DV1566 - Project

Group: 17
Student 1: Isac Svensson
Student 2: Robin Stenius

Description of the problem addressed

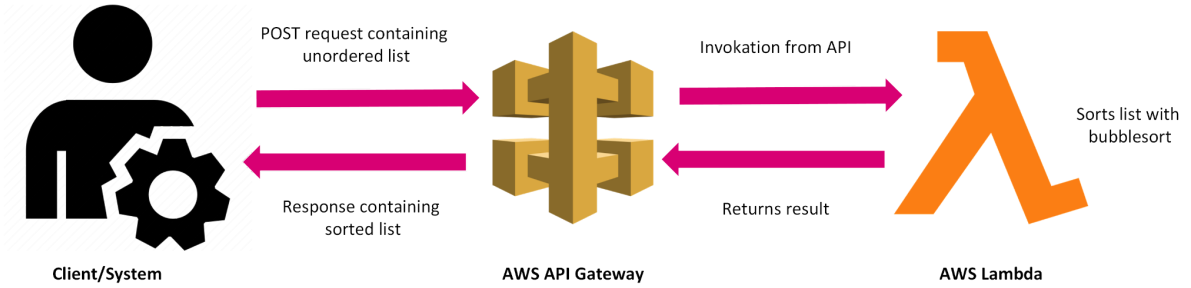
In this project we have addressed the scenario where a system is in need of a certain functionality, in this case, a sorting algorithm. This necessary functionality may be small, but has the potential to be under heavy stress in short and long periods of time, and it needs to be reliable and scalable for the overall functionality of a system.

Description of the design of the solution

In order to achieve this we used the AWS API Gateway service together with the AWS Lambda service. The API takes the incoming requests and sends them to the Lambda for processing and sends the response from the Lambda back to the user. The Lambda takes incoming requests for processing and when finished sends a response back to the API.

The API and Lambda service can be set with a timeout, meaning if enough time passes before it finishes, it will terminate prematurely. The maximum time for the Lambda timeout is 15 minutes, but in this case it was set to 20 seconds, this is because the API has a maximum of 29 seconds[1,2]. This restricts the array size and elements size range, and depending on the sorting algorithm, that Lambda could handle before timing out.

With the free tier, AWS API Gateway and Lambda service can process 1,000,000 requests for free. The default scaling capabilities of a Lambda is 1,000 instances, which can be used to restrict the scaling of certain Lambdas[3]. By using the AWS CloudWatch we can observe the scaling and availability of the Lambda functionality with various metrics described below.



[1] https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html
[2] https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
[3] https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html

Description of the Implementation of the solution

We began with creating the Lambda function, a simple Bubble sort algorithm, written in python found in Appendix 1. Which sorts the requested array in ascending order before returning with status code '200' and the finished sorted array. We restricted the Lambda to 200 instances. This made it easier for our personal computers to handle all the sending and receiving of requests. Which also simplifies the observation of AWS CloudWatch metrics from Lambda. Afterwards we created the API Gateway and connected it to the Lambda function.

Then we set up the necessary metrics for observing the scaling and availability of Lamba in CloudWatch. We chose the '**Invocations**', '**Throttles**' and '**ConcurrentExecutions**' from the Lambda functionality. The Invocations are the amount of times that the Lambda function has executed either successfully or with an error outcome, but it does not count requests that end up being throttled.
Throttles shows the number of Lambda requests that could not be handled due to a cap on the outward scaling of the Lambda service (i.e. 200 in our case). This means that when 200 instances are already up and running, busy with execution, further invocations become throttles instead. The ConcurrentExecutions track the number of Lambda instances that are up and running[4].

From the API Gateway we chose the '**Count**' and '**5XXError**'. The Count metric shows the total number of requests to the API in a given timeframe, this helps tracking the total number of requests that have been accumulated. It can also be used to validate that the requests actually reach the API. The 5XXError accumulates the number of server-side errors in a given period, which is helpful for knowing when something went wrong with either the API or Lambda, for example, timeouts or invalid inputs[5].

Test/validation design

To test the solution, both control the sorting of given arrays and the status code of the response. We built two tests with three parameters each, found in appendix 2 together with the logic to create threads, send and receive the data in appendix 3. Each test was run nine times with different arguments.

The first test tests the reliability of the service, validating that the result is a correctly sorted list whenever the service is run and returns a response with status code 200.

The second test tests the availability by checking the status code in the response when sending concurrent requests. The assertion is based on the fact that you should receive the same number of responses with status code 200 as the number of sent requests.

