Sizing Servers Lab
www.sizingservers.be

Example report

SIZING SERVERS LAB
2017-04-06

Sizing Servers Lab (Howest) Graaf Karel de Goedelaan 5, 8500 Kortrijk, Belgium
+32 (0) 56 24 12 82 info@sizingservers.be www.sizingservers.be

Table of contents

Table of figures 1
Charts 1
Screenshots 2
Positioning 3
Goal 3
Sizing Servers Lab 3
Test setup 4
Terminology 4
Methodology 5
Scenario(s) and user actions 5
AWS setups 6
Summary 7
Scenario 1: Disregard requests from disconnected device 7
Scenario 2: Create a virtual key 7
Scenario 4: Get keys, create a key, post synthesis and unlock doors 8
Scenario 5: Post synthesis 8
Conclusions 10
Detailed report 11
Production monitoring 11
Scenario 1: Disregard requests from disconnected device 13
AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS) 13
AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS) 17
AWS setup 3: ELB --> 2x c4.large + RDS Aurora 18
Scenario 2: Create a virtual key 24
AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS) 24
AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS) 30
AWS setup 3: ELB --> 2x c4.large + RDS Aurora 36
Scenario 4: Get keys, create a key, post synthesis and unlock doors 42
AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS) 42
AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS) 45

AWS setup 3: ELB --> 2x c4.large + RDS Aurora 51
Scenario 5: Post synthesis 59
AWS setup 3: ELB --> 2x c4.large + RDS Aurora 59

Table of figures

Charts

Chart 1 Calls / hour - Production monitoring 11
Chart 2 - Response time vs throughput - scenario 1, AWS setup 1 13
Chart 3 - Errors vs throughput - scenario 1, AWS setup 1 14
Chart 4 - RDS CPU - scenario 1, AWS setup 1 15
Chart 5 - RDS Write IOPS - scenario 1, AWS setup 1 16
Chart 6 - Response time vs throughput - scenario 1, AWS setup 2 17
Chart 7 - Response time vs throughput - scenario 1, AWS setup 3 18
Chart 8 - Response time vs throughput (without the first - API access device - call - scenario 1, AWS setup 3 19
Chart 9- EC2 CPU - scenario 1, AWS setup 3 20
Chart 10 - EC2 Network in - scenario 1, AWS setup 3 20
Chart 11 - EC2 Network out - scenario 1, AWS setup 3 21
Chart 12 - RDS CPU - scenario 1, AWS setup 3 22
Chart 13 - RDS DML throughput - scenario 1, AWS setup 3 23
Chart 14 - Response time vs throughput - scenario 2, AWS setup 1 24
Chart 15 - 95th %iles response times - scenario 2, AWS setup 1 25
Chart 16 - RDS CPU - scenario 2, AWS setup 1 26
Chart 17 - RDS Write IOPS - scenario 2, AWS setup 1 27
Chart 18 - EC2 CPU - scenario 2, AWS setup 1 28
Chart 19 - EC2 Network in - scenario 2, AWS setup 1 28
Chart 20 - EC2 Network out - scenario 2, AWS setup 1 29
Chart 21 - Response time vs throughput - scenario 2, AWS setup 2 30
Chart 22 - 95%iles response times - scenario 2, AWS setup 2 31
Chart 23 - RDS CPU - scenario 2, AWS setup 2 32
Chart 24 - RDS Write IOPS - scenario 2, AWS setup 2 33
Chart 25 - EC2 CPU - scenario 2, AWS setup 2 34
Chart 26 - EC2 Network in - scenario 2, AWS setup 2 34
Chart 27 - EC2 Network out - scenario 2, AWS setup 2 35
Chart 28 - EC2 CPU- scenario 2, AWS setup 3 37
Chart 29 - EC2 Network in - scenario 2, AWS setup 3 38
Chart 30 - EC2 Network out - scenario 2, AWS setup 3 39
Chart 31 - RDS CPU - scenario 2, AWS setup 3 40
Chart 32 - RDS DML throughput - scenario 2, AWS setup 3 41
Chart 33 - Response time vs throughput - scenario 4, AWS setup 1 42
Chart 34 - RDS CPU - scenario 4, AWS setup 1 43
Chart 35 - RDS Write IOPS- scenario 4, AWS setup 1 44
Chart 36 - Response times vs throughput - scenario 4, AWS setup 2 45
Chart 37 - Errors vs throughput - scenario 4, AWS setup 2 46
Chart 38 - RDS CPU - scenario 4, AWS setup 2 47
Chart 39 - RDS Write IOPS - scenario 4, AWS setup 2 48

