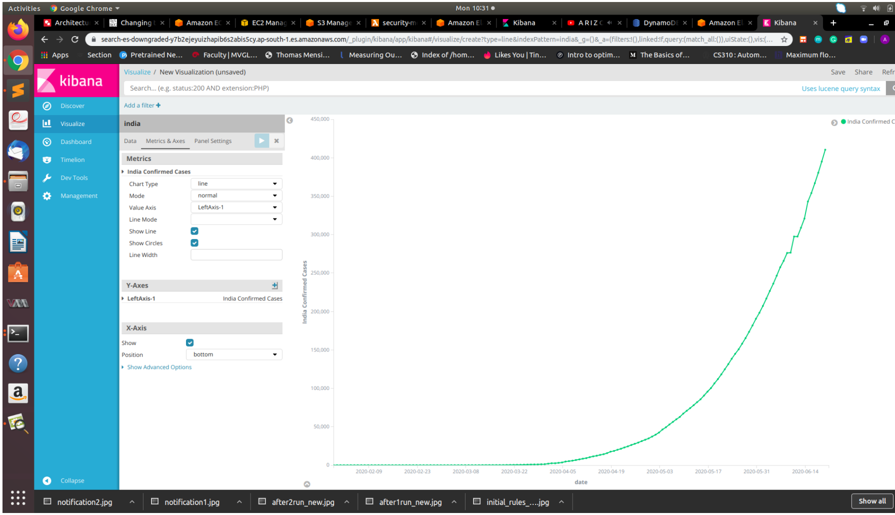
Field: Date

Interval: Daily

Custom Label: date

Visualisations Contd.

Datewise increase of cases in India. (Still in exponential rise phase)



Visualize / New Visualization (unsaved)

india

Data | Metrics & Axes | Panel Settings

Metrics

India Confirmed Cases

Chart Type	line
Mode	normal
Value Axis	LeftAxis-1
Line Mode	
Show Line	✓
Show Circles	✓
Line Width	

Y-Axes

LeftAxis-1 India Confirmed Cases

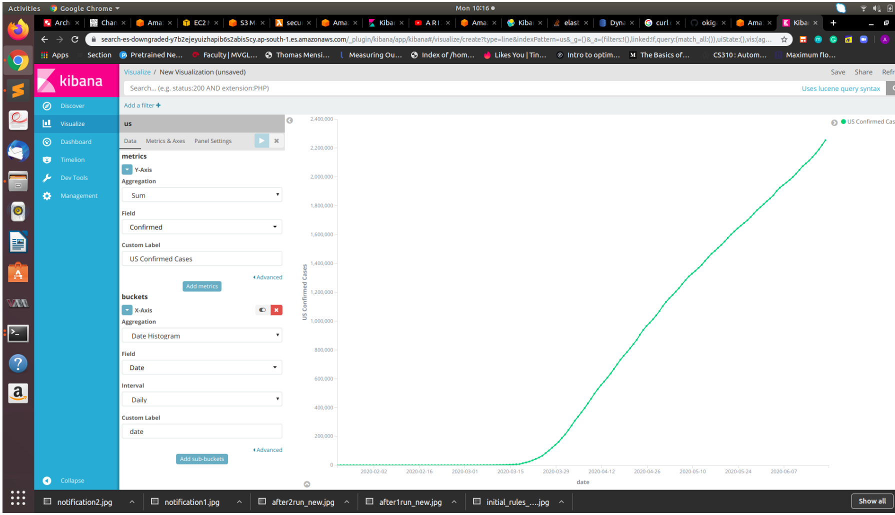
X-Axis

Show	✓
Position	bottom

Show Advanced Options

Visualisations Contd.

Datewise increase of cases in US. (In linear phase)


Visualize / New Visualization (unsaved)

Search... (e.g. status:200 AND extension:PHP)

us

Data | Metrics & Axes | Panel Settings

metrics

Y-Axis

Aggregation: Sum

Field: Confirmed

Custom Label: US Confirmed Cases

buckets

X-Axis

Aggregation: Date Histogram

Field: Date

Interval: Daily

Custom Label: date

Expt Design

- This experiment deals with randomly shutting down instances which are part of our ECS cluster.
- The configuration file for this experiment is stored in an S3 bucket named "config–bucket". For setting the configuration, a line has to be added to the config-file, in the format "ECS_Cluster_Name Blast_Radius start_time stop_time". The blast_radius determines percentage of chaos in our cluster.
- I have created an AWS lambda function which runs every 1 minute, fetches the config file from S3, and checks if current time lies in the period mentioned in the config file, then it randomly chooses blast-radius percent of instances from the cluster and shuts them down. And when the period has ended, it starts those again.

