Show Submission Credentials

# P1. Elasticity Horizontal Scaling and Advanced Resource Scaling

- ✔ Introduction
- ✔ AWS Horizontal Scaling
- ✔ AWS Autoscaling
- ✔ AWS Autoscaling with Terraform
- ✔ Project Reflection Task (Mandatory, graded)
- ✔ Success
- ✔ Project Discussion

Introduction

## Introduction to Resource Scaling

Information

### Learning Objectives

This project will encompass the following learning objectives:

1. Design solutions and invoke cloud APIs to programmatically provision and deprovision cloud resources for a dynamic load.
2. Configure and deploy an Elastic Load Balancer and an Auto Scaling Group on AWS.
3. Develop solutions that monitor cloud resource metrics to manage cloud resources with the ability to deal with resource failure.
4. Analyze a workload pattern and develop elasticity policies to maintain the Quality of Service (QoS) of a web service.
5. Account for cost as a constraint when provisioning cloud resources and analyze the performance tradeoffs due to budget restrictions.
6. Orchestrate infrastructure on the cloud using Terraform as part of the deployment process.
7. Recognize the utility of cloud SDK's & CLI's and navigate through documentation pages for insightful information.

Information

#### General Details

The following table contains some general information about this project:

| | |
|---|---|
| Prerequisites | Java 8 or Python 3 |
| Primers | Git Best Practices; Intro to Maven and Checkstyle (for Java user); Jupyter Notebook (for Python user); Amazon Web Services APIs; Infrastructure as Code |

| | |
|---|---|
| Applicable Languages | We recommend that you use *Java 8 (JDK 1.8 preferred) and Python 3* so that the teaching staff can support you. Otherwise, you may use any language that your AWS instance supports. However, the teaching staff may not be able to help you resolve problems in other languages. Note: If you use Maven, the *Maven central repository* is the only remote repository you are allowed to use. |
| Applicable Cloud Platform | Amazon Web Service (AWS) |

Information

## Grade Distribution

| | |
|---|---|
| AWS Horizontal Scaling (Task 1) | 35 points |
| AWS Auto Scaling (Task 2) | 40 points |
| AWS Auto Scaling using Terraform (Task 3) | 10 points |
| Manual Grading | 10 points |
| Reflection and Discussion | 5 points |

**Danger**

## Intense Project Warning!

You should start this project early to avoid any potential stress or roadblocks before the deadline. This project requires an understanding of cloud APIs in AWS as well as iterative development, tuning and/or testing cycles where some tests can take ~40 minutes to complete per run. Learning and using new APIs can be time-consuming due to the lack of consistent documentation and examples in the real world. Therefore, we recommend that you start the project as early as possible to be successful.

**Danger**

## Accessing Course Resources

Before you provision any resources, make sure that you have updated your 12-digit AWS account ID in your course profile. Failing to do so will get your login blocked when using the student VM image provided by us.

**Warning**: Make sure that your instance limits for the following types of instances are at least 20 `m5.large` instance type. You can check your EC2 service limits on the console (https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Limits:). If you are facing an issue with the vCPUs limit, please **contact Amazon** to increase the limit (https://console.aws.amazon.com/support/v1#/case/create?issueType=service-limit-increase&limitType=service-code-ec2-instances) instead of posting this issue on Piazza. Amazon customer service will handle your request in a timely fashion (typically in a few days).

Information

## Cloud Resource Details

The following table contains information regarding AWS resource tagging and the technologies used in this project.

AWS tag keys and values themselves are case-sensitive, e.g., `Project` and `project` are different. However, the tagging requirement in this course is case-insensitive when we monitor if you have properly tagged your resources.

| | |
|---|---|
| Tag Key | Project |
| Tag Value | vm-scaling |
| Cloud Technologies explored | Elastic Load Balancing, Auto Scaling Group |
| IaC Technologies explored | Terraform |

We suggest that you plan the budget carefully. You may want to use AWS Budgets (https://aws.amazon.com/aws-cost-management/aws-budgets) to monitor and forecast your spending on AWS.

Note that the budget portal on AWS and the Sail() platform is not updated in real-time. It takes around 6 hours for AWS to update the costs, and it takes the Sail() platform around 24 hours to update the project costs under the AWS Cost section. There is a delay for the cost logs to update for CSPs. With the number of users, implementing a real-time, accurate cost report system is not financially feasible for the cloud service providers as of today. Instead of relying on the reported spending provided by others, as a cloud practitioner, you need to calculate the hourly cost and plan the budget before you provision resources.

In this project, you are allowed to use either Spot or On-Demand instances. That said, we recommend that you use On-Demand instances for the project as Spot requests might not be fulfilled every time due to capacity overload. Also, as it normally takes longer to fulfill the spot request, we recommend you to use On-Demand instances for the formal submissions. You may consider using Spot instances in test runs so along as you programmatically tag the instances.

**Danger**

## Project Grading Penalties

| Violation | Penalty of the project grade |
|---|---|
| Spending more than $20 for this project phase on AWS | -10% |
| Spending more than $40 for this project phase on AWS | -100% |
| Failing to tag all your resources in either part (EC2 instances, ELB, ASG) for this project with the tag: key= `Project` , value= `vm-scaling` (AWS only) | -10% |
| Submitting your cloud/submission credentials or any Personal Identifiable Information (PII) in your code for grading | -100% |
| Using instances other than m5.large for Horizontal scaling on AWS | -100% |
| Using instances other than m5.large for Autoscaling on AWS | -100% |
| Submitting only executables ( `.jar` , `.pyc` , etc.) instead of human-readable code ( `.py` , `.java` , `.sh` , etc.) | -100% |
| Attempting to hack/tamper the autograder in any way | -200% |
| Cheating, plagiarism or unauthorized assistance (please refer to the university policy on academic integrity and our syllabus) | -200% & potential dismissal |

When it comes to VM clusters, you may have heard of managed Hadoop PaaS such as AWS EMR to perform big data analytics at scale. Unfortunately, PaaS services like AWS EMR, Azure HDInsight, or GCP Dataproc can be expensive, and the internals are not transparent to you. Although such services support a fair number of frameworks, which are mostly geared towards the needs of batch processing and analytics, they are still limited and inflexible. You might need to deploy another framework, or a custom configured web service on cloud resources. To enable you to do this, we will now shift gears to look at ways in which we can deploy performant web services that respond to dynamic requests on the cloud.

If you recall, in the previous project, we intentionally required you to use a T-family micro instance which has very limited hardware resources. You may observe that code that ran fairly fast on your own machine would take much more time to run on a T-family micro instance. Sometimes, inefficient code can exceed the capacity of a small machine and make the program crash or stuck.

One option is address the resource limit is to optimize your code in order to require fewer resource. Still, there is a limit to how much you can optimize the code. For certain workloads, you simply must be able to assign the appropriate hardware resources in order to complete processing the workload efficiently. In cloud computing, the process of adjusting the number of resources assigned to a particular workload/task/service is known as **scaling**.

# Types of Scaling

For the purposes of this course, resource scaling can be categorized broadly into two main parts:

## Vertical Scaling

The simplest technique used to scale an application is vertical scaling, which involves changing the capacity of the resources. Recall that you have explored vertical scaling in the previous project, where you benchmarked the performance of various types of instances on AWS, GCP and Azure.

For instance, if the resource is a virtual machine, vertical scaling could involve changing the number of cores, amount of memory, or CPU speed used by the VM. You may resize an existing VM, which requires you to restart the VM with a downtime. Sometimes, you may need to provision a new set of resources and migrate the application or service.

Note that there is an upper limit of the resources for a single machine, and vertical scaling may face challenges when handling dynamic workload.

## Horizontal Scaling

Horizontal scaling involves adding or removing resources such as VMs to/from the system, respectively. For example, if a web service is currently utilizing a cluster of 5 nodes and the load continues to grow. Using horizontal scaling, we can increase the number of nodes to deal with the increased load. Horizontal scaling is more complicated than vertical scaling because you now have a task distribution and assignment problem, for example, to which VM should you send a new incoming request. In the simplest case, resources can be scaled using replicas which can be used interchangeably. Horizontal scaling can also be performed by partitioning the problem domain into smaller “shards” and making each individual resource responsible for a single partition (or shard).

## Scenario

You are a competitive job seeker who is extremely interested in working for clandestine organizations that may or may not have Orwellian goals. Recently, you received the following letter offering you a chance to join one of the top organizations in this domain, the Massive Surveillance Bureau (MSB).


