Auto Scaling and Load Balancing Assignment 1 Report

Michael Roddy

23/10/2024

---

1. Introduction

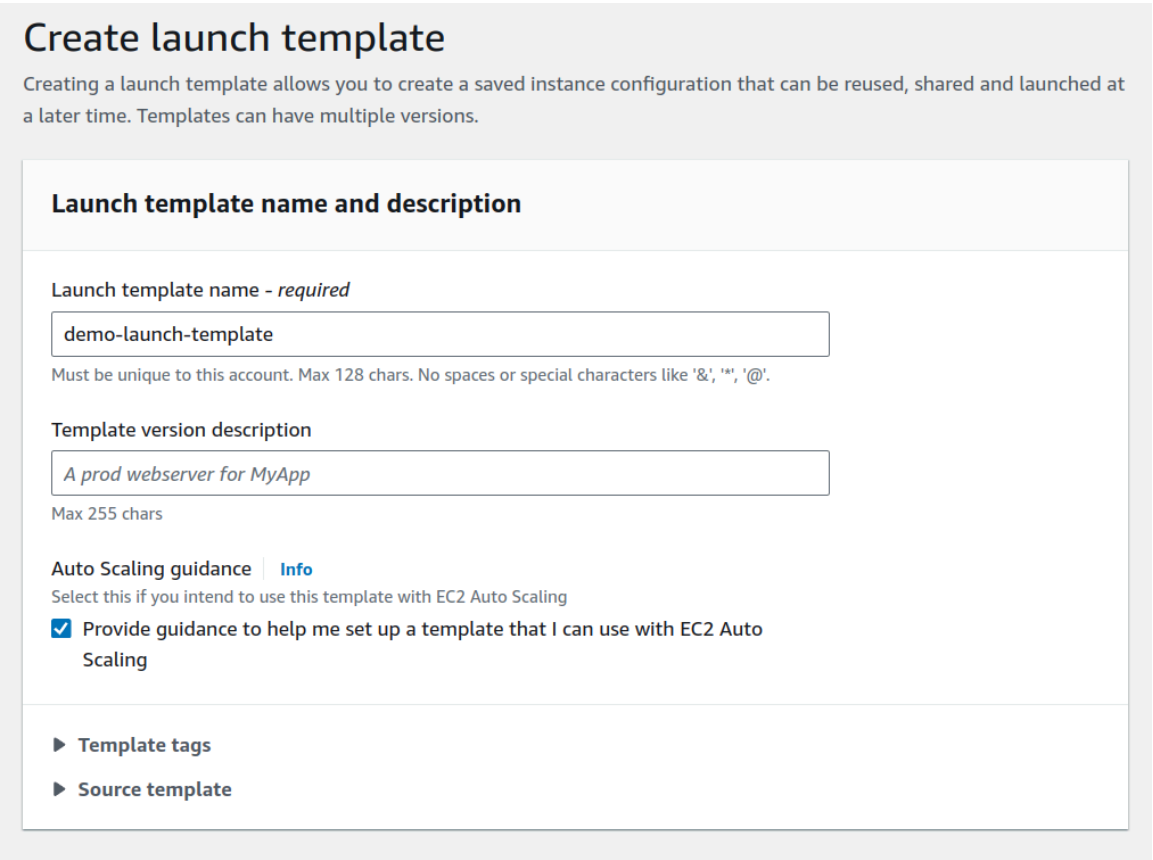
EC2 is essentially a compute service offered by Amazon. It allows you to spin up remote servers or instances in the cloud and use them like you would any other machine. When we say cloud here what we really mean is the EC2 instance is served on some physical hardware in an AWS data center somewhere across the globe. Amazon has data centers all around the world in different regions, for example eu-west-1 which is Ireland. EC2 is typically used to run cloud based microservices. In EC2 you can configure which OS to use like Amazon Linux, Ubuntu or Windows, how much compute power you need for the instance like RAM and CPU etc.. and network rules. EC2 instances can be easily scaled up or down depending on service needs, either by horizontal scaling which is increasing or decreasing the number of instances used to distribute the load across multiple servers or by vertical scaling, which is increasing or decreasing the amount of hardware resources needed to run the service. This makes EC2 very flexible and efficient when determining how to scale your application according to service needs or usage. Scaling in EC2 can be done manually or by auto scaling which allows for automatic scaling of services depending on some configurations and scaling policies set on an auto scaling group in EC2. An auto scaling group is just a collection of instances that is auto scaled up or down according to the policy and that web traffic is routed to using a load balancer such as the Application Load Balancer. This is done to distribute the load evenly and maintain the average instance CPU utilization at a certain percentage. The auto scaling group uses a Launch Template to determine how to configure the instances that are started up, for example you can configure the Launch Template with the OS, hardware type and security groups for the instances.

