HI-ALAN

Load Test Results

(Java versus NodeJS)

# Document Identification

|  |  |
| --- | --- |
| Document Type | Load Test |
| Customer Name | HiAlan Core |
| Document Location |  |

# Document Version Control

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Created By | Change | Approved By |
| 1.0 | 02.01.2020 | Alper Ozdamar | First version of Load Test Document |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Table of Contents

[Document Identification 2](#_Toc39933656)

[Document Version Control 2](#_Toc39933657)

[Table of Contents 3](#_Toc39933658)

[1. Glossary 3](#_Toc39933659)

[1. Abbreviations 4](#_Toc39933660)

[2. Introduction 4](#_Toc39933661)

[3. Test Simulator 4](#_Toc39933662)

[3.1. Apache JMeter 4](#_Toc39933663)

[4. Test Architecture 5](#_Toc39933664)

[5. Systems Under Test 6](#_Toc39933665)

[5.1. Test System #1 (TS1) 6](#_Toc39933666)

[6. HiAlan End-to-End Load Tests 7](#_Toc39933667)

[6.1. Java Lambda Load Tests 7](#_Toc39933668)

[6.1.1. Java LOAD TEST-1 (100 threads-438 TPS) 7](#_Toc39933669)

[6.1.2. Java LOAD TEST-2 (50 threads-381 TPS) 10](#_Toc39933670)

[6.2. Node.js Lambda Load Tests 12](#_Toc39933671)

[6.2.1. Node.js LOAD TEST-1 (100 threads-902 TPS) 12](#_Toc39933672)

[6.2.2. Node.js LOAD TEST-2 (50 threads- 694 TPS) 13](#_Toc39933673)

[7. HiAlan End-to-End Endurance Tests 16](#_Toc39933674)

[8. SUMMARY & COMPARISON 16](#_Toc39933675)

[9. 3rd Party Test Results 17](#_Toc39933676)

[There is an interesing announcement from two weeks ago: 17](#_Toc39933677)

# Glossary

|  |  |
| --- | --- |
| RESTful API | A RESTful API is an application program interface ([API](https://searchmicroservices.techtarget.com/definition/application-program-interface-API)) that uses HTTP requests to GET, PUT, POST and DELETE data. |
| AWS Lambdas |  |
| AWS API Gateway |  |
| MySQL |  |

# Abbreviations

|  |  |
| --- | --- |
| AWS | Amazon Web Services |
| Rps | Request per second |
|  |  |
|  |  |

# Introduction

The purpose of this document is to specify/compare the limits and capabilities of NodeJS and Java Lambda function.

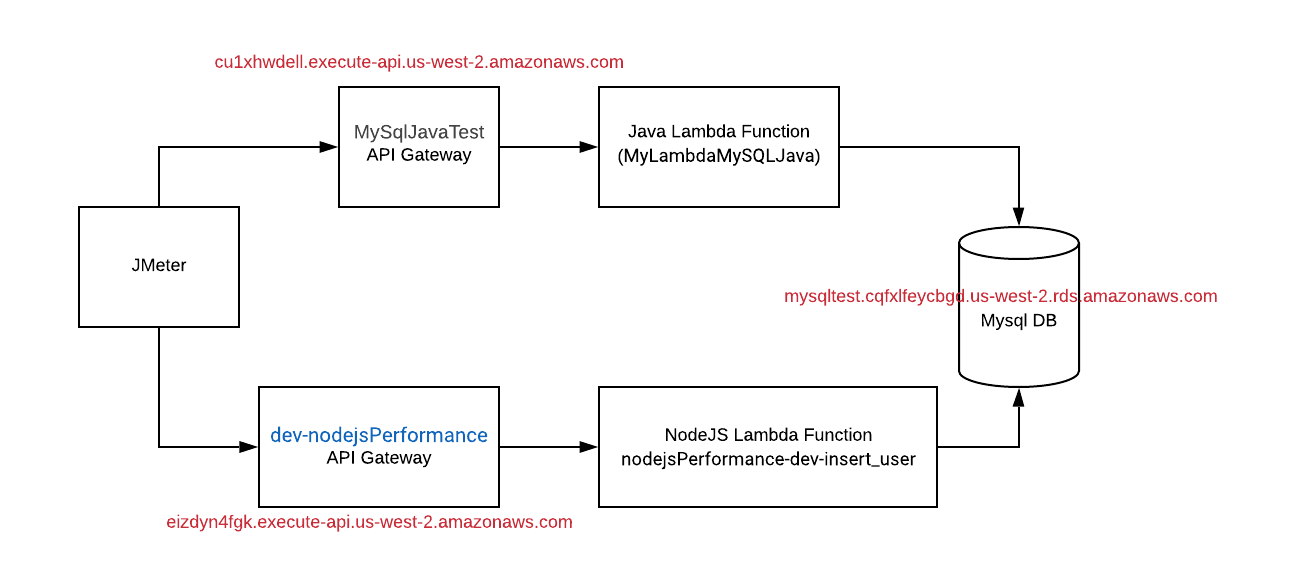
**Important Note:** All test results, graphs, timings can be found in here: <https://drive.google.com/open?id=1wmR3H5LnzurRZFJBVUEkAoCGvKVLbWw_>