First-round interview invitation letter

Dear Student,

We are pleased to confirm that you have been shortlisted for the first round of selection for the System Architect Program (fast track) at the Massive Surveillance Bureau (MSB).

For this interview, you need to pass several Horizontal Scaling tests using different techniques in order to show that you have a clear idea of how to elastically provision resources to maintain the Quality of Service. You should complete these Horizontal Scaling tests on AWS. These tasks are time and cost-intensive! Good luck!

Harry Q. Bovik

Human Resources
Massive Surveillance Bureau
Quis Custodiet Ipsos Custodes

Information

## Getting Ready

The theme of this project is **elasticity**, and you will gain first-hand experience with dynamic scaling and elasticity. This project introduces you to the methods which allow you to write programs that manipulate and dynamically control resources on a public cloud. Though this project primarily relies on stringing together API calls in the right order, it is important for your code to be reliable and fault-tolerant. Making cloud API calls in a networked program is not as easy as sequential programming. Each API call is merely a request to a cloud provider, that the cloud provider may or may not be able to meet, due to global resource capacity, etc. You must account for all possibilities. **Defensive programming is crucial this week.**

At the end of this project, you will learn how to provision AWS resources with either AWS SDKs or Terraform to host auto-scalable web services.

***Note: You are only allowed to use the official APIs/SDKs provided by the cloud providers. Do not use third-party or cloud agnostic libraries like libcloud. They defeat the learning objectives of the assignment as you won't be able to learn the differences between the APIs and their complexities effectively.***

Information

# Starter Code

We provide you with starter code in Java and Python. You can find the starter code in the instance launched with the student workspace VM image.

**Note: The starter code is only for your reference. You are welcome to start from scratch or use partial snippets of the starter code.**

The starter code for `task1` and `task2` contains the basic dependencies you will need and stubs to interact with the APIs.

The starter code also contains the terraform file template you will need for `task3`.

Put all your configuration parameters for `task1` and `task2` in the JSON files provided.

## Getting the starter code

**1.** Provision a `t3.micro` instance in the `us-east-1` region using the AWS console with the AMI `ami-04537cfe22bace769`. You will have to open both port 80 and port 22 for this instance. You may reuse the Terraform templates provided in the previous project. **Make sure to tag the VM instance using the required tags of this project.**

**2.** Open the URL `http://[YOUR-EC2-DNS]` in your web browser for the instance. Enter your submission username and submission password. Then click the Launch Project button of this project. Wait for several minutes until the log tells you that the instance is ready. You can then log into the instance using the PEM/PPK file as the user `clouduser`.

**3.** You can find the starter code for each task under `/home/clouduser/VM_Scaling_Project/`.

AWS Horizontal Scaling

# AWS Horizontal Scaling

For this task, MSB requires you to write code and build a system that is capable of handling the large volume of requests generated by the Load Generator with minimal cost. Your program will have to scale the number of web service instances until your entire system is able to handle a specified target of requests per second (RPS).

You will be scored on your ability to scale out and achieve the target RPS within the allotted time.

## Tasks to Complete

Your task is to implement the program using the Amazon EC2 API to meet the following specifications:

**1.** Create two security groups to allow incoming traffic on port 80 and all outgoing traffic on all ports. One of them should be associated with the Load Generator and one with your Web Service Instance.

**2.** Launch a `m5.large` load generator instance using `ami-0273735cd4c306ac0` with the correct tags.

**3.** Launch a `m5.large` web service instance using `ami-042de649749923897` with the correct tags.

**4.** Read the values of your submission username and password from the environment variables. **Do not store your username or password in the code or the configuration file. Submitting your username or password will incur a 100% penalty. You MUST read these values from environment variables.**

**5.** Authenticate with the load generator. This can be done by submitting your password and username to the load generator using the following request:

```
http://[load-generator-dns]/password?passwd=[your-submission-password]&username=[your-username]
```

**6.** After the authentication is successful, you will submit the web service instance DNS name to the load generator to start the test. This can be achieved by making a request to the following URL:

```
http://[load-generator-dns]/test/horizontal?dns=[web-service-instance-dns]
```

During the test, the load generator will send a large amount of traffic to your web service VM. You would be provided with a test ID if the launch of the test was successful.

**7.** Monitor the test from the log and fetch the total RPS of the web service instance(s) using the following URL:

```
http://[load-generator-dns]/log?name=test.[test-number].log
```

A sample log can be accessed here (https://s3.amazonaws.com/cmucc-public/p21/f20/resources/submission.log).

**8.** Launch more web service instances, one at a time, in order to achieve a cumulative RPS of 50. The cumulative RPS means the total RPS of all web service instances added together. For example, in the above log file, the cumulative RPS at minute 4 is 24.61. Once the instance is launched and running, new instances can be added to the running test by sending the following request to the load generator:

```
http://[load-generator-dns]/test/horizontal/add?dns=[web-service-instance-dns]
```

**9.** Read the Testing Rules section carefully to pass the test. Once a test is complete, you should be able to visit the submissions page to see your score for this task.

The starter code for this task can be found under `/home/clouduser/VM_Scaling_Project/<java/python>/task1` on your EC2 VM launched with the Cloud Computing Project Image. The starter code has implemented the steps that authenticate with the LG, start the test and parse the log file for the RPS. You will need to implement the rest of the steps.

## Testing Rules

**1.** To complete the test, you have a maximum of `30` minutes to reach your target RPS of `50`.

**2.** Please make sure you start with a blank slate. All your instances must be launched by your code. All web service instances (except for the first one) should be launched **after** the test starts. The autograder checks for this.

**3.** The Horizontal Scaling Test will end once the sum of the RPS of your web service instance VMs reaches the target threshold. You can make use of the test log to check the progress of the test.

**4.** Make sure your code processes the current requests per second and then decide whether to launch another instance.

- **Do not hardcode** the number of instances into the code.
- There should be a time window of at least 100 seconds between any pair of web service instances VMs' launch time. That is, if your program just launched a web service VM, your program should wait at least 100 seconds before it launches another one. This window is known as the cooldown period, and the cooldown period helps to ensure that your scaling doesn't launch or terminate additional instances before the previous scaling activity takes effect.
- Do not hardcode a fixed delay between launch operations. Please understand the difference between **hard coded delays** and **polling intervals**. Hardcoded delays refers to the explicit use of time.sleep(X seconds) or any similar operation to launch a new instance or wait for an instance to transition to a new state. On the other hand, you can use polling intervals to record the last time you added an instance and check that 100s have elapsed since that time before adding a new one.
- However, you may use sleep for <= 1 seconds while retrying a request.
- The code snippet below demonstrates the correct usages of `sleep`

```
request_success=false
while(!request_success){        //Usage 1 (Polling)
        try{
                make_http_request()
                request_success = true
        }
        catch (Exception){
                Thread.sleep(<=1 second)
        }
}

while (test is not complete) {    //Usage 2 (Polling)
        Thread.sleep(<=1 second); // Very small sleep time to save clock cycles
        if (current_time - previous_launch_time > 100 seconds){
                //check or do something
        }
        current_time = update_current_time()
}
```

The following usage of sleep is **NOT ALLOWED**:

```
if (test is not complete){
        Thread.sleep(100000) //100 seconds
        launchInstance()
}
```

**5.** Your program should be fully automated. Executing the program should complete the following steps: launch an LG VM and WS VM, fill in your submission password and username, start the test, add more VMs if needed, and exit once the test is completed.

**6.** You can terminate all the web service instances (either manually or through code).

**7. You are not allowed to access the Load Generator or Web Generator instances via SSH. The project does not require you to perform any operations on the instances.**

## Validate your Work

Before you make a submission, please verify that you can compile and execute the program without any errors.

## What to Submit

In order to submit the Horizontal Scaling test, you will need to execute the task 1 submitter. The submitter will run the test and submit the source code. **Note that if you run your code without the submitter, you will be able to run the test, but the results of the test will not be reported to the learning platform for grading.**

A reminder that your last submission of each task must include a `references` file where you cite all the reference sources when you work on this task. We strongly recommend that you have your reference file ready in the task folder to avoid re-running the test. Please do not include any binaries (.jar) or cloud credentials (.pem, access_key.csv) files. You will incur a 100% penalty if we find any credentials files in your submitted code.

