Cloud Computing & Big Data

Riley Evans - re17105

University of Bristol

Abstract. The use of cloud computing for executing embarrassingly parallel tasks is now easier that ever. Tasks are able to scale many instances with ease, without the overhead of managing many machines. This report will implement an example of one of these tasks - a coursework submission auto marker.

1 The Task & Motivation

The embarrassingly parallel task chosen for implementation is an auto-marker for students Game of Life coursework submissions. This involves ensuring that all the tests pass and running benchmarks on the solution so that markers can compare the efficiency of each submission.

This task is suited to processing in an embarrassingly parallel fashion - each submission has the same process run on it. It is also beneficial to run this task in a scalable environment for several reasons. Firstly, each submission can take from a few minutes to mark up to 84 minutes in the worst-case scenario depending on if a student has deadlocks present in their code. With the ever-increasing demand for quicker feedback and increasing student numbers, it would be helpful if this task could be parallelised to speed up the marking of coursework. Secondly, if there is one slow submission it would not be ideal if it blocked the marking of every other submission.

Another justification for running it on the cloud is that it provides multiple machines all of the same specification. One could argue that if each marker were to run separate batches of submissions on their own personal machines, it could lead to inconsistencies when marking. Especially, if each marker has a different specification machine. On a cloud provider, using the same instance tier results in identical machines being provisioned.

2 The Architecture

The architecture can be divided into two sections - client facing and submission processing. Fig. 1, shows how a client interacts with the system and the components that are accessible to it. A client will initially invoke a Lambda function to register itself as a client in the system. The Lambda function will then return a client ID along with an SQS queue for it to listen for results on. The client then uploads zip files to an S3 bucket. Once a submission has been processed a message will be sent to the client's SQS queue. The client is then able to download

the file from an S3 bucket. The client has two different modes: asynchronous or synchronous. The asynchronous mode is helpful when marking can take longer that the 3 hours than the AWS Educate access keys are active for. By assigning client IDs the system is capable of differentiating work between each client. This means that it is able to handle multiple clients at the same time.

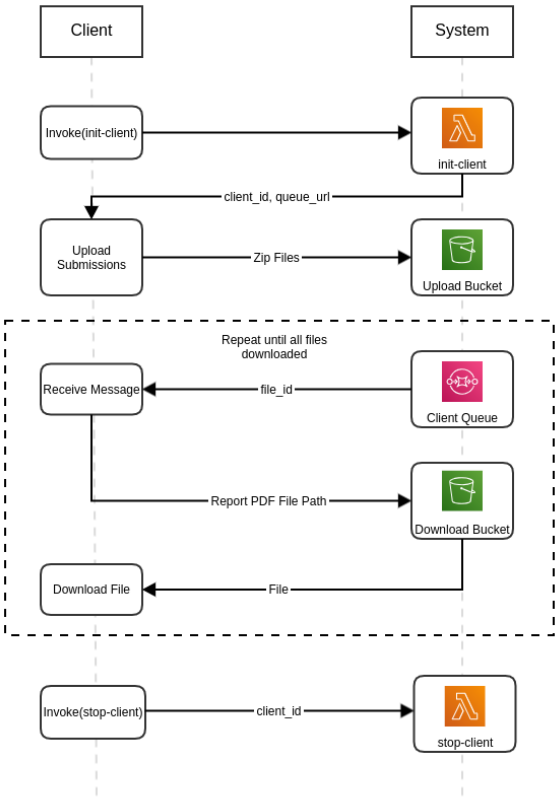

Fig. 1. Client interaction diagram

The submission processing part of the system is illustrated in Fig. 2. This depicts how a submission is taken from one S3 bucket and converted into a report PDF in a second bucket. The architecture has 3 key components, the Elastic Container Service (ECS) cluster, the Simple Queuing System (SQS) and the Lambda functions used for auto-scaling.

The auto-mark script that is used to produce a report for the submission is run inside a docker container, there are several reasons why this is suitable. Firstly, it means that the marking of each submission can be contained, this prevents the student from running malicious code, that could either cause problems with the host machine or another students submission. Secondly, it allows a specification to be given to students so that they can ensure that their sub-

