# Programming Assignment#4 Report

## Introduction:

This programming assignment involves implementation of dynamic provision for a task execution framework on Amazon EC2, making use of Simple Queue Service (SQS) and CloudWatch.

- Amazon EC2 is used for instance creation and computation. The Amazon EC2 functionality is as follows:
  1. Select a pre-configured, templated Amazon Machine Image (AMI) to get up and running immediately. Or create an AMI containing your applications, libraries, data, and associated configuration settings.
  2. Configure security and network access on your Amazon EC2 instance.
  3. Choose which instance type(s) you want, then start, terminate, and monitor as many instances of your AMI as needed, using the web service APIs or the variety of management tools provided.
  4. Determine whether you want to run in multiple locations, utilize static IP endpoints, or attach persistent block storage to your instances.
- Amazon SQS for storing messages and move data between different instances. The main functionality of SQS as follows:
  1. Can create an unlimited number of Amazon SQS queues with an unlimited number of messages. Messages can be sent, received or deleted in batches of up to 10 messages or 64kb. Batches cost the same amount as single messages, meaning it can be even more cost effective for customers that use it.
  2. When a message is received, it becomes "locked" while being processed. This keeps other computers from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again. In the case where the application needs more time for processing, the "lock" timeout can be changed dynamically via the ChangeMessageVisibility operation.
  3. Can securely share Amazon SQS queues with others. Queues can be shared with other AWS accounts.
- Amazon CloudWatch offers monitoring AWS cloud resources i.e. Amazon EC2 and the applications. Once registered for Amazon CloudWatch all Amazon EC2 instances are automatically enabled for Basic Monitoring which collects and reports metrics for CPU utilization, data transfer, and disk usage activity from each Amazon EC2 instance at a five-minute frequency. Amazon CloudWatch Detailed Monitoring provides these same metrics at one-minute intervals, and also enables data aggregation by Amazon EC2 AMI ID and instance type.

## Problem:

The major goal of this assignment is as follows:

1. To test the performance for dynamic provisioned instances in relative to the manually scaled instances.
2. To scale up and down the instances using CloudWatch depending the workloads.
3. To read the tasks in the file by the client and send it to the scheduler.
4. Further the scheduler takes all the tasks and places in a memory queue for local workers. Whereas in case of remote workers store in SQS.
5. Here local workers are the threads which execute the tasks in from in-memory queue.
6. At last, the results are stored back into the in-memory queue for local workers i.e. –lw and remote workers i.e. –rw, which doesn't specifies the number of workers stores the result in queue on AWS.

# The Task Execution Framework:

## The Client:

The Client is a command line tool, which submits the task to the front-end scheduler. The major roles of the client are:

1. The client reads the file from the local and sends to the server task by task.
2. The command used for running the client as below
   *java -cp prog4.jar Client <<Server Name>> <<Port No>>*
3. In above command, *Server Name* is the name of server to which the data is sent and *Port No* specifies the port number on which the server is listens. Local host is provided as the server is the current machine.
4. Hash Map is also implemented, in order to match the received task ID results with the corresponding submitted ID.

**Extra Credit**: We have also optimized the results with batching results together and implemented notification handler, to eliminate the polling need. This involves following steps:

1. The receipt results are batched together and sent to the server.
2. Each task is separated by the % symbol.
3. Further, the results are put together on the result queue which is as in-memory queue for local workers and SQS for remote workers.
4. All the results are then batched together and sent to the client.

## The Front-End Scheduler:

This offers the network interface to the client, in order to transfer the tasks from client to the server. The design of this scheduler as follows:

1. The server listens to the port no. it specified in *scheduler –s <PORT> -lw <NUM> -rw*.
2. The tasks are sent in batches or sequentially.
3. If switch set as *'lw'*, the tasks are written to a in-memory queue which is implemented using linked list.
4. Otherwise if switch is *'rw'*, the tasks are written to the SQS on AWS assigned name as *'myQ'*.
5. In case of 'lw' switch, we may specify the number of threads.
6. The thread executor service pops and item from the queue and submits it to the thread pool.
7. The free threads take the next task and execute it.

## Local Back-End Workers:

In this implemented a pool o threads that will process tasks from the submit queue, and when complete will put results on the result queue. The functionality as below:

1. The numbers of local workers are defined by the number of threads as specified in the scheduler.
2. This method handle the sleep functionality by taking tasks sleeps for a specified length in milliseconds.
3. The results are written in another queue as soon as the thread completes running the tasks.
4. An exception is raised on the failure of the thread and the return value is stored in the result queue.

