Practical: Application load balancer and auto scaling groups

Table of Contents

AWS EC2 includes functionality that allows you to automatically add or remove instances and distribute load between them. We will explore how this works to create a simple load-balanced web service that automatically scales in and out in response to load.

Prerequisites and references

- **AWS Application Load Balancers (https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html)**
- **AWS EC2 auto scaling (https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html)**
- **Practical: Using EC2 Launch Templates and AMIs (https://canvas.qut.edu.au/courses/20367/pages/practical-using-ec2-launch-templates-and-amis-2)**

Walk-through video

CAB432 Autoscaling Practical

Practical: Application load balan… scaling groups

Table of Contents

- Prerequisites and references
- Overview of construction
- Creating the application AMI
- Creating the Launch template
- Creating the target group
- Creating the application load balancer
- Creating the auto scaling group
- Testing autoscaling
  - Set up the testing script
  - Testing scale-out
  - Testing scale-in

0:00 / 1:01:35 1x

Search Hide transcript

Hello. Welcome to Week ten, and this week, we're going to have a look at using auto scaling groups and load balancers in AWS to create an automatically scaling application. So there's a bunch of parts to make this go. The main plan here is that we're going to, first of all, configure our virtual machine, which is going to house the

Overview of construction

We'll be using several different AWS services together to build our application:

- **Virtual machine image (AMI)**: this is the virtual machine image that contains our application, with the application started on boot.
- **Launch template**: this contains all the necessary information for how to launch an EC2 instance for our application.
- **Target group**: this is a set of EC2 instances that are running our application. It is used to organise the instances so that the load balancer knows what instances to send requests to.
- **Application load balancer**: this distributes HTTP requests amongst the EC2 instances in the target group.
- **Auto-scaling group**: this monitors a target metric on the target group (eg. average CPU load) and adds or removes EC2 instances from the target group to maintain a consistent metric.

Here's the plan:

- Create a simple web server with a synthetic CPU load so that we can load an EC2 instance's CPU down. Set up an AMI and launch template for it.
- Create a target group and application load balancer to distribute requests amongst multiple instances of our server
- Create an auto scaling group for the target group that scales the application in and out (adding or removing instance of our server) to maintain an average CPU utilisation of about 50%
- Run a load testing script to send multiple requests and load down the application. This will increase the CPU utilisation and trigger the auto-scaling group to scale out.
- Stop the script. The CPU load will go down and the auto-scaling group will scale in.

This demonstrates all the structures required to achieve automatic horizontal scaling in response to load.

Creating the application AMI

Before we can do anything else we need an application running on an EC2 instance. We'll use a simple service that generates a lot of CPU load by calculating a large number of logarithms before responding.

- Create an EC2 instance in the usual way, but take care to use these settings:
  - For instance type choose `t2.micro`
  - In the *Advanced settings* section
  - Under *Detailed CloudWatch monitoring* choose *Enable*. This means AWS will measure the average CPU utilisation every minute, where the default is every 5 minutes.
  - Under *Credit specification* choose *Unlimited*. This means that our `t2.micro` instance will always allow full utilisation of the CPU, where the *Standard* only allows a burst to 100% for a short time, then limits to 15%.

Next we will create our little server. We'll use JavaScript. Something similar could be done with Python, but the language is not important for our purposes so we will only show the JavaScript version. Note that our method of artificially generating load is not acceptable for the assessment; your project should generate load in a way that is meaningful to your application.

- Log in to the EC2 instance over ssh or the session manager. If using the session manager ensure that you change to the `ubuntu` user with `sudo -iu ubuntu`.
- Install node as in **Practical: Node.js application on EC2 (https://canvas.qut.edu.au/courses/20367/pages/practical-node-dot-js-application-on-ec2)**.
- `md server` then `cd server` to create a folder for our little server. Then `nano index.js` to create our server file and fill with the following contents:

```
const http = require("http");
const port = 3000;
const MAX = 1000000000;
const server = http.createServer((req, res) => {
   let start = Date.now();
   for (let i = 0; i <= MAX; i++) {
      let dummy = Math.log(i + 1);
```

```
    }
    let timing = Date.now() - start;
    res.statusCode = 200;

    res.setHeader("Content-Type", "text/plain");
    res.end("Hello World\n" + timing + "(ms)\n");
});

server.listen(port, () => {
    console.log(`Server listening on ${port}`);
});
```

- Create a small bash script to start up the server. While still in the `~/server` directory, do `nano start.sh` and populate with the following contents:

```
#!/bin/bash

# Assuming node is installed with nvm, we need to set up the environm
ent
# variables and set the PATH to be able to run node
export NVM_DIR="$HOME/.nvm"
. /home/ubuntu/.nvm/nvm.sh

cd /home/ubuntu/server
node index.js
```

- `chmod +x start.sh` to make the script executable.
- `./start.sh` to run the script.
- Make sure that the server is working by going to `http://<your EC2 public DNS name>:3000`. You should see `Hello World` plus some timing information.
- Press Ctrl-C on your EC2 instance terminal to stop the script and the web server.