---

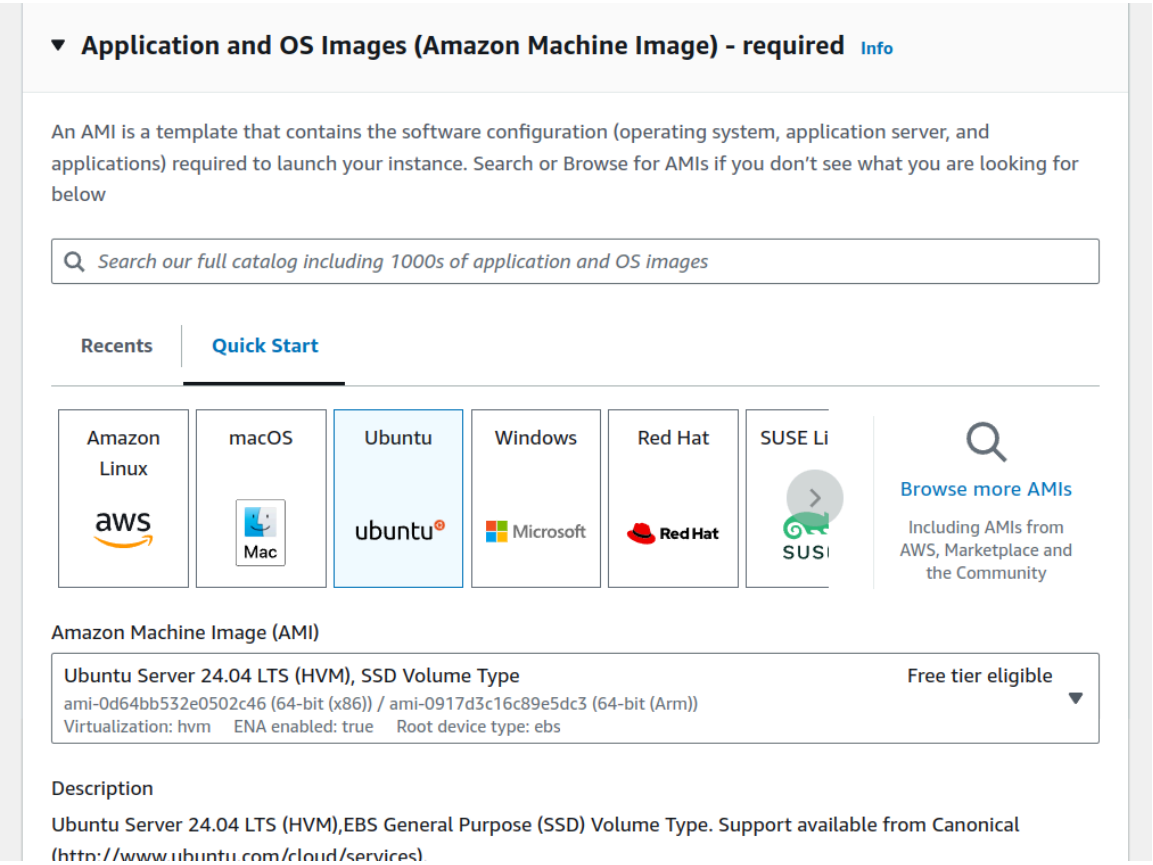
2. Configuration

Configuring a Launch Template

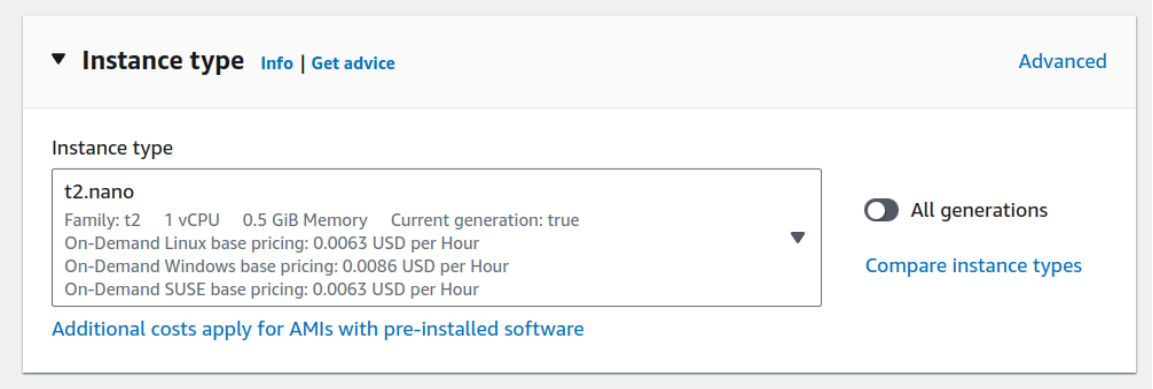
To create a launch template go to the launch template section on the EC2 console and click create template. From here we can set the configurations needed for the template.



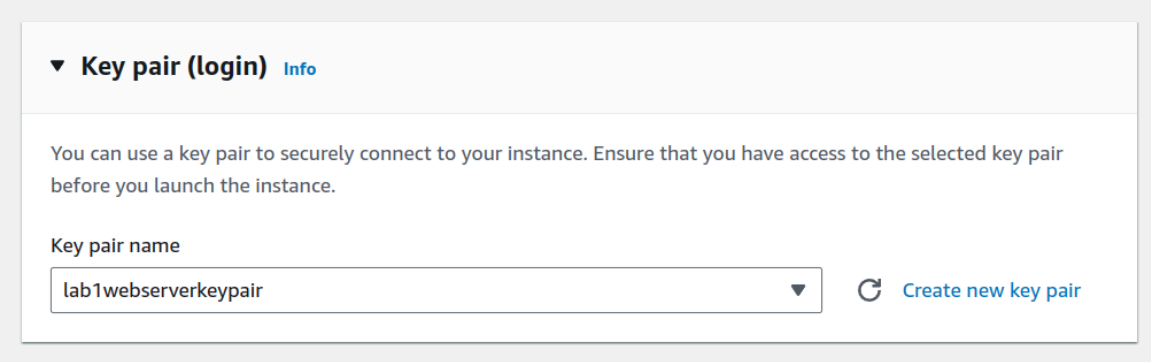
- Set the template name
- Check the auto scaling guidance box to see what sections are required for creating a launch template. This is useful.



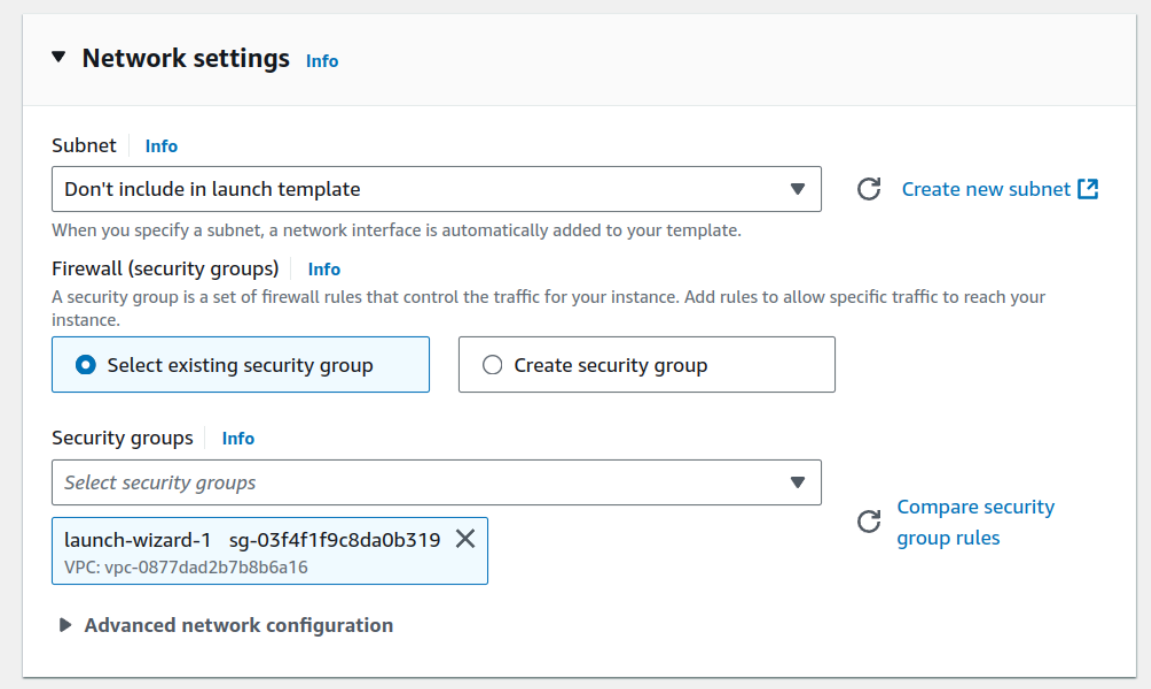
- Here we will set the OS. You can choose from various operating systems but I have chosen the Ubuntu OS. You can choose whatever OS is suitable for your needs or preference.



