454A Demo

FinOps, Performance & E2E Architecture

Logan Zhang | Edgar Palomino

TABLE OF CONTENTS

Introduction

Stress API (CPU & Memory)

Terraform and Auto Scaling

Performance Testing Tools

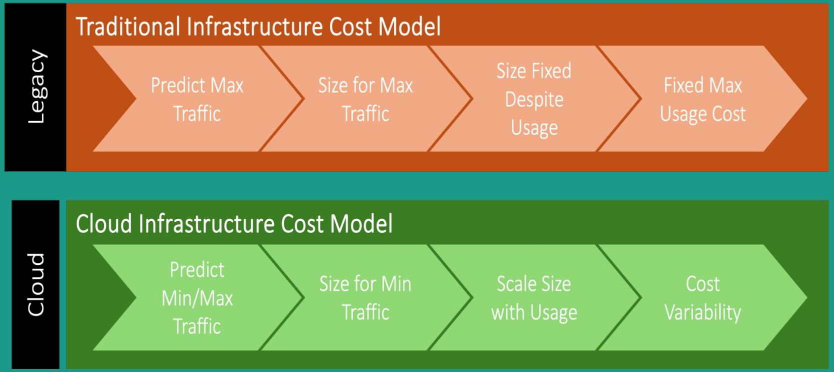
Test Results

Summary and Q&A

INTRODUCTION

FinOps

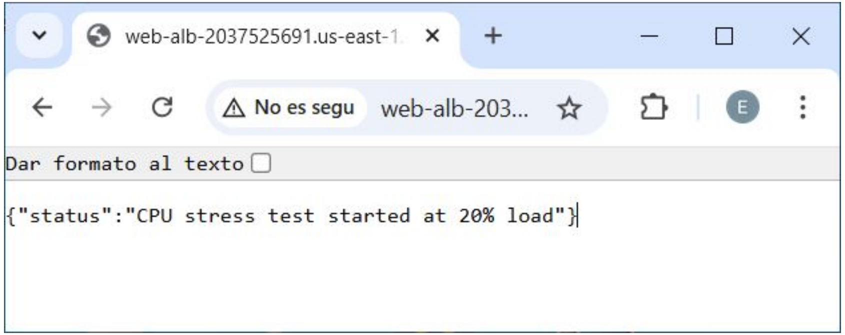
- Aligning product features, architecture and strategic direction with the cost in the cloud
- Prioritization of infrastructure spend in development of architecture and design decisions



Stress APIs

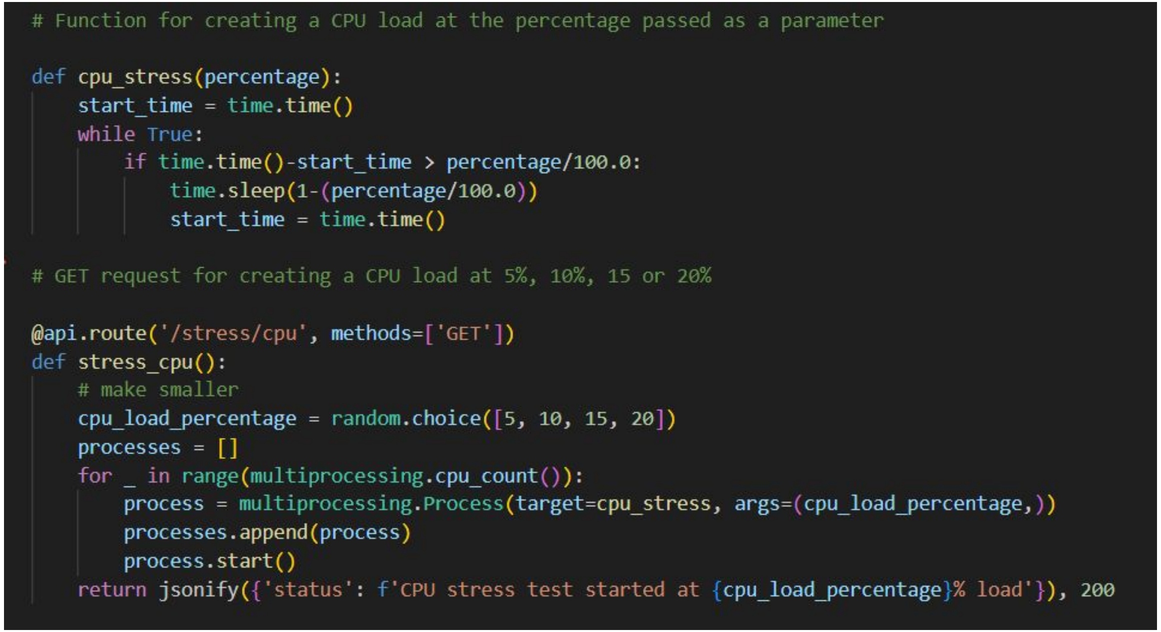
Route 1: /stress/cpu

1. Selects a random CPU load percentage between 5%, 10%, 15% and 20%
2. Detects the number of CPUs of the machine currently running
3. Simulates stress by executing an endless loop for a fraction of a second



```
{"status":"CPU stress test started at 20% load"}
```

