# Instituto Superior Técnico

## Cloud Computing and Virtualization

---

# RadarScanner@Cloud

(Checkpoint)

Group 17

---


**Sara Machado**

**86923**

**Rafael Figueiredo**

**90770**

**Ricardo Grade**

**90774**

# 1. Implementation

## 1.1. WebServer

The WebServer was already parallelized, therefore it could compute multiple requests for the same image at the same time, the only thing that we noticed was that the created images were overriding each other, since their filename would be the same, to solve this issue we added a nonce to the filename of each of them.

The WebServer can collect the metrics of a given request at the end of its processing because they are kept in an in-memory static map in the instrumentation class.

## 1.2. Instrumentation

The goal of the instrumentation is to collect metrics about a specific request, therefore as multiple requests are being processed at the same time, it was necessary to come up with a way of avoiding conflicts between them.

Our implementation associates each set of metrics to the id of the current thread that is computing it, in a static map, after the request is completed, the metrics collected are written to a file and removed from the map.

Our implementation collects the following metrics dynamically:

- Number of bytecode instructions;
- Number of basic blocks;
- Number of data accesses (loads and stores);
- Number of allocations;
- Number of method calls.

The invocation stack depth is not being collected by our instrumentation, since after testing this property it was made clear that its value wouldn't vary in any requests, therefore becoming irrelevant for our decision making.

Our implementation has the following structure:

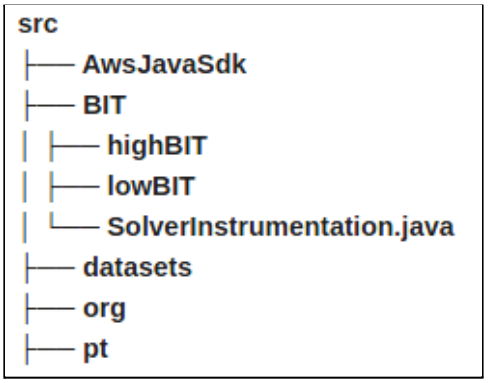


**Figure 1**: Packages Structure

This structure allows for our implementation to be maintainable because if in the future there was a need to add more instrumentation classes, they could be easily placed in the BIT package which can be accessed by the WebServer. Furthermore, it becomes more organized, allowing for the instrumentation and programming logic to be well divided.

## 1.3. Load Balancer and Auto-Scaler

To implement the load balancer and the auto-scaler, we first added our web server code to an AWS instance listening on port 8000 and created an AMI image of that instance.

Next, we created a load balancer that listens on port 80, and performs health checks on *:8000/scan* on each instance to detect if an instance is working correctly.

For this to work, the WebServer was modified to return a *200 OK* on that request without any arguments.
To create the auto-scaler we added a launch configuration with information about the load balancer and AMI, and a group size with minimum capacity of 1, desired capacity of 2 and maximum capacity of 5, this capacity properties should be enough to support multiple requests of multiple clients.

In order to give the auto-scaler scaling policies to increase or decrease the number of instances, some metrics, available in the AWS, were taken into consideration. Those metrics are the following:

- *CPUUtilization*: Which gathers the percentage of utilization that the CPU of each instance is having and it allows to track when CPU intensive requests are taking a lot of computational space from the instances. Its threshold is set between 30% and 80%;
- *NetworkOut*: Which counts the number of bytes that each instance is sending in order to reply to a request, allows to track the size of the image that is sent on the reply to each request. Its threshold is set between 10K and 1M bytes.

When the auto-scaler performs a scaling action, it is based on the alarms that were created in order to track the mentioned metrics. When one of those metrics reaches a threshold, minimum or maximum, a scaling action is triggered, to remove or add instances, respectively.

# 2. Next Phase

Regarding the improvement of the load balancer and auto-scaler which remains to be implemented, we foresee the following structure:

- MSS: The repository to where each instance is going to send its newly collected metrics after a certain amount of time;
- Manager: The instance that is going to play the role of the load balancer and the auto-scaler, based on the metrics collected from time to time from the MSS;
- Set of worker instances: The instances to which the manager is going to forward the incoming requests to be computed.

## 2.1. Algorithm

The manager is going to keep the cost associated with each worker instance.
This cost is estimated based on the complexity of the requests forwarded to that instance.
The cost that each request adds to the worker instance to which the request is going to be forwarded is calculated based on a weighing of the previously collected metrics.

The load balancer's manager is going to forward an incoming request to the worker instance who has the lowest associated cost.

Regarding the auto-scaler's manager:

$$S = \sum_{i=0}^{N} (WorkerInstance[i].Cost)$$

- If $S > ThresholdMax \times N$:
  - Adds an instance
- If $S < ThresholdMin \times N$:
  - Removes an instance