When submitting, please make sure the following files are in your project folder:

- The reference under `~/VM_Scaling_Project/<java/python>/task1/references` where you cite all the reference sources when you work on this task.
- The source code under `~/VM_Scaling_Project/<java/python>/task1/src/`
- The XML file under `~/VM_Scaling_Project/java/task1/pom.xml` if you use Java

Note: You should only execute the submitter from the student VM and not from your local machine. Executing the submitter with your own machine is subject to an unbounded debugging scope of the OS environment and such debugging is not in the scope of the project learning. The teaching staff will not be able to support you if you attempt to run the code locally. However, you may run your java/python code locally for testing purposes without submission.

If you use Java, you can use the following command to run the compiled code after compiling your Java code:

```
mvn compile && mvn exec:java -Dexec.mainClass="horizontal.HorizontalScaling"
```

## How to Submit

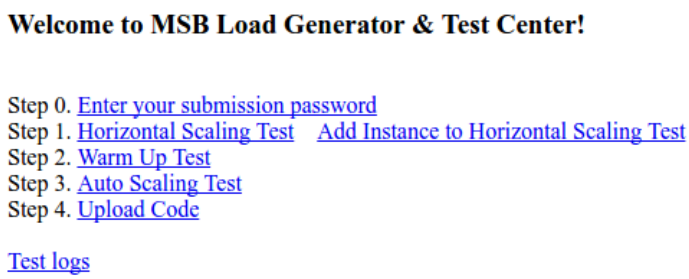
Since you will be communicating with the AWS APIs, you will first need to authenticate with AWS with `aws configure` and entering the required fields. You can refer to this documentation (https://docs.aws.amazon.com/powershell/latest/userguide/specifying-your-aws-credentials.html) for the location of the credentials on your local machine.

After `aws configure`, provide the submission username and password, and then execute the submitter.

```
    export SUBMISSION_USERNAME=your_submission_username
    export SUBMISSION_PASSWORD=your_submission_password
    ./task1_code_submitter
```

## Hints

**1.** We highly recommend that you try a dry run using only the browser before you write code, to get a feel of the process. You can start coding once you are confident with how the different modules are connected. You should see a page similar to this:




You should be careful the number of times you run the test with m5.large since they are more expensive than t3.micro.

The test results will be reported only when the correct submission instance type `(m5.large)` is used.

---

The following video covers the basics of a dry run. **Please note that you should use the AMI ID(s), VM instance type, and tags as per the writeup instead of the example ones in the video.**

**Video 1:** Horizontal Scaling Dry Run

**2.** If you are unfamiliar with what requests you need to make and what output to expect, view the process on your browser. You can use developer tools in Chrome and Firefox to analyze the requests that are being sent when you perform some operation on the UI. You can also see the responses received for each and every request.

**3.** Launching all instances within the same subnet should yield better performance than having them in different subnets. You do not need to create your own private subnet and VPCs. Every region has a default VPC attached to it and every VPC has a set of subnets that you can choose from. When you create an instance in a specific region, the instance is directly launched into a default VPC. When using the API, you are required to obtain the **ID of the default VPC** and the **Subnets** for the **US-EAST-1** region.

**4.** You need to use the public DNS rather than public IP address as the endpoints of virtual machines in this project, e.g., when you access `http://[load-generator-dns]/test/horizontal?dns=[web-service-instance-dns]`.

**5.** During the test, you want to monitor the instance graphs on the EC2 Console. You can optionally create a custom Dashboard to monitor multiple parameters. Note: Custom dashboards are chargeable on a monthly basis.

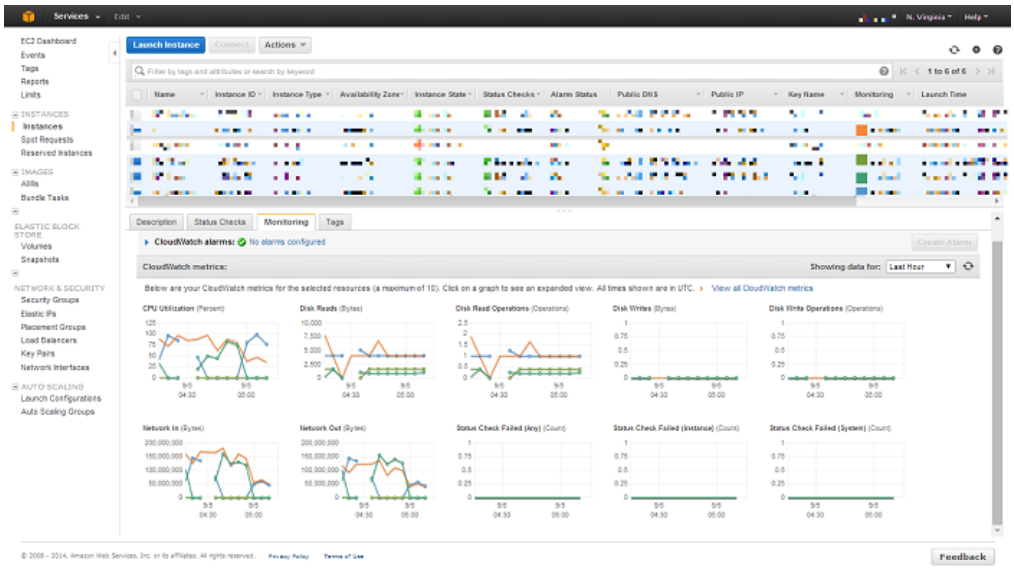



**Figure 2:** CloudWatch on the EC2 Console

**6.** Try to make your code modular and reusable. This can help greatly in the next task.

**7.** You might want to make an AWS API call to get the latest state once your resource is created completely. For example, when you create an EC2 instance through the API, the response of the API call does not have an IPv4 address since the VM is still starting. The IPv4 is assigned one upon successful creation. For this purpose, you will find the Waiters class in the AWS SDK for Java (https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/ec2/waiters/AmazonEC2Waiters.html) and Python (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html#EC2.Instance.wait_until_running) particularly useful in waiting for a resource to transition into the desired state, which is a very common task when you're working with services that have a lead time for creating resources (Amazon EC2).

**8.** If your test is interrupted before completion, remember to destroy any provisioned resources to save money.

AWS Autoscaling

Information

# Introduction

**Quality of service (QoS)** and cost are two very important factors when deploying a scalable web service in the real world. It is guaranteed to lose users in the competition if you provide web services that fail to meet acceptable QoS guarantees. Performance, availability, reliability and security are some dimensions of QoS. One can always provide high QoS by over-provisioning resources in order to handle peak load, however, the ability to compete on cost-efficiency will become an issue.

To effectively address the above, AWS provides a service **Elastic Load Balancing** that automatically distributes traffic to connected instances and handles instance failures. Elastic Load Balancing can tie in with an **Auto Scaling Group** which can dynamically shrink or grow a set of instances that trigger **scaling policies** according to some **CloudWatch** alarms. The following sections will provide you with the information about these AWS resources.

## Elastic Load Balancer

The Elastic Load Balancer (ELB) acts as a network router that will forward incoming requests to multiple resources (e.g., EC2 Instances as in this project) in a round-robin fashion. In this project, we will use an **Application Load Balancer** which is a type of ELB. The group of instances serving requests behind a given Application Load Balancer is called a **target group**. Instances can be added/removed to/from a target group manually through the web console, programmatically through an API, or dynamically with an Auto Scaling Group. ELB can also perform a Health Check to check if the instance is responsive (if not, it will stop sending requests to it). An ELB basically acts as a front door of your web service or system. When a new instance is added to the target group, a minimum number of health checks at predefined intervals are performed before the ELB starts sending requests to the new instance. You can read more about the health check configuration here (https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-healthchecks.html#health-check-configuration)