Next we need to set up a system service to start up our web server. We'll use `systemd` which is the main system for managing system services on Ubuntu.

- `sudo nano /etc/systemd/system/startnode.service` to create a new file and fill with these contents:

```
[Unit]
Description=Node service
After=network.target

[Service]
ExecStart=/home/ubuntu/server/start.sh
StandardOutput=/home/ubuntu/server.log
StandardError=/home/ubuntu/server.err
User=ubuntu
Group=ubuntu

[Install]
WantedBy=multi-user.target
```

- `sudo systemctl daemon-reload` to reload the service specifications so that `systemd` knows about our new service
- `sudo systemctl enable startnode` to tell `systemd` to start our service on boot
- `sudo systemctl start startnode` to tell `systemd` to start our service now
- `sudo systemctl status startnode` to check the logs and make sure that the service started up. Look for any errors.
- Check the server is running from your web browser again.
- Reboot the EC2 instance (`sudo reboot` or use the AWS console). After a minute or two, check the server again using the web browser.
- If the server is not running, do `sudo systemctl status startnode` on the EC2 instance to check for errors.

Our application is ready to go! Now create an AMI as in Practical: Using EC2 Launch Templates and AMIs.

Keep your EC2 instance for the next step.

Creating the Launch template

The AMI that you just created is the image of the OS and software for your application EC2 instance. A Launch template is the configuration for the virtual machine. Later, the auto-scaling group will use a launch template to launch new EC2 instances when scaling out, so we will need to create one.

Follow the steps for creating a Launch Template from your EC2 instance as in Practical: Using EC2 Launch Templates and AMIs, but make the following adjustments to the configuration:

- For instance type, choose `t2.micro`. These have only one CPU so they are easier to load down. In particular, Node is single-threaded, so it will only ever load down one CPU at a time.
- In Advanced settings:
  - Enable *detailed monitoring*. With this enabled AWS will collect statistics (notably, CPU utilisation) every 1 minute instead of every 5 minutes. Without this scaling out will take a *long* time.
  - Make sure that *credit specification* is set to *unlimited*. The *standard* option uses a bursting mode which limits the CPU utilisation to 15% after a short burst of high CPU usage. This would interfere with our attempts to load the CPU down to trigger auto-scaling.

Creating the target group

A *target group* is AWS's way of organising EC2 instances (and other resources that we won't use here) into a set that a load balancer can direct requests to.

- Search for *Target groups* in the AWS console.
- Click *Create target group* in the top right of the list of target groups.
- Give your target group a name like `n1234567-autoscale-practical`

- Under *Protocol:Port* change the port number to 3000 to match the port that our server is listening on.
- Under *VPC* select the `aws-controltower-VPC`
- Under *Advanced health check settings*, set the *Timeout* to 20 seconds.
- Add tags for `qut-username` with value like `n1234567@qut.edu.au` and `purpose` with value `practical`.
- *Create target group*

We'll use this target group in the next section.

Creating the application load balancer

AWS has several types of load balancer. We'll use a *Application load balancer* which understands HTTP. This means that we can use it for additional functionality such as having multiple routes corresponding to different target groups, similar to how an API Gateway works. We can also add TLS support.

- Search for *Load Balancers* in the AWS console
- Click *Create load balancer* in the top right of the list
- Click *Create* under *Application Load Balancer*
- Give your ALB a name like `n1234567-autoscale-practical`
- Under *VPC* select `aws-controltower-VPC`
- Select all three availability zones
- For each availability zone select the subnet named like `aws-controltower-PublicSubnetN` (where `N` is 1, 2 or 3.)
- For *Security group* select *CAB432SG*.
- Under *Listeners and routing* for the *Listener HTTP:80* rule, set the *Default action* to the target group that you created in the previous step.
- Add *Load balancer tags* with key `qut-username` and value set to your username like `n1234567@qut.edu.au`, and key `purpose` set to `practical`.
- *Create load balancer*

Your load balancer is now set up to listen on port 80 and forward packets to EC2 instances in your target group.

Later on you'll need the *DNS name* for your ALB. Find your ALB in the list of load balancers, click on it to go to its details page, and copy the DNS name. Keep this handy for later.

If you try going to your ALB's DNS name on your browser at this point you will see an error because there are no EC2 instances in the target group.

Creating the auto scaling group

We don't want to manually add and remove EC2 instance to the target group. Instead we'll set up an auto scaling group to handle that for us in response to load.