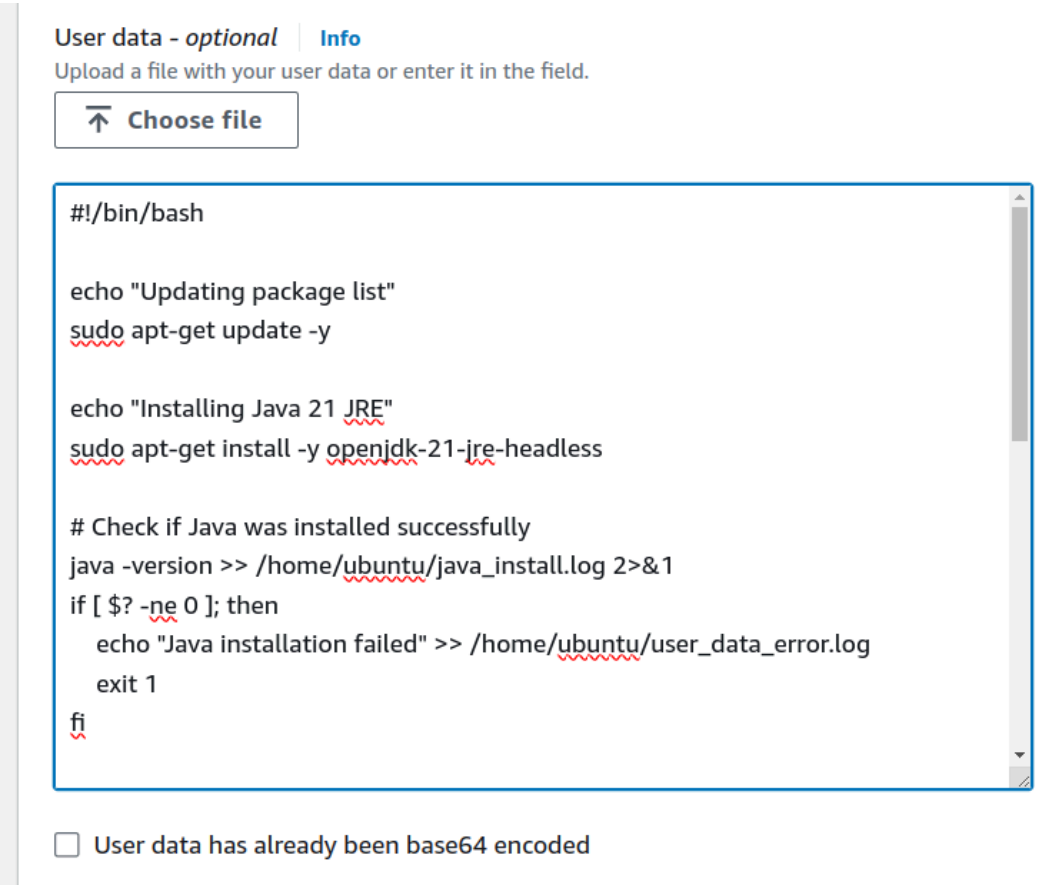
- Set the instance type. This will determine how much hardware resources are allocated for the instance. I have chosen the smallest instance type which is t2.nano.



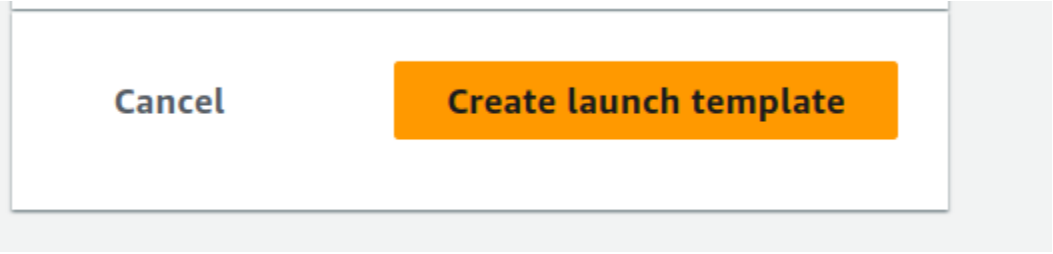
- If you want to connect to the instance over SSH you can set the key pair.
- Here I am using a key pair that I already have configured.
- If you do not have a key pair configured you can create one.



- Set a security group on the template.
- The security group controls inbound and outbound network access to the instance such as the protocol and ports used like port :80 for HTTP :443 for HTTPS or a custom TCP on port :9090 for example.
- Here I have chosen a security group that I previously configured.
- If you do not have a security group you can create one.



