Cloud Computing and Virtualization EcoWork@Cloud Project

Madalda Ferreira *92513*
João Travassos *105710*
Olga Silva *105714*

1 Introduction

This project consists on designing and implementing the EcoWork@Cloud system running within the Amazon Web Services ecosystem. The system is composed by workers, including EC2 instances and Lambdas, a Load Balancer, an Auto-Scaler and a metrics storage system hold by DynamoDB. Figure 1 represents the architecture of our system implementation.

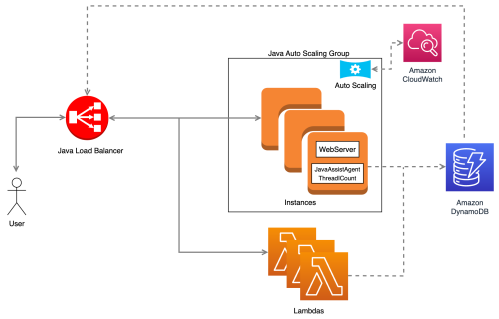

Figure 1: System architecture

2 Methodology

This section outlines the implemented features within the system. To determine the cost of a request, metrics are collected and stored in DynamoDB by each worker and subsequently applied to new requests by the Load Balancer. The Load Balancer (LB) is a Java application running on a single VM. It utilizes a task scheduling algorithm to forward requests to the appropriate Worker VMs based on their respective costs, and when this is not possible, it triggers a Lambda function invocation. Additionally, the system employs an Auto-Scaler (AS) algorithm, which operates as a separate thread alongside the LB, to determine when to scale the pool of workers.

2.1 Instrumentation Metrics

The Javassist framework contains a tool called "ThreadICount", used to record the total number of instructions and the number of load and store instructions performed by each request. It tracks the thread ID of the thread that is executing the current request, stores the request parameters collected by the same thread in the request handler, and increments the number of executed instructions and memory accesses at each basic block. Upon completion of the request, both the total and the load/store instruction count, along with the captured parameters, are recorded. Even though the load/store instructions are already accounted in the total number of instructions, they are also individually stored with the intent of giving them more weight (2x) in the metric acquisition phase, since these instructions typically take longer to execute than non-memory access ones.

2.1.1 Webserver Instrumentation

Meanwhile, a separate thread tries to update DynamoDB with the current existing values recorded by the instrumentation. In an infinite loop, sleeping fifteen seconds in between each iteration and, for each type of request (Foxes and Rabbits, Insect War, Image Compression), the thread loads the current metrics stored in DynamoDB and tries to update them locally with the recorded values from the 'ThreadICount' instrumentation. If impactful changes were made to the loaded metrics, they are updated and written to DynamoDB, otherwise nothing happens.

During each iteration, readings from DynamoDB are performed a number of times equivalent to the existing applications, namely Foxes and Rabbits, Insect War, and Image Compression. This is because each application has its own dedicated table within DynamoDB.

Writes are performed whenever significant changes that are crucial to update the metric occur. Since this thread sleeps for a while, it can gather multiple values from executed requests that require tables to be updated. At most, it would have to update all entries from all tables, which are limited as explained in section 2.3.

2.1.2 Lambdas Instrumentation

Ideally, instrumentation should be independent of the type of environment. Therefore, Lambda functions also execute instrumented bytecode for Foxes and Rabbits and Insect War to gather instrumentation metrics. As a result, as soon as the event handler finishes processing the requests and before returning the response, it obtains the total work, i.e., the total number of instructions and the number of memory accesses, and updates the metrics in DynamoDB if required.

2.2 Request Cost Estimation

To better understand the instructions executed by the applications, a python script was developed to create plots that were used to analyze the correlation between different parameters and workloads. From now on, we refer to the total number of instructions plus the number of memory accesses as the workload of a request.

2.2.1 Foxes and Rabbits

After some iterations, we observed that, in Foxes and Rabbits, the number of instructions is proportional to the number of generations, as depicted in Figure 2. However, since the workload is different for each combination of world and scenario, we decided to divide the analysis for each. For instance, since there are 4 worlds and 3 scenarios, we have a total of 12 different outcomes based on the number of generations. After executing multiple requests, we decided that the best approach for workload estimation would be the usage of slopes.