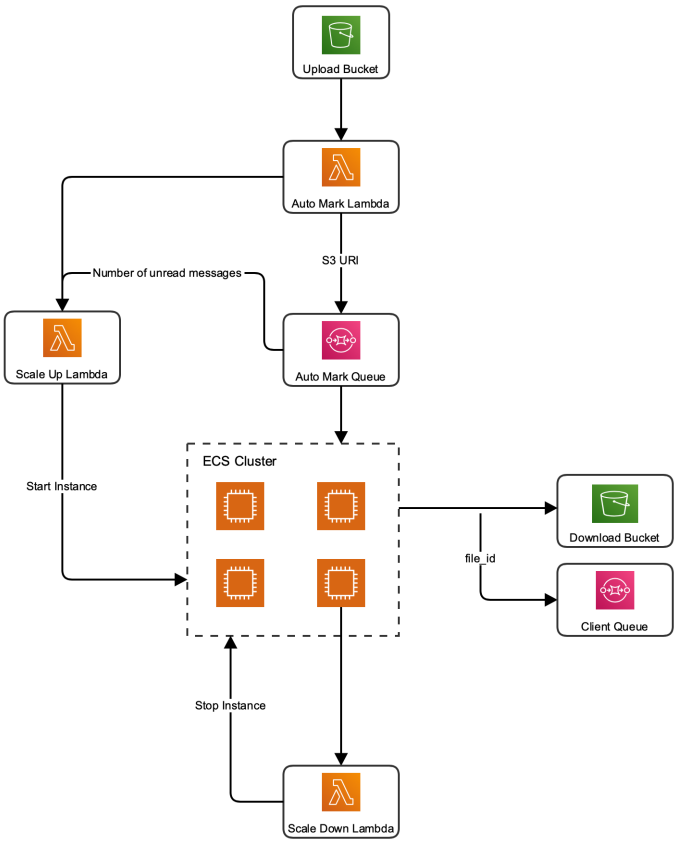

Fig. 2. System Architecture diagram

mission will run inside the environment without problems. This will help build confidence between students and markers.

Each container will run within the Elastic Container Service. In order to do this a task definition is needed, this states what container image will be run and any resource allocations that it requires. For example the auto-mark task requires 4096 CPU units, this is equivalent to 4 full CPU cores. A service is then specified within the cluster, this will be able to schedule tasks on available EC2 container instances. The service is able to control the number of desired tasks running. It ensures that the containers running on an EC2 instance do not exceed the maximum resources that it has available. If a container crashes or becomes unhealthy, the service will be able to stop that container and start a new one in its place.

The auto-mark SQS queue is used to manage a list of all the submissions that need marking. This helps to provide fault tolerance within the system - discussed later in the report. Uploads to the S3 upload bucket will trigger a message to be sent to the queue, and tasks within the ECS cluster will request messages from this queue so they know which zip files to mark.

3 Auto Scaling

There are many parts of the academic year where having this system running EC2 instances would be a waste of money, as there is no work to mark. However, towards the end of term there could be large demand from lecturers who would like to auto-mark work. Therefore, the ability for this system to auto-scale is crucial. Ideally, when this system is not being used it uses the least amount of resources possible to avoid costs being incurred.

Initially, one solution that presented itself was to use the auto-scaling built into ECS. However, after investigation, there were several problems with this approach. The auto-scaling system did not easily allow for control over which instance would be terminated, which could be problematic. If we are marking submissions on 7 out of the 8 running container instances with no more work to do, then ideally the one not marking would be terminated. However, this is difficult to do when not controlling which instance is terminated. Instead, an instance that is currently marking a submission could be terminated causing it to fail to mark a submission.

Another solution to this problem could be to invoke a Lambda function on a schedule - similar to a cron job. This would be able to check the state of the queue and decide when to start more instances or which instances to terminate. However, this comes with the problem that a lambda will be invoked, even when there is no need to. Additionally, AWS Educate prevents recurring Lambdas.

The auto-scaling solution implemented aims to solve both of these problems. When work is added the the auto-mark queue then a Lambda function will be triggered to check if an instance needs to be started. The Lambda function is able to start at most 4 instances in a single invocation. This allows for the number of instances to rapidly grow when tasks are added to the queue. Instances are shutdown by reporting their status to another Lambda when they have not processed any work for a set period of time - 200 seconds. One could question what happens if a container is faulty and not able to report its state. In this scenario it will be reported as unhealthy and stopped. The effect of this auto scaling solution can be seen in Fig. 3, this example uses a maximum of 8 instances.

One benefit of this system is that when no jobs running you can have no instances running. This will reduce the standby cost of the system, especially since Lambda functions are only charged when they are run - the standby cost of this system is $0.00.

4 System Performance

Being able to process submissions quickly would be an ideal feature of this system. The system runtime, however, will be limited by the length of time each submission takes to mark - this can vary significantly. Using an average timed solution, the throughput of the system can be calculated. To do this the time will be measured for 100 submissions to be processed, from the first upload time to the last download time. The system executed for a total of 133 minutes and completed 100 submissions giving a throughput of 0.75 submissions per minute.