Expt Design Contd.

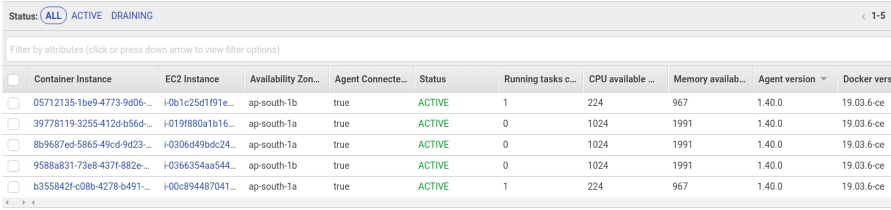
- For controlling the shutting down and startups of the instances I have used the "boto3" SDK which is the AWS SDK for Python.
- I had initially thought of terminating ec2 instances when the period starts and spawning new ones when when the period ends. But, if an ec2 instance is terminated, the ECS agent automatically creates a new one to maintain cluster size. This caused the expt to not run properly.
- So, instead of that, I change the state of the container instance to "Draining" when period starts and back to "Active" when period ends. In draining state, existing tasks are stopped and reassigned to the active container instances based on the healthy task percentages configured in the service, and no additional tasks can run on that instance.

Observations

- I ran this expt 4 times with blast radius as 20, 40, 60, and 80 respectively. Since, I had 5 instances in the cluster and 2 tasks running, in the first 3 cases the tasks stopped and restarted on the remaining instances.
- However, in the case where blast radius is 80, only one instance is left in Active state. Due to this, only one task out of the two are able to run on our cluster (this is because AWS allows to run maximum 1 task on an ECS container instance, multiple tasks can't run on the same instance even if there are sufficient resources present).
- This affects the application seriously, because either the React task stops running or the scraping task stops running.

Observations Contd.

For this expt, we define the steady state as the state when both the tasks are healthy and running. The state before running expt is captured below.



Status: ALL ACTIVE DRAINING

1-5

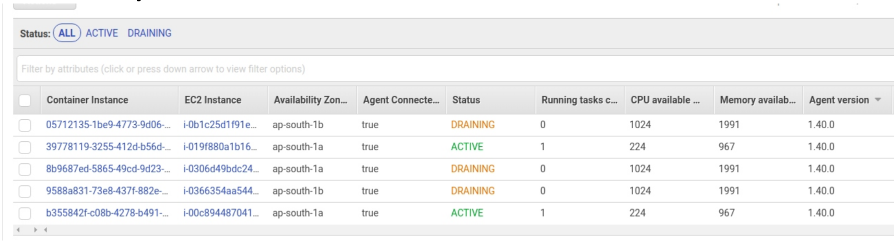
Filter by attributes (click or press down arrow to view filter options)

Container Instance	EC2 Instance	Availability Zon...	Agent Connecte...	Status	Running tasks c...	CPU available ...	Memory availab...	Agent version	Docker vers
05712135-1be9-4773-9d06-...	i-0b1c25d1f91e...	ap-south-1b	true	ACTIVE	1	224	967	1.40.0	19.03.6-ce
39778119-3255-412d-b56d-...	i-019f880a1b16...	ap-south-1a	true	ACTIVE	0	1024	1991	1.40.0	19.03.6-ce
8b9687ed-5865-49cd-9d23-...	i-0306d49bdc24...	ap-south-1a	true	ACTIVE	0	1024	1991	1.40.0	19.03.6-ce
9588a831-73e8-437f-882e-...	i-0366354aa544...	ap-south-1b	true	ACTIVE	0	1024	1991	1.40.0	19.03.6-ce
b355842f-c08b-4278-b491-...	i-00c894487041...	ap-south-1a	true	ACTIVE	1	224	967	1.40.0	19.03.6-ce

Since 800 CPU units and 1024 MB memory is allotted for each task:-
Effective CPU Utilisation = ((800*2)/(1024*5))*100 ie 31.25%
Effective Memory Utilisation = ((1024*2)/(1991*5))*100 ie 20.57%

Observations Contd.

