Games@Cloud Report

Miguel Capitão (Nr. 98957) Vasco Silva (Nr. 99132) Tiago Teixeira (Nr. 102638)

1 Architecture

Our solution implements a distributed architecture consisting of three main components: a Load Balancer (LB) and an Auto-Scaler (AS) working in the same host, and several Workers. The system follows a master-worker pattern, where the LB acts as the entry point for all client requests, distributing them across available workers based on current load and request characteristics. The control flow begins when a client sends an HTTP request to the LB. The `RequestAssigner` component processes the request and decides whether to:

- Forward the request to an existing worker;
- Send it to AWS Lambda for low-complexity tasks under heavy load;
- Queue the request when no workers are available;

Data flows through the system as follows:

- Client → LB (request)
- LB → Worker/Lambda (forwarded request)
- Worker/Lambda → LB (response)
- LB → Client (response)

The AS continuously monitors system metrics (CPU usage, queue complexity) and adjusts the worker pool size accordingly. Communication between components uses asynchronous patterns with `CompletableFuture` for non-blocking operations.

The request parameters and their corresponding collected metrics are stored in an Amazon DynamoDB table, which is created by the Load Balancer at the beginning of the deployment. The responsibility for deciding which requests need to have their details and metrics stored in persistent storage lies with the Load Balancer.

In order to instrument the workloads, the Javassist module provided by the teachers was used as a starting point. Specifically, the ICount tool is used to determine a complexity score based on the number of instructions (`ninsts`) and methods called (`nmethods`), with the approach explained in detail later.

Each request received by the workers includes an additional parameter, `storeMetrics`, which is added by the Load Balancer with the purpose of signaling the worker to add a new entry to the Amazon DynamoDB table with the request parameters and the associated metrics. The Load Balancer will only signal the worker to store the metrics if it fails to find an exact match of the request in its local cache or the Amazon DynamoDB table.

2 Instrumentation metrics

Initially, a choice was made to extract as many different metrics via instrumentation as possible (each one at a time) to make a study of the usefulness of each one. For this study, we developed a Python script that sends multiple requests to the Web server with a wide range of different parameters for each game. In doing this, we were able to see how different values of each metric influenced the response time to the request. The result of this study can be seen in Table 2 where we present the R^2 value of the linear relationship between each one of the metrics and the request response time for each one of the three workloads.

The results of this study showed that there is no apparent benefit in extracting multiple metrics, as virtually any of them besides `ndataWrites` and `ndataReads` explain the evolution of the response time very accurately.

With this in mind, we performed another study with the goal of understanding which of the metrics introduced the least overhead. To do this, we sent multiple requests with different parameters to each one of the three workloads and compared the average elapsed time when using different bytecodes that extracted different metrics. The results of this study are shown in Table 3 and clearly show that the extraction of method calls is the one that introduces the least overhead.

Based on the results of these two studies, an initial decision was made to extract only the number of methods calls through instrumentation, as it maximizes the quality of the information that can be used to estimate the complexity of a request at the expense of as little as possible execution overhead. After this study, we were advised to use at least one more metric, so that our approach could generalize better to any workload and not only the provided ones. This advice led us to choose the `ninsts` and `nmethod` metrics to be combined in a complexity score. To have an accurate measure of complexity, we used the following formula for each game:

$$Complexity = \frac{ninsts}{a} \times b + nmethods$$

where a corresponds to how many more instructions there are on average than methods called and b corresponds to how many times on average it is more expensive to count instructions than method calls.

3 Data Structures

As previously mentioned, we utilize the `storeMetrics` parameter to order the worker to send the request data to the Amazon DynamoDB that the load balancer creates, so that the values obtained from new, unknown requests are stored, and used in the future for better estimation of complexity and assignment from the Load Balancer. This parameter is only placed by the Load Balancer if no other stored request, both in the DynamoDB and in the local cache, matches the values of the new request, preventing repetition of requests and unnecessary usage of these systems and resources.

4 Request Cost Estimation

The request cost estimation component plays a pivotal role in enabling intelligent resource management and dynamic task scheduling in our distributed system. Its primary goal is to provide a quantifiable measure of how computationally expensive a client request is likely to be, thereby allowing the system to make informed decisions regarding scheduling, load balancing, and auto-scaling.

To learn how to transform the parameters of a request in an estimation of its complexity, we first had to build a large dataset with pairs of parameters and the corresponding real complexity score. Then, we used this dataset to learn three models, one for each game, that could make complexity estimations with acceptable performance. Ideally, in addition to this offline-learned heuristics, we'd also like to factor in for the complexity estimation the data already available from previous served requests in memory and in storage, but our current implementation does not support this and uses only the heuristic for the estimation.