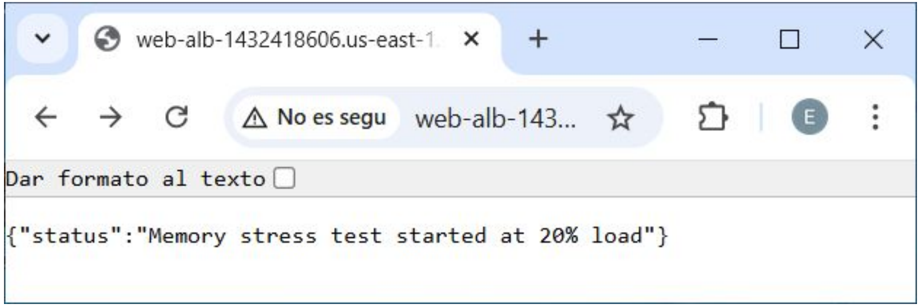
Route 1: /stress/cpu



```python
# Function for creating a CPU load at the percentage passed as a parameter

def cpu_stress(percentage):
    start_time = time.time()
    while True:
        if time.time()-start_time > percentage/100.0:
            time.sleep(1-(percentage/100.0))
            start_time = time.time()

# GET request for creating a CPU load at 5%, 10%, 15 or 20%

@api.route('/stress/cpu', methods=['GET'])
def stress_cpu():
    # make smaller
    cpu_load_percentage = random.choice([5, 10, 15, 20])
    processes = []
    for _ in range(multiprocessing.cpu_count()):
        process = multiprocessing.Process(target=cpu_stress, args=(cpu_load_percentage,))
        processes.append(process)
        process.start()
    return jsonify({'status': f'CPU stress test started at {cpu_load_percentage}% load'}), 200
```

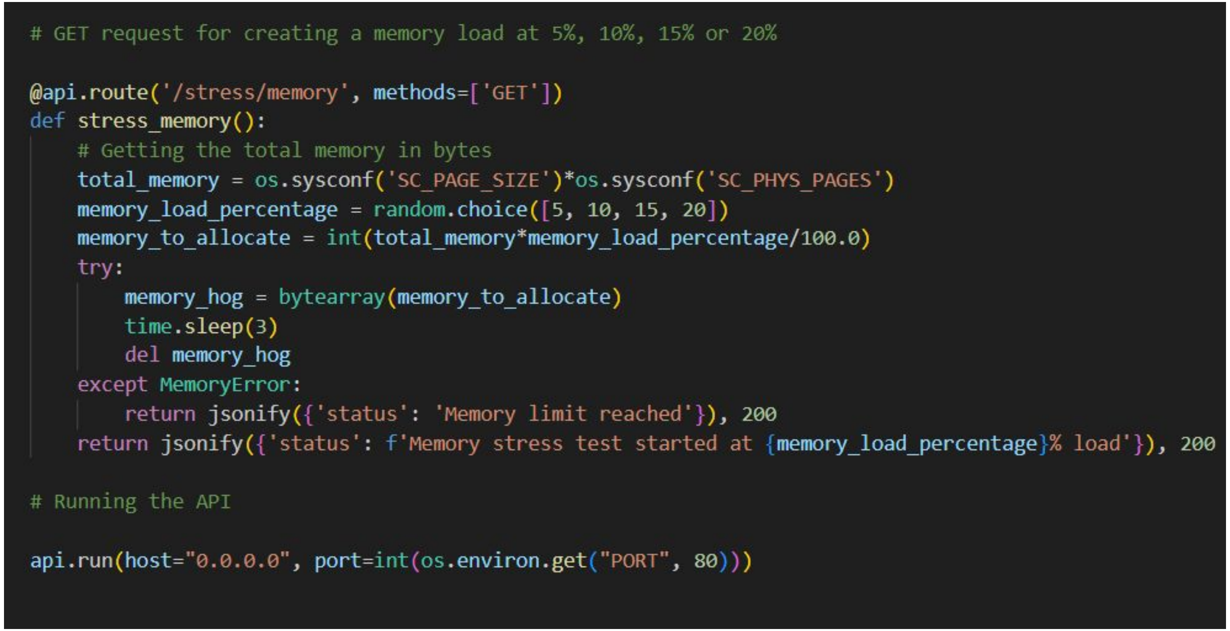
Route 2: /stress/memory

1. Selects a random memory load percentage between 5%, 10%, 15% and 20%
2. Detects the number of pages and their sizes of the machine currently running
3. Simulates stress by allocating and freeing a byte array that hogs the memory space provided by the pages



```
{"status":"Memory stress test started at 20% load"}
```

Route 2: /stress/memory



```python
# GET request for creating a memory load at 5%, 10%, 15% or 20%

@api.route('/stress/memory', methods=['GET'])
def stress_memory():
    # Getting the total memory in bytes
    total_memory = os.sysconf('SC_PAGE_SIZE')*os.sysconf('SC_PHYS_PAGES')
    memory_load_percentage = random.choice([5, 10, 15, 20])
    memory_to_allocate = int(total_memory*memory_load_percentage/100.0)
    try:
        memory_hog = bytearray(memory_to_allocate)
        time.sleep(3)
        del memory_hog
    except MemoryError:
        return jsonify({'status': 'Memory limit reached'}), 200
    return jsonify({'status': f'Memory stress test started at {memory_load_percentage}% load'}), 200

# Running the API

api.run(host="0.0.0.0", port=int(os.environ.get("PORT", 80)))
```