## Remote Back-End Workers:

In this some back end workers are implemented that run on different machines to facilitate the scalability. The workflow is as follows:

1. The tasks are stored into SQS on AWS for switch *'rw'*.
2. A custom AMI is also created for holding the jar file.
3. The remote worker polls the SQS queue. If it finds any tasks, they are retrieved and executed. Otherwise it polls again after some time.

4. If the poll remains idle for too long, the remote worker is terminated.

**Extra Credit:** We have also allowed multiple tasks retrieval simultaneously and allow multiple tasks to run concurrently and all the completed tasks are returned back to the scheduler. The design of this functionality is as follows:

1. Implemented threads to run multiple tasks in one local worker.
2. The numbers of threads are identified with the command line argument in the remote worker command.
3. Now, each remote worker is allocated the number of tasks equivalent to the no. of threads.
4. Each remote worker implements a thread executor service and executes the tasks concurrently in each remote worker.

### Dynamic Provisioning of Workers:

Dynamic provisioning of resources provides scalability and elasticity. Thus in case of SQS service is empty it doesn't allocate any resources. Otherwise it has at least 1 worker. The implementation of this functionality is as follows:

1. The CloudWatch is implemented for monitoring SQS queue every 1 minute time interval.
2. In case of number tasks = 0, the CloudWatch does not start any instances.
3. If there are at least 1 instance or more and less than 500, CloudWatch instantiates 16 instances.
4. Above each instance is same AMI, which manual workers uses and the tasks executions starts with the start of the instance.
5. In order to fetch number of messages in SQS and the number of instances are running, a CloudWatch is implemented.
6. On the basis of all available informations, the scaling up of the information is done.

## Code Design:

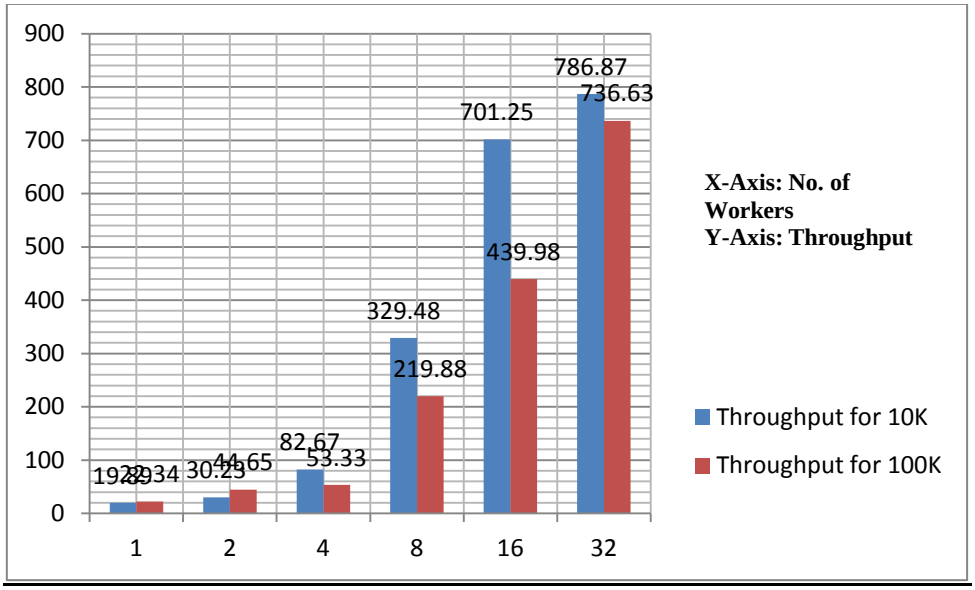
The detailed description of source code files as follows:

i. *AmazonInstance.java*: It creates the instances, security groups and the key-pair for the specified AMI.
ii. *Client.java*: The file with all the jobs is passed from the client to the server. And the acknowledgment is received from the server.
iii. *CloudWatch.java*: It monitors the activity of the instances as and when the jobs are executed.
*iv.* *LocalWorkers.java:* Function to insert the tasks into the thread pool.
*v.* *QueApp.java:* Creates the Linked List that holds the jobs to be executed.
*vi.* *RemoteWorkers.java*: It gets the tasks from SQS and adds it into the lists and then sends the lists to the process to start thread pooling.
*vii.* *Scheduler.java*: Waits for the connection form the client and sends the tasks from the client file to the SQS.
*viii.* *SimpleQueueServiceSample.java*:
*ix.* *ThreadPoolService.java*: Insert the task into the executor as a runnable task which will execute the sleep method and push the result to resultQueue.