4.1 Game of Life Estimator

The Complexity Estimator for the Game of Life workload uses only the parameter `iterations` in a simple linear regression model. This is possible for this game since the computational cost scales linearly with the number of simulation steps.

$$\hat{Complexity} = 67.178 + 893.856 * iterations$$

4.2 Fifteen Puzzle Estimator

For the Fifteen Puzzle workload, the empirical data showed that the difficulty grows non-linearly with the board size and the number of shuffles with cubic interaction terms significantly improving the estimation accuracy. Because of this, the complexity estimation uses as features the `shuffles` and `size` parameters and their polynomial combinations up to degree 3. The complete list of features is available in the `FifteenPuzzleEstimator` Class.

$$\log(\hat{Complexity}) = 13.911 + Coefficients * Features$$

4.3 Capture The Flag Estimator

The Capture The Flag workload similarly to the Fifteen Puzzle also showed that the complexity grows non-linearly with the parameters with quadratic interaction terms significantly improving the estimation accuracy. Since this workload has the non-numerical parameter `flagPlacementType`, we had to use one-hot encoding for the flag placement strategies. Complexity estimation once again uses as features the game parameters and their polynomial combinations up to degree 2. The complete list of features is available in the `CaptureTheFlagEstimator` Class.

$$\log(\hat{Complexity}) = 16.445 + Coefficients * Features$$

4.4 Complexity Estimators Performances

	Train R^2	Test R^2	Test MSE	Test RMSE
CTF	0.99	0.99	0.04	0.20
FP	0.69	0.77	2.02	1.42
GOL	0.99	0.99	0.18	0.34

Table 1: Complexity Estimators Performances

5 Task Scheduling Algorithm (Load Balancer)

The load balancer dynamically distributes incoming requests across virtual machines (VMs) using a hybrid strategy that balances load spreading and packing based on current system utilization. Each request's computational complexity is estimated and used to guide assignment decisions. We defined 20 seconds as the maximum acceptable response time, extracting the associated complexity score to serve as the total capacity for a single worker. The presented thresholds are defined as a fraction of this capacity and only workers who have enough capacity to attend the request are considered. In all strategies, if no worker has enough available capacity to attend to the request, it will be added to a global overflow queue which is attended to and drained after each request is completed or after a new worker is commissioned by the autoscaler.

5.1 Spreading Strategy

Triggered when average worker load is over 70%, the load balancer will assemble a list of candidate workers which are sorted in ascending order by current load, therefore giving priority to less loaded workers. If no worker has enough capacity to handle the request and the request's complexity is below the lambda threshold, it is handled through a lambda function call in order to minimize latency. Three Lambda functions are setup, one for each game.

5.2 Packing Strategy

Triggered when average worker load is under 30%, the load balancer will assemble a list of candidate workers which are sorted in descending order by current load, therefore giving priority to higher load workers.

5.3 Balanced Strategy

Triggered when the average worker load is between 30% and 70%, the load balancer selects candidate workers based on a balance between spreading and packing. Each candidate is scored according to a weighted combination of its current load and its distance from full capacity. The weights dynamically adjust depending on the system's average load, allowing the strategy to favor spreading under higher loads and packing under lower loads. Candidates are then sorted in descending order by score, giving preference to workers that achieve a better balance between spreading and packing.

The score for each worker is computed as follows:

$$\text{packScore} = \frac{\text{currentLoad}}{\text{VM\_CAPACITY}}$$
$$\text{spreadScore} = 1 - \text{packScore}$$
$$\text{spreadWeight} = 0.3 + 0.7 \times \left(\frac{\text{avgLoad}}{\text{VM\_CAPACITY}}\right)$$
$$\text{packWeight} = 1 - \text{spreadWeight}$$
$$\text{score} = \text{spreadScore} \times \text{spreadWeight} + \text{packScore} \times \text{packWeight}$$

Example Consider the following scenario:

- VM capacity: 1000 (100%)
- Request complexity: 200
- Current worker loads:
  - Worker A: 200 (20%)
  - Worker B: 400 (40%)
  - Worker C: 600 (60%)

The average load is:

$$\frac{20\% + 40\% + 60\%}{3} = 40\%$$

Thus,

$$\text{spreadWeight} = 0.3 + 0.7 \times 0.4 = 0.58$$
$$\text{packWeight} = 1 - 0.58 = 0.42$$

Now, compute the score for each worker:

- Worker A (20% load):
$$\text{packScore} = 0.20, \quad \text{spreadScore} = 0.80$$
$$\text{score} = (0.80 \times 0.58) + (0.20 \times 0.42) = 0.464 + 0.084 = 0.548$$
- Worker B (40% load):
$$\text{packScore} = 0.40, \quad \text{spreadScore} = 0.60$$
$$\text{score} = (0.60 \times 0.58) + (0.40 \times 0.42) = 0.348 + 0.168 = 0.516$$
- Worker C (60% load):
$$\text{packScore} = 0.60, \quad \text{spreadScore} = 0.40$$
$$\text{score} = (0.40 \times 0.58) + (0.60 \times 0.42) = 0.232 + 0.252 = 0.484$$

The workers would then be ranked as follows:

Worker	Score
A	0.548
B	0.516
C	0.484

As can be seen, the balanced strategy slightly favors Worker A, but still considers the other workers according to their load.

6 Auto-Scaling Algorithm

The system integrates a custom-built auto scaler to dynamically adapt the number of worker instances based on real-time workload and performance metrics. This approach ensures optimal resource utilization, responsive service, and cost-efficiency by scaling the worker pool between defined bounds.

The AutoScaler continuously monitors the system's global state, including:

- Average CPU Utilization of all active EC2 instances;
- Length of the global overflow queue, which indicates unhandled load;
- Average VM load, relative to the known capacity of each worker;

These metrics are periodically evaluated every two minutes, unless a recent scaling operation has occurred—preventing excessive or rapid fluctuations in worker count. The system is configured to maintain at least 1 and at most 5 workers.

6.1 Scale-Out Conditions

A scale-out is triggered when either of the following conditions is met:

- The average CPU usage across all active workers exceeds 85 percent (with the value computed using an Exponential Moving Average, giving weight to recent usage and smoothing spikes).
- The global queue is non-empty, indicating that the current capacity cannot keep up with incoming requests.

When these conditions are met and the number of workers is below the maximum threshold, a new EC2 instance is launched using a predefined AMI. After creation, the AutoScaler polls AWS until the instance is assigned a public IP address. Once available, the instance is registered with the load balancer as a new worker. It will initially be marked as unhealthy but a fast acting monitoring task will query the /test endpoint in 1 second intervals in order to allow for faster health checking and worker availability registering during scale out.

After being marked as available, the worker immediately begins receiving requests from the overflow queue, helping reduce system latency and balance workload.

6.2 Scale-In Conditions

A scale-in is triggered under the following conditions:

- The average CPU usage is below 25 percent
- The average load is also below 25 percent of the estimated VM capacity

To prevent premature termination during transient dips in demand, the autoscaler also ensures that scale-in operations are separated by at least two minutes and that the system does not drop below the minimum number of workers.

If conditions are met, the least loaded non-draining worker is selected for removal. The load balancer initiates a graceful shutdown, marking the worker as draining to stop it from receiving new requests while allowing it to complete any ongoing ones. Once idle, the instance is terminated via the EC2 API and removed from the system state.

6.3 Implementation Details

The auto scaler uses the AWS SDK to:

- Launch new instances with `RunInstancesRequest`
- Terminate instances with `TerminateInstancesRequest`
- Query CPU utilization via `GetMetricStatisticsRequest`
- Identify instances using `DescribeInstancesRequest`

A dedicated thread `AutoScaler-Thread` continuously runs the evaluation loop. The thread uses a lock and condition variable to sleep between checks or be woken up by events (e.g., changes in workload). New instances are integrated asynchronously using `CompletableFuture`, ensuring the autoscaler thread remains non-blocking while waiting for new instances to acquire a public IP.

By combining proactive monitoring, CPU-based heuristics, and queue-aware logic, this auto scaler ensures that the system remains resilient, responsive, and cost-efficient under varying workloads.

7 Fault-Tolerance

To ensure high availability, resiliency, and robustness in a dynamic and potentially unreliable cloud environment, our system integrates a comprehensive fault tolerance strategy. This strategy includes proactive health monitoring, intelligent retry logic, graceful failure handling, and asynchronous task management, enabling the platform to process requests reliably even under failure conditions.

7.1 Retry System for Failed Requests

The system incorporates an automatic retry mechanism that reattempts failed requests transparently. When a request fails, due to a timeout, internal error, or loss of connectivity, the load balancer will retry the request on a different machine from the pool of healthy virtual machines (VMs):

- Retries are limited by a maximum attempt threshold to avoid infinite loops and mitigate cascading failures.
- Each retry selects the next most suitable VM, balancing load and avoiding machines marked as unhealthy.
- This design shields users from transient backend issues and significantly increases the likelihood of successful request completion.
- Health status is re-evaluated periodically, and VMs that recover can be reintroduced to the active pool.