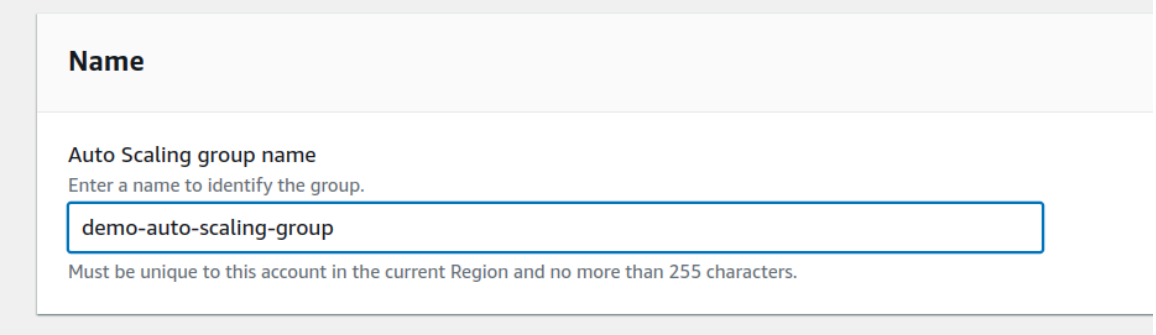
- In advanced settings you can set a user data script which is just a BASH script that will be run when the instance is starting up.
- Here you download and install software or dependencies needed for your service.
- Maybe you might need to reroute network traffic from default ports to custom ports etc.. Whatever it is you need to do to get your service up and running.
- Then you can run the service. e.g java -jar couponservice.jar



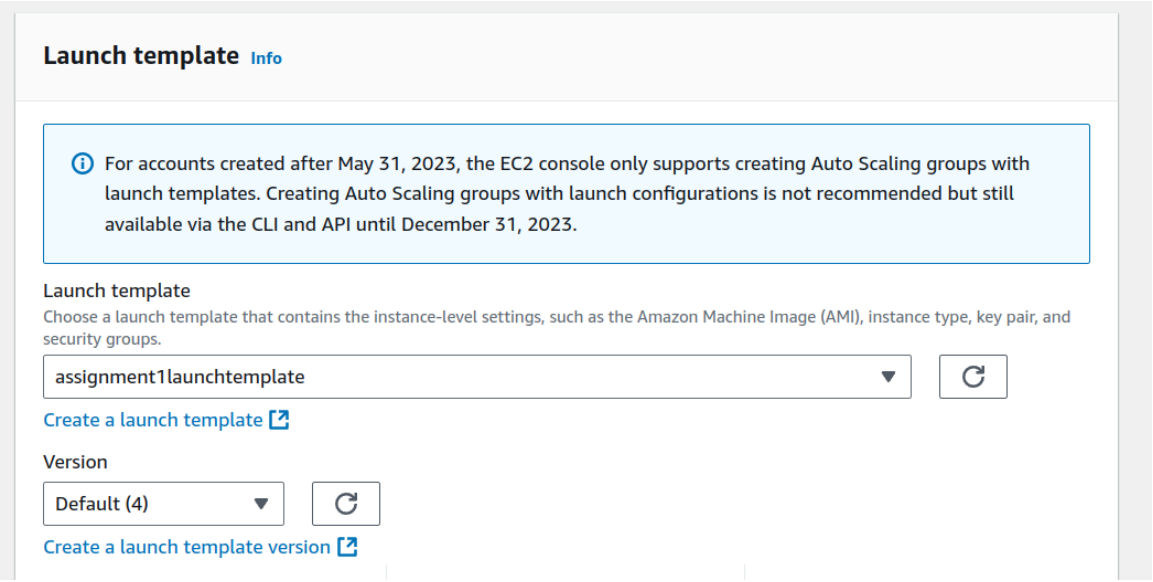
- Now we can click create launch template.
- After this the template should show up in your list of launch templates ready for use with an auto scaling group.

Configuring Auto Scaling

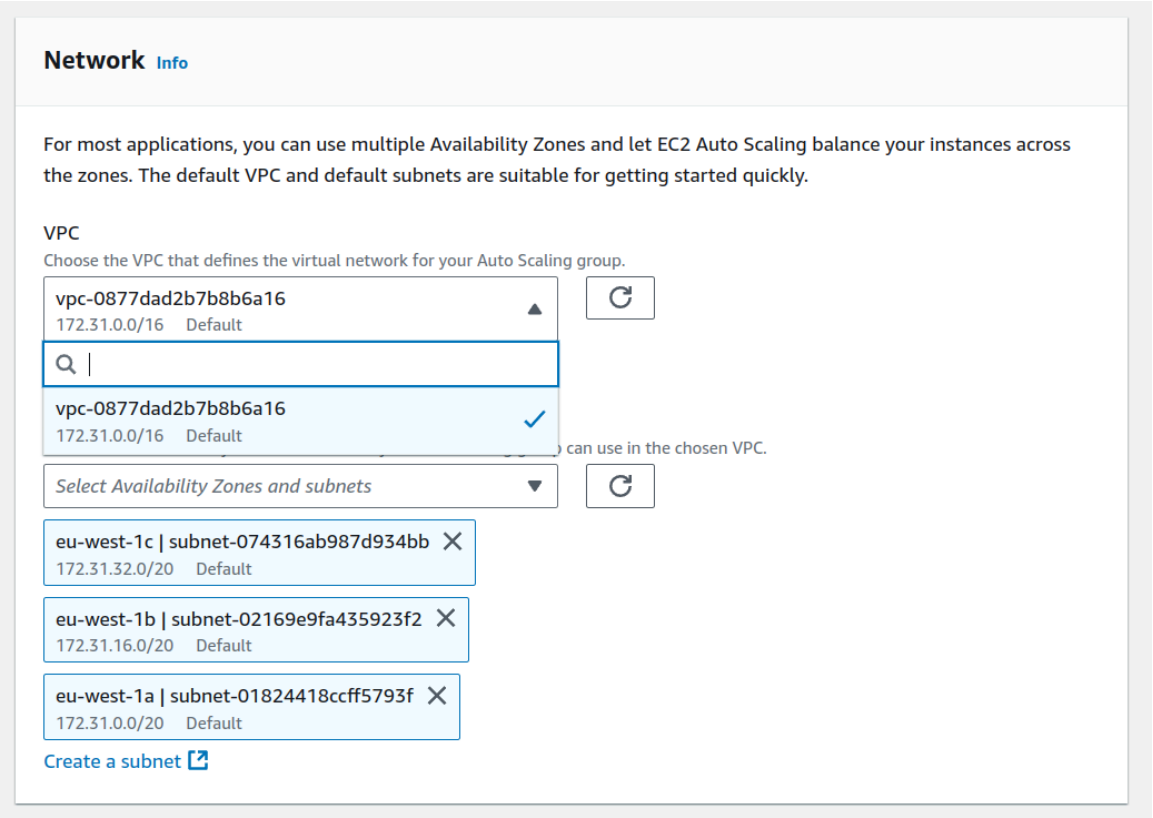
To create an auto scaling group go to the auto scaling tab in the EC2 console and click create an auto scaling group.