# Test Simulator

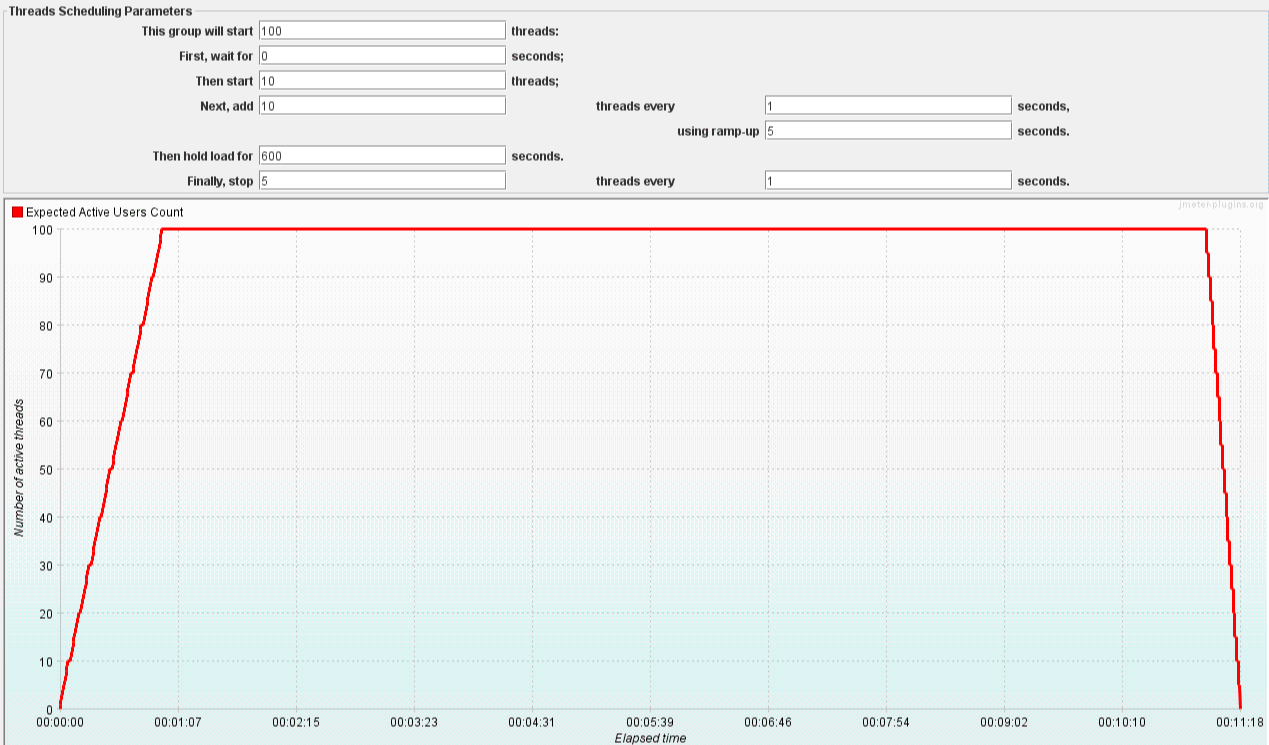
# Apache JMeter

The **Apache JMeter** application is an open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.