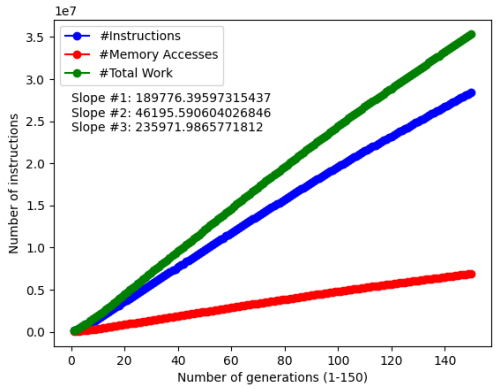

Figure 2: Workload variation with different number of generations for world 2 and scenario 2.

In this case, we track the lowest and highest number of generations (X axis) and store the corresponding number of workload (Y axis). Then, the slope is simply be the result of

$$slope = (max_{work} - min_{work})/(max_{gens} - min_{gens})$$

Afterward, the estimated workload for a request with a different number of generations is:

$$estimated\_work = slope \cdot num_{generations}$$

With this approach, we achieved very similar estimated workloads. However, to achieve an ideal slope, we would need to perform numerous requests. Therefore, we store, in our metric storage system, the slope alongside the four values, i.e., minimum and maximum number of generations and the corresponding workload for each. Then, if the number of generations of a new request is smaller or higher than one of the values stored in DynamoDB, a new slope is computed. Ultimately, both values and slope, named as `Metric` in our database, are updated. This is because the new slope value is calculated using values that represent more distant datapoints:

```
if new_gens > max_gens:
    max_gens = new_gens
    max_work = new_work
else if new_gens < min_gens:
    min_gen = new_gens
    min_work = new_work
```

2.2.2 Insect War

Regarding the estimation cost of Insect War requests, we observed a similar tendency as the one presented previously in Foxes and Rabbits. For instance, the workload also increases linearly with the increase of the number of rounds as well as the increase of the total number of army size. The latter can be observed in Figure 3.

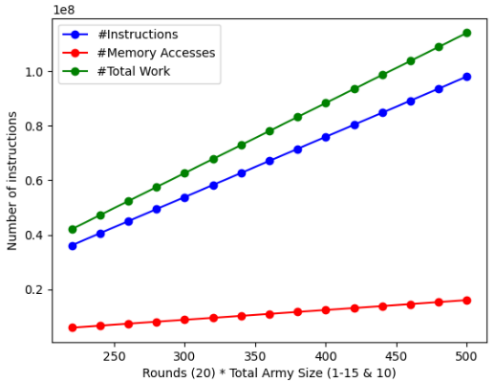

Figure 3: Workload variation with different total army size for 20 rounds.

On the other hand, as expected, the ratio between the highest army size and the lowest one impacts the workload, as seen in Figure 4.

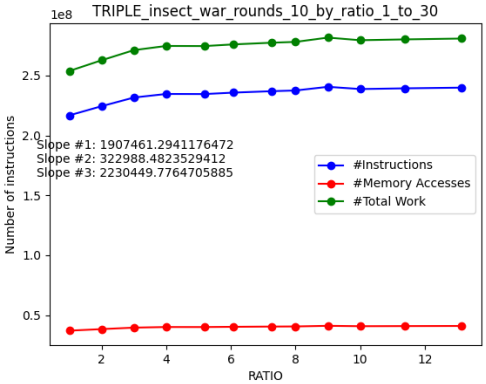

Figure 4: Workload variation with different total army size for 10 rounds and 200 of total army size.

As a result of these observations, we decided that, for each ratio interval, we would use a slope that depends on the number of rounds and total army size, since they contribute to the workload with the same weight. This statement can be supported by observing that the workload increases roughly twofold when either the number of rounds or the total army size doubles. Hence, the slope must depend on these two parameters:

$$roundArmy = rounds \cdot (army1 + army2)$$

$$slope = (max_{work} - min_{work})/(max_{roundArmy} - min_{roundArmy})$$

Then, similarly to Foxes and Rabbits, these four values (maximum `roundArmy`, minimum `roundArmy` and corresponding workloads) are stored in DynamoDB and the `Metric` value, i.e., the slope, is updated when new values result in a computed slope that is based on more distant datapoints. Ultimately, we are able to estimate new workloads for new requests as follows:

$$estimated\_work = slope \cdot roundArmy$$

2.2.3 Image Compression

Finally, for the Image Compression application, we determined that the workload varies a lot with the output format and, since there are only three possibilities (*png*, *jpeg* and *bmp)*, we divided the analysis for each case. Then, we conclude that, the variation pattern is similar to the previous types of requests, since the workload increases linearly with the increase of the number of pixels. Furthermore, we observed some variation spikes depending on the compression factor. Both results can be observed in Figures 5 and 6.