Instantiating the Cloud via Terraform

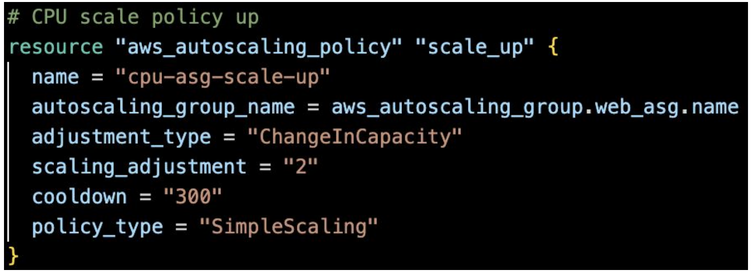
Terraform

- Terraform:
  - An Infrastructure as Code (IaC) tool that simplifies deployment into an AWS environment
    - Define and provision infrastructure using a declarative configuration language: HashiCorp Configuration Language (HCL)
    - HCL is a high level language that makes it easy to define the desired state of the infrastructure

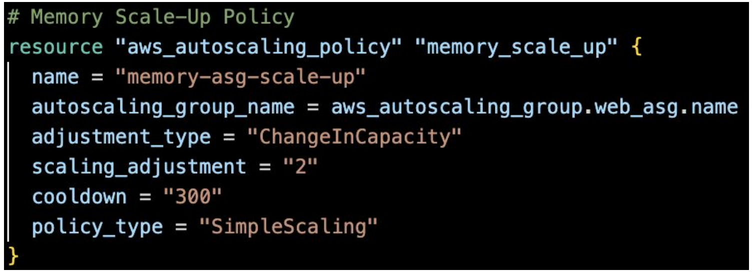
Auto Scaling Policies

Auto Scaling

- Auto Scaling (Simple Scaling):
    - Using IaC, we defined rules for the EC2 cluster
        - These rules scale the capacity of your Auto Scaling group in predefined increments
        - These increments are measured by CloudWatch alarms which is a AWS feature that measures the stress we induce on an EC2 instance



```
# CPU scale policy up
resource "aws_autoscaling_policy" "scale_up" {
  name = "cpu-asg-scale-up"
  autoscaling_group_name = aws_autoscaling_group.web_asg.name
  adjustment_type = "ChangeInCapacity"
  scaling_adjustment = "2"
  cooldown = "300"
  policy_type = "SimpleScaling"
}
```

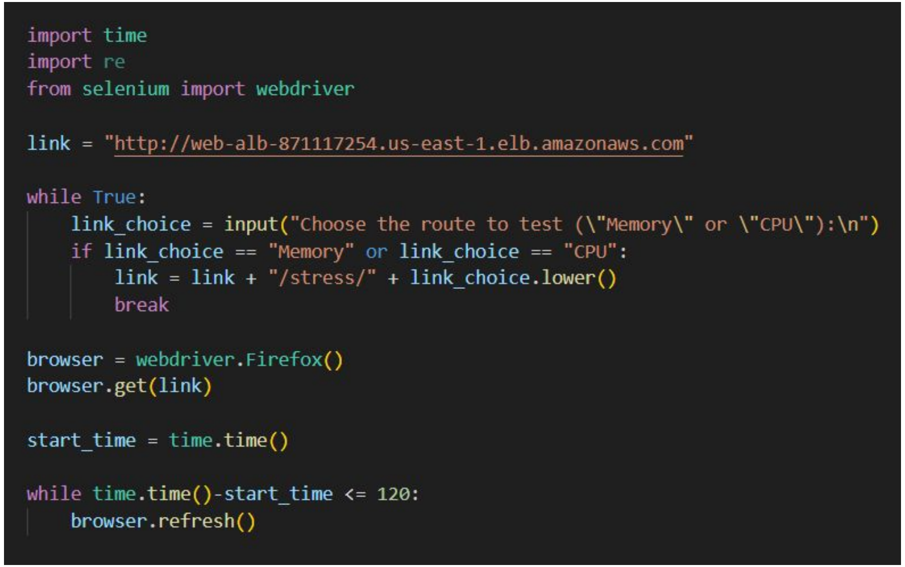


```
# Memory Scale-Up Policy
resource "aws_autoscaling_policy" "memory_scale_up" {
  name = "memory-asg-scale-up"
  autoscaling_group_name = aws_autoscaling_group.web_asg.name
  adjustment_type = "ChangeInCapacity"
  scaling_adjustment = "2"
  cooldown = "300"
  policy_type = "SimpleScaling"
}
```

How to test these routes?

Naive approach

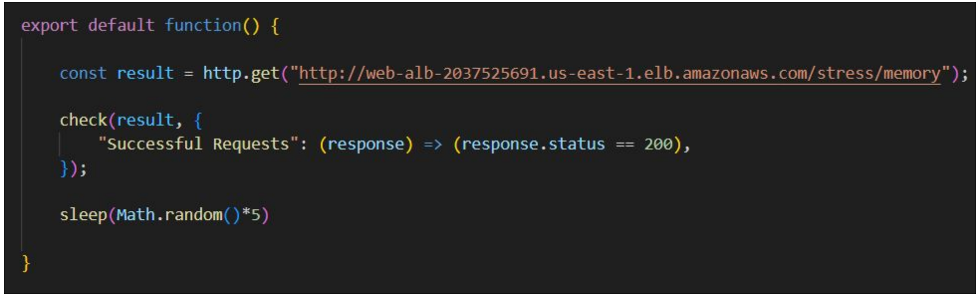
- Our first idea was to use libraries that provided functions for making HTTP requests
- We wrote two simple scripts using requests and selenium
- But the amount of information we were getting was very limited and not enough to make a comprehensive analysis