First, we set up the basics and networking:

- Find *Auto Scaling groups* in the AWS console by going to the EC2 service and scrolling down to the bottom of the sidebar.
- Click *Create Auto Scaling group* in the top right of the list
- Add a name like `n1234567-autoscaling-practical`
- Under *Launch template* select the launch template that you created earlier.
- Note that if you edited your launch template then you will have more than one version. Make sure that you have selected the correct version. The default version does not change when you create a new version.
- *Next*
- Under *VPC* select `aws-controltower-VPC`
- Under *Availability Zones and subnets* select the three subnets like `aws-controltower-PublicSubnetN` for `N` equal to 1, 2, 3.
- *Next*

Now we need to connect things up to the load balancer and target group.

- Select *Attach to an existing load balancer*
- Under *Existing load balancer target groups* select the target group that you created earlier
- Under *Health checks* select *Turn on Elastic Load Balancing health checks*
- *Next*

Here we create the scaling policy, which determines how EC2 instance are added and removed in response to load.

- Under *Scaling*
  - Set *Max desired capacity* to `3`
  - Select *Target tracking scaling policy*.
  - Set *Target value* to `80`. This means that the auto-scale group will add or remove EC2 instances to try to keep the average CPU utilisation around 80%.
  - Under *Instance warmup* enter 90. This is the number of seconds that is given for a new instance to boot up before routing traffic to it. In our testing this takes about 55 seconds, so we are using a conservative value here.
- *Next*
- *Next*

And finally, we make sure that we add the proper tags.

- Add tags with key `qut-username` and value set to your username like `n1234567@qut.edu.au`, and key `purpose` set to `practical`. Ensure that the *Tag new instances* is selected for both tags.
- *Next*
- *Create Auto Scaling group*

At this point your auto scaling group should launch an EC2 instance in your target group.

- Go to the details page for your auto scaling group by finding it in the list and clicking on its name.

- Click on the *Activity* tab. You should see an event in the history for launching a new EC2 instance
- Click on the *Instance management* tab to see the list of instances.
- Click on the *Monitoring* tab to see graphs for the number of instances. *Desired capacity* indicates the number of instances that the auto scaling group wants to have. *In Service Instances* is the number of instances that are ready to receive traffic. These may be different if an EC2 instance has just been launched and is not ready yet.
- Click on the *EC2* tab within the monitoring tab. Here you'll see the *CPU Utilization*, which is averaged across the instances in the auto scaling group.

By default the scaling policy has a *cool down* period, set to 5 minutes. After scaling in or out, it won't make further changes for this amount of time. With a cool down of 5 minutes it will take a long time to scale back in. You can change this after creating your auto scaling group by editing the *Advanced configurations* in the details page for your group.

Testing autoscaling

Now we'll do some testing to see how the load balancer and auto scaling group respond.

Set up the testing script

To test autoscaling we'll use a simple node script. You can run this from your own computer or from a separate EC2 instance. Edit the `endpoint` variable with your ALB instance public DNS name. We'll call this `loadtest.js`.

```
const endpoint = "http://";
const numberOfRequests = 10000;
const targetResponseTime = 1800;
const targetTimeHysteresis = 1.2;
const minTargetConcurrentRequests = 2;
const maxTargetConcurrentRequests = 8;
const rollingAveragePastWeight = 0.95;
const scaleoutTime = 10000;


const rollingAverageCurrentWeight = 1 - rollingAveragePastWeight;
var currentRequests = 0;
var targetConcrurentRequests = minTargetConcurrentRequests;
var rollingAverage = targetResponseTime;
var lastScaleoutTime = performance.now();

// Helper function to pause for a time but keep the event loop free t
o do other things
function sleep(ms) {
   return new Promise((resolve) => {
      setTimeout(resolve, ms);
   });
}
```

```
// Helper function to create a non-blocking fetch request which repor
ts timing
function makeRequest(requestNumber) {
   currentRequests += 1;
   return new Promise((res) => {
      console.log(`Request ${requestNumber} started, currently ${curr
entRequests} outstanding, target ${targetConcrurentRequests} request
s.`);
      const startTime = performance.now();
      fetch(endpoint, { method: "GET" }).then((res) => {
         if (!res.ok) {
            console.error(
               `Request ${requestNumber} failed with status: ${res.st
atus}`
            );
            currentRequests -= 1;
            return;
         }

         // Calculate a rolling average, weighted towards the most re
cent requests
         const responseTime = performance.now() - startTime;
         rollingAverage = rollingAverage * rollingAveragePastWeight +
responseTime * rollingAverageCurrentWeight;

         // Print out some status information
         console.log(
            `Request ${requestNumber} completed in ${responseTime.toF
ixed(2)}ms, rolling average ${rollingAverage.toFixed(2)}ms.`
         );

         // Scale the target number of concurrent requests to keep th
e rolling average close to the target response time
         // Limit how quickly we scale
         if (performance.now() > scaleoutTime + lastScaleoutTime) {
            // Reduce concurrent requests if rolling average is too b
ig
            if (currentRequests <= targetConcrurentRequests && rollin
gAverage > targetResponseTime * targetTimeHysteresis) {
               targetConcrurentRequests -= 1;
               lastScaleoutTime = performance.now();
               if (targetConcrurentRequests < minTargetConcurrentRequ
ests) {
                  targetConcrurentRequests = minTargetConcurrentReque
sts;
               }
            // Increase concurrent requests if rolling average is too
l low
            } else if (currentRequests >= targetConcrurentRequests &&
rollingAverage < targetResponseTime  / targetTimeHysteresis) {
               targetConcrurentRequests += 1;
               lastScaleoutTime = performance.now();
               if (targetConcrurentRequests > maxTargetConcurrentRequ
ests) {
```

```
                        targetConcrurentRequests = maxTargetConcurrentReque
sts;
                    }
                }
            }

            currentRequests -= 1;

        });
    });
}

// Make a bunch of non-blocking requests, up to the target number of
concurrent requests
async function loadTest() {
    for (let i = 0; i < numberOfRequests; i++) {
        makeRequest(i);

        while (currentRequests >= targetConcrurentRequests ) {
                await sleep(10);
        }
    }
}

loadTest();
```