## Performance Evaluation:

The maximum throughput is determined by dividing the number of tasks processed by the total time from submission of the first task to the completion of the last task. Below table demonstrates the throughput values of 10K tasks and then tasks are increased to 100K.

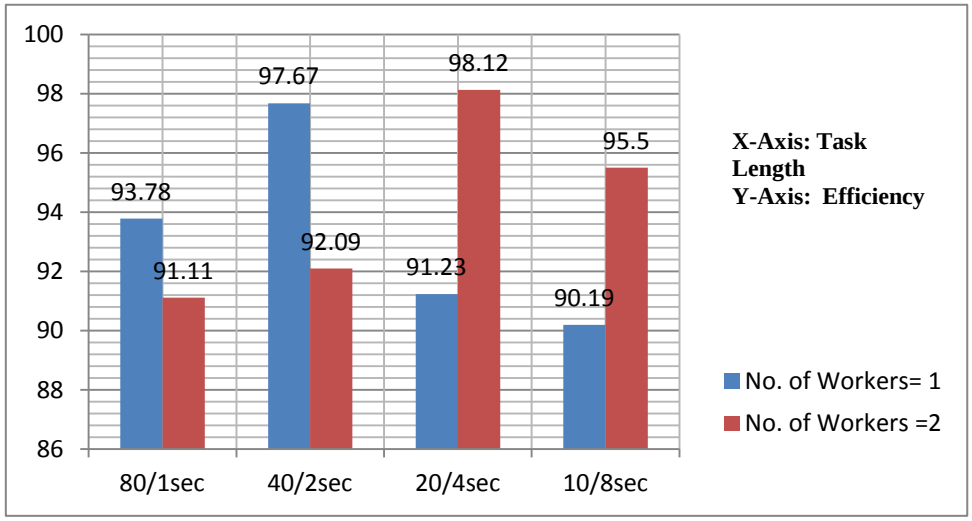
| | Number of Workers | | | | | |
|---|---|---|---|---|---|---|
| **No. of Tasks** | **1** | **2** | **4** | **8** | **16** | **32** |
| **10K** | 19.89 | 30.23 | 82.67 | 329.48 | 701.25 | 786.87 |
| **100K** | 22.34 | 44.65 | 53.33 | 219.88 | 439.98 | 736.63 |




**Graph1: Throughput at varying number of workers**

The efficiency is measured at varying number of workers from 1 to 32 and at varying sleep time from 1 sec, 2 sec, 4 sec and 8 sec. The numbers of tasks are 80 per worker for 1 sec tasks, 40 per worker for 2 sec tasks, 20 per worker for 4 sec tasks and 10 per worker for sec tasks.

| | Tasks Length | | | |
|---|---|---|---|---|
| **Number of Workers** | **80/worker for 1 sec** | **40/worker for 2 sec** | **20/worker for 4 sec** | **10/ worker for 8sec** |
| **1** | 93.78 | 97.67 | 91.23 | 90.19 |
| **2** | 91.11 | 92.09 | 98.12 | 95.50 |




**Graph2: Efficiency at varying number of workers**

## Observation and Comparison with Falkon:

From the above benchmarks we conclude that:

1. Throughput is directly proportional to the number workers.
2. Throughput for 100K tasks is varying significantly as compared to 10K tasks.
3. As per graph 2, the average efficiency single workers by varying the task lengths is approximately 93 seconds.
4. In graph 2, the increase in number of workers has no impact on the efficiency as the number of tasks also increases on the whole for the workers.

While comparing with Falkon, we infer that

1. By increasing the data size in experiment, it provides more accuracy, which is demonstrated in our benchmark also in Graph 1 for 10K and 100K number of tasks.
2. Our experiments don't implements more than 32 executors and thus throughput increases. While in Falkon, it is tested on 256 numbers of executors and throughput is steady.
3. Our benchmarks do not take more than 10 seconds to run 10K tasks of sleep0. Therefore our throughput is more than 700 tasks.

## Dynamic Provisioning:

1. The experiment was performed for only 3 stages.
2. In first stage we passed workload with 500 sleeps with 0 tasks and 16 instances were also created. The total time taken by tasks to complete is around 3-4 seconds.
3. While in stage 2, 1000 more tasks are passed from client to server and number of instances are scaled up to 32 which took around 3 seconds.
4. At last in 3$^{rd}$ stage, 500 jobs are sent to the scheduler and are executed on 32 instances. This took approximately 2-3 seconds.

***Submitted By***:
Shivanshu Misra (A20279849)
Pavan Praveen Kulkarni(A20277967)