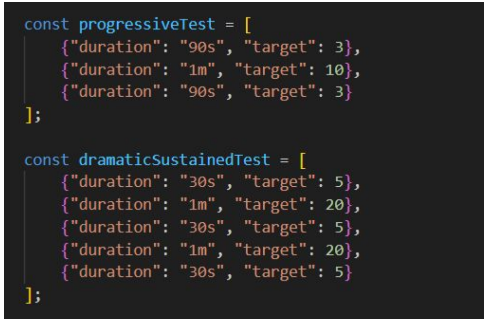
```
import time
import re
from selenium import webdriver

link = "http://web-alb-871117254.us-east-1.elb.amazonaws.com"

while True:
    link_choice = input("Choose the route to test (\"Memory\" or \"CPU\"):\n")
    if link_choice == "Memory" or link_choice == "CPU":
        link = link + "/stress/" + link_choice.lower()
        break

browser = webdriver.Firefox()
browser.get(link)

start_time = time.time()

while time.time()-start_time <= 120:
    browser.refresh()
```

Grafana k6 OSS

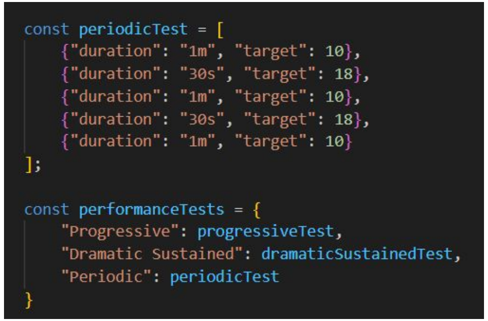
- Our client suggested us to use the k6 OSS tool to be able to conduct more refined testing
- This tool also provided real-time graphics and metrics that allowed us to follow our performance tests step by step
- It also permitted us to define more specific and detailed types of loading tests which was extremely helpful



```
export default function() {

    const result = http.get("http://web-alb-2037525691.us-east-1.elb.amazonaws.com/stress/memory");

    check(result, {
        "Successful Requests": (response) => (response.status == 200),
    });

    sleep(Math.random()*5)

}
```



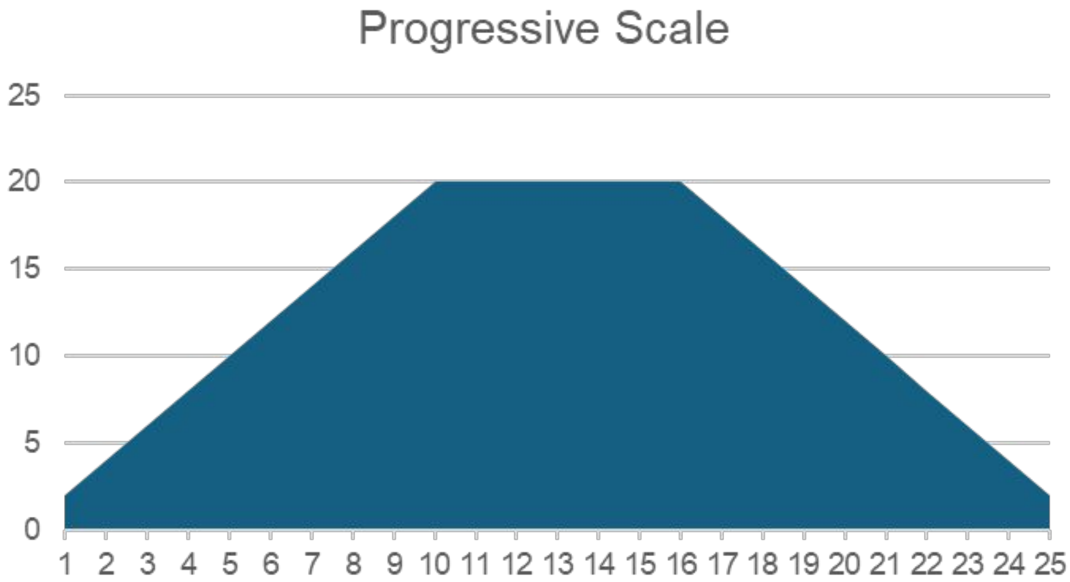
```
const progressiveTest = [
    {"duration": "90s", "target": 3},
    {"duration": "1m", "target": 10},
    {"duration": "90s", "target": 3}
];

const dramaticSustainedTest = [
    {"duration": "30s", "target": 5},
    {"duration": "1m", "target": 20},
    {"duration": "30s", "target": 5},
    {"duration": "1m", "target": 20},
    {"duration": "30s", "target": 5}
];
```



