EcoWork@Cloud

Cloud Computing and Virtualization
Project - 2022-23
MEIC / METI / MECD - IST - ULisboa

1 Introduction

The goal of this project is to design and implement a service hosted in an elastic public cloud to give support to an hypothetical study of ecosystems. The service is meant to execute a number of computationally-intensive tasks, namely simulation of ecosystems and compression of nature-related imagery. You are given an already built simplified Java application (solely aimed at generating a realistic CPU load) that handles the following types of requests:

- Foxes and Rabbits: carries out a simulation of evolution of an ecosystem of foxes and rabbits in a given terrain, over a number of rounds;
- Insect War Simulation: carries out a simulation of a war between several insect species in a terrain, where each species has several insect roles (soldiers, workers, drones and queens);
- Image Compression: returns a compressed version of a nature-related image, using a given algorithm and compression factor;

With this application, your task is to deploy it in the cloud and scale it by increasing or decreasing the number of workers according to the number and complexity of user requests. You will be using two types of cloud deployments: EC2 instances and Lambda functions. Note that while EC2 instances are cheaper per processed request, they suffer from a long startup time. On the other hand, Lambdas are more expensive per processed request but fast to start. Therefore, you need to balance invocations between the two types of deployments to minimize request latency and cost. Measuring the complexity of requests will be done by instrumenting the application code with Javassist. Complexity is an estimate of the amount of work involved in processing the request. As you will notice, simply using processing time instead of estimating the complexity won't help since processing time quickly becomes unpredictable when multiple tasks execute at the same time and compete for possibly overcommitted resources. The project specification is accompanied by a Frequently Asked Questions document available at: `https://tinyurl.com/CNV-22-23-FAQ`.

2 Architecture

The EcoWork@Cloud system should run within the Amazon Web Services ecosystem. The system (see Figure 1) will be organized in four main components:

- **Workers** receive web requests to perform an EcoWork@Cloud operation. There will be a varying number of identical VMs and Lambdas running simualtion and image compression operations;
- **Load Balancer (LB)** is the entry point of the system. It receives all web requests, and for each one, it selects an active VM to serve the request and forwards it to that server. In alternative, it can also trigger a Lambda function invocation;
- **Auto-Scaler (AS)** is in charge of collecting system performance metrics and, based on them, adjusting the number of active VM instances;
- **Metrics Storage System** uses Amazon DynamoDB to store request performance metrics. These will help the LB choose the most appropriate worker.

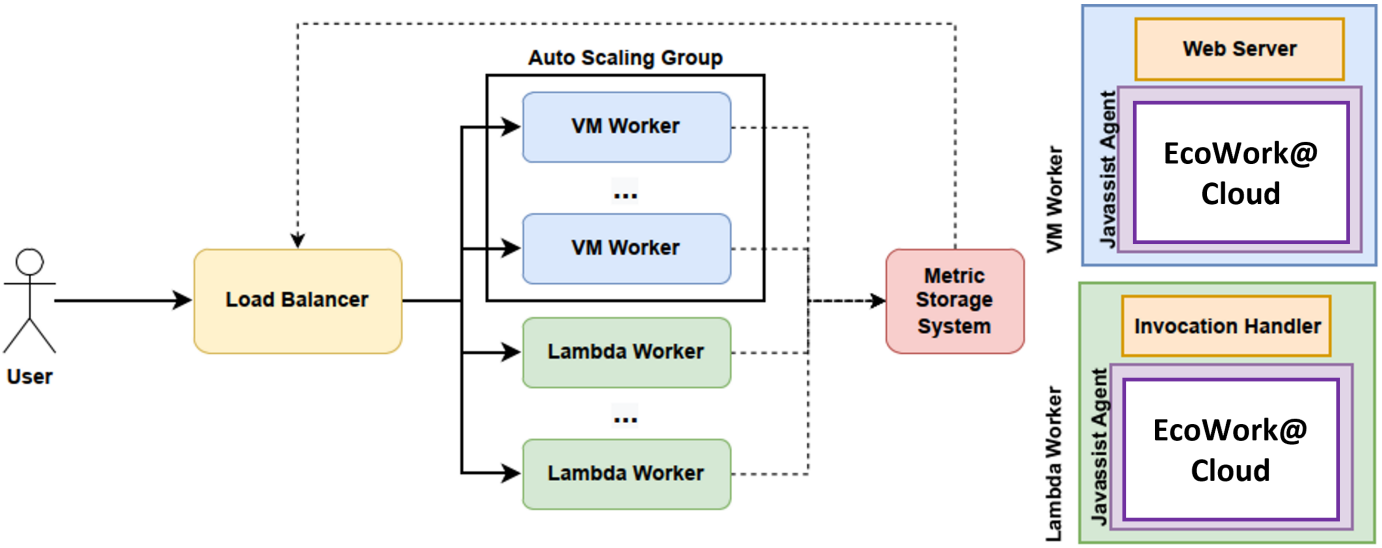

Figure 1: Architecture of EcoWork@Cloud

2.1 Workers

Workers are responsible for executing operations. In EcoWork@Cloud , there are two types of workers: VM workers (EC2 instances), and FaaS workers (Lambda invocations). Both types of workers execute the same code (see Figure 1): application executing EcoWork@Cloud code instrumented with Javassist. It is your task to use the JavassistAgent tool to instrument request handling. The main difference between both types of workers is how the invocation is performed.