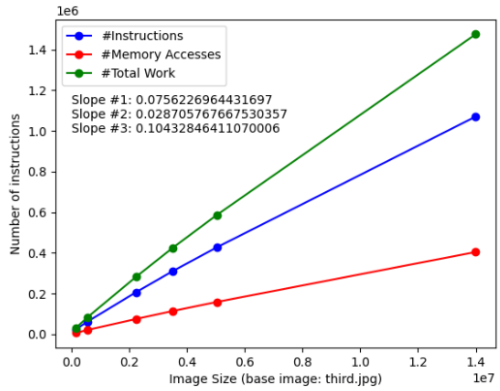

Figure 5: Workload variation by different image size

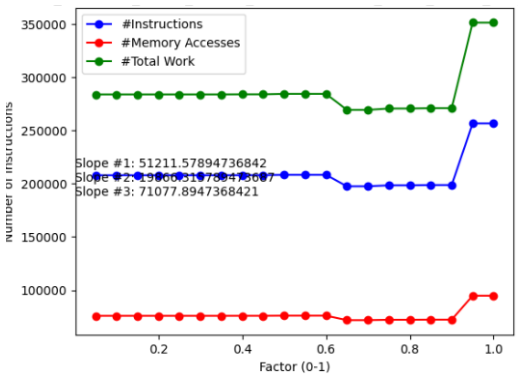

Figure 6: Workload variation by different compression factor values.

For Image Compression we also used the same technique as before. We started by separating each combination of output format and intervals of compression factor, from 0.0 to 1.0 in a tenth scale. Then, for each combination, we obtain the resulting slope that depends on the image size, i.e., number of pixels:

$$slope = (max_{work} - min_{work})/(max_{pixels} - min_{pixels})$$

When new values are obtained from the instrumentation metrics, we can then verify if we are able to obtain a new slope based on more distant datapoints. If this is the case, the workers update the DynamoDB database accordingly. With this, we can obtain the slope (`Metric` value) from DynamoDB and estimate the workload for new requests for images with different size using the following formula:

$$estimated\_work = slope \cdot pixels$$

2.3 Metric Storage System

Since our system is composed of elastic properties that rely on dynamic and constant worker deployment and execution in order to meet the client demands, it is necessary to store the collected metrics information in a persistent manner. To achieve this, we rely on DynamoDB to store metrics regarding the three types of requests in three distinct tables. All tables use a Hash Key that is defined according to the conclusions taken previously in the analysis of the metrics, in section 2.2.

Foxes and Rabbits metrics are stored by their `World:Scenario`, Insect War by `<={ratio}`, where `ratio` is the quotient of the two armies rounded to the upper even whole number. If `ratio > 8`, then the entry is stored as `≤ infinity`. Image Compression is stored by `{format}:{quality}`, where the `quality` ranges from 0.0 to 1.0 in a tenth scale. There is one exception when a *BMP* format is received. *BMP* has no quality association, so it is stored as `{format}` only.

2.4 Task Scheduling Algorithm

Task scheduling is a crucial property of the Load Balancer, whose chosen responsibility is to spread the work evenly across all EC2 workers and, if necessary, take advantage of Lambdas.