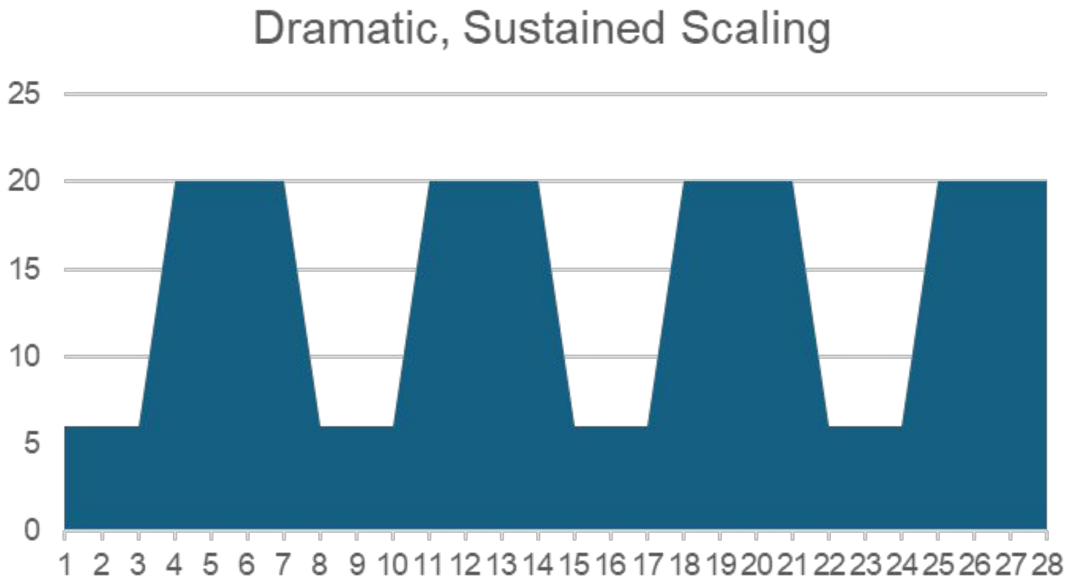
```
const periodicTest = [
    {"duration": "1m", "target": 10},
    {"duration": "30s", "target": 18},
    {"duration": "1m", "target": 10},
    {"duration": "30s", "target": 18},
    {"duration": "1m", "target": 10}
];

const performanceTests = {
    "Progressive": progressiveTest,
    "Dramatic Sustained": dramaticSustainedTest,
    "Periodic": periodicTest
}
```

Types of tests

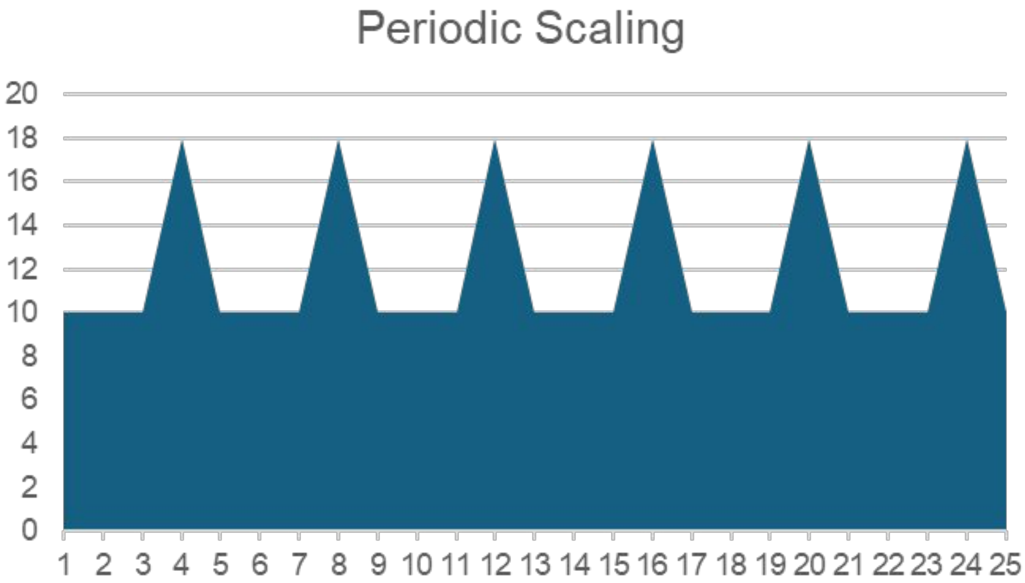
Progressive testing



Dramatic sustained testing

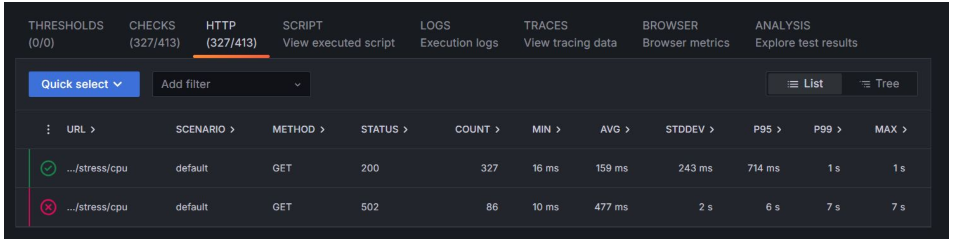


Periodic testing



What were our actual results?