# Test Architecture



**Figure 1: Architecture**



**Figure 2: Thread Scheduling for Test Scenarios (100 threads, 600 seconds)**

# Systems Under Test

# Test System #1 (TS1)

 **cu1xhwdell.execute-api.us-west-2.amazonaws.com**

**Description:**

**Software Version:** Experimantal Code (<https://github.com/alperozdamar/aws_lambda_java_mysql>)

**Hostname:** cu1xhwdell.execute-api.us-west-2.amazonaws.com

**Memory:** 512 Mbyte for Lambda functions.

**OS: ??**

**Cpu:** ??

# HiAlan End-to-End Load Tests

# Java Lambda Load Tests

# Java LOAD TEST-1 (100 threads-438 TPS)

**Aim:** To see the stability, response times and memory usage of the system under load with high call rate.

**Test Configuration:**

**System Under Test :**  **TS1**

**Tested Version :** [MyLambdaMySQLJava](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions/MyLambdaMySQLJava)

**Tps : 451 tps (Throughput)**

**Log Level :** Error

**Test duration :** 11 minutes 18 seconds (678 seconds)

**Test Time : February 14, 2020 9:58:44.213 PM**

**Additional Comment :**

**Test Results:**

**Total Call Created :** **305.717 requests**

**Success Percentage : %100.00 (No message lost)**

**Average Response Time : 209 ms**

**Min Response Time : 47 ms**

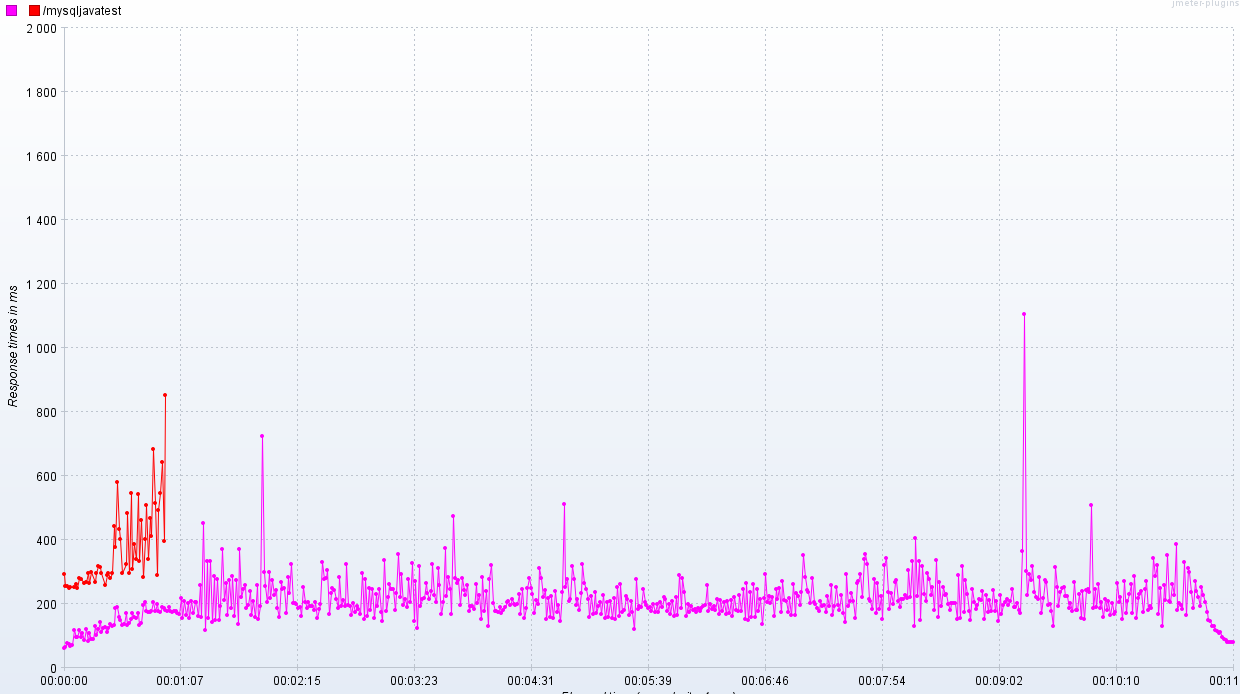
**Max Response Time : 935 ms**

**Cpu usage : % ??**

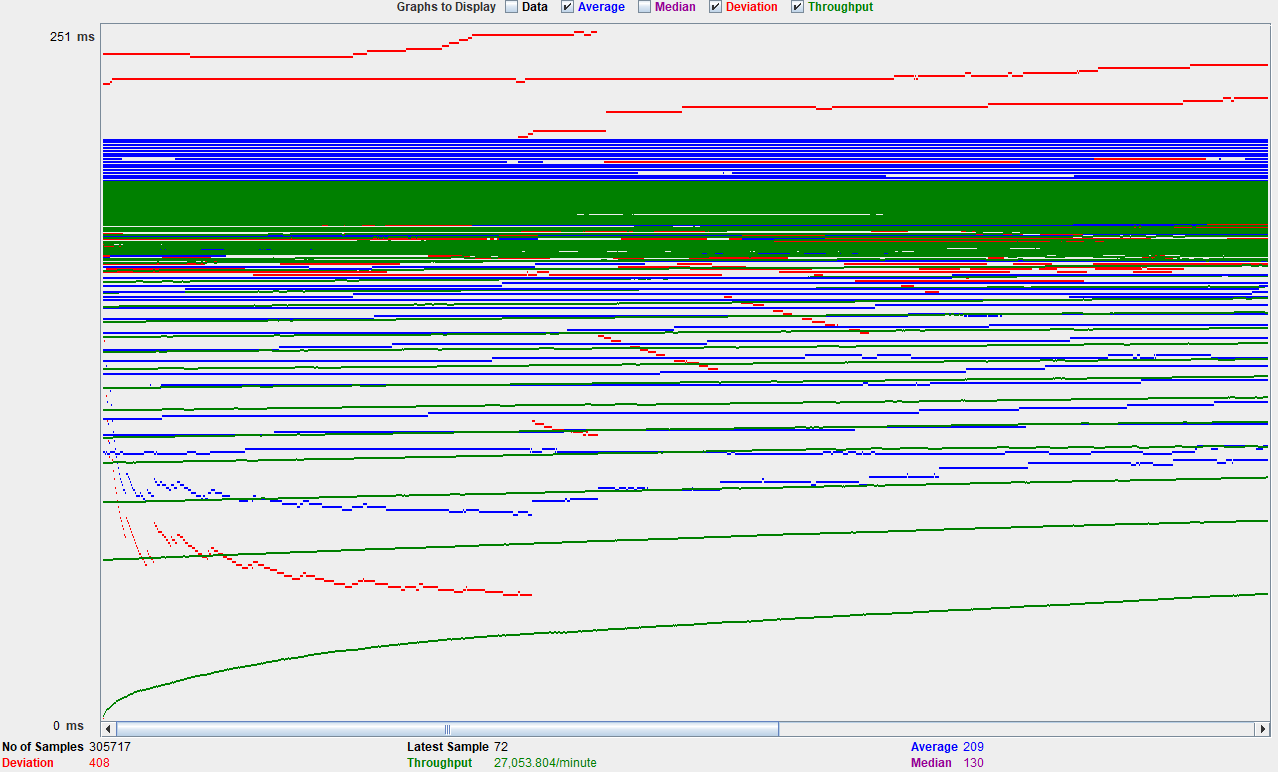
**Memory Increament : 167 Mbyte(Max)**

**Max. Sim. Threads(users) : 100 threads**

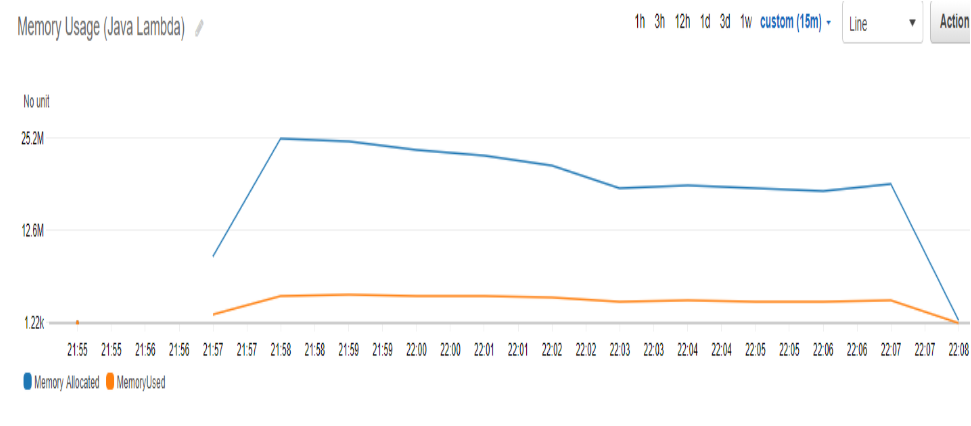
**Additional Comment :**



**Figure 3: Response Time Graph**

****

**Figure 4: Graph Results (Throughput, Average, Deviation)**

****

**Figure 5: Memory Usage/Allocated Graph**

# Java LOAD TEST-2 (50 threads-381 TPS)

**Aim:** To see the stability, response times and memory usage of the system under load with high call rate.

**Test Configuration:**

**System Under Test :**  **TS1**

**Tested Version :** [MyLambdaMySQLJava](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions/MyLambdaMySQLJava)

**Tps : 381 tps (Throughput)**

**Log Level :** Error

**Test duration :** 10 minutes 38 seconds (638 seconds)

**Test Time : February 