In order to achieve this, the Load Balancer resorts to a data structure that keeps track of the number of concurrent requests and the current workload of each EC2 worker. Prior to the task scheduling, the Load Balancer starts by estimating the workload of the current request as mentioned in the previous section 2.2. Then, the scheduling algorithm concerns two distinct steps. First, it tries to retrieve an available EC2 worker, as depicted in Listings 1 and 2. Then, if it is not possible to obtain a worker, a Lambda function is invoked if certain conditions are met, as presented in Listing 1. Note that the latter only happens when there is an ongoing scale up, as it will be further explained.

```
String processRequest() {
    long workload = estimateWorkload(requestType,
        params);
    while (true) {
        InstanceProperties instance =
            getEC2Instance(workload);
        if (instance != null) {
            String response =
                forwardRequestEC2(instance, uri,
                requestBodyBytes, workload);
            finishRequest(instanceId, workload);
            if (response != null) return response;
        }
        if (workload <= MAX_WORKLOAD_LAMBDA) {
            String response =
                LambdaUtils.invokeFunction(requestType,
                params);
            if (response != null) return response;
        }
    }
}
```

Listing 1: Scheduler algorithm - process request

By default, as depicted in Listing 2, the Load Balancer always selects the EC2 worker with the least current workload, if instances are available (lines 12-15). This happens since the work is expected to be balanced between instances and the Auto Scaler can adjust the number of workers if needed.

On the other hand, if the Auto Scaler is doing a scale up, the Load Balancer's scheduling algorithm iterates over the sorted list of instances by ascending value of workload. Then, it only selects an EC2 worker whose expected instance workload and number of concurrent requests do not surpass the maximum workload threshold and maximum concurrent requests (lines 4-11). These values take into account the expected outcome of forwarding the request to that worker. Additionally, it is important to note that these thresholds were determined offline by sending multiple requests to an instance and observing its behavior. Therefore, if sending the request does not imply exceeding these thresholds, the request is forwarded to the EC2 worker. Otherwise, the Load Balancer verifies if the request is a good candidate for a Lambda worker, meaning that the workload is below 20% of the maximum workload an instance can withstand. If this is verified, a Lambda worker is invoked, otherwise, the Load Balancer keeps iterating until an available worker is found.

```
InstanceProperties getEC2Instance(long workload) {
  while (true) {
      if (!availableInstances.isEmpty()) {
          if (AutoScaler.isScalingUp()) {
              for (InstanceProperties instance :
                  availableInstances) {
                  if (instance.getRequests() <=
                      MAX_CONCURRENT_REQUESTS &&
                      workload +
                      instance.getWorkload() <=
                      MAX_WORKLOAD_EC2 ) {
                      registerRequest(instance.getId(),
                          workload);
                      return instance;
                  }
              }
              return null;
          } else {
              InstanceProperties instance =
                  availableInstances.get(0);
              registerRequest(instance.getId(),
                  workload);
              return instance;
          }
      } else {
          return null;
      }
  }
}
```

Listing 2: Scheduler algorithm - obtain available EC2 worker

Although it would be simpler if Lambda functions were used to execute every small request, it is important to take into account that these serverless functions have limited execution time and memory, with very fine-grained control over the execution cost. Hence, executing requests in EC2 instances is often cheaper than in Lambdas. As a result, we only resort to them if the workload is minimal and if there is an ongoing scaling up. This is because, when the Auto-Scaler is increasing the cluster size, some or all workers might be overloaded. If large requests need to be processed and forwarding them to an instance would mean exceeding the workload threshold, then it is best to wait until a new instance is finally created than overwhelming even more the instance, which would risk a crash and slow its execution on current requests. On the other hand, if small requests do not fit in any instance, they can simply be forwarded to Lambdas.

2.5 Auto-Scaling Algorithm

The Auto Scaler adjusts the scale of running machines according to their overall average CPU utilization every 1 minute. This metric is acquired through the AWS CloudWatch service. Instances are created or terminated, on a 10% factor, when the average CPU falls below 15% or exceeds 85%, respectively, as represented in lines 11-14 in Listing 3. For instance, with a cluster of 20 instances with an average CPU utilization of 10%, the downscale operation would involve terminating 2 instances simultaneously, calculated as 20 * 0.1 = 2. This approach proves to be more efficient when dealing with larger clusters, as terminating a set of instances collectively is more efficient than requesting the termination of each individual instance separately due to reducing overhead.

```
void autoscale() {
```

```
    Set<String> instanceIds = getRunningInstances();
    double cpuUtilization = 0.0;
    for (String instanceId : instanceIds) {
        double instanceCpuUtilization =
            getCpuUtilization(cloudWatch,
            instanceId);
        cpuUtilization += instanceCpuUtilization;
    }
    int size = instances.size();
    double averageCpuUtilization =
        cpuUtilization/size;
    int numOfInstances =
        (int)(Math.ceil(size*SCALE_FACTOR));
    if (averageCpuUtilization > MAX_CPU_UTILIZATION
        && size < MAX_INSTANCES) {
        createInstances(numOfInstances);
    } else if (averageCpuUtilization <
        MIN_CPU_UTILIZATION && size >
        MIN_INSTANCES) {
        terminateInstances(numOfInstances);
    }
}
```