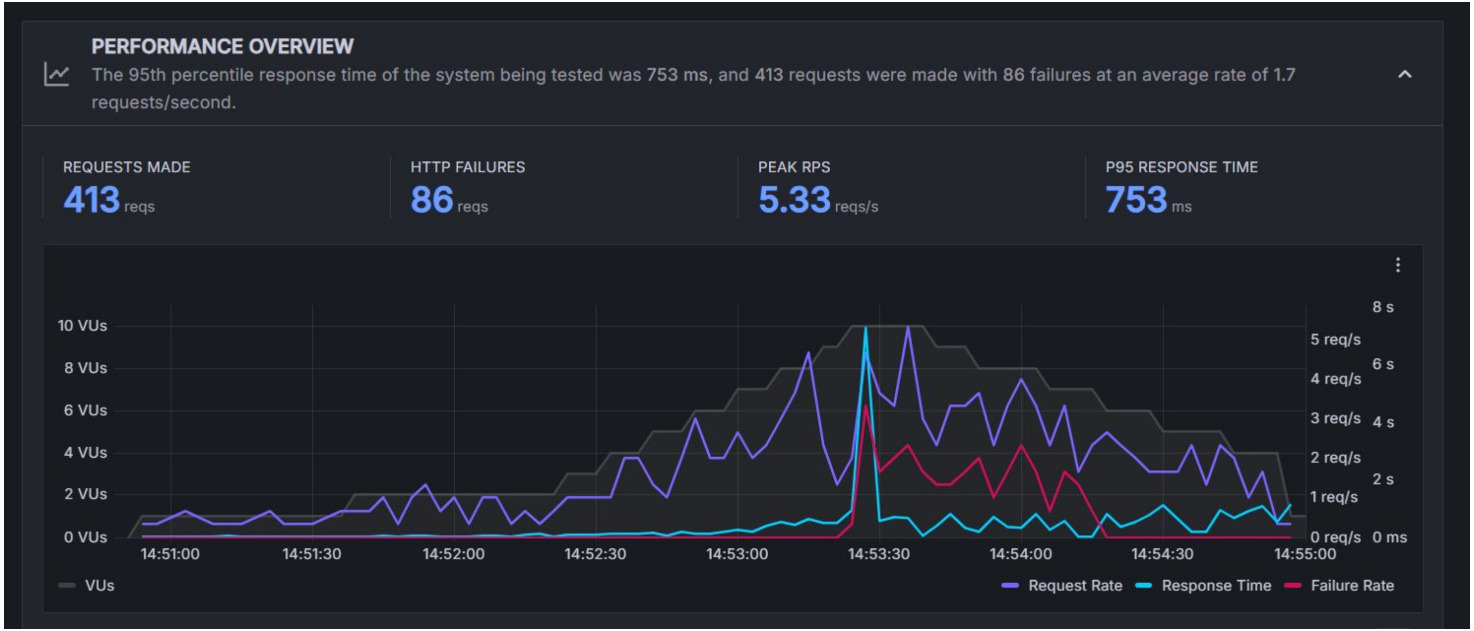
Progressive testing



THRESHOLDS (0/0) | CHECKS (327/413) | HTTP (327/413) | SCRIPT View executed script | LOGS Execution logs | TRACES View tracing data | BROWSER Browser metrics | ANALYSIS Explore test results

Quick select | Add filter | List | Tree

	URL	SCENARIO	METHOD	STATUS	COUNT	MIN	AVG	STDDEV	P95	P99	MAX
✓	.../stress/cpu	default	GET	200	327	16 ms	159 ms	243 ms	714 ms	1 s	1 s
✗	.../stress/cpu	default	GET	502	86	10 ms	477 ms	2 s	6 s	7 s	7 s

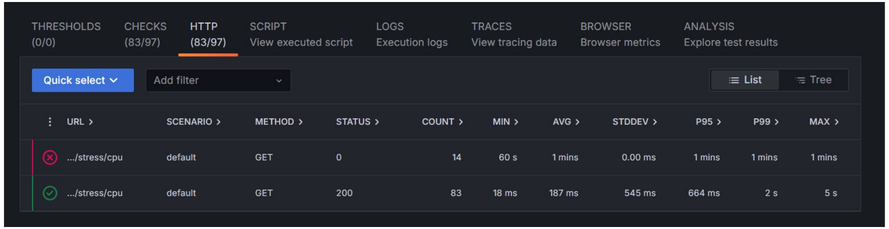


PERFORMANCE OVERVIEW

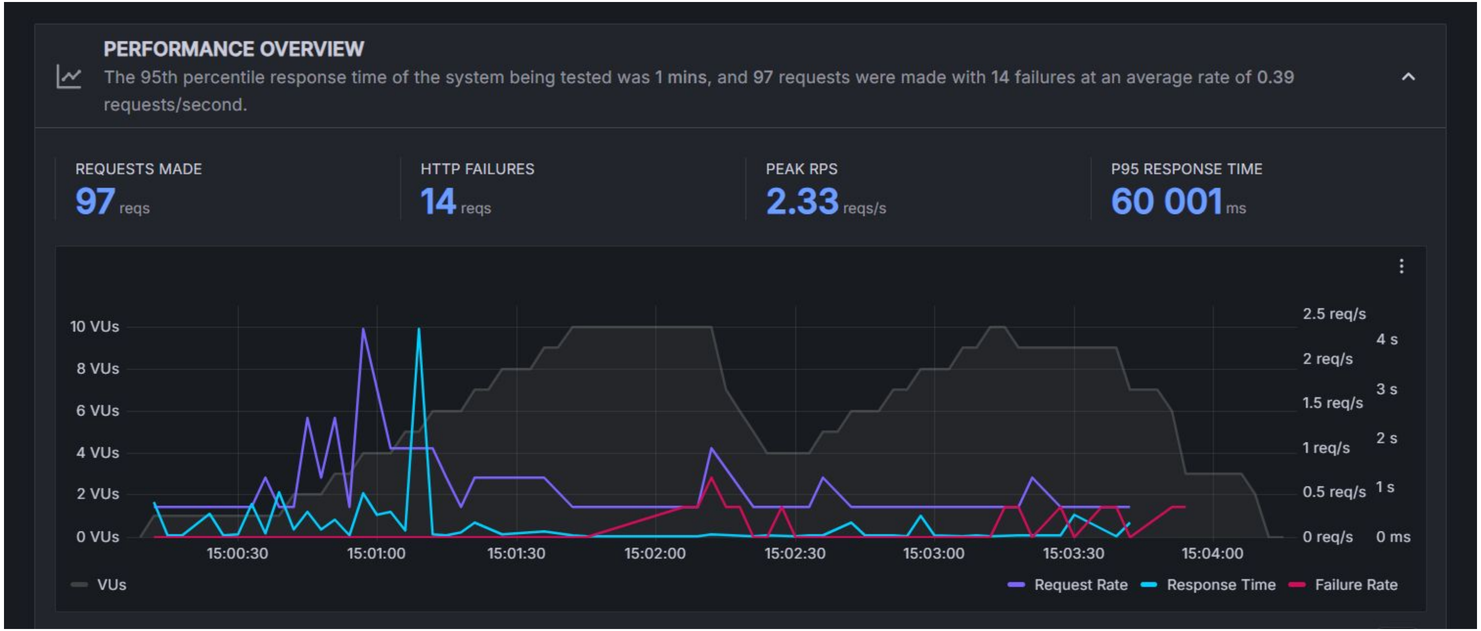
The 95th percentile response time of the system being tested was 753 ms, and 413 requests were made with 86 failures at an average rate of 1.7 requests/second.