15, 2020 11:00:14.321 PM**

**Additional Comment :**

**Test Results:**

**Total Call Created :** **214.891 requests**

**Success Percentage : %100.00 (No message lost)**

**Average Response Time : 143 ms**

**Min Response Time : 52 ms**

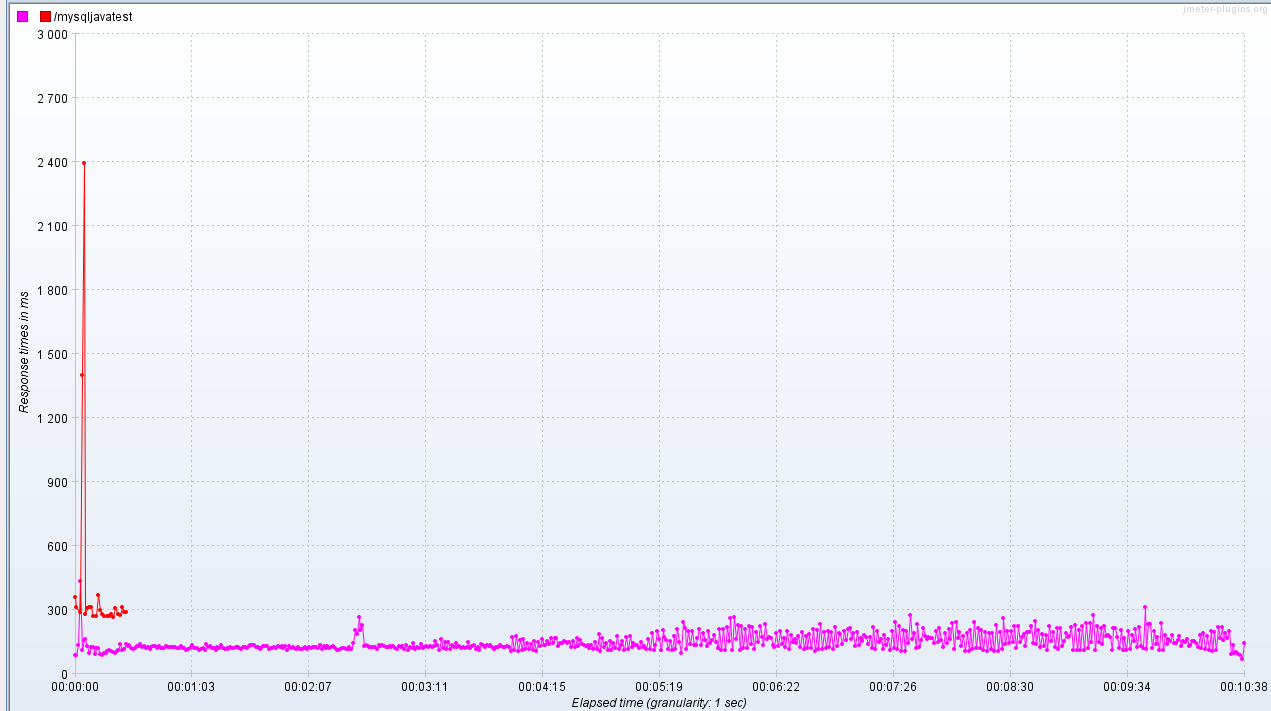
**Max Response Time : 2470 ms**

**Cpu usage : % ??**

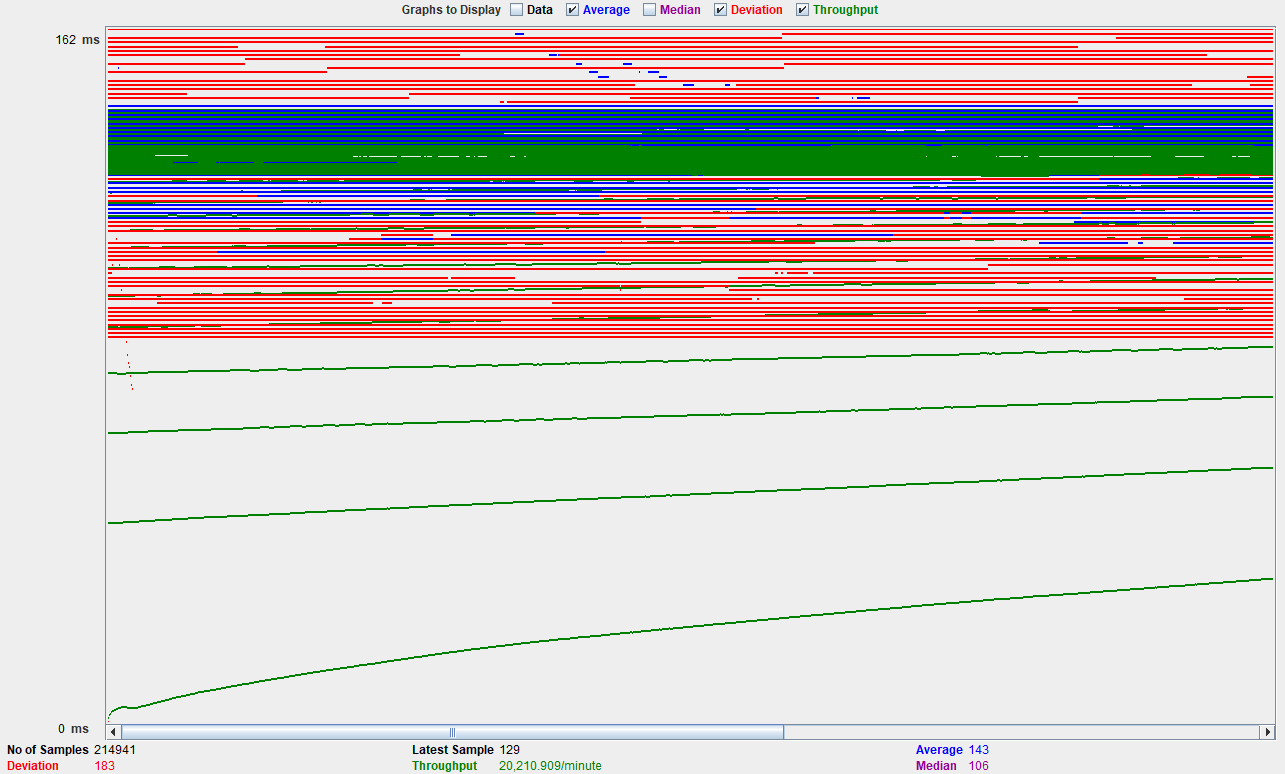
**Memory Increament : 164 Mbyte(Max)**

**Max. Sim. Threads(users) : 50 threads**

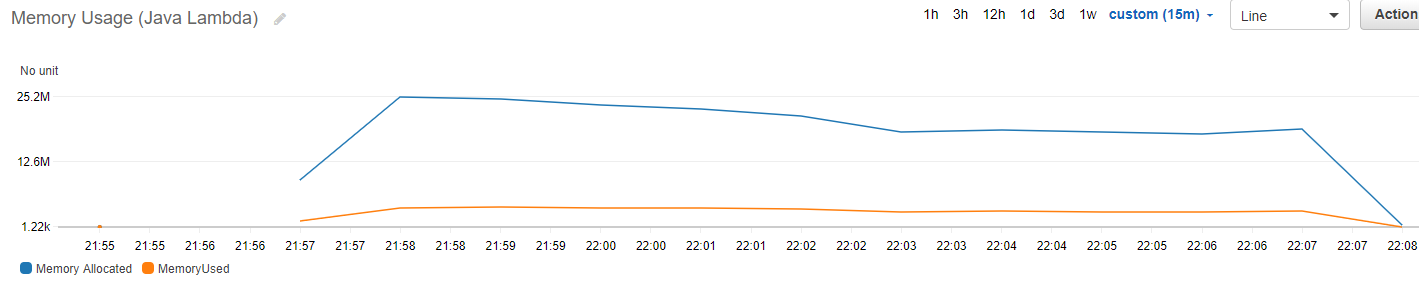
**Additional Comment :**



**Figure 6: Response Time Graph**



**Figure 7: Graph Results (Throughput, Average, Deviation)**



**Figure 8: Memory Usage/Allocated Graph**

# Node.js Lambda Load Tests

# Node.js LOAD TEST-1 (100 threads-902 TPS)

**Aim:** To see the stability, response times and memory usage of the system under load with high call rate.

**Test Configuration:**

**System Under Test :**  **TS1**

**Tested Version :** [nodejsPerformance-dev-insert\_user](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions/nodejsPerformance-dev-insert_user)

**Tps : 902 tps (Throughput)**

**Log Level :** Error

**Test duration :** 10 minutes 24 seconds (624 seconds)

**Test Time : February 15, 2020 11:40:10.212 PM**

**Additional Comment :**

**Test Results:**

**Total Call Created :** **563.753 requests**

**Success Percentage : 82.58% (17.42% requests failed)**

**Average Response Time : 105 ms**

**Min Response Time : 0 ms (??-Jmeter shows 0.)**

**Max Response Time : 45.984 ms**

**Cpu usage : % ??**

**Memory Increament : 130 Mbyte**

**Max. Sim. Threads(users) : 100 threads**

**Additional Comment :**

Big amount of requests failed due to the Too many open connections in database.

Decided to decrease simultaneous thread count to 50 for this test.

# Node.js LOAD TEST-2 (50 threads- 694 TPS)

**Aim:** To see the stability, response times and memory usage of the system under load with high call rate.

**Test Configuration:**

**System Under Test :**  **TS1**

**Tested Version :** [nodejsPerformance-dev-insert\_user](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions/nodejsPerformance-dev-insert_user)