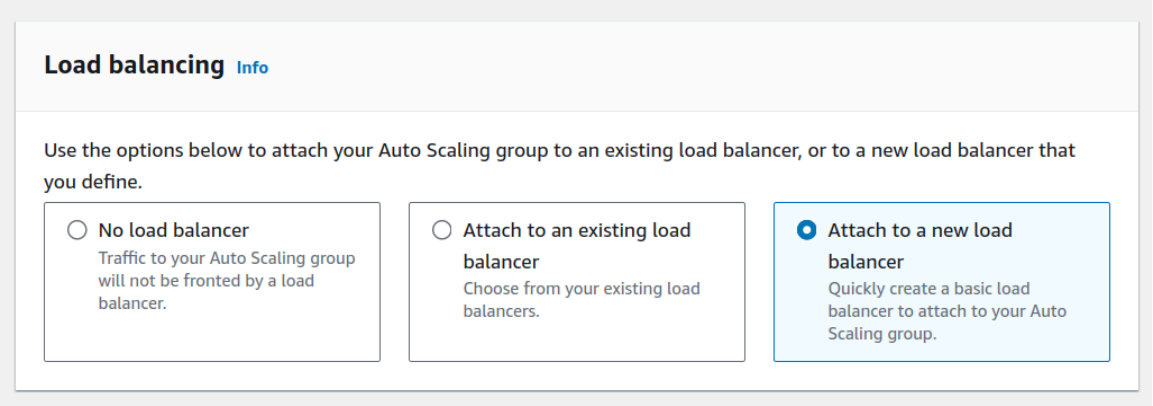
- Set the auto scaling group name.
- Give it a unique name.



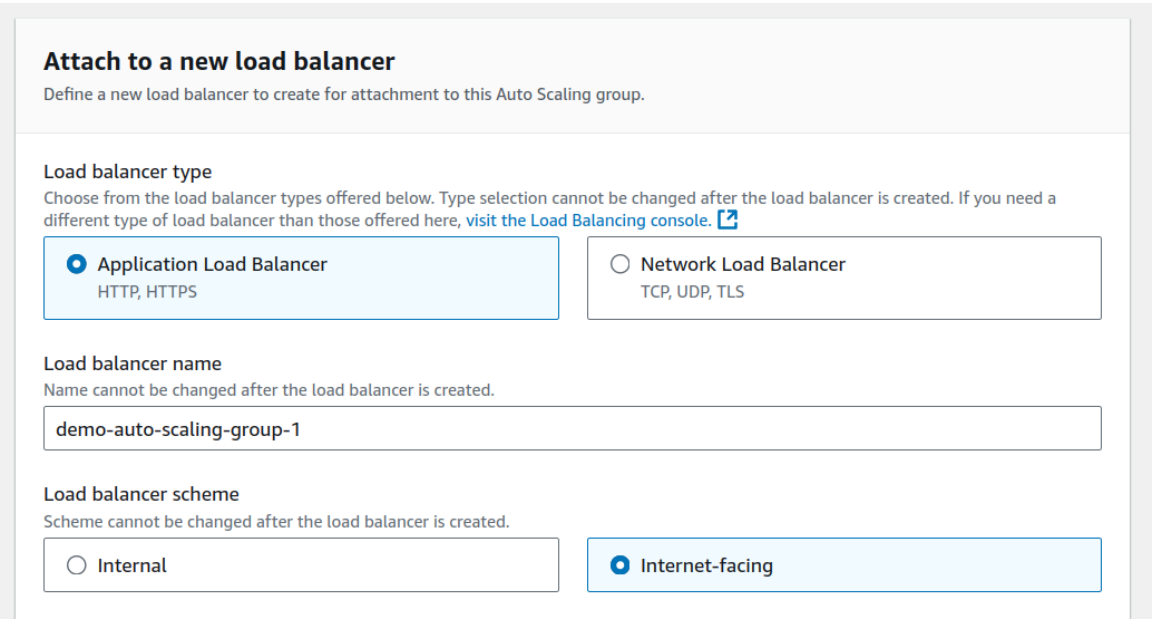
- Choose a launch template to use.
- Here I am choosing a template that I have previously configured.
- Make sure to pick the correct version of the template to use otherwise changes made to the launch template will not be reflected in the instances that are started up because changes made to the template create a new version.



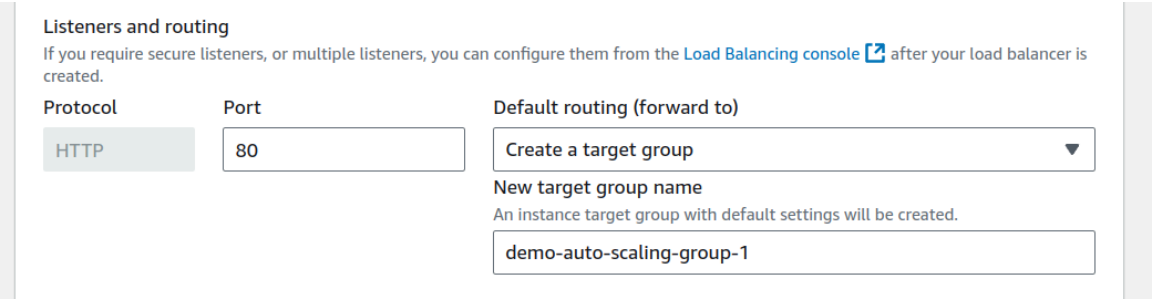
- Choose the VPC and the availability zones/subnets.
- The default options will suffice.