REQUESTS MADE	HTTP FAILURES	PEAK RPS	P95 RESPONSE TIME
413 reqs	86 reqs	5.33 reqs/s	753 ms

Dramatic sustained testing



	URL	SCENARIO	METHOD	STATUS	COUNT	MIN	AVG	STDDEV	P95	P99	MAX
⊗	.../stress/cpu	default	GET	0	14	60 s	1 mins	0.00 ms	1 mins	1 mins	1 mins
✓	.../stress/cpu	default	GET	200	83	18 ms	187 ms	545 ms	664 ms	2 s	5 s



PERFORMANCE OVERVIEW

The 95th percentile response time of the system being tested was 1 mins, and 97 requests were made with 14 failures at an average rate of 0.39 requests/second.

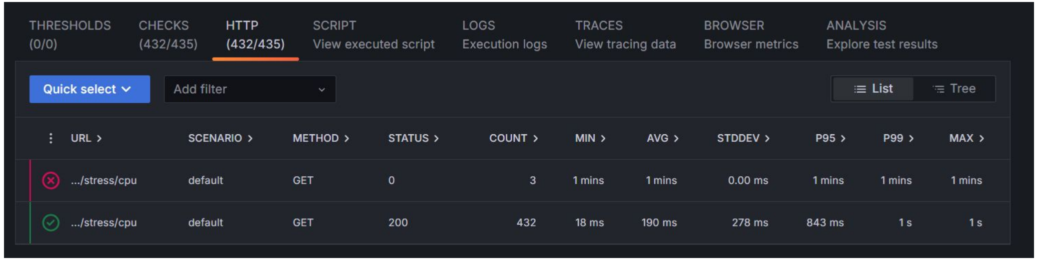
REQUESTS MADE
97 reqs

HTTP FAILURES
14 reqs

PEAK RPS
2.33 reqs/s

P95 RESPONSE TIME
60 001 ms

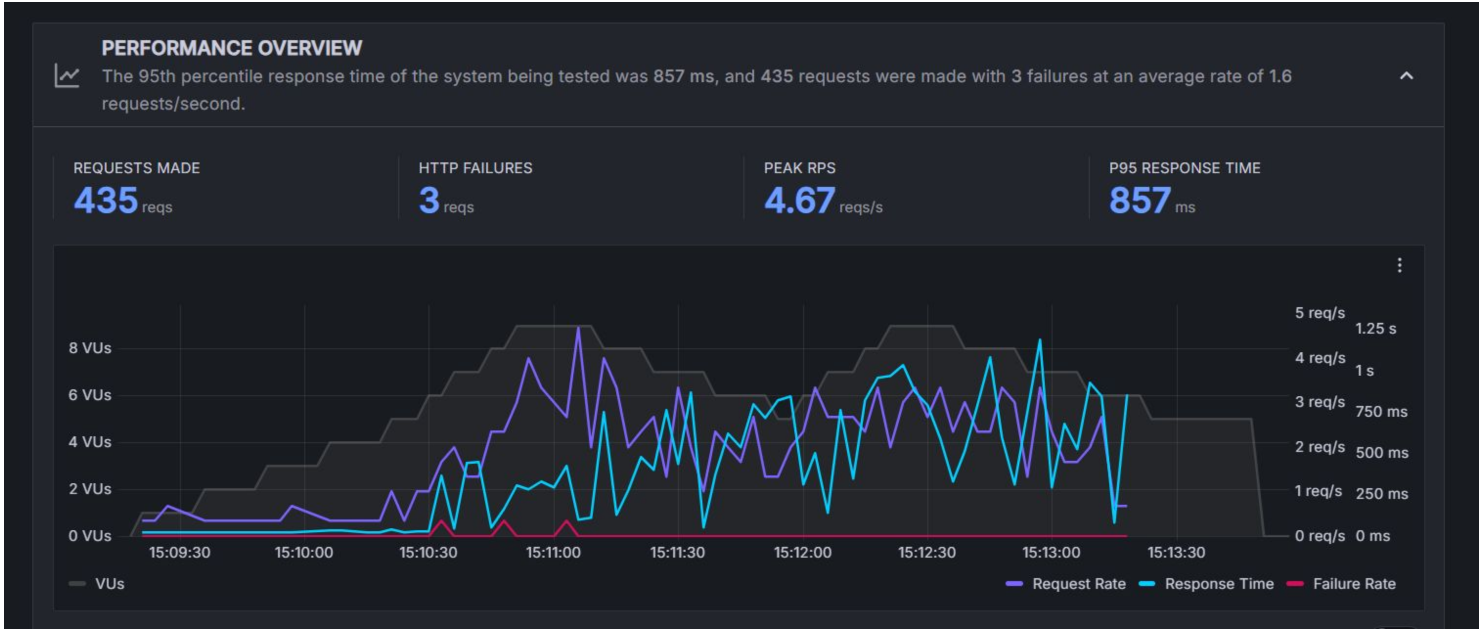
Periodic testing



THRESHOLDS (0/0) | CHECKS (432/435) | HTTP (432/435) | SCRIPT View executed script | LOGS Execution logs | TRACES View tracing data | BROWSER Browser metrics | ANALYSIS Explore test results