**Tps : 694 tps (Throughput)**

**Log Level :** Error

**Test duration :** 10 minutes 38 seconds (638 seconds)

**Test Time : February 15, 2020 11:55:15.212 PM**

**Additional Comment :**

**Test Results:**

**Total Call Created :** **563.753 requests**

**Success Percentage : 82.58% (17.42% requests failed)**

**Average Response Time : 69 ms**

**Min Response Time : 43 ms**

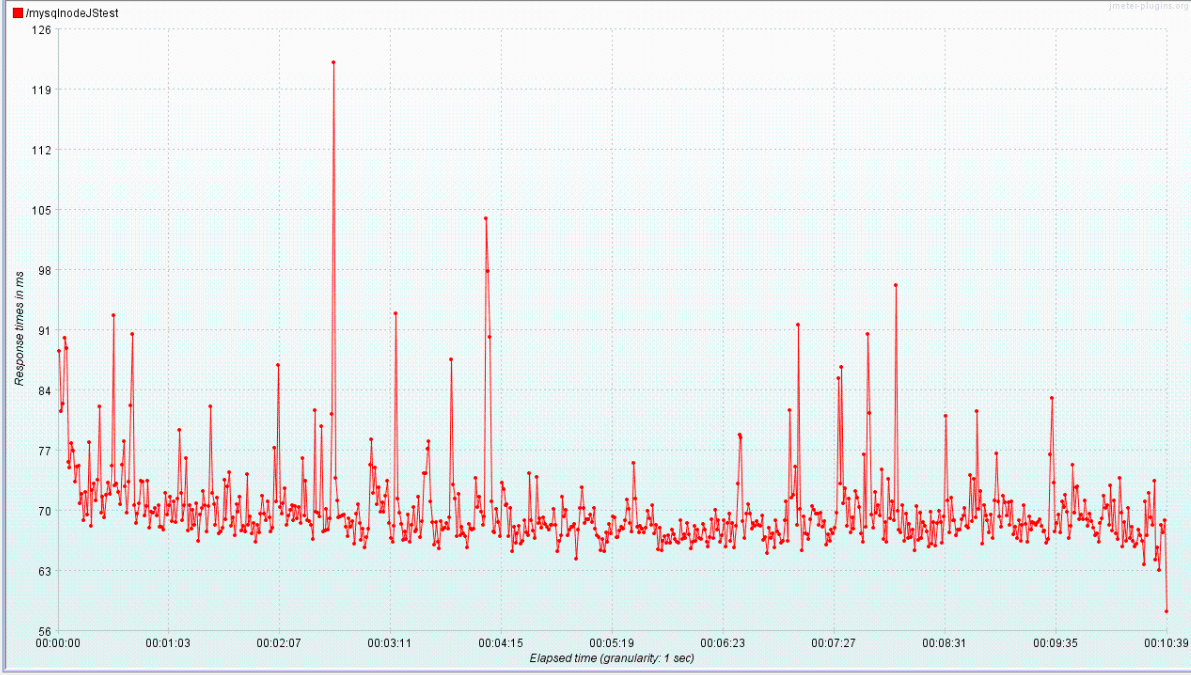
**Max Response Time : 3694 ms**

**Cpu usage : % ??**

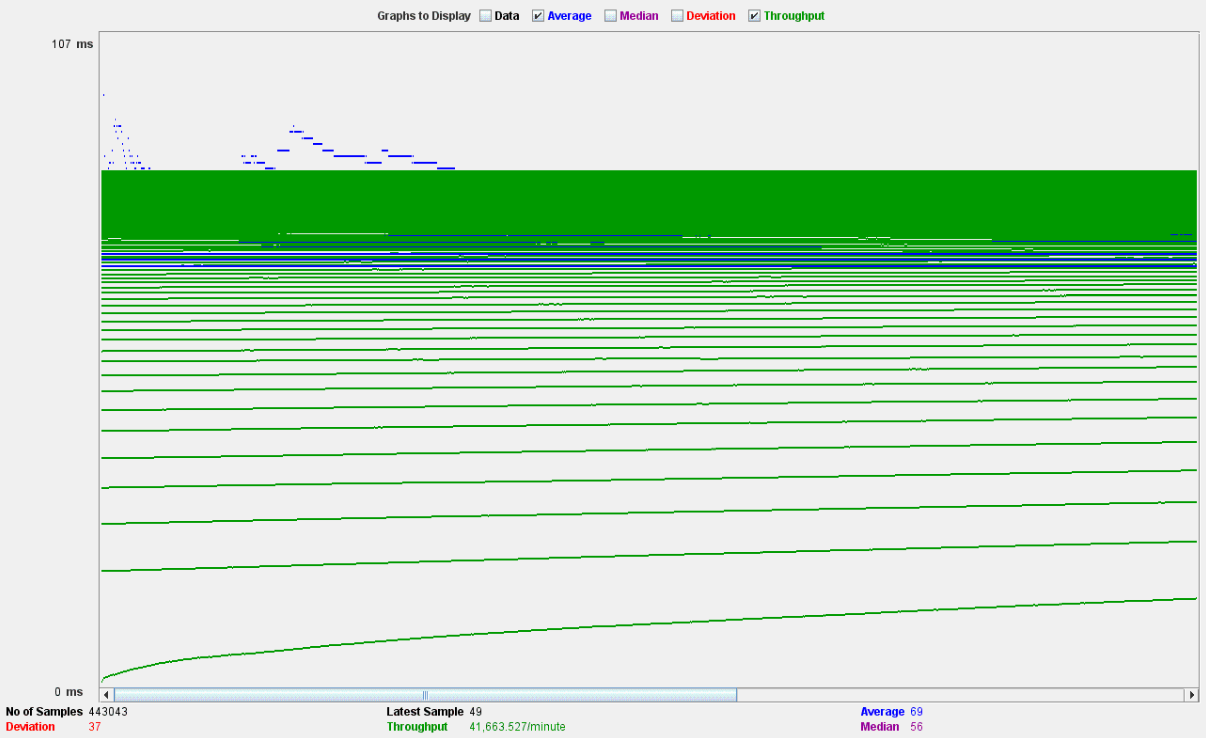
**Memory Increament : 126 Mbyte (Max)**

**Max. Sim. Threads(users) : 50 threads**

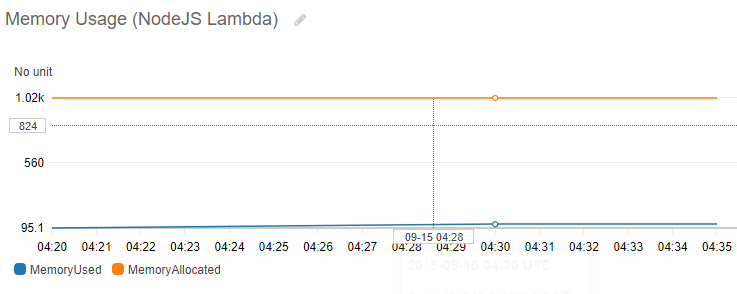
**Additional Comment :**



**Figure 7: Response Time Graph**



**Figure 8: Graph Results (Throughput, Average, Deviation)**

****

**Figure 8: Memory Usage (NodeJs Lambda)**

# HiAlan End-to-End Endurance Tests

TO BE DONE!

# SUMMARY & COMPARISON

There is no significance difference between Java and NodeJs Lambda functions. Java’s only disadvantage is cold start and there are solutions for cold start. Java is using slightly more memory than NodeJs.

|  |  |  |
| --- | --- | --- |
| **50 Threads- 10 Minutes TEST** | **JAVA** | **NODE JS** |
| **MAX Mbyte** | 164 MByte | 126 MByte |
| **Average Response Time** | 143 ms | 69 ms |
| **Max Response Time** | 2470 ms | 3694 ms |
| **Min Response Time** | 52 ms | 43 ms |
| **Max Seen RPS** | 381 rps | 694 rps |
| **FAILED REQUESTS** | 0 | 0 |
| **Total Requests** | 214.891 requests | 563.753 requests |

|  |  |  |
| --- | --- | --- |
| **100 Threads- 10 Minutes TEST** | **JAVA** | **NODE JS** |
| **MAX Mbyte** | 167 Mbyte | 135 Mbyte |
| **Average Response Time** | 209 ms | 105 ms |
| **Max Response Time** | 935 ms | 45.984 ms |
| **Min Response Time** | 47 ms | 0 - ?? Jmeter stg wrong |
| **Max Seen RPS** | 451 rps | 902 rps |
| **FAILED REQUESTS** | 0 | **(17.42% requests failed)**  **DB bottle neck** |
| **Total Requests** | 305.717 requests | 563.753 requests |