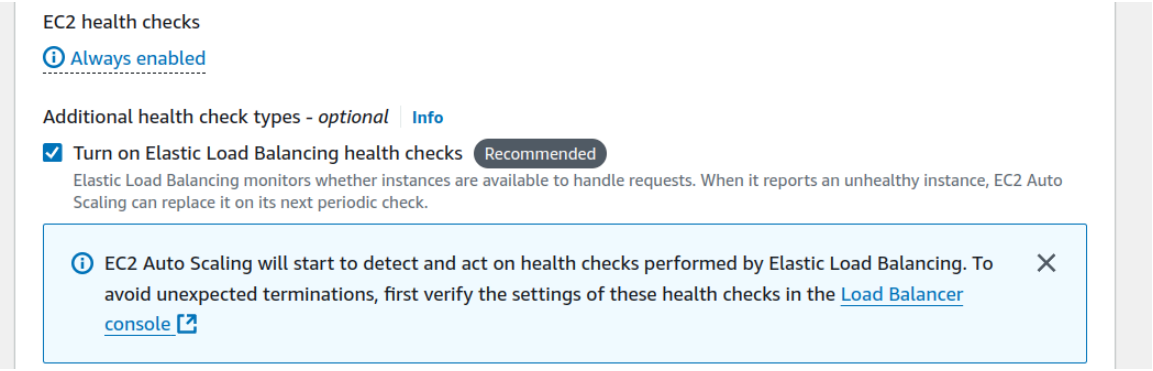
- Attach a load balancer to the auto scaling group.



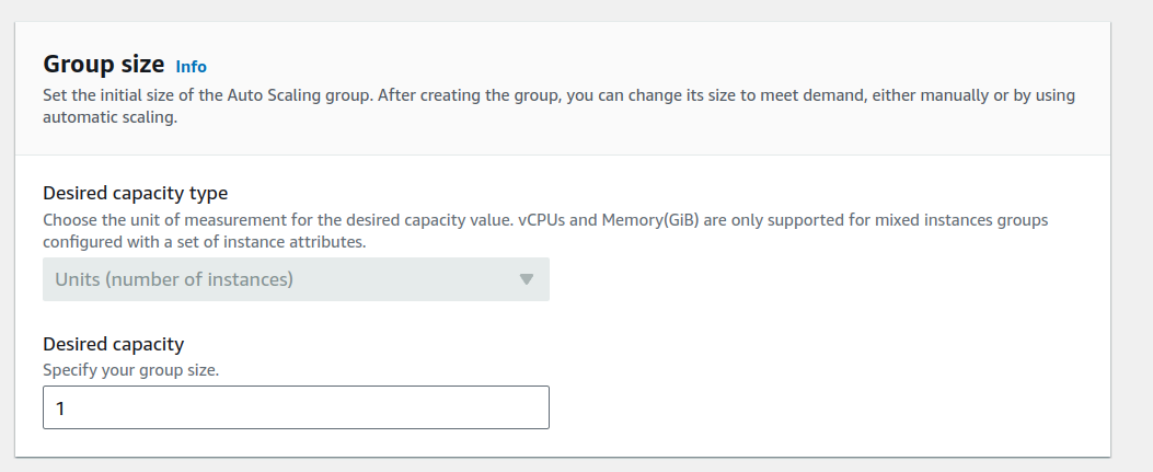
- Choose the Application Load Balancer for HTTP and HTTPS as this type of load balancer is more applicable to our use case. The Network Load Balancer would be used for lower level network requests like UDP or TLS.
- Make sure the load balancer is internet facing for the web.



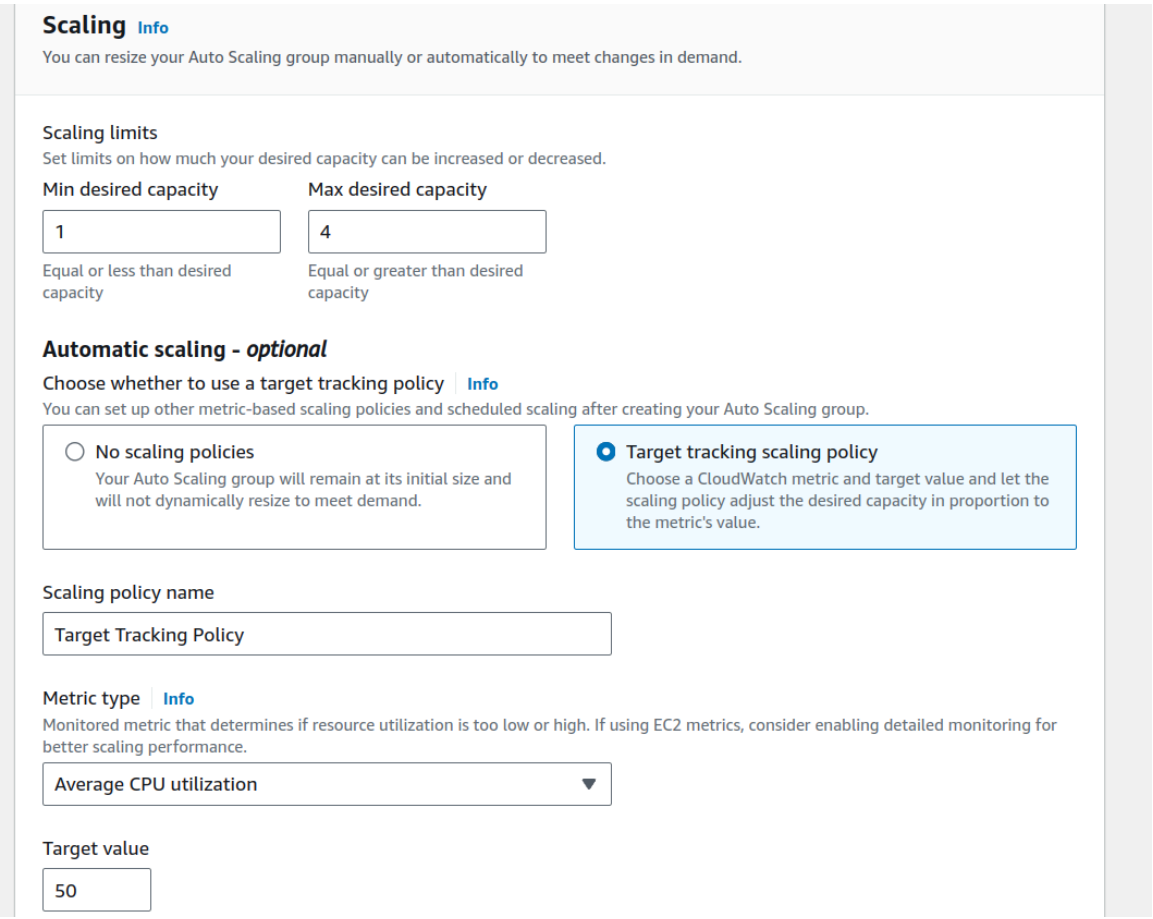
- Choose a target group and port.
- The target group is where the requests are forwarded too.
- If you do not have a target group already you can create one.
- Here requests will be forwarded to targets listening on port :80



- Turn on health checks for the auto scaling group. This is useful.