Since, the first 3 cases are similar, I'll just show the screenshot and metrics analysis for the case when blast_radius=60 :-



Status: ALL ACTIVE DRAINING

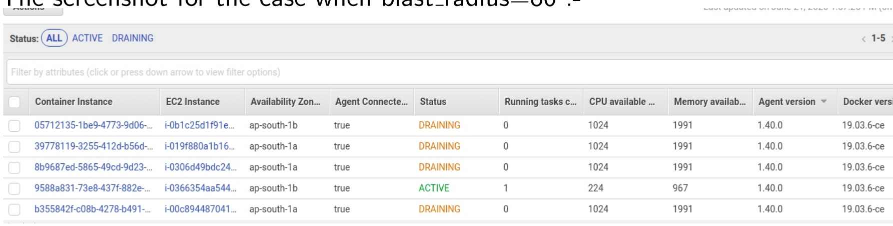
Filter by attributes (click or press down arrow to view filter options)

Container Instance	EC2 Instance	Availability Zon...	Agent Connecte...	Status	Running tasks c...	CPU available ...	Memory availab...	Agent version
05712135-1be9-4773-9d06-...	i-0b1c25d1f91e...	ap-south-1b	true	DRAINING	0	1024	1991	1.40.0
39778119-3255-412d-b56d-...	i-019f880a1b16...	ap-south-1a	true	ACTIVE	1	224	967	1.40.0
8b9687ed-5865-49cd-9d23-...	i-0306d49bdc24...	ap-south-1a	true	DRAINING	0	1024	1991	1.40.0
9588a831-73e8-437f-882e-...	i-0366354aa544...	ap-south-1b	true	DRAINING	0	1024	1991	1.40.0
b355842f-c08b-4278-b491-...	i-00c894487041...	ap-south-1a	true	ACTIVE	1	224	967	1.40.0

As seen above, only two instances are left in Active state, meaning:-
Effective CPU Utilisation = ((800*2)/(1024*2))*100 ie 78.125%
Effective Memory Utilisation = ((1024*2)/(1991*2))*100 ie 51.43%.

Observations Contd.

The screenshot for the case when blast_radius=80 :-


Status: ALL ACTIVE DRAINING 1-5

Filter by attributes (click or press down arrow to view filter options)

Container Instance	EC2 Instance	Availability Zon...	Agent Connecte...	Status	Running tasks c...	CPU available ...	Memory availab...	Agent version	Docker versi
05712135-1be9-4773-9d06-...	i-0b1c25d1f91e...	ap-south-1b	true	DRAINING	0	1024	1991	1.40.0	19.03.6-ce
39778119-3255-412d-b56d-...	i-019f880a1b16...	ap-south-1a	true	DRAINING	0	1024	1991	1.40.0	19.03.6-ce
8b9687ed-5865-49cd-9d23-...	i-0306d49bdc24...	ap-south-1a	true	DRAINING	0	1024	1991	1.40.0	19.03.6-ce
9588a831-73e8-437f-882e-...	i-0366354aa544...	ap-south-1b	true	ACTIVE	1	224	967	1.40.0	19.03.6-ce
b355842f-c08b-4278-b491-...	i-00c894487041...	ap-south-1a	true	DRAINING	0	1024	1991	1.40.0	19.03.6-ce

As seen above, only 1 task is now running on the instance in Active state which compromises the functionality of the application.

Remediation

- To remediate such a situation, we can have multiple ECS instances per ec2 instance.
- For eg. if there were 2 ECS instances per ec2 instance, then, after 80% shutdown, 2 container instances would still be left to run both the tasks.