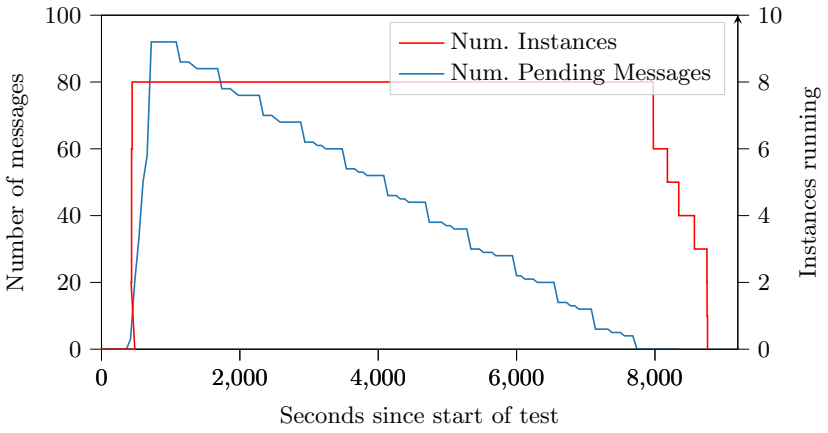

Fig. 3. How the number of active instances scales with the number of pending messages.

The system startup time can be estimated from the latency it takes to complete the first submission. In this case the latency was 12 minutes. With each submission taking between 10-11 minutes, this would suggest that our system has a startup time of roughly 1-2 minutes. A breakdown of where time is spent can be seen in Table 1. Each task in the startup process is dependent on the previous, therefore giving a startup time of 81 seconds.

Table 1. Startup time of each component in the system

Task	**Runtime (s)**
Auto-Mark Lambda	1
Scale Up Lambda	6
EC2 Instance Start	43
ECS Image Pulling	31
Total	81

It is also worth comparing this cloud based solution to another possible way of marking the submissions. They could be marked on a single machine in a sequential fashion. It would not be possible to run them in parallel as the submissions need full access to *all* CPU cores. A single submission took 10 minutes 42 seconds to run on a 4-core i5-4690 processor. From this It can be estimated that 100 submissions would 17 hours 15 minutes. This is significantly slower than the 2 hours 13 minutes for the cloud based solution. This would mean that a lecturer spends more time waiting for the submissions to be marked, and would therefore have less time to give more detailed feedback.

5 Cost

To evaluate the cost of the system, a sample of 100 submissions can be run on the system, and then evaluating the number of resources used. The breakdown of

the costs can be seen in Table 2. To estimate the cost of the EC2 instances, the length of time they ran for needs to be calculated. This can be calculated from the area under the curve of the number of instances shown in Fig. 3. This results in a value of 63830 seconds of running instances - EC2 instances are charged per second [1]. The EC2 instances are the most major cost in this system - they represent 98.3% of the total costs. The instance type chosen was c4.xlarge, this is because that type was recommended to the students to use. Therefore, it would not be possible to use a different one to reduce the costs.

The usage of Lambda, SQS and S3 are all included in the free-tier, however, the costs excluding the free-tier are also calculated. They are small enough to end up with a cost of $0.00. Lambda invocations are charged at $0.0000166667 for every GB of RAM used per second. The average runtime of each Lambda function is estimated to be 3 seconds and all Lambdas use just 128MB of RAM. S3 is billed at $\$3.2 \times 10^{-5}$ per GB-hour, with each submission using on average 4.1MB this can be estimated to 0.4GB. The number of hours is rounded up to 3 hours as any usage within a one hour period will get rounded up, for example 2 hour 20 minutes of usage will be billed at 3 hours.

Table 2. Costs of the system

Resource	**Usage**	**Cost per unit ($)**	**Cost**
EC2 (c4.xlarge)	63830s	0.00005528	$3.53
EBS (general purpose)	63830s × 30GB	3×10^{-8}	$0.06
Lambda Invocations	208 × 3s × 0.125GB	0.0000166667	$0.00
SQS	200	4×10^{-7}	$0.00
S3 (general storage)	3hr × 0.4GB	3.2×10^{-5}	$0.00
		Total	$3.59

A key comparison for the cost could be comparing it to the costs of running in on a local machine. Previously it was calculated that running on a single machine would take 17 hours 15 minutes. Assuming the machine draws 450W of power, this converts to 8 kWh. The cost of power can be calculated as 8×0.163[2] = £1.304 ≈ $1.74. The cost of ownership has not been factored in here, as the machine will already be owned by the marker. This cost is lower than the cost of running the system on AWS, however, it does lockup a machine for 16 hours that cannot be used to avoid interfering with the benchmark times - which could have downsides if other work needs completing at the same time.

6 Fault Injection

Fault tolerance is crucial when marking the submissions: it would not be good if a student has a viva and their submission had failed to process. This system is capable of resolving faults that may occur.