# 3rd Party Test Results

<https://medium.com/the-theam-journey/benchmarking-aws-lambda-runtimes-in-2019-part-ii-50e796d3d11b>

**Summary of 3rd Party Test Result :** It is apparent for all the CRUD operations above that Node.js and Haskell are being penalized in the average execution duration due to a really high delta between the maximum and minimum values. This negative impact is more noticeable in Node.js, where the minimum execution duration is usually no more than 2 ms away from the best performers. Furthermore, according to the data displayed above, C#, Python, and Java have similar durations if we don’t take into consideration cold-starts. The single digit execution durations of C#, Python, Node.js, and Java are great, bearing in mind that a NoSQL database is being accessed. On the other hand, Haskell, in spite of being slower than other runtimes, has acceptable execution durations that we are very confident will improve in the future.It can also be seen that Create, Get, Update and Delete operations have similar performance, while the List operation has a slightly slower performance, around a 20 percent difference, due to the scan on the whole table. **Memory usage:** Memory usage increases when performing AWS DynamoDB reads or writes. However, it is still below 128 MB for all the runtimes except Java, which demanded a maximum of 147 MB in some of the requests.

# There is an interesing announcement from two weeks ago:

We’re excited to announce a major improvement to how [AWS Lambda](https://aws.amazon.com/lambda/) functions work with your [Amazon VPC](https://aws.amazon.com/vpc/) networks. **With today’s launch, you will see dramatic improvements to function startup performance and more efficient usage of elastic network interfaces.** These improvements are rolling out to all existing and new VPC functions, at no additional cost. Roll out begins today and continues gradually over the next couple of months across all Regions. (For details: <https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/>)

# 