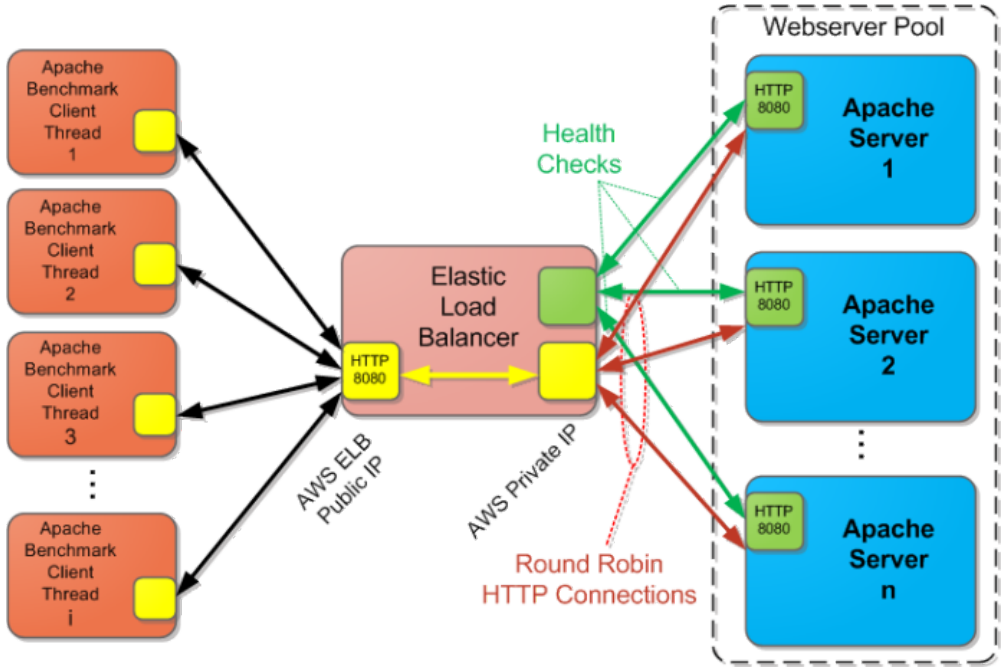



**Figure 3:** Elastic Load Balancer

The following video covers the basics of ELB:

**Video 1:** Basics of ELB

You can programmatically interact with an Elastic Load Balancer in many ways, including:

**1.** The AWS Command Line Interface (AWS CLI) (https://aws.amazon.com/cli/)

**2.** The AWS SDK for Java (https://aws.amazon.com/sdk-for-java/)

**3.** boto3 for Python (https://boto3.readthedocs.io/en/latest/)

**4.** Terraform (https://www.terraform.io/)

We will give you more information about working with these APIs after introducing the next section on Auto Scaling.

## A Note on ALB Warmup

Application Load Balancer (ALB) is designed to handle large loads of network traffic provided the traffic increases gradually over a long period of time (several hours). However, if the traffic load increases rapidly over a short period of time, it becomes difficult for the ALB to handle this load fast enough.

AWS considers that if the traffic increases more than 50% in less than 5 minutes, then it means that the traffic is sent to the load balancer at a rate that increases faster than the ELB can scale up. In order to deal with these issues, the clusters need to be scaled up even before the actual load spikes through a process called warming up. Warming up an ALB using an expected pattern allows the ALB to be vertically scaled before it can handle the real test load.

## Amazon Auto Scaling

AWS Auto Scaling service dynamically adds or removes computing resources allocated to an application, by responding to changing workload. AWS Auto Scaling is a type of Horizontal Scaling, which is the act of increasing or decreasing the capacity of an application by changing the number of identical servers assigned to it. The following video explains Auto scaling:



**Video 2:** Auto Scaling

AWS Auto Scaling service provides the following deployment modes on a group of EC2 instances:

- Maintain a fixed number of running EC2 instances at all times, by performing periodic health checks on instances and automatically launching new instances when one or more are unhealthy;
- Scale instances manually, which will increase or decrease the number of running instances on your behalf;
- Predictable scaling based on a developer-specified schedule (for example, a condition you could specify to increase the servers every Friday at 5 p.m. and to decrease them every Monday at 8 a.m);

- Dynamically scale based on conditions specified by the developer (for example, CPU utilization of your EC2 instances).

## Using Elastic Load Balancing and Auto Scaling

An important use case for Auto Scaling is dynamic infrastructure management. In the case of MSB, it should be clear now how Auto Scaling can help deal with variable traffic patterns. As the traffic increases and the load on the servers increases, the number of servers can be increased dynamically. Similarly, as the traffic decreases and the load on the servers decreases, the number of servers can be decreased dynamically. You can program Auto Scaling to respond to changes in CPU load (or other metrics, such as response time) of provisioned servers.

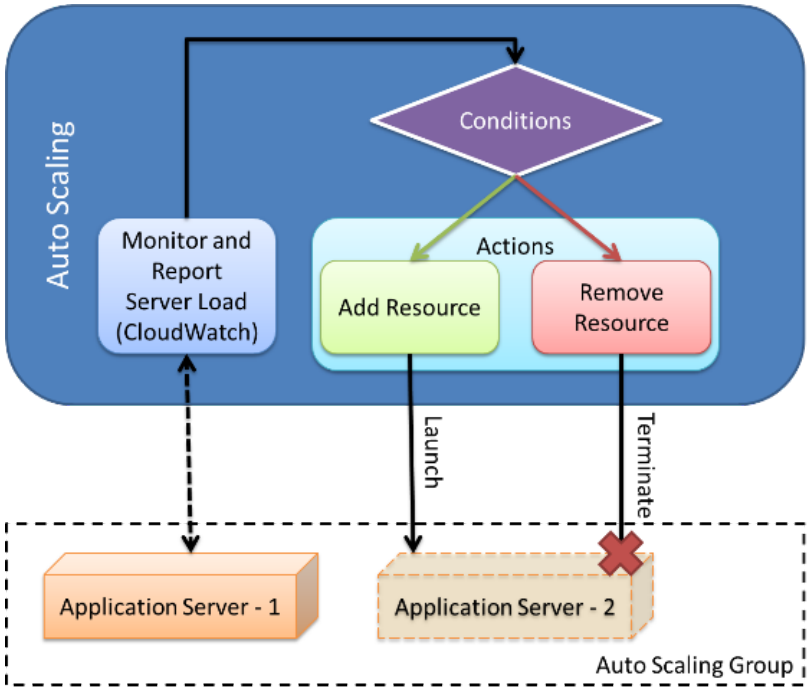



**Figure 4:** Auto Scaling Architecture in Amazon EC2

## AWS Auto Scaling Internals

Using Amazon's Auto Scaling in a dynamic fashion requires, at minimum, the following:

- An **Auto Scaling Group (ASG)** should be defined with a minimum, maximum and desired number of instances defined during the group creation. ASG is also associated with an Elastic Load Balancer and a corresponding Target Group. More information can be found here (http://docs.aws.amazon.com/autoscaling/latest/userguide/GettingStartedTutorial.html) on creating auto-scaling groups.
- A **Launch Configuration** template should be defined, which includes the AMI ID, instance types, key pairs and security group information, among others. Auto Scaling is meant to scale automatically based on application demands; i.e., the instance should be configured to automatically start the required application services and work seamlessly on launch.
- An **Auto Scaling Policy** should be created, which defines the set of actions to perform when an event, such as a CloudWatch alarm is triggered.

The following video will demonstrate how you can use Auto Scaling (keep in mind that the latest UI is slightly different):

**Video 3:** Auto Scaling Demo

## Amazon CloudWatch

Amazon CloudWatch enables developers to monitor various facets of their AWS resources. Developers can use it to collect and track metrics from various AWS resources that are running on the AWS Cloud. CloudWatch also allows you to programmatically retrieve monitoring data, which can be used to track resources, spot trends and take automated action based on the state of your cloud resources on AWS. CloudWatch also allows you to set alarms, which constitute a set of instructions to be followed in case of an event that is tracked by CloudWatch is triggered.

CloudWatch can be used to monitor various types of AWS resources. In this project, you will monitor the following resources:

- EC2 instances
- Application Load Balancer
- Auto Scaling Group

For EC2 instances, CloudWatch allows you to monitor CPU, memory and disk utilization. For Application Load Balancer, CloudWatch allows monitoring of such metrics as request/response counts, response time, failures, etc.

**You will be using CloudWatch extensively in this task to come up with near-optimal auto scaling policies.**

For more information on CloudWatch please refer to the Amazon CloudWatch documentation (http://aws.amazon.com/documentation/cloudwatch/).

**Video 4:** CloudWatch

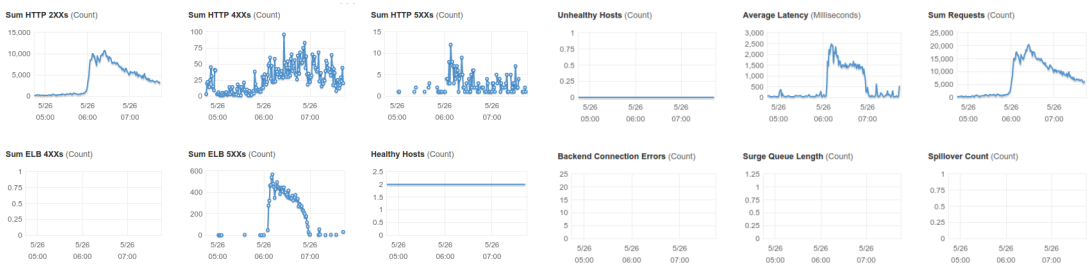



**Figure 5:** Example of CloudWatch Metrics

The above image shows some sample metrics captured and displayed by the CloudWatch service. These metrics can be useful when devising a policy for auto-scaling:

- **HTTPCode_ELB_2XX:** This is the count of HTTP 2XX status codes that ELB receives from the client when the request is successfully processed. A large number of 200 requests in the graph means that the policy is working fairly well.