There are several ways this system could fail, firstly, the Python script that is marking could fail. If this occurs then the container stops running and a new one will be started on the container instance. However, the submission that it

was processing will have already been taken out of the SQS queue. In order to be fault tolerant this submission will need to be re-added to the queue, this is possible through an SQS feature - the visibility timeout. When a message is received from the queue, a timeout is set, if this timeout expires before the recipient has deleted the message then it will be re-added to the queue automatically. Each submission could take from 5 minutes up to 84 minutes to mark depending on how good the submission is. Therefore, it would make sense to set the higher as the timeout, however, if one submission fails it could extend the total processing time significantly. It is possible to extend the visibility timeout multiple times after receiving the message. This could be used to allow the timeout to be updated regularly, preventing a large delay from the task failing to when it is decided that it failed. Using this incremental timeout between each subset of tests, the failure reporting delay can be reduced to at most 36 minutes and as little as 30 seconds - a reduction of between 57% and 99%.

This can be tested by adding random crashes to the python script, and observing its effects on the throughput of the system. Between each set of testing a failure can be added at a rate of 1% - giving each task a probability of failure of 8%. As previously done, 100 test submissions can be run through the system, the results of which can be seen in Fig. 4 & Fig. 5.

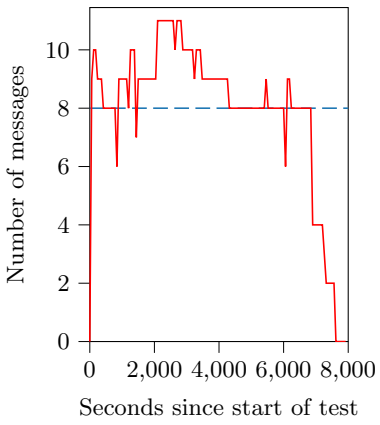

Fig. 4. The number of hidden messages.

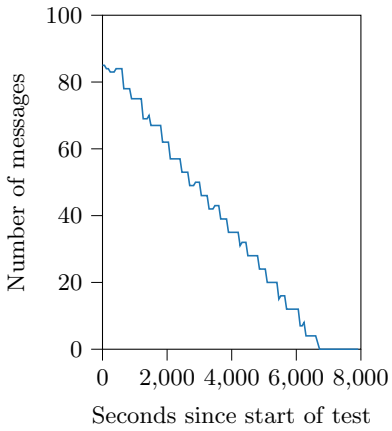

Fig. 5. The number of visible messages.

Fig. 4, visualises the number of hidden messages in the queue. The hidden messages will include all messages that have been accepted out of the queue, but not marked as complete. In this example - where 8 marker tasks are being run in parallel - any value in this graph above 8 shows that there has been a failed task. This is useful for monitoring the system to view how many failed tasks have occurred. At the 2000 second mark, there were 3 task that had all failed but their timeout had not been exceeded yet. Over the next 2000 seconds it can be visualised when the timeouts trigger and the task is moved from hidden to visible again. Fig. 5, demonstrates the number of visible messages in the queue

at any point in the marking process. You can align any uptick in the number visible messages to points in Fig. 4 where the number of hidden tasks decrease back down towards 8. This trend does not apply past 6500 seconds.

The auto-marker container also has a health check to ensure that stays healthy. In a similar method to the queue timeout, a status update is output to a file. This file can be checked to decide if the container is still healthy or if there is a problem. For example, the Game of Life makes use of parallel workers, this could result in deadlocks. Although timeouts have been used, there is the possibility that the Python script could get stuck waiting for a submission that has deadlocked. In this case the health check would fail as the script would not report a new status in the correct period of time. If the health check fails multiple times the container will be stopped and a new one started to replace it.

7 Conclusion & Future Work

In this report, a system has been presented that allows for the auto marking of students coursework submissions in the cloud. It has been demonstrated how the system is able to successfully scale based on demand in the queue, and how when there is no demand the system costs $0.00 to keep in standby. Finally, presenting method for ensuring that the system remains fault tolerant, that ensures that all work is processed with the minimal delay when certain tasks fail.

There are, however, some possible improvements that could be made to make the system even more capable. Firstly, an adversary could take advantage of a client to interfere with processing as they have full access to the AWS API. It would be more secure if a client could be granted access tokens that only allowed access to the certain areas that they need. They could then use a public REST API to invoke the Lambda function, which would return their credentials. Unfortunately due to the restrictions on AWS Educate this was not possible, but if this system was used more widely in the future this would be critical.

Secondly, in its current state the system only supports marking of Game of Life courseworks. A future expansion could allow for the marking of different types of coursework, for example, marking C code submissions. It can be hypothesised that when a client is created, they are able to also specify a container of their own construction. This would then have 2 attached volumes: a submission volume and an artefacts volume. The container could then store the results of auto-marking in the artefacts volume, which would be forwarded to the client. This would allow the system to be more widely used.

References

1. AWS News Blog. Per-Second Billing for EC2 Instances and EBS Volumes. https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/
2. Department for Business, Energy & Industrial Strategy. Annual domestic energy bills. https://www.gov.uk/government/statistical-data-sets/annual-domestic-energy-price-statistics