[4] https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html
[5] https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-metrics-and-dimensions.html

Since we have limited computing power in our hands we have set the reservations for concurrent invocations to 200, making it easier to reach the limit and testing. We chose a high time complexity $O(n^2)$ sorting algorithm that gives a clear difference between different iterations.

By observing the chosen CloudWatch metrics, we could validate the scalability of the Lambda service reaching the predefined cap of instances for which it can scale.
This can be further confirmed once the Throttles metric shows requests being declined when the scaling capabilities has reached its limits, meaning in a real-life scenario, further increase of the scalability would be necessary to investigate.

Experimental results

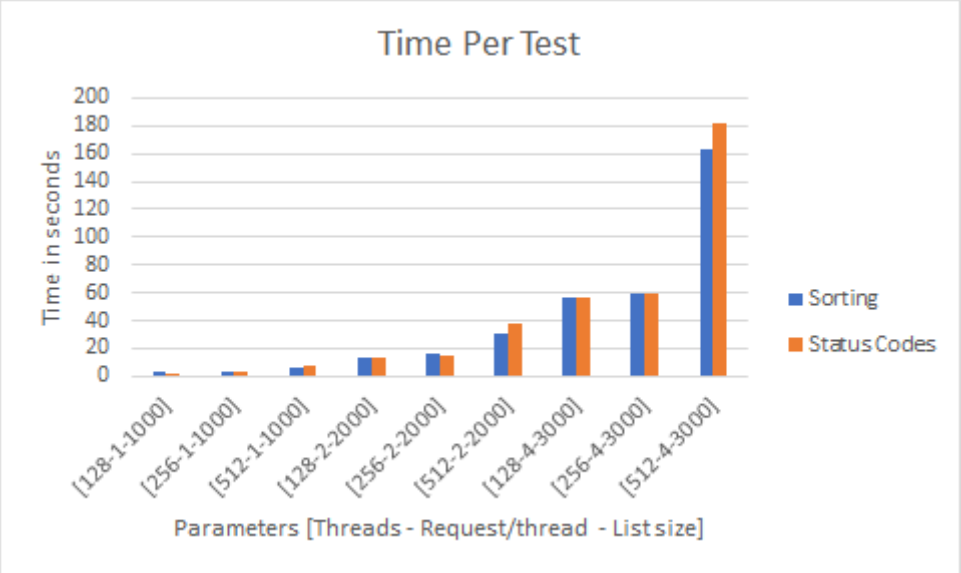
In order to test the performance of the system stress tests were performed. The tests performed were iterations of requests sent from pytest with different number of threads, requests per threads and sizes of lists.

The performed tests had the following arguments found in table 1.

Threads	Requests/Thread	Array size
128	1	1000
256	1	1000
512	1	1000
128	2	2000
256	2	2000
512	2	2000
128	4	3000
256	4	3000
512	4	3000

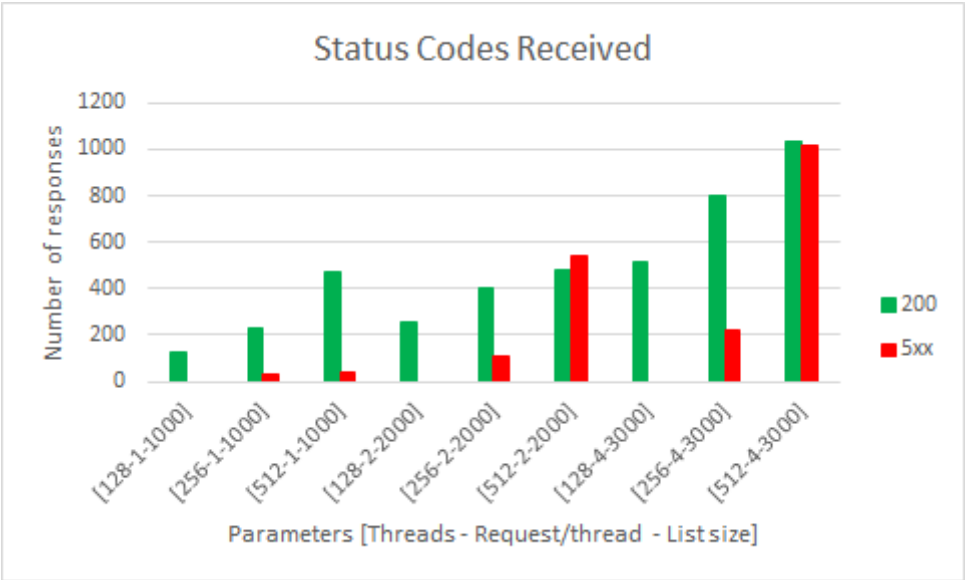
(Table 1. Parameter arguments for the tests)

As we increased the stress on the system, the time per test went up. From a couple of seconds on the first test to around three minutes for the heaviest test as displayed in diagram D1.