- **HTTPCode_ELB_5XX:** This is the count of HTTP 5XX (which indicates a server error) that the ELB responds to the client. There are several scenarios where the ELB can also generate 5XX error e.g., if no instance is registered, the instance is unhealthy, or the request rate is higher than the ELB instance's current capacity.
- **HTTPCode_ELB_4XX:** This is the count of HTTP 4XX (which indicates a client error) status codes which the ELB responds to the client. One scenario where the ELB can respond with this status code is when the instances behind the load balancer have not fully started.
- **Latency:** This tells you about the performance of your web server instances. It measures the amount of time the webserver is taking to process a request.
- **Request Count:** This is the total number of requests which are received by the load balancer and routed back to instances. This CloudWatch metric can help the user understand the load pattern which can be used to make changes to the policy alarms.
- **Backend Connection Errors:** This is the number of failed conncetions to the instance behind the load balancer.
- **UnHealthyHost Count:** The UnHealthyHost count instances that have failed more health checks than the given threshold. The instance can be unhealthy if the instance health check is giving non-200 response or time-out when performing the health check.
- **Healthy Count:** The HealthyHost count instances are ready to receive requests from the Load Balancer.

---

# AWS Autoscaling

## Getting Ready

This is the final test before you pass the first round of interviews of the MSB System Architect. You will need to use the AWS APIs to create or launch everything you need for the final test. In the final test, the LG will send a changing load to your ELB, and your autoscale policy will adjust the number of instances running to achieve a required request per second (RPS) rate within reasonable instance hours (defined later). **The focus of this task is to provide you with hands-on experience with balancing QoS and cost.**

During a typical day, there are several load phases reflecting activity in different parts of the world (where the application has users). During the peak phases, MSB receives a large number of requests to lookup access codes (requests to the `/lookup/random` page of the Web Service Instance). You are required to achieve the highest average RPS possible, with at least 35 RPS during a peak. In other words, not reaching a max RPS of 35 during the 24-minute period will reduce your score. Additionally, system architects at MSB do not have an unlimited budget. MSB keeps track of their resources using a normalized "instance-hour" measure. As the name suggests, 1 instance-hour means that one instance is running for one hour. For example: An m5.large running for 1 hour counts for 4 instance-hours. At the MSB, system architects are limited to 280 instance-hours per day. However, they are encouraged to use fewer instance-hours, to save budget. Hence, they need to balance between RPS and resource consumption. See more details in the Scoring section below.

Information

### A Note on Time Warping

Unfortunately, we do not have the budget (and most of you do not have the time) to conduct 24-hour long tests. For the rest of this module you should assume that we will simulate 1 hour of MSB time in 1 minute of real time. Thus, we will simulate a full **24 hour test** in only **24 real world minutes**.

### Instance-Hours

Instance-hours are meant to keep track of the total resources used to complete a task. As such, different instances have a different multiplier to account for their costs. Furthermore, since we are simulating MBS time in this project (remember 1 minute of test time equals one hour of simulated MBS time) an instance-hour in simulated time is 1 minute in test-time. So, if you are limited to 280 instance-hours in the simulation your real test can only use ~3 m5.large instances if you run them for the entire 24-minute test. **You need to use fewer than 220 instance hours to get a full score.**

### Average RPS

The system architect candidate gets interview points relative to a day's performance. However, to get those points you must meet the minimum requirement of creating a system with a mean RPS of over 7 for the entire 24 hours of traffic. (i.e., in our simulation that signifies a mean RPS >= 7 during the 24-minute test to get a non-zero score). **You need to get an average RPS of 12 to get a full score.**

### Peak RPS

During a typical day, there are several load phases reflecting activity in different parts of the world. During the peak phases, MSB receives peak traffic to lookup access codes (requests to the `/lookup/random` page of the Web Service Instance). **You need to achieve at least 35 RPS as the highest RPS in order to handle the peak.** Not reaching a max RPS of 35 during the 24-minute period will reduce your score.

### Performance Target and Scoring

| Performance Metric | Threshold | Score calculation | Score |
|---|---|---|---|
| Instance Hours | 220 <= x <= 280 | $score_{ih} = \frac{280 - x}{280 - 220} * 100$ | min( $score_{ih}$ , 100) |
| Average RPS | 7 <= x <= 12 | $score_{avgrps} = \frac{12 - x}{12 - 7} * 100$ | min( $score_{avgrps}$ , 100) |
| Max RPS | x < 35 | $score_{maxrps} = -5$ | min( $score_{maxrps}$ ,0) |

Your total auto-graded score for this phase is the average of the instance-hour score and the average RPS score.

This task is worth 40 points. The autograded total score will be scaled to 40 points from 100 points before calculating your final project grade.

$$\text{Total score} = \frac{(score_{ih} + score_{avgrps})}{2*100} * 40 + score_{maxrps}$$

Apart from the autograded score, your code will also be manually graded. Those points will go to proper use of cloud APIs (such as polling the status of resources to check if they are ready rather than waiting for a fixed amount of time), your description of the traffic patterns you've observed and how/why you chose your Auto Scaling policy parameters (see **Step 16** of **"Tasks to Complete"** below).

In order to optimize the tradeoff between increased RPS and decreased capacity instance-hours, your auto-scaling solution should use the bare minimum number of instances required for most of the day, quickly scale up to handle the spikes as they occur, and then scale down as soon as the spikes are fading.

### Fault Tolerance

As the scale of an infrastructure increases, the chances for failure also grow. In order to simulate such a situation in this project, **one of the instances will be killed during the runtime of the application**. The Load Balancer will continue to send traffic to this server until they are no longer marked as unhealthy. You need to ensure that most of these requests are not lost by setting up the ELB and/or the Auto Scaling Group to detect and respond to failures using health checks.

## Putting It All Together

Using the AWS SDKs in Java/Python:

**1.** Create two security groups to allow **incoming traffic on port 80 and all outgoing traffic on all ports**. One of them should be associated with the Load Generator and one with your Elastic Load Balancer and Auto Scaling Group.

**2.** Launch a `m5.large` load generator using `ami-0273735cd4c306ac0` with the correct tags.

**3.** Create a Launch Configuration for the instances that will become part of the Auto Scaling Group, with the following parameters:

- AMI ID: `ami-042de649749923897`
- Instance Type: `m5.large`
- Detailed Monitoring: enabled

**4.** Create a Target Group for instances using HTTP:80. Use `/` (which is the heartbeat endpoint of the web service in this project) as the Health Check path and use HTTP. Note that ELB health checks may become queued behind other requests. A congested server may fail to pass health checks.

**5.** Create an Application Load Balancer that forwards the HTTP:80 requests from the load balancer to HTTP:80 on the instance. You will have to use the target group from step 4 to configure the listener.

**6.** Create an Auto Scaling Group after analyzing the traffic pattern from the Load Generator (through several dry runs) figure out a good rule for Scale Out and Scale In operations. Below we define the following default rules to give you an idea of the various parameters.