Quick select | Add filter | List | Tree

⋮	URL ›	SCENARIO ›	METHOD ›	STATUS ›	COUNT ›	MIN ›	AVG ›	STDDEV ›	P95 ›	P99 ›	MAX ›
⊗	.../stress/cpu	default	GET	0	3	1 mins	1 mins	0.00 ms	1 mins	1 mins	1 mins
✓	.../stress/cpu	default	GET	200	432	18 ms	190 ms	278 ms	843 ms	1 s	1 s



PERFORMANCE OVERVIEW

The 95th percentile response time of the system being tested was 857 ms, and 435 requests were made with 3 failures at an average rate of 1.6 requests/second.

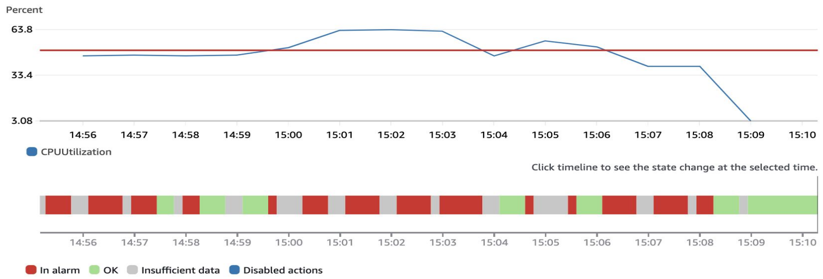
REQUESTS MADE
435 reqs

HTTP FAILURES
3 reqs

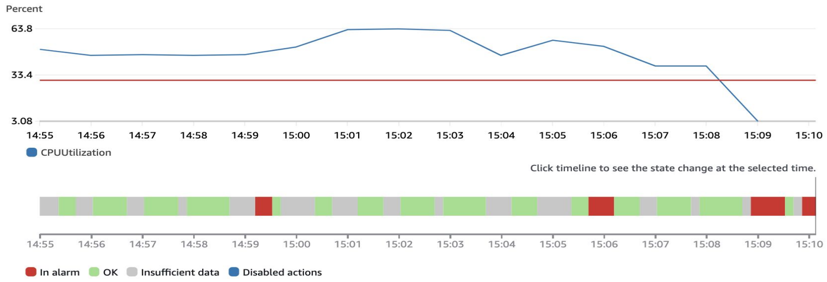
PEAK RPS
4.67 reqs/s

P95 RESPONSE TIME
857 ms

AWS Cloud Watch



Click timeline to see the state change at the selected time.

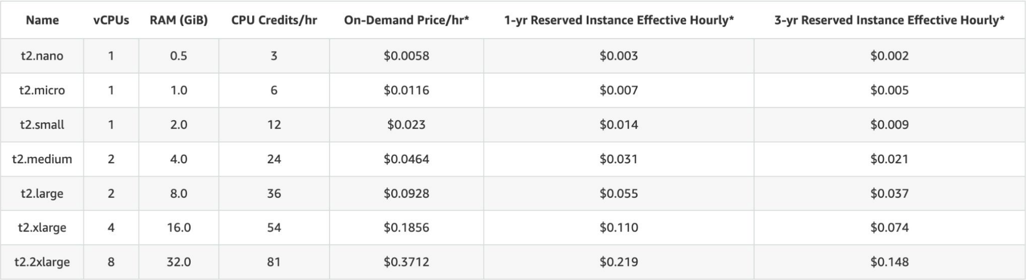


Click timeline to see the state change at the selected time.

Preliminary Pricing Evaluations

Full Capacity vs Dynamic Allocation Pricings

- AWS EC2 T2 Instances Pricing:



Name	vCPUs	RAM (GiB)	CPU Credits/hr	On-Demand Price/hr*	1-yr Reserved Instance Effective Hourly*	3-yr Reserved Instance Effective Hourly*
t2.nano	1	0.5	3	$0.0058	$0.003	$0.002
t2.micro	1	1.0	6	$0.0116	$0.007	$0.005
t2.small	1	2.0	12	$0.023	$0.014	$0.009
t2.medium	2	4.0	24	$0.0464	$0.031	$0.021
t2.large	2	8.0	36	$0.0928	$0.055	$0.037
t2.xlarge	4	16.0	54	$0.1856	$0.110	$0.074
t2.2xlarge	8	32.0	81	$0.3712	$0.219	$0.148

- Cost Comparison (Assuming 24 hour services required daily):
  - Yearly Non Dynamic Full Capacity Pricing (5 instances): $467.712
  - Yearly Dynamic Pricing (night and weekends only one instance–hypothetical best case): $227.1744
  - 48% Saving via dynamic resource allocation (FinOps)

Summary and Q&A

Summary and Q&A

- Auto Scaling and cloud computing is a powerful tool for industry to optimize budget when creating the necessary frameworks dedicated to their services
- FinOps is an evolving and revolutionary discipline that has allowed organizations to align product features, architecture and strategic direction with the cost in the cloud
- By adhering to FinOps principles and utilizing scaling policies, industry is able to pay for the amount of computer resources in response to customer needs and general usage trends
- As a technology that is not bounded by any particular field of industry, FinOps is very essential to many different organizations catered to serving to any group of customers

Thank You!