(Diagram D1. Time for each test with different arguments passed)

For the availability tests we can see (diagram D2) a clear trend that as long as the number of threads, ie. concurrent users are less than 200, all requests pass. But as soon as we pass 200 concurrent users the system starts to throttle (pic. 4) and returns status code 500. Thus, as long as you are in your range of reserved concurrency, the scaling works as expected[6].



(Diagram D2. Status codes received with different parameters passed)

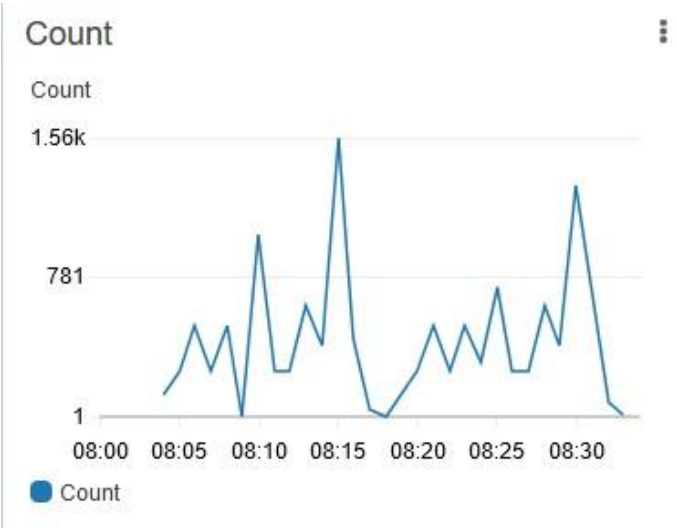
[6] https://docs.aws.amazon.com/lambda/latest/operatorguide/reserved-concurrency.html

The result observing the scalability of the Lambda can be seen in picture 1, 2, 3 and 4.
In picture 1 we see the number of concurrent executions goes up and down in correspondence to the metrics of requests in picture 2. Keeping in mind that between each finished parameter test there is a 60 second delay before the next one begins, this was to ensure easier differentiation between each parameter test on CloudWatch.

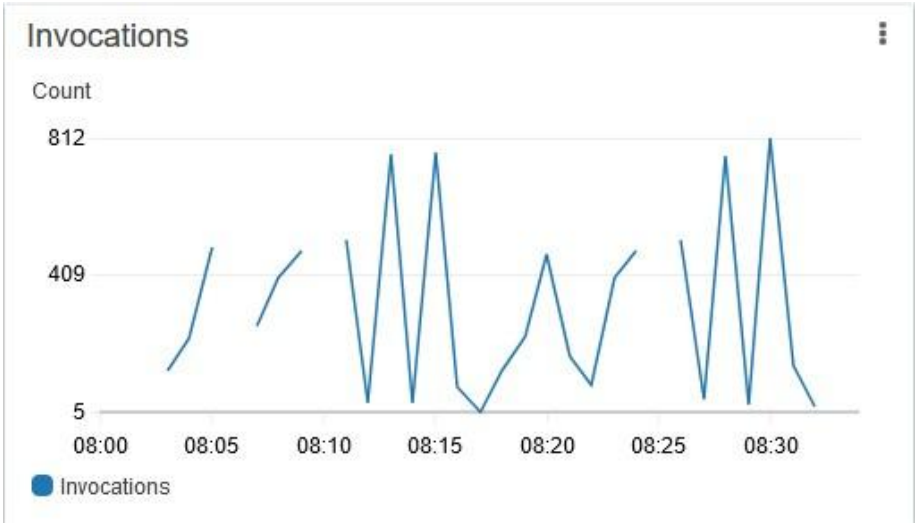
We can observe that as the number of requests at the API Gateway (pic 2) increases, so does the invocations and throttles at the Lambda seen in picture 3 and 4 respectively.
By analysing all the metrics, we can see that once the concurrent executions reach its maximum capacity, the throttling increases in correspondence to the increasing number of requests. This shows that the scaling of our necessary Lambda functionality has been successful, but only to the degree which we allow it.

This is especially observable at the 08:15 and 08:30 mark, when the maximum 512 threads are sending 4 requests each. In such a scenario, the current Lambda service would not be able to handle all requests, and further action would be necessary to increase the scalability and availability of the overall system. But as seen right after the 08:10 mark, the Lambda service holds up since it stays within the reserved concurrency.