- Group Size: Start with 1 instance
- Subnet: We recommend that you choose the same availability zone(s) corresponding to your ELB
- Load Balancing: Receive traffic from the created Target Group
- Health Check Type: EC2
- Detailed Monitoring: enabled

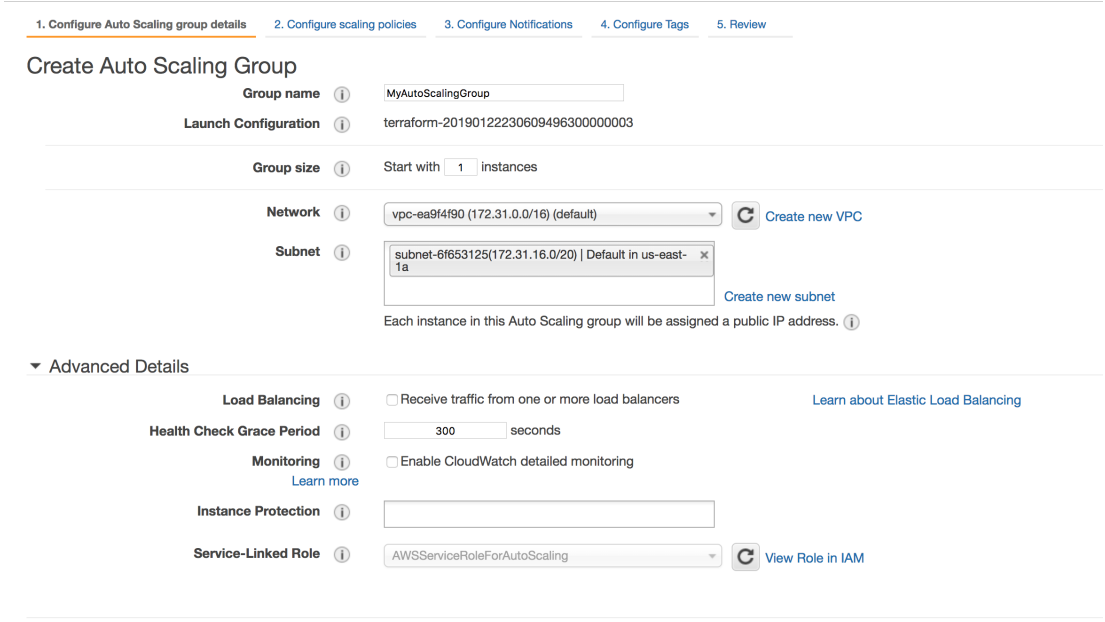



**Figure 6:** An example of Auto Scaling Group configuration

**7.** Create the following *example* scaling policies *as the starting point*. Note that you may find it **difficult** to achieve a passing grade using an Auto Scaling Group with the following parameters.

- Minimum Instance Size: 1
- Maximum Instance Size: 2
- Create a Scale Out policy that automatically adds 1 instance to the Auto Scaling Group.
- Create a Scale In policy that automatically removes 1 instance from the Auto Scaling Group.

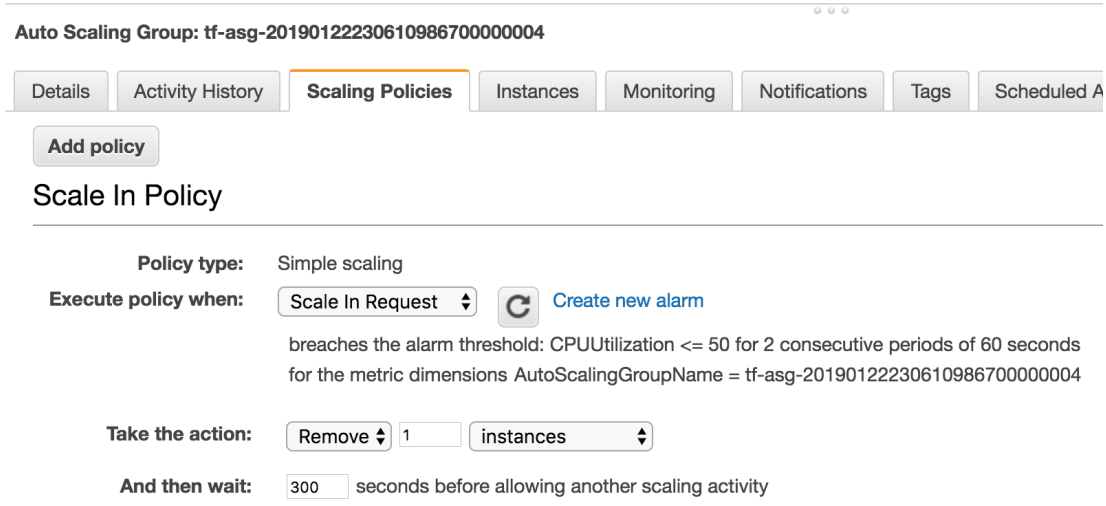



**Figure 7:** An example of Auto Scaling Group configuration

**8.** Create CloudWatch Alarms that invoke the appropriate policy for the following scenarios. Note that you may find it **difficult** to achieve a passing grade using an Auto Scaling Group with the following parameters.

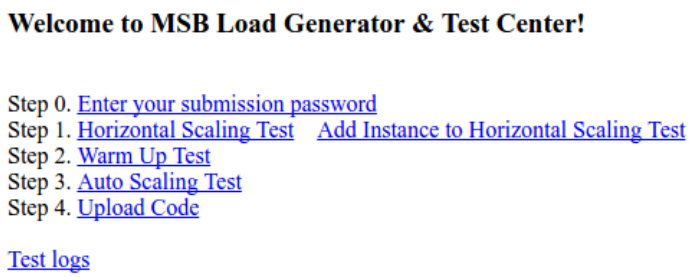
- Scale Out when the group's CPU load exceeds 80% on average over a 5-minute interval.
- Scale In when the group's CPU load is below 20% on average over a 5-minute interval.

**9.** Link the CloudWatch Alarms to the Scale Out and Scale In rules of your Auto Scaling Group. You will be able to add an Auto Scaling policy with the alarms and scaling actions by editing the Auto Scaling Group.

**10.** Configure correct tags for your Auto Scaling Group using your code. Java and Python with Boto3 have support for ELB tagging.

**11.** Authenticate with the load generator. This can be done by submitting your password and username to the load generator using the following request:

```
http://[load-generator-dns]/password?passwd=[your-submission-password]&username=[your-username]
```




**12.** Make a request to the following HTTP URL to start the Autoscaling test:

`http://[load-generator-dns]/autoscaling?dns=[elb-dns]`.

You can view the progress of this test using the page:

`http://[load-generator-dns]/log?name=test.[testId].log`.

**13.** You may want to use the following structure to run a test:

```
//Start Test
while(!isTestComplete){ //Parse and check the INI file to see if the test is complete
    Thread.sleep(<=1 second) //Very small sleep time
}
//Terminate Resources
```

**14.** At the end of the 24 minutes, your average RPS for the entire test and total instance hours consumed will be displayed. A sample log file for Task 2 can be accessed here (https://s3.amazonaws.com/cmucc-public/p21/s21/resources/submission_task2.log).

**15. The program needs to terminate all the resources after the test.**

**16.** Once you are done with the test and confident with your score, create a **patterns.pdf** answering the following questions:

```
(a) What traffic patterns did you see in the load sent to the ELB by the load generator? How did you actually figure
out the load pattern? (Please provide appropriate screenshots from the AWS dashboard wherever necessary) (3 pts)

(b) How did you model the policies for the Auto scaling Group in response to the insights gained in the above questi
on? (2 pts)
```

**17.** (Optional) You can also repeat the testing if you have a budget left and hope to try with different parameters/policies. Note that only your latest submission will be graded for every task.

The starter code for this task can be found under `/home/clouduser/VM_Scaling_Project/<java/python>/task2` on your EC2 VM launched with the Cloud Computing Project Image. The starter code has implemented the steps that authenticate with the LG, start the test, get the test ID and parse the log file for the RPS. You will need to implement the rest of the steps.

Initially, for testing, you may ignore setting up CloudWatch alarms and an Auto Scaling Group (Steps 3-9 above could be replaced by attaching a fixed number of instances in the Auto Scaling Group to the Elastic Load Balancer) to have a sense of the traffic pattern. Moreover, you can consider using Spot instances during initial experimentation, to save your project budget for further fine-tuning and optimization. Once you have analyzed the traffic pattern using a fixed set of instances (e.g., 5), you could continue with the full set of steps below:

---

## Validate your Work

Before you make a submission, please verify that you can compile and execute the program without any errors.

## What to Submit

**1.** In order to submit the Auto-Scaling test, you will need to execute the task 2 submitter. The submitter will run the test and submit the source code. Note that if you run your code without the submitter, you will be able to run the test, but the results of the test will not be reported for grading. A reminder that your last submission of each task must include a `references` file where you cite all the reference sources when you work on this task. We strongly recommend that you have your reference file ready in the task folder to avoid re-running the test. Please do not include any binaries (.jar) or cloud credentials (.pem, access_key.csv) files. You will incur a 100% penalty if we found any credentials in your submitted code.