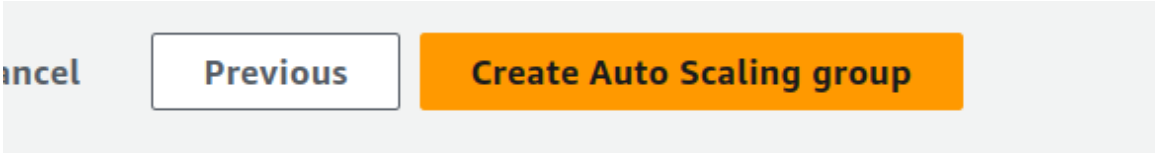


- Set the group size. This is the desired number of running instances.



- Set the minimum desired capacity and the maximum desired capacity for the auto scaling group.
- This setting will determine how much the auto scaling group can scale up or down depending on service needs.

- Here the minimum number of instances running will be 1 and the maximum the maximum number of instances will be 4.
- Choose the target tracking scaling policy. The target tracking policy will allow our auto scaling group to scale based on some numerical metric such as average CPU utilization.
- Here I have set the target value to 50.



- Finally on the review page you can now click create auto scaling group.
- The auto scaling group should now show up in the list of auto scaling groups.

---

3. Test Results

To generate load on our microservice we can read a file 5000 times to apply some CPU intensive operations.

```
@GetMapping(value = "/generateload")
public ResponseEntity<Coupon> generateLoad() {

    ObjectMapper objectMapper = new ObjectMapper();

    Coupon dummyData = new Coupon();

    System.out.println("Generating load..");

    for (int i = 0; i < 5000; i++) {
        try {
            ClassPathResource resource = new ClassPathResource(path:"data.json");
            InputStream inputStream = resource.getInputStream();
            dummyData = objectMapper.readValue(inputStream, valueType:Coupon.class);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    System.out.println("Done");

        return new ResponseEntity<>(dummyData, HttpStatus.OK);
}
```

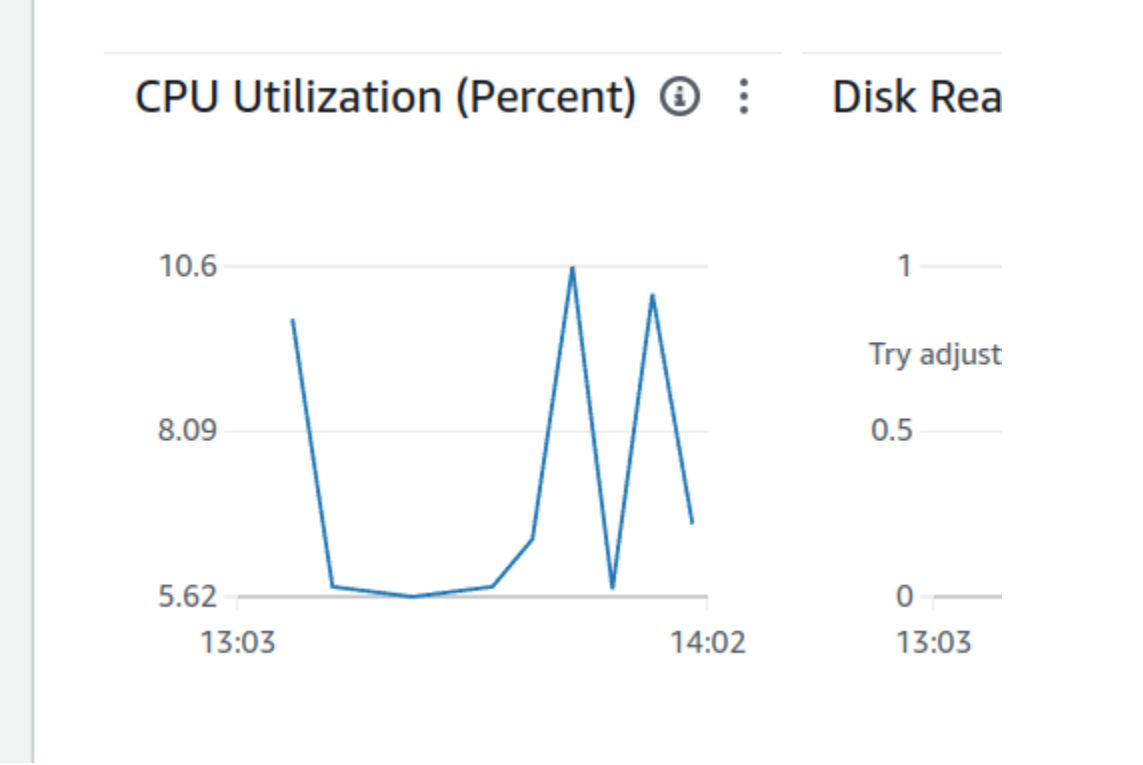
- Given that the instance type is t2.nano this operation should be sufficient in overloading the service if the endpoint is repeatedly called.

send_requests.py

home > michaelroddy > Documents > Masters > Container_Design_And_Deployment > assignment1 > send_requests.py > ...

```python
from concurrent.futures import ThreadPoolExecutor
import requests

url = "http://assignment1autoscalinggroup-1-lb-1645138976.eu-west-1.elb.amazonaws.com/couponapi/generateload"

def send_request(request_id):
    try:
        response = requests.get(url)
        print(f"Request {request_id}: Status Code {response.status_code}")
        print(f"Response: {response.text}\n")
    except requests.exceptions.RequestException as e:
        print(f"Request {request_id} failed: {e}")

total_requests = 5000
concurrent_requests = 5

try:
    with ThreadPoolExecutor(max_workers=concurrent_requests) as executor:
        executor.map(send_request, range(1, total_requests + 1))

except KeyboardInterrupt:
    print("\nProgram terminated by user.")
```