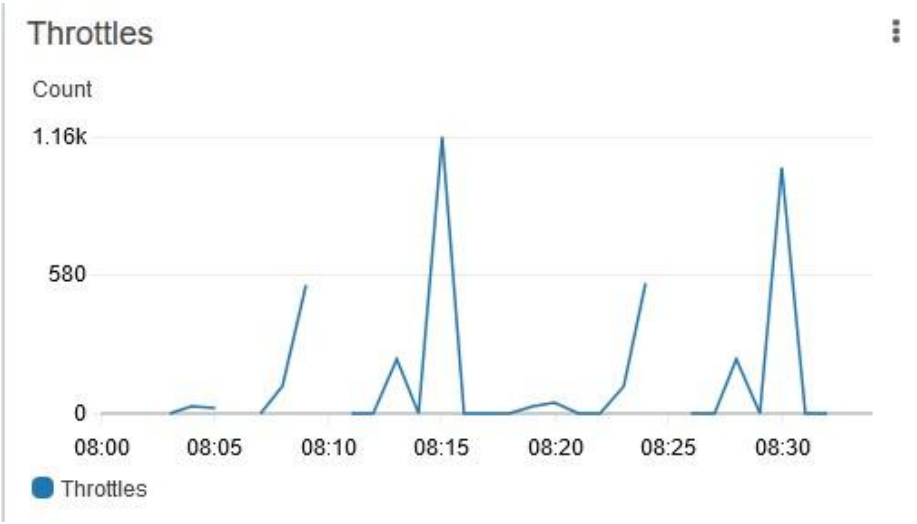
(pic 1. The number of concurrent executions from the two tests)



(pic 2. The number of requests at the API Gateway from the two tests)



(pic 3. The number of invocations form the two tests)



(pic 4. The number of throttles from the two tests)

Appendix 1

```
import json

def lambda_handler(event, context):
    arr = json.loads(event['body'])
    n = len(arr)

    # Traverse through all array elements
    for i in range(n-1):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return {
        'statusCode': 200,
        'body': json.dumps(arr)
    }
```

Appendix 2

```
import collections, pytest, helpers
url = "xxx"
parameters = [
    (128,1,1000),
    (256,1,1000),
    (512,1,1000),
    (128,2,2000),
    (256,2,2000),
    (512,2,2000),
    (128,4,3000),
    (256,4,3000),
    (512,4,3000),
]

@pytest.mark.parametrize("no_threads, no_req, array_size", parameters)
def test_concurrent_requests_sorting(no_threads, no_req, array_size):
    global url
    results = helpers.perform_web_requests(no_req, no_threads, array_size, url)
    correctly_sorted = True

    for result in results:
        if correctly_sorted == False:
            break
        for arr, status_code in zip(result['arr'], result['status_code']):
            if status_code != 200:
                continue
            correctly_sorted = arr == sorted(arr)
            if correctly_sorted == False:
                break
    assert correctly_sorted == True

@pytest.mark.parametrize("no_threads, no_req, array_size", parameters)
def test_concurrent_requests_status(no_threads, no_req, array_size):
    global url
    results = helpers.perform_web_requests(no_req, no_threads, array_size, url)

    status_codes = []

    counter = 0
    for result in results:
        for status_code in result['status_code']:
            status_codes.append(status_code)
            if status_code == 200:
                counter += 1
    assert counter == no_req*no_threads
```

Appendix 3

```
import time, json, requests, random
from threading import Thread

class Worker(Thread):
    def __init__(self, no_req_to_send, arr, url):
        Thread.__init__(self)
        self.no_req_to_send = no_req_to_send
        self.arr = arr
        self.url = url
        self.result = {
            'status_code': [],
            'arr': []
        }

    def run(self):
        for i in range(self.no_req_to_send):
            try:
                response = requests.post(self.url, data=json.dumps(self.arr))
            except Exception as e:
                continue
            self.result["status_code"].append(response.status_code)
            self.result["arr"].append(json.loads(response.content))

def perform_web_requests(no_req_to_send, no_threads, arr_size, url):
    threads = []
    for _ in range(no_threads):
        random.seed(0)
        # Create the randomised array
        arr = [random.randint(0,100000) for i in range(arr_size)]
        # Create the object
        thread = Worker(no_req_to_send, arr, url)
        threads.append(thread)

    # Start the threads
    for thread in threads:
        thread.start()
    # Join threads to wait till they finished
    for thread in threads:
        thread.join()
    # Combine results from all threads
    results = []
    for thread in threads:
        results.append(thread.result)
    time.sleep(60)
    return results
```