VM workers implement a web server that receives requests directly from the LB. These requests include the input and parameters which are then passed to the application code. The result is returned to the LB and then returned to the user.

FaaS workers implement an invocation handler that receives Lambda invocations triggered by the LB. Similarly to VM worker requests, Lambda invocations include an input and parameters and return an output after applying the requested operation.

2.2 Load Balancer (LB)

The LB is the only entry point into the system. It receives requests and either selects one of the active VM workers to handle each of the requests or triggers a Lambda invocation. Note that the LB should itself be a VM instance running a web server that uses metrics obtained in earlier requests (stored in the Metrics Storage System, MSS) to decide how and where to execute (i.e. schedule) requests.

The LB should estimate the approximate complexity of requests based on the requests' parameters combined with data previously stored in the MSS. The LB may know which VM workers are busy, how many requests there are currently executing in each VM, and how much work is left in each VM taking into account a request complexity estimate.

2.3 Auto Scaler (AS)

For this project, you will design an auto-scaling component that decides how many web server nodes should be active at any given moment. You should design and implement an auto-scaling strategy that provides the best balance between performance and cost. The AS should detect that the VM workers are overloaded and start new instances and, on the other hand, reduce the number of nodes when the load decreases. For simplicity, you can deploy both the AS and the LB code in the same VM (and in the same web server application).

2.4 Metrics Storage System

EcoWork@Cloud should also include a Metrics Storage System (MSS) that stores load and performance metrics collected from the worker nodes. These metrics result from running instrumented code which collect relevant dynamic performance metrics regarding the application code executed (e.g. number of bytecode instructions or basic blocks executed, data accesses, number of function calls executed, invocation stack depth, and/or others deemed relevant). These metrics allow estimating the task complexity realistically, irrespective of variable wall-clock time delays that can be caused by frequent resource overcommit.

The final choice of the metrics extracted, instrumentation code, and structure used to store the metrics data is thus subject to analysis and decision by the students. Students should consider the usefulness/overhead trade-offs of all utilized metrics. The selected storage system can be updated directly or you may resort to some intermediate transfer mechanism. For realism, you must take into account that continuously querying this storage system is expensive and may also become a performance bottleneck for the LB.

3 Design and Implementation Guidelines

- Automate all cloud deployments and delete all cloud resources after each work session;
- Design and experiment with your instrumentation code locally on your or lab PCs to save resources;
- For simplicity, use `t2.micro` VM instance types and Lambdas with 512 MB of RAM.
- There are multiple possible design options for each of the components required for the project.
  - More important than the picked design is the reason for picking it, understand and explain it!

4 Checkpoint

Students may submit an initial version of their system by **May 26th**, 23h59. This version should support running EcoWork@Cloud on multi-threaded VM workers, with at least an AWS-configured LB and AS operating (no need for lambdas at this stage). Instrumentation gathering metrics should be working. Ideally, it should also include an initial version of your LB and AS code. Note that the algorithms for load balancing, auto scaling may not be fully implemented at this stage. However, it is expected that some logic is already thought out even if simplified. The code submitted for the checkpoint will be evaluated on the following labs.

The checkpoint submission bundle must include an intermediate report (1-page, double column) clearly describing: a) what is already developed and running in the current implementation (architecture, data structures and algorithms); b) the specification of what remains to be implemented or completed (namely pseudo-code for your final LB and AS algorithms). The report should be submitted in Fénix until 23:59 on **May 27th**.

5 Final Submission

The final submission is to be delivered by **June 16th**, 23h59. It should include a complete implementation of the system. In addition to the checkpoint, the project should include: i) an instrumentation tool that balances the instrumentation overhead with the precision of the extracted information; ii) an auto scaling algorithm that balances cost and performance efficiently; iii) a load balancing algorithm that minimizes cost and request latency using request complexity estimates based on previous requests.

Student groups should write a report (up to 5 double column pages after the cover) describing the implemented solution, clearly explaining and justifying the algorithms, as well as any results, measurements, charts and analysis that support the design decisions and configurations. Groups are encouraged to provide information about the experiments conducted while experimenting with different design tradeoffs (e.g., charts, datasets). The report should submitted in Fénix until 23:59 on **June 17th**.