7.2 Worker Health Monitoring

To avoid continuously sending requests to unresponsive or overloaded machines, the system maintains a health status registry for each VM:

- Each VM is continuously monitored for request success rates and responsiveness.
- When a VM fails to respond correctly to a request (e.g., returns an error or times out), it is flagged as unhealthy.
- Health status is re-evaluated periodically (every 30s), and VMs that recover can be set as available again.

This dynamic health-checking mechanism allows the system to automatically isolate faulty or degraded components and reintroduce them once they stabilize, reducing the risk of cascading failures.

7.3 Graceful Shutdown

When a worker VM is scheduled to be terminated (e.g., for scaling down or maintenance), it enters a graceful shutdown phase we call draining. During this phase:

- The machine stops accepting new requests.
- In-progress tasks are allowed to complete to avoid partial computation loss.
- Once idle, the worker is safely decommissioned.

This prevents request interruptions and prevents dropout of results due to abrupt termination, ensuring the reliability and completeness of requests.

7.4 Fallback Handling

If all retry attempts fail or if no healthy VM is available to handle a request, the system invokes a fallback handler, which can:

- Log the failure with contextual information for offline debugging.
- Return a predefined error response to the client.

This fallback mechanism ensures that even in worst-case scenarios, failure is handled predictably, and diagnostic data is preserved.

7.5 Asynchronous Execution and Resilience

All requests are handled asynchronously from the moment they are received:

- Upon arrival, each request is queued and delegated to a worker thread or VM based on load and health metrics.
- The system decouples request submission from execution, enabling better parallelism and fault isolation.

- If a worker fails during execution, the job can be retried on another VM without blocking the client or affecting other on-going tasks.

Asynchronous execution also enables more efficient resource usage and faster failure recovery, as failed tasks can be detected and migrated quickly without system-wide stalls.

7.6 Load-Aware Retries and VM Selection

When a retry occurs, the load balancer uses the request cost estimation model (via complexity estimators and EMA smoothing) to intelligently select the next VM with:

- The lowest expected load impact,
- Sufficient remaining capacity,
- And a healthy status.

This avoids overloading the same node repeatedly and improves system-wide throughput.

7.7 Logging, Observability, and Recovery Analytics

All critical events related to failures, retries, health status changes, and shutdowns are recorded and time-stamped. These logs are used to debug and understand what the system is doing, where it is failing and how it is failing, allowing it to be corrected and returned to normal functioning.

By combining health monitoring, intelligent retrying, graceful shutdowns, fallback mechanisms, and asynchronous task handling, the system achieves a high level of fault tolerance. These measures work together to maintain availability and correctness in the face of partial failures, transient instability, or dynamic scaling operations.

8 Deployment in the cloud

In order to easily deploy the project in the cloud, a number of shell scripts are used with the intention of automating the steps to create the necessary components of this system.

To create an Amazon Machine Image (AMI) for the VM Workers, the script `create-image.sh` is used to essentially make a clone of a properly set-up VM instance, and store its ID for later usage.

We then use `./create-lambda.sh` to create the three lambda functions that will be utilized by each game to run the low complexity requests. These functions are created using .jars files present in `lambda-jars/`.

Afterwards, we use the script `./create-launch-lbas.sh`, to launch a new EC2 instance with the Load Balancer and Auto Scaler Webserver set up for autostart.

Finally, once the Load Balancer and Autoscaler instance appears running on AWS, requests can be sent to the Load Balancer's IP Address on port 80.

A Results Tables

	Capture The Flag	**Fifteen Puzzle**	**Game of Life**
`nblocks`	0.8741	0.9987	0.9921
`ninsts`	0.8744	0.9987	0.9921
`nmethods`	0.9755	0.9938	0.9301
`ndataWrites`	0.8512	0.9935	0.0000
`ndataReads`	0.5696	0.9996	0.9964

Table 2: R^2 value per metric and game

	Capture The Flag	**Fifteen Puzzle**	**Game of Life**
`nmethods`	0.1190	0.0096	0.0395
`nblocks & ninsts`	0.3932	0.0261	0.3101
`ndataReads`	0.1857	0.0365	0.3489
`ndataWrites`	0.0343	0.0064	0.0278
`all metrics`	0.6234	0.0627	0.6747

Table 3: Average elapsed times per game and set of metrics extracted