Expt Design

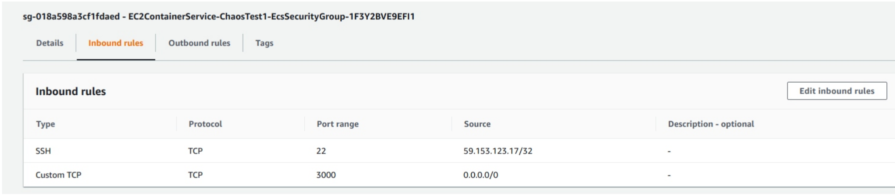
- This experiment deals with making random changes in the inbound rules of our cluster's security group to build confidence in the security alarm/notification system.
- The majority of security intrusions occur because hackers take advantage of ports unexpectedly left open and gain access to our system's data. For tackling such issues, most cloud systems have a notification mechanism such that whenever any changes occur in the inbound rules of our cluster, the administrator gets notified.
- I have built a similar notification system for my application. I have created a cloudwatch rule with the event pattern that matches any event associated with changes in our security group, and rule's target is an SNS topic which sends a notification to my gmail address.

Expt Design Contd.

- The configuration file for this experiment is stored in an S3 bucket named "config–bucket". For running this expt. with a security group, it's security group ID has to be added to the config file.
- I have created an AWS lambda function which when run, fetches the config file from S3. For each security group in the config, it randomly chooses to either add an inbound rule or delete one of the rules already present. While adding an inbound rule, it randomly chooses the CIDR IP (v1.v1.v1.v1/v2 where v1 is chosen randomly from 0 to 255 and v2 randomly from 0 to 32), the port (randomly b/w 0 and 65535), and the protocol(UDP or TCP).
- With this expt. we hope to determine whether our cloudwatch event rules and SNS notification system work fine.

Observations

First, I add the ID of our security group to the config file. Initially, the inbound rules look like this:-



sg-018a598a3cf1fdaed - EC2ContainerService-ChaosTest1-EcsSecurityGroup-1F3Y2BVE9EFI1

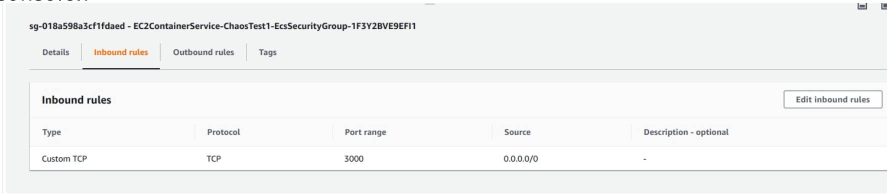
Inbound rules

Type	Protocol	Port range	Source	Description - optional
SSH	TCP	22	59.153.123.17/32	-
Custom TCP	TCP	3000	0.0.0.0/0	-

As in above image, there is one rule for SSH from my machine and another rule for accessing the React form from anywhere.

Observations Contd.

Then, I run the lambda function which makes random changes in our security group. I check whether these are reflected in the security group console.:-



sg-018a598a3cf1fdaed - EC2ContainerService-ChaosTest1-EcsSecurityGroup-1F3Y2BVE9EFI1

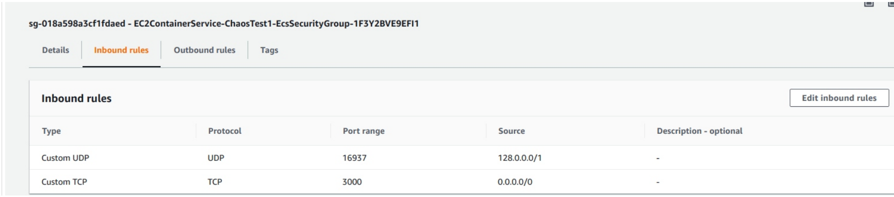
Inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	3000	0.0.0.0/0	-

It randomly revoked one of the existing inbound rules.

Observations Contd.

I run the lambda function again. I check whether there are any changes in the security group console.:-



sg-018a598a3cf1fdaed - EC2ContainerService-ChaosTest1-EcsSecurityGroup-1F3Y2BVE9EFI1

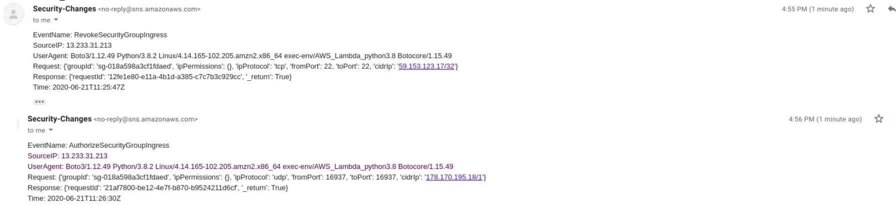
Inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom UDP	UDP	16937	128.0.0.0/1	-
Custom TCP	TCP	3000	0.0.0.0/0	-

This time it randomly, authorised a new inbound rule.