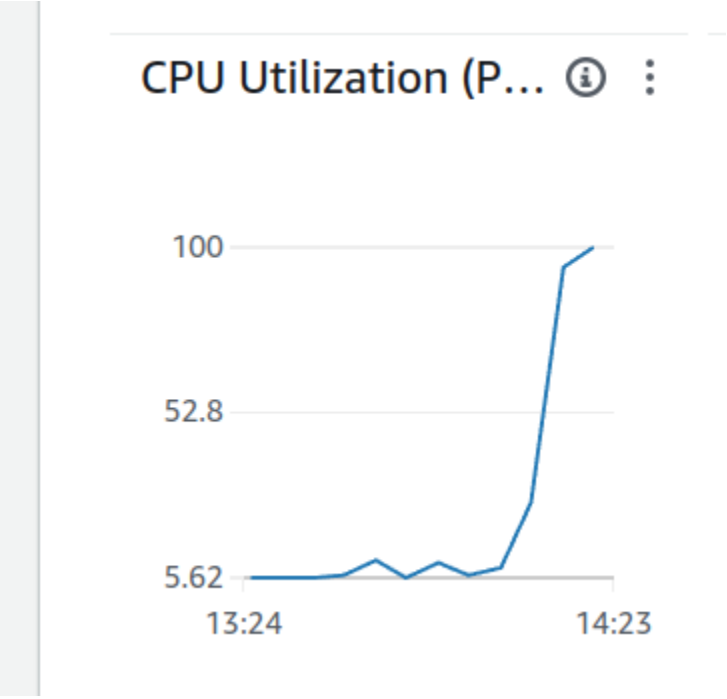
- Here I have a python script that will call the /generateload endpoint and execute 5000 requests with 5 concurrent requests to overload the service.
- We send requests over HTTP to the load balancer's DNS name / our endpoints.



- Here you can see that when the service just started up the CPU utilization was around 10%. I tested the service with a few postman requests.

```
Generating load..
Done
Done
Done
Done
Done
Generating load..
Generating load..
Generating load..
Generating load..
Generating load..
Done
Done
Done
Generating load..
Done
Done
```

- Then I ran the Python script to generate load.
- This is output from the terminal on the EC2 instance so we can see that the generateload() method is being called correctly.



- After some time we can see that the CPU utilization on the instance has increased to 100%.

Status	Description	Cause	Start time
Successful	Launching a new EC2 instance: i-0779e7c9ef546e254	At 2024-10-20T14:25:00Z a monitor alarm TargetTracking-assignment1autoscalinggroup-AlarmHigh-ff9ae4e6-0029-4ef0-a74d-460a26edd573 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 4. At 2024-10-20T14:25:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 4.	2024 October 20, 03:25:08 PM +01:00
Successful	Launching a new EC2 instance: i-0cc680560cadab693	At 2024-10-20T14:25:00Z a monitor alarm TargetTracking-assignment1autoscalinggroup-AlarmHigh-ff9ae4e6-0029-4ef0-a74d-460a26edd573 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 4. At 2024-10-20T14:25:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 4.	2024 October 20, 03:25:08 PM +01:00
Successful	Launching a new EC2 instance: i-06a1e75991d3f3f57	At 2024-10-20T14:19:00Z a monitor alarm TargetTracking-assignment1autoscalinggroup-AlarmHigh-ff9ae4e6-0029-4ef0-a74d-460a26edd573 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2024-10-20T14:19:04Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2024 October 20, 03:19:06 PM +01:00
Successful	Launching a new EC2 instance: i-0277a57d11f08b51c	At 2024-10-20T13:10:10Z a user request update of AutoScalingGroup constraints to min: 1, max: 4, desired: 1 changing the desired capacity from 0 to 1. At 2024-10-20T13:10:16Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 1.	2024 October 20, 02:10:17 PM +01:00

- After some more time has passed and as the initial service became overloaded with requests, you can see that the auto scaling group has started up 3 more EC2 instances bringing us to the desired max capacity of 4 instances running.

Name	Instance ID	Instance state	Instance type	Status check
	i-0cc680560cadab693	Running	t2.nano	Initializing
lab1webserver	i-085298e65593c2759	Stopped	t2.nano	–
	i-06a1e75991d3f3f57	Running	t2.nano	2/2 checks passec
assignment1m…	i-0262ebc2260d4bb4a	Running	t2.micro	2/2 checks passec
	i-0779e7c9ef546e254	Running	t2.nano	Initializing
	i-0277a57d11f08b51c	Running	t2.nano	2/2 checks passec

- These are the instances that are currently active (the instance with name assignment1m… is an EC2 instance running a mysql database that each of the microservices connects to).

```
*** System restart required ***
Last login: Sun Oct 20 13:49:47 2024 from 18.202.216.51
ubuntu@ip-172-31-31-190:~$ sudo systemctl status mysql
Warning: The unit file, source configuration file or drop-ins of my
● mysql.service - MySQL Community Server
     Loaded: loaded (/usr/lib/systemd/system/mysql.service; enable
     Active: active (running) since Sun 2024-10-20 14:24:50 UTC; 1h
    Process: 2124 ExecStartPre=/usr/share/mysql/mysql-systemd-start
   Main PID: 2133 (mysqld)
     Status: "Server is operational"
      Tasks: 88 (limit: 1130)
     Memory: 384.0M (peak: 407.0M)
        CPU: 21.909s
     CGroup: /system.slice/mysql.service
             └─2133 /usr/sbin/mysqld
```

- This is the centralized mysql database up and running on an EC2 instance.

---

4. Evaluation

I achieved auto scaling correctly. Basically the idea is to distribute incoming traffic across multiple instances and balance the load accordingly rather than having 1 instance do all the work or risk becoming overloaded. This allows for horizontal auto scaling up or down depending on service needs, for example if you happen to experience a rush in users on an application that is using a particular service and the amount of network traffic increases significantly, service resources will need to increase to meet the demands or else the server could become overloaded and experience an outage. This is the problem that auto scaling helps us to solve. This sounds great but there may be disadvantages to auto scaling as well as the setup is lengthy and depends quite a bit on certain configurations that are set correctly, so for example if the scaling policy isn't set correctly this could lead to instances being started unnecessarily thus increasing costs. Also, there may be some performance degradation experienced when instances are being started up as they require time to initialize.

For this assignment I used a 1 centralized mysql database on a separate EC2 instance.

---

5. References

- https://www.youtube.com/watch?v=cf9jQc4xzpo&t=358s (Academind)
- https://docs.aws.amazon.com/ec2/ (AWS)