When submitting, please make sure the following files are in your project folder: * The reference under `~/VM_Scaling_Project/<java/python>/task2/references` with all the links that were referred when completing this task. * The source code under `~/VM_Scaling_Project/java/task2/src/` or `~/VM_Scaling_Project/python/task2/<autoscaling.py/auto-scaling-config.json>` * The XML file under `~/VM_Scaling_Project/java/task2/pom.xml` if you use Java

**2.** The second file that you will submit is the **patterns.pdf** file. This can be submitted after you are satisfied with your results from the Auto Scaling test. In order to submit this file, make sure the name of the file is the same as above with all lowercase letters. Make sure this file exists under `~/VM_Scaling_Project/<java/python>/task2/patterns.pdf` when submitting. You do not need to re-run the test

to submit your patterns.pdf file. Please read the How to Submit section about how to submit the patterns report.

You may choose to run your code without the submitter for testing purposes.

Note: You should only execute the submitter from the student VM and not from your local machine. Executing the submitter with your own machine is subject to an unbounded debugging scope of the OS environment and such debugging is not in the scope of the project learning. The teaching staff will not be able to support you if you attempt to run the code locally. However, you may run your java/python code locally for testing purposes without submission.

If you use Java, you can use the following command to run the compiled code after compiling your Java code:

```
mvn compile && mvn exec:java -Dexec.mainClass="autoscaling.AutoScale"
```

## How to Submit

Since you will be communicating with the AWS APIs, you will first need to authenticate with AWS with `aws configure` and entering the required fields. You can refer to this documentation (https://docs.aws.amazon.com/powershell/latest/userguide/specifying-your-aws-credentials.html) for the location of the credentials on your local machine.

To submit your program for auto-grading:

```
    export SUBMISSION_USERNAME=your_submission_username
    export SUBMISSION_PASSWORD=your_submission_password
    ./task2_code_submitter
```

To submit your **patterns.pdf** file:

```
    export SUBMISSION_USERNAME=your_submission_username
    export SUBMISSION_PASSWORD=your_submission_password
    ./task2_code_submitter -p
```

**Danger**

## Timezone awareness

**Timezone awareness** matters when getting the test start time. In the previous project, we have discussed implicit environment reliance, and the timezone is another example. "2019/9/20 3:00 PM" in Pittsburgh is not the same as "2019/9/20 3:00 PM" in SV, but "2019/9/20 3:00 PM ET" always means the same time for anyone in any timezone. When you set up a global meeting, you don't want to omit the timezone information. Similarly, please make sure your program is timezone aware, and you should never use any timezone-naive local time to compare with other times. For example, when you're parsing the time from the logs, the datetime in the logs will not reflect the datetime as displayed on your local system.

## Hints

- One full test with warmup takes `39` minutes (not including time to provision and destroy resources). Plan your time accordingly. **Start early!**.
- You may want to set up AWS resources in the AWS console (the web GUI) before you start programming; you'll likely find the console more intuitive to understand the process. It's not much overhead, since you'll be able to reuse the parameters set in the console for the programming portion. The documentations for Boto, Terraform, etc. assume that you know how AWS works.
- You may wish to perform several manual runs without Auto Scaling, the next with a set of Auto Scaling rules based on your observations of the static test, followed by a run with modified rules to perform well.
- In order to understand the characteristics of the load pattern, you may want to run the test with a fixed amount of resources a few times. Note that the pattern is not the same between runs, meanwhile the patterns are designed so that a good scaling policy can pass the test deterministically.
- Note that CloudWatch provides very interesting metrics that you can use to understand the behaviour of the system. Make sure to read the **Example of CloudWatch Metrics** section carefully.
- In order to fine-tune your Auto-Scaling policy without waiting for 24 minutes in each run, you can use a synthetic benchmarking tool such as loadtest (https://www.npmjs.com/package/loadtest). An example command to send requests to your load balancer with 25 RPS rate for 1 minute: `loadtest -c 8 --rps 25 -t 60 -r http://YOUR_LB_DNS_NAME/lookup/random`. Our "real" Load Generator uses a similar approach (i.e., the generated load does not change within minute boundaries).
- Scaling out quickly is tricky. Find a way to mitigate the impact of adding/removing new instances. The 24-minute simulation of 24 hours effectively amplifies the overhead of instance launching/termination overhead.
- Pay close attention to the peak RPS to determine the maximum number of instances required.
- The rules to scale out and scale in are important, but you should also consider configurations such as the scale out/in cool-down period, auto-scaling group cool-down period, draining, health checks, etc.
- The availability zone you choose may have an impact on your test result. Creating all instances in the same availability zone would potentially lead to better results.
- A good scaling policy can and should pass the test deterministically. Your final score will be decided by your most recent submission, not the highest!
- Do not store your username, submission password, or any personally identifiable information (PII) in the code. Read it from a file or environment variables. For submission, we will expect submission credentials to be passed through environment

variables; AWS credentials will be passed through ~/.aws/credentials. You may find it helpful to use the AWS CLI to generate a credentials file by using **aws configure**.

- If you believe that your API calls are correct, but they fail to get resources within a reasonable amount of time (5-10 minutes), it's possible that your request is asking for resources from an availability zone in high demand. Spot instances may be hard to acquire in certain availability zones. Auto Scaling groups with spot instances may fail to acquire instances if you restrict instances to one availability zone. We suggest that you use a single availability zone first, if you experience high delay of resource provisioning, then consider switching the availability zone or using multiple availability zones.
- Be sure to delete Auto Scaling groups before terminating the instances created by Auto Scaling. Note that, terminating instances without deleting the Auto Scaling Group first may result in Auto Scaling Group trying to boot more instances.
- Endpoint: `/warmup?dns=[]` can be used to start a warmup test, which
  - Returns 200 if the warmup has been started successfully.
  - Returns 400 if your ELB is not responding or there is another test running. Please check the Troubleshooting section for more information.
  - Returns 500 if your web service is not responding. Please check the Troubleshooting section for more information.
  - Ignore the RPS reported during the warmup at the beginning, as it takes time for your ELB to scale and start accepting requests at a large scale.
- API: `/autoscaling?dns=[]` can be used to start a new Auto Scaling test
  - Returns 200 if a test has been started successfully.
  - Returns 400 if your ELB DNS is not working as expected or there is another test running. Please check the Troubleshooting section for more information.
  - Returns 500 if your web service is not responding. Please check the Troubleshooting section for more information.
- You should be able to access your web service via `http://[ELB Public DNS]/lookup/random` before you can submit (the URL may return a 504 error if the host instance is busy)
- Your RPS should eventually grow during the warmup.
- If your autoscaling group has been provisioned but fails to launch instances, make sure you are passing in security group IDs as opposed to security group names into your APIs.
- If your test is stopped before completion, remember to destroy any provisioned resources to save money.
- The "elasticloadbalancingv2" module is used in the java skeletal code we provide you. Make sure you are not conflating usage of the elbv1 and elbv2 modules in your code. This is not to be confused with Java AWS SDK v1 and v2, the skeletal code uses Java AWS SDK v1 with elbv2. You can reference the correct elbv2 docs for Java (https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/elasticloadbalancingv2/AmazonElasticLoadBalancing.html) here.
- We provided you with a script `student_scaling_script` which transparently mimics the load generation from the load generator. You may use it by invoking `./student_scaling_script LOAD_BALANCER_DNS` and following the prompted instructions.

Warning

## Cloudwatch Monitoring Hints

During previous semesters, we observed that students who used the trial and error approach to tune the policy tended to spend too much time and budget. It is advisable that you learn and follow best practices in order to create policies that can work with the test patterns. The load pattern is designed to encourage structured approaches to monitor the pattern and tune the policy. Observe and analyze the pattern, experiment with a policy, collect data to verify why it achieved a certain performance, and iterate until you achieve your goal.

You can navigate to the Auto Scaling Group in the web console, and enable Auto Scaling group metrics collection under the `Monitoring` tab of the Auto Scaling group. These metrics describe the group rather than any single instance in the target group.

We suggest that you monitor the pattern by the metrics included but not limited to:

- ELB
  - Total Request Count
  - Number of 200 requests
  - Number of 400 requests
  - Number of 500 requests
- EC2
  - CPU Utilization
- ASG
  - Number of healthy/unhealthy instances

Information

## Troubleshooting

- If you observe unhealthy hosts for more than one short period of time during the run, check your health check configuration, timeouts, etc. Note that sometimes ALB health checks are less reliable when instances are under a heavy load because health check requests are queued together with regular requests. An improper health check configuration may misidentify healthy instances as unhealthy when the instances are under a heavy load.