Remediation

Checking my gmail account, I've received notifications for both these changes:-



Security-Changes <no-reply@sns.amazonaws.com> 4:55 PM (1 minute ago)
to me

```
EventName: RevokeSecurityGroupIngress
SourceIP: 13.233.31.213
UserAgent: Boto3/1.12.49 Python/3.8.2 Linux/4.14.165-102.205.amzn2.x86_64 exec-env/AWS_Lambda_python3.8 Botocore/1.15.49
Request: {'groupId': 'sg-018a598a3cf1fdaed', 'ipPermissions': {}, 'ipProtocol': 'tcp', 'fromPort': 22, 'toPort': 22, 'cidrIp': '59.153.123.17/32'}
Response: {'requestId': '12fe1e80-e11a-4b1d-a385-c7c7b3c929cc', '_return': True}
Time: 2020-06-21T11:25:47Z
```

Security-Changes <no-reply@sns.amazonaws.com> 4:56 PM (1 minute ago)
to me

```
EventName: AuthorizeSecurityGroupIngress
SourceIP: 13.233.31.213
UserAgent: Boto3/1.12.49 Python/3.8.2 Linux/4.14.165-102.205.amzn2.x86_64 exec-env/AWS_Lambda_python3.8 Botocore/1.15.49
Request: {'groupId': 'sg-018a598a3cf1fdaed', 'ipPermissions': {}, 'ipProtocol': 'udp', 'fromPort': 16937, 'toPort': 16937, 'cidrIp': '178.170.195.18/1'}
Response: {'requestId': '21af7800-be12-4e7f-b870-b9524211d6cf', '_return': True}
Time: 2020-06-21T11:26:30Z
```

Thus, the application's alarm system can be trusted.

Expt Overview

- This experiment involves injecting latency in one of the network links in our cloud architecture. (Eg. in my case the link b/w ECS cluster and DynamoDB)
- If a large latency is injected in the DynamoDB requests, then the scraping-service may not be able to function properly, and changes in data might not get reflected in ElasticSearch.
- For this, I screated another service which scrapes the data and directly transfers it to ElasticSearch (DynamoDB isn't involved). This would act as a fallback for the original service.
- We expect our application to shift to the fallback service, in case the original service doesn't function. This experiment is to build confidence in our application's fallback mechanism.

CapitalOne

Expt Design: Work in progress

- For this experiment I initially planned to just write wrappers around the high-level DynamoDB API requests and add the line "sleep(latency)" to run the experiment. But that would've been hardcoding. What is actually required is to run the same code without changes, but experience latency in the link.
- This requires working with the low-level API for DynamoDB. For Java, I found the functionality of beforeRequest() and afterRequest() hooks, which can facilitate latency injection.
- But I haven't been able to find a similar functionality in Python. I am currently working on this to find other solutions.

Importing these tools as packages: loading code and setting up config

- The Chaos Monkey and Security Monkey tools that I've designed are not application-specific and can be imported to use alongside any AWS application.
- To work with these in a new application, create a lambda function for that application, and upload the code for the appropriate tool using "Upload zip file" option and remove any pre-existing file.
- Now an S3 bucket has to be created by the name of "config–bucket", and add config files "config.txt" for Chaos Monkey and "sgconfig.txt" for Security Monkey.

Importing these tools as packages: setting up policies

- Finally, the requisite permissions have to be granted to the lambda functions.
- For Chaos Monkey's IAM role, "UpdateContainerInstancesState" and "S3AccessPolicy" have to be added.
- For Security Monkey, "AuthorizeSecurityGroupIngress", "RevokeSecurityGroupIngress", "DescribeSecurityGroup", and "S3AccessPolicy" have to be added.
- Now, these experiments are ready to run.

Examples of other Chaos tools and their uses

- Chaos Kong is a tool created by Netflix, which is similar to Chaos Monkey, but simulates entire region downtimes.
- ChaosSlingr is a security chaos engineering tool, which has support for experimenting with ports and IAM roles and policies.
- Chaos toolkit is another tool which supports random ec2 shutdowns, CPU and memory spikes, and also has support for kubernetes.
- And many more....

Thank You