Cloud Computing and Virtualization - AWS

Renato Vivar - Joao Bettencourt
Instituto Superior Tecnico

August 25, 2025

1. Introduction

The system is designed to handle a variety of workloads, including Capture-the-Flag, the 15-Puzzle, and the Game of Life, all of which are executed on an instrumented web-server hosted on an EC2 virtual machine. To monitor performance, the system employs Javassist's "ICount" instrumentation, which records per-thread bytecode counts for each handler. For scalability and efficient traffic distribution, the system utilizes a prototype setup of AWS services, combining an Application Load Balancer (ALB) with an Auto Scaling Group (ASG).

2. Implemented Components

We have successfully built and tested the provided Games@Cloud application, verifying that the WebServer handles multiple simultaneous requests across the supported workloads: capturetheflag, fifteenpuzzle, and gameoflife.

We implemented dynamic workload instrumentation using Javassist. We extended the ICount agent to:

- Collect metrics per thread using ThreadLocal to separate request statistics and track executed methods, basic blocks, and bytecode instructions.
- Print and log per-thread and global statistics to files and include reset methods to clear counters between requests, avoiding accumulation.

Instrumentation was injected using -javaagent, and the pom.xml was updated to support class transformation. We built a custom Amazon Machine Image (AMI) with the auto-starting web server on instance boot using a systemd-compatible /etc/rc.local setup. Maven and dependencies are pre-installed during image creation through a scripted pipeline (create-image.sh, install-vm.sh, test-vm.sh), ensuring reproducibility.

For Elasticity, we configured:

- AWS Elastic Load Balancer (ELB) to route requests to workers.
- Auto Scaling Group (ASG) with defined policies and alarms based on CPU utilization.
- Scale-out: CPU $> 50\%$ and Scale-in: CPU $< 25\%$.

3. Current Architecture

The current architecture consists of EC2-based VM workers running the Games@Cloud application with Javassist instrumentation, allowing for runtime collection of execution metrics such as method calls, basic blocks, and instructions. These VM workers are fronted by an AWS Elastic Load Balancer (ELB), which routes incoming HTTP requests to available instances. The system is designed to scale dynamically using an AWS Auto Scaling Group (ASG) configured with scaling policies and CloudWatch alarms based on CPU utilization—triggering scale-out when usage exceeds 50% and scale-in when it drops below 25%. A custom Amazon Machine Image (AMI) was created to automatically start the instrumented web server on boot, ensuring new instances are deployment-ready. Instrumentation metrics are logged per request, distinguishing concurrent threads, and will be integrated with a Metrics Storage System (MSS) to support future load balancing and scaling decisions based on request complexity.

4. Next Steps and Design Plans

The next step involves integrating the collected statistics—currently stored in .txt files per request—into a centralized Metrics Storage System (MSS) using Amazon DynamoDB. These statistics will be indexed by the input parameters of each request to allow the Load Balancer (LB) to estimate request complexity in real time. Based on this complexity estimation, the LB will decide whether to forward the request to an EC2 VM or invoke a Lambda function—favoring Lambdas for heavier, more complex tasks to reduce latency.

The Auto-Scaler (AS) logic will also be enhanced to make decisions not only based on CPU usage but also on aggregated complexity metrics and the number of pending requests in the system. The goal is to dynamically adjust the number of active VM instances to optimize cost and performance, scaling out when workload complexity exceeds the current capacity and scaling in when utilization drops for an extended period. So far, we have implemented the ICount instrumentation tool, which tracks methods, basic blocks, and instructions per request.

In the next phase, we plan to explore and possibly integrate additional instrumentation tools that offer complementary or more efficient metrics to improve request complexity estimation with minimal overhead.

5. Conclusions

All base components (instrumented workloads, AMI, LB/AS setup) are operational and validated. Metrics are being collected in a fine-grained and thread-aware manner, ready for MSS integration. The current AWS-provided ELB and ASG offer a functional starting point, with plans to evolve toward custom LB/AS logic based on dynamic request complexity estimation.

6. Initial Code for Scheduling strategy for LB and scaling policy for the AS

```
#!/bin/bash

source config.sh

# Create Load Balancer and configure health
    check
aws elb create-load-balancer \
  --load-balancer-name $LB_NAME \
  --listeners "Protocol=HTTP,
      LoadBalancerPort=80,InstanceProtocol=
      HTTP,InstancePort=8000" \
  --availability-zones
      $AWS_AVAILABILITY_ZONE

aws elb configure-health-check \
  --load-balancer-name $LB_NAME \
  --health-check "Target=HTTP:8000/,
      Interval=30,UnhealthyThreshold=2,
      HealthyThreshold=2,Timeout=5"

# Create Launch Configuration
aws autoscaling create-launch-configuration
    \
  --launch-configuration-name
      $LAUNCH_CONFIG_NAME \
  --image-id $(cat image.id) \
  --instance-type t2.micro \
  --security-groups $AWS_SECURITY_GROUP \
  --key-name $AWS_KEYPAIR_NAME \
  --instance-monitoring Enabled=true

# Create Auto Scaling Group
aws autoscaling create-auto-scaling-group \
  --auto-scaling-group-name $ASG_NAME \
  --launch-configuration-name
      $LAUNCH_CONFIG_NAME \
  --load-balancer-names $LB_NAME \
  --availability-zones
      $AWS_AVAILABILITY_ZONE \
  --health-check-type ELB \
  --health-check-grace-period 60 \
  --min-size 1 \
  --max-size 3 \
  --desired-capacity 1

# Create Scaling Policies and store their
    ARNs
increase_policy_arn=$(aws autoscaling put-
    scaling-policy \
  --auto-scaling-group-name $ASG_NAME \
  --policy-name $POLICY_INCREASE_NAME \
  --scaling-adjustment 1 \
  --adjustment-type ChangeInCapacity \
  --cooldown 300 \
  --query PolicyARN --output text)

decrease_policy_arn=$(aws autoscaling put-
    scaling-policy \
  --auto-scaling-group-name $ASG_NAME \
  --policy-name $POLICY_DECREASE_NAME \
  --scaling-adjustment -1 \
  --adjustment-type ChangeInCapacity \
  --cooldown 300 \
  --query PolicyARN --output text)

# CloudWatch Alarms
aws cloudwatch put-metric-alarm \
  --alarm-name $ALARM_HIGH_NAME \
  --alarm-description "Scale out when CPU >
      50%" \
  --metric-name CPUUtilization \
  --namespace AWS/EC2 \
  --statistic Average \
  --period 60 \
  --threshold 50 \
  --comparison-operator
      GreaterThanThreshold \
  --evaluation-periods 1 \
  --alarm-actions $increase_policy_arn \
  --dimensions "Name=AutoScalingGroupName,
      Value=$ASG_NAME"

aws cloudwatch put-metric-alarm \
  --alarm-name $ALARM_LOW_NAME \
  --alarm-description "Scale in when CPU <
      25%" \
  --metric-name CPUUtilization \
  --namespace AWS/EC2 \
  --statistic Average \
  --period 60 \
  --threshold 25 \
  --comparison-operator LessThanThreshold \
  --evaluation-periods 1 \
  --alarm-actions $decrease_policy_arn \
  --dimensions "Name=AutoScalingGroupName,
      Value=$ASG_NAME"
```

Listing 1: Inital-code