Listing 3: Auto-Scaling algorithm - autoscale

When the `terminateInstances` method is invoked, the corresponding number of instances are terminated after completing all pending requests assigned to them. This method, presented in Listing 4, gets the instances to terminate based on their individual CPU utilization, i.e., it gets the *numberOfInstances* of the less utilized running instances. At line 5, the AS modifies the status of an instance to `onHoldToTerminate`, ensuring that the LB recognizes that it should no longer route requests to that particular machine. In lines 7-14, the AS waits until all the instances scheduled for termination have completed processing their requests, meaning their workload has reached 0.

```
1  void terminateInstances(int numberOfInstances) {
2      List<String> instanceIds =
           getInstancesToTerminate(numberOfInstances);
3      for(String instanceId : instanceIds) {
4          InstanceProperties instance =
               instances.get(instanceId);
5          instance.setOnHoldToTerminate();
6      }
7      while(true) {
8          int count = 0;
9          for (String instanceId : instances.keySet())
10             if (instance.isHoldingToTerminate() &&
                   instanceId.getValue().getWorkload()
                   == 0)
11                 count++;
12         if (count == numberOfInstances) break;
13               Thread.sleep(SECOND_SLEEP_INTERVAL);
14     }
15     terminateInstancesById(instanceIds);
16 }
```

Listing 4: Auto-Scaling algorithm - terminate instances

Additionally, the AS also terminates instances whose status was set do `dead` by the LB.

2.6 Fault-Tolerance

Fault-Tolerance is a property that ensures the system remains operational upon failure, which allows for complete transparency to clients while recovering from failures. Therefore, requests have two states: (i) being processed by a worker and (ii) on hold until a worker is available.

In order to acknowledge EC2 instance failures, we employed timeout mechanisms in conjunction with heartbeats. When sending a request to a worker, a dynamic connection and response timeout is set according to the estimated workload. As a result, if the worker does not respond within a time period, a heartbeat is sent to the worker. If the worker responds, then the Load Balancer retries sending the request. Otherwise, the instance is marked as `dead` and, later on, the Auto Scaler will instantly terminate it. On the other hand, it is also necessary to ensure the proper execution of Lambda functions. Hence, when a response is received from a Lambda, the Load Balancer simply verifies if the response contains an error. In both EC2 instance failure and Lambda function execution error, the Load Balancer ensures the request is subsequently processed by another available worker.

3 Conclusions

In conclusion, the EcoWork@Cloud system, designed and implemented within the Amazon Web Services (AWS) ecosystem, incorporates various features to optimize resource utilization and enhance performance. The system comprises workers such as EC2 instances and Lambdas, a Load Balancer (LB), an Auto-Scaler (AS), and a metrics storage system using DynamoDB.

Metrics obtained from Foxes and Rabbits, Insect War and Image Compression operations are collected and stored in DynamoDB by each worker. These metrics are then utilized by the Load Balancer to estimate the cost of new requests and determine the most suitable worker for processing. In cases where a request cannot be handled by any worker but requires minimum work, it is redirected to a Lambda function.

The Auto-Scaler plays a crucial role in maintaining the optimal number of running machines. It dynamically adjusts the scale of instances based on their average CPU utilization, as measured by the AWS CloudWatch service. Instances are created or terminated when the cluster's average CPU exceeds 85% or falls below 15%, respectively.

The system's fault-tolerance mechanisms ensure operational continuity in the event of failures. Timeout mechanisms, combined with heartbeats, are employed to detect and handle worker failures. Dead instances are promptly terminated by the Auto-Scaler, while Lambda function execution errors are mitigated by assigning the request to an available worker.