This script will generate multiple concurrent requests to the load balancer, automatically adjusting the number of concurrent requests to max out the CPU(s) without overloading the EC2 instances to the point where they get marked as unhealthy by the health checks.

Testing scale-out

We've set up the auto scaling group to respond to high CPU load by launching more EC2 instances in the target group. We'll test this now.

- Run the script with `node loadtest.js`
- Open the AWS Console to your auto scaling group details page. Go to the *Monitoring* tab and watch *Desired capacity*. After a few minutes the auto scaling group will add another instance.
- The new instance will take some time to boot up. It will be marked as unhealthy until the warm-up period is over, at which point the ALB will start sending traffic to it.
- Watch the auto scaling group monitoring tab. A 3rd instance should be added.

If you don't see scaling out happening, check the following:

- Detailed monitoring is turned on in the Launch Template. Without this the CPU utilisation is only collected every 5 minutes, so things will happen very slowly.
- Credit specification is set to *unlimited* in the Launch Template. Without this the CPU will run out of boost credits and be limited to about 15%.
- You can check the average CPU utilisation in the auto scaling group monitoring tab.

- If you made changes to the Launch Template, first ensure that you set the default version to the new version that you create. Then you can trigger the auto scaling group to create new instances by setting max, min and desired number of instances to 0. Wait for the instance(s) to stop, then set the max, min and desired values back to their previous values.

Testing scale-in

Scaling in will happen when the auto scaling group detects that removing an instance will bring the CPU utilisation closer to the target value.

- We'll assume that at this point your auto scaling group has already scaled out to 3 instances.
- Stop the load testing script.
- Wait several minutes while the auto scaling group removes two instance in response to the lower CPU utilisation.
- Scaling in can take a long time (5 to 10 minutes), so be patient.

Interpreting errors

The ALB will return 5xx error codes when there is a problem with the target group:

- **502 Bad gateway:** this indicates that the target (EC2 instance handling the request) sent an unexpected response, such as closing the connection early.
- **503 Service unavailable:** this indicates that the target group is empty; there are no healthy EC2 instances to send the request to. This might happen if your EC2 instance(s) has not responded to health checks, in which case it is marked as unhealthy.
- **504 Gateway timeout:** this indicates that the target didn't respond within the timeout period (10 seconds). This is the error that you are most likely to see during this practical. It happens because the target is overloaded with responses and can't serve requests fast enough.
- On the ALB details page you can go to the monitoring tab and see various metrics including how many of each type of error has occurred recently.

Lights out policy

Since you won't need to have your app up continuously, the CAB432 AWS account implements a policy which sets the desired, min, and max number of instances to 0 every night. If you want to start your app up again, just change these back to 1, 1 and 3.

To explore on your own (optional)

You can add additional listeners and rules to your ALB to mimic API gateway functionality.

- On the details page for your ALB, in the *Listeners and rules* tab.
- Select the *HTTP:80* listener and then click on *Manage rules* then *edit rules*.
- You'll see a list of rules, which only has the default rule set.
- Click *Add rule*
- Give your rule a name and click *Next*
- Click *Add condition*

- Explore the different types of conditions that you can set up, such as path, query type, and so on. Click *Next* after adding at least one condition.
- Explore the types of routing actions. You can route traffic to a separate target group, to another URL, or return a fixed response.

You can use this functionality in your project to route client requests to separate microservices. Note that you can also route to a target group representing a Lambda.