- If your CloudWatch alarm is stuck in `INSUFFICIENT_DATA` state, make sure you set all parameters correctly for the alarm (especially `Unit` and `Namespace`).
- If your RPS is nearly zero (0 <= RPS <= 1), please
  - Check if there is at least one healthy instance connected to the ELB. Note that your solution should be tolerant to instance failures (i.e., when an instance is stopped or terminated - e.g., due to a hardware failure).
  - Check if appropriate parameters are used when creating Scaling policies and CloudWatch alarms.
- If you are receiving 400 status code requests, make sure that the web service instances behind the load balancer are healthy/have completely started
- A large number of 500 requests on the graph shows that you're scaling slowly such that the web server on your instance is getting overloaded
- If your code never terminates at warmup or autoscaling even though the logs on the load generator show that your respective test has finished: This may be a networking issue. Wait a few minutes after the warmup/autoscaling test has finished on the load generator, then stop the execution of your code. Check for log files in your working directory (your code copies over the log files from the load generator). If the test has started and the respective log files for warmup/autoscaling do not exist or if the existing log files are not up to date with the logs at the load generator, there was likely a networking issue. If you are working on a student vm, you can try executing your code on your local machine to verify there was a networking issue on your vm. If you continue to encounter issues on your student vm, try waiting a few minutes and resubmitting on the same vm.

# Additional Resources and References

Autoscaling Developer Guide (http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-dg.pdf)

Autoscaling API Guide (http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-api.pdf)

CloudWatch Documentation (http://aws.amazon.com/documentation/cloudwatch/)

Elastic Load Balancing Documentation (http://aws.amazon.com/documentation/elastic-load-balancing/)

AWS Autoscaling with Terraform

# AWS Autoscaling with Terraform

You have passed the first round of interviews of the MSB System Architect by completing the design and implementation of auto-scaling group with fine-tuned scaling policies using AWS SDK.

Here comes the final round of interview. MSB asks you to reproduce the auto-scaling group with fine-tuned scaling policies you designed and implemented using Infrastructure as Code.

Before completing this task, review the Infrastructure as Code primer to gain a better understanding of infrastructure automation and the relevant tools.

Information

## Task Notes

1. You are not required to run the 24-minute test for this task.
2. You are only required to complete the Terraform template and run `terraform plan` to validate your script.
3. You should assign the values to all the variables you defined so that Terraform will not ask the user for manual input of unassigned variables when running `terraform plan`.
4. You are only allowed to use the following resources to complete this task. Using different variants of these resources will not fetch you a full score.
   - aws_cloudwatch_metric_alarm
   - aws_lb_listener
   - aws_launch_configuration
   - aws_autoscaling_group
   - aws_lb_target_group
   - aws_autoscaling_policy
   - aws_subnet
   - aws_lb
   - aws_instance
   - aws_security_group

Information

# Tasks to Complete

In the previous task, you used the AWS APIs to orchestrate various AWS resources and launch an autoscaling web service. From the examples provided in the Infrastructure as Code primer, you may expect that infrastructure automation tools could be used to deploy similar or even more complex architectures.

Starting from the provided Terraform configuration template file - `task3-terraform.tf` under the `task3` folder, you will complete the configuration so that it achieves the autoscaling architecture defined in the previous task. Your task is to implement the Terraform template to:

1. Create two security groups to allow **incoming traffic on port 80 and all outgoing traffic on all ports**. One of them should be associated with the Load Generator and one with your Elastic Load Balancer and Auto Scaling Group.
2. Create a Load Generator instance of size `m5.large` using `ami-0273735cd4c306ac0` with one of the security groups you created in step 1.
3. Create a Launch Configuration for the web-service instances that will become part of the Auto Scaling Group, with the following parameters:
    - AMI ID: `ami-042de649749923897`
    - Instance Type: `m5.large`
    - Detailed Monitoring: enabled
4. Create an Application Load Balancer that forwards the HTTP:80 requests from the load balancer to HTTP:80 on the instance. You will have to use the target group from step 5 to configure the listener.
5. Create a Target Group for instances using HTTP:80. Use / (which is the heartbeat endpoint of the web service in this project) as the Health Check path and use HTTP. Note that ELB health checks may become queued behind other requests. On a congested server, this may cause health checks to fail.
6. After analyzing the traffic pattern from the Load Generator, create an Autoscaling Group. Below we define the following default rules to give you an idea of the various parameters. Part of the configuration will be covered in the next steps.
    - Group Size: Start with 1 instance
    - Subnet: We recommend that you choose the same availability zone(s) corresponding to your ELB
    - Load Balancing: Receive traffic from the created Target Group
    - Health Check Type: EC2
    - Detailed Monitoring: enabled
7. Create the following scaling policies as the starting point: Please note that you may find it **difficult** to achieve a passing grade using an Auto Scaling Group with the following parameters.
    - Minimum Instance Size: 1
    - Maximum Instance Size: 2
    - Create a Scale Out policy that automatically adds 1 instance to the Auto Scaling Group.
    - Create a Scale In policy that automatically removes 1 instance from the Auto Scaling Group.
8. Create CloudWatch Alarms that invoke the appropriate policy for the following scenarios: Please note that you may find it **difficult** to achieve a passing grade using an Auto Scaling Group with the following parameters.
    - Scale Out when the group's CPU load exceeds 80% on average over a 5-minute interval.
    - Scale In when the group's CPU load is below 20% on average over a 5-minute interval.
9. Link the CloudWatch Alarms to the Scale Out and Scale In rules of your Auto Scaling Group. You will be able to add an Auto Scaling policy with the alarms and scaling actions by editing the Auto Scaling Group.
10. Configure correct tags for your Load Generator/Auto Scaling Group using your code.

## What to Submit

In order to submit the code for this module, you will need to execute the task 3 submitter. The submitter will run the test and submit the source code. A reminder that your last submission of each task must include a `references` file where you cite all the reference sources when you work on this task. We strongly recommend that you have your reference file ready in the task folder to avoid re-running the test. Please do not include any binaries (.jar) or cloud credentials (.pem, access_key.csv) files. You will incur a 100% penalty if we found any credentials files in your submitted code.

Please do not include any terraform directories or state file ( `.terraform` , `.tfstate` ) or cloud credentials (.pem, access_key.csv) files. You will incur a 100% penalty if any credentials files are found in your last submission of each task.

When submitting, please make sure the following files are in your project folder:

- the reference under `~/VM_Scaling_Project/<java/python>/task3/references` with all the links that were referred when completing this task.
- The Terraform script under `~/VM_Scaling_Project/<java/python>/task3/task3-terraform.tf`

## How to Submit

Since you will be communicating with the AWS APIs, you will first need to authenticate with AWS with `aws configure` and entering the required fields. You can refer to this documentation (https://docs.aws.amazon.com/powershell/latest/userguide/specifying-your-aws-credentials.html) for the location of the credentials on your local machine.

```
export SUBMISSION_USERNAME=your_submission_username
export SUBMISSION_PASSWORD=your_submission_password
./task3_code_submitter
```

Project Reflection Task (Mandatory, graded)

# Project Reflection Task (Mandatory, graded)

Upon completing this project you will make one (1) post in the [Forum] P1. Elasticity (https://projects.sailplatform.org/cloud-forum/topic/publish/1029/) to reflect on your experience before the project deadline. **Before you publish the post, please double check that the category is the project category, NOT the course category "Cloud Computing" or any other project category**.

Consider the following topics when creating your post, however, you should never share any code snippets in your reflection:

- Describe your approach to solving each task in this project. Explain alternative approaches that you decided not to take and why.
- Describe any interesting problems that you had overcome while completing this project.
- If you were going to do the project over again, how would you do it differently, and why?

After completing this task, confirm that your ***Reflection Score*** has been updated on the scoreboard before the project deadline.

Success

If you have successfully completed all the tasks and obtained a score, congratulations and welcome to the MSB.

Now you have already got some experience working with virtual machines and adding scalability, performance and fault-tolerance to your applications. In the next project, we will show you how to play with docker and Kubernetes, which are very popular in the industry. Apart from their scalability, performance and fault-tolerance, they are also very easy to use and benefit the continuous integration and continuous deployment.

Please leave us feedback for this project here. This will help us strengthen this project for future offerings.

Project Discussion

# Project Discussion (Mandatory, graded)

After the project deadline, all reflection posts by all students will become visible for review.

Your task is to reply and provide feedback to **3** posts in the [Forum] P1. Elasticity (https://projects.sailplatform.org/cloud-forum/category/1029/), within **7** days after the project deadline.

After completing this task, confirm that your Discussion Score has been updated on the scoreboard within 7 days after the project deadline.