Chart 40 - Response time vs throughput - scenario 4, AWS setup 3 51
Chart 41 - 95th %iles response times - scenario 4, AWS setup 3 52
Chart 42 - Errors vs throughput - scenario 4, AWS setup 3 53
Chart 43 - EC2 CPU - scenario 4, AWS setup 3 54
Chart 44 - EC2 Network in - scenario 4, AWS setup 3 55
Chart 45 - EC2 Network out - scenario 4, AWS setup 3 56
Chart 46 - RDS CPU - scenario 4, AWS setup 3 57
Chart 47 - RDS DML throughput - scenario 4, AWS setup 3 58
Chart 48 - Response times vs user actions - scenario 5, AWS setup 3 59
Chart 49 - EC2 CPU - scenario 5, AWS setup 3 60
Chart 50 - EC2 Network in - scenario 5, AWS setup 3 61
Chart 51 - EC2 Network out - scenario 5, AWS setup 3 62
Chart 52 - RDS CPU - scenario 5, AWS setup 3 63
Chart 53 - RDS DML throughput - scenario 5, AWS setup 3 64

Screenshots

Screenshot 1 Number of cars and phones at the busiest hour - Production monitoring 12
Screenshot 2 - Results - scenario 2, AWS setup 3 36
Screenshot 3 - Response times at concurrency 50 - scenario 4, AWS setup 1 42
Screenshot 4 - Response times at concurrency 50 - scenario 4, AWS setup 2 45
Screenshot 5 - Slow query - scenario 4, AWS setup 2 49

Positioning

Goal

This is the first final report about the performance of the clients' API. It includes the results from Intermediate report 1 (2017-02-21).
These test results provide insight in how well the API performs currently. The API was tested on three AWS setups, set-up by another party.

Sizing Servers Lab

Sizing Servers Lab is an independent research lab at university-college Howest. The lab was founded in 2003 and is since 2007 recognized by the Belgian government as an official research lab.

Our purpose is to offer best practices and directly applicable results to companies in Flanders, Belgium and internationally by researching cutting-edge server technologies. To accomplish this goal, we combine the three strongest factors of our lab:

A profound knowledge of hardware and software interfaces and optimization technics: virtualization, power management, etc.

Feedback of professionals in the field by publishing our results on popular IT websites (Anandtech.com, ZDNet, etc.)

A close cooperation with the engineers of market leaders as Intel, AMD, VMware, Facebook, etc.

Test setup

Terminology

vApus is an in-house developed framework to stress test and monitor different applications. The power of vApus lies in its flexibility to adjust the framework to the required platform.

A **User action** is a sequence of requests/calls sent by vApus, initiated by a **simulated user** / device (car, phone, …). An action can be defined as "all needed requests to unlock the doors of a car" or "post a synthesis". An action is complete if all requests received a response. Where after, the virtual user will proceed with the next action. In this test set-up we can define an action at best with a **page or a "click"** on the website.

Concurrency / concurrent users determines the number of user actions executed per second. While testing a website the concurrency can be best interpreted as **"number of pages per second"**. In the Sizing Servers Lab tests we linearly increase the concurrency. By doing this we gradually increase the stress on the server(s) (ramp-up testing).

The number of **runs** defines the number of times a test with a certain concurrency is (re)done, this to smoothen out average concurrency results.

Response time translates itself best as "user experience". The response time determines the average time it took for a simulated user to receive a full page. This is measured from the moment the request is sent until the last byte is received. **A lower response time is always better.**

Throughput gives a general idea of the server(s) performance, but doesn't translate itself directly as user experience. The throughput determines the quantity of responses per second a server can process and send. **A higher throughput is always better.**

Think time can be defined as the average time between actions: the average time a user stays on the page before clicking through to next page. Usually we consider an average think time of 1 second (900-1100ms), consequently we speak about "pages per second".

Methodology

We use vApus to create workflows consisting of typical user actions. In our stress tests we increase the number of users ("concurrency") linearly to determine how a server-application performs while under increased stress.

Each *simulated user* clicks through to the next page with an average *think time* of 1 second (we agreed 500-1500ms). This means *the concurrency* can be roughly translated as the number of user **actions per second** the client requested.

Scenario(s) and user actions

Seven scenarios (use cases) where determined by the client. We focused on four of them: scenario 1, 2, 4 and 5.

1 - Disregard requests from disconnected device

- Disconnect (x 5 or x 10) *
- API access device (x 5 or x 10)
- SDK access device (x 5 or x 10)

* Each simulated user executes all user actions 10 times. The correct order of those user actions is respected.

2 - Create a virtual key

- API - Create a virtual key (x 10)

3 - Disregard requests from disconnected device

- API - Create a virtual key with 100 single action tokens (x 10)

4 - Get keys, create a key post synthesis and unlock doors

- API - Get keys, create a key, post synthesis and unlock doors (x 5)

5 - Post synthesis

- API - Post synthesis (x 20)

6 - Get a vehicle synthesis

- API - Get a vehicle synthesis (x 30)

7 - Get keys, enable a key post synthesis and unlock doors

AWS setups

1 - 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS)

2 - ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS)

3 - ELB --> 2x c4.large + RDS Aurora (Fixed indexes, disabled query cache)

Summary

Scenario 1: Disregard requests from disconnected device

AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS)
Maximum throughput of 16 responses / s at concurrency 20. At concurrency 50 accessing the device (/rest/api/v3/authenticate/accessDevice) via the API takes almost 10 seconds.

RDS was the bottleneck in this case: CPU and IOPS.

AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS)
The maximum throughput was 12 responses / s. The second try it was 10. No bottlenecks were detected in the monitoring. The problem was that devices were not parameterized in this scenario. This was addressed for the next AWS setup.

AWS setup 3: ELB --> 2x c4.large + RDS Aurora
Throughput reaches 144 responses / s at concurrency 185. **This is 10 times better than previously**.

In this scenario, the setup can handle 185 unique phones/cars (2 of 3 calls → cars, 1 of 3 calls → phones. accessing the SDK/API every second).

Which calls coming from cars and which calls coming from phones are deducted from *Calls per day.txt* (in the zip file), containing information provided by the client.

Please do note that vehicles to authenticate where parameterized. This was not the case previously. Furthermore, as requested by the client, the disconnect call was moved to the end. So results cannot be entirely matched.

There were no particular hardware bottlenecks. RDS CPU usage only gets over 60%. There is a fake push-client in place to enable this test. Maybe this causes a delay.

Scenario 2: Create a virtual key

To simulate authentication the server-side waits between 150 and 4500 milliseconds before returning the response. This means that throughput is not a helpful metric in this case, since we match throughput (responses / second) to concurrency (+- requests / second).

AWS setup 1
At concurrency 200 the application is saturated. From there on the maximum response times get higher than 4 seconds. RDS is at its Write IOPS limit. No other bottlenecks.

AWS setup 2
Around concurrency 265 the application is saturated. Network traffic was rather high.

AWS setup 3
We only tested concurrency 300, because there was no more time left. Creating a virtual key at concurrency 300 is not a problem. Max response times are slightly above 4.5 seconds.

In this scenario, the setup can handle 300 unique phones (accessing the SDK every second).

There were some errors. All the results of a key Id that could not be fetched from a response. Meaning that the expected response was not returned. See the detailed report.

No hardware bottlenecks were detected. We maybe could get good results with even higher concurrencies.

Scenario 4: Get keys, create a key, post synthesis and unlock doors

To simulate authentication the server-side waits between 250 and 4000 milliseconds before returning the response on two calls.

Furthermore, is it necessary to get all keys in order to do this scenario correctly?

AWS setup 1
At around concurrency 30 the application is saturated. Getting all key takes a long time (7.3 seconds at concurrency 50).

The RDS CPU usage was maxed out.

AWS setup 2
At concurrency 35 the application is saturated. Getting all keys takes even longer (7.7 seconds at concurrency 50).

The RDS CPU usage was maxed out.

The problem was a very slow query for getting the keys. This problem was resolved for the next AWS setup.

AWS setup 3
We see that the application is much faster now with this scenario. Mainly fixing the 'get all keys SQL request' is responsible for this.

Getting keys takes now 83ms on average instead of 7700ms before. The application meets its saturation point at concurrency 60-65 , making it twice as fast as before.

In this scenario, the setup can handle 60-65 unique phones (accessing the SDK every second).

There were some errors. Al the results of a key Id that could not be fetched from a response. Meaning that the expected response was not returned.

The RDS CPU usage poses a bottleneck.

Scenario 5: Post synthesis

This scenario was not tested on AWS setup 1 and 2.

AWS setup 3

Posting a synthesis reaches its plateau at concurrency 95-100, with a throughput of 65 responses per second. On average, it takes around 460ms at concurrency 95 to get a response back.

In this scenario, the setup can handle 95-100 unique cars (accessing the CSM every second).

No bottlenecks where seen in the monitors or the slow query log. We could try testing with higher concurrencies.

Conclusions

The clients' API came a long way since our first testing. The efforts of the parties involved resulted in an API that is **2, and even 10 times faster** with certain scenarios on the last AWS setup compared to the first two.

In between the testing bugs got fixed and new AWS setups set into place.

In our tests **AWS setup 3: ELB --> 2x c4.large + RDS Aurora** can handle in the worst case 60-65 unique phones (accessing the SDK every second – Scenario 5*)* and 95-100 unique cars (*accessing the CSM every second*).
At best, it can handle 300+ unique phones (accessing the SDK every second – Scenario 2).

On the current (not AWS) production environment **the API-, CSM- and SDK-service combined handle 81 calls per minute (1.4 calls per second) in peak moments, in a per hour basis analysis.**

During the tests on the last AWS setup, RDS Aurora CPU posed a bottleneck or there was no immediate hardware bottleneck at all. The EC2 usage never went above 50%.

See the summary report and/or detailed report sections for more details.

There was one error during the tests: sometimes a key Id that could not be fetched from a response. Meaning that the expected response was not returned.

Detailed report

Note: All charts and information that were not put in this report can be found in the zip file.

Production monitoring

To be able to determine the load on the current production environment, and match cars and phones to the stress test results for the AWS setups an analyses for the load between March 27th and April 2nd was done based on the info provided by the client.

The clients' services run currently in production in two VMWare VMs, FYI:

MySQL VM
2 vCPUs (Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz)

API (+ CSM + SDK) VM
2 vCPUs (Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz)

Based on the API-, CSM-, SDK log and info provided by the client, Thursday March 30th was found to be the busiest day in terms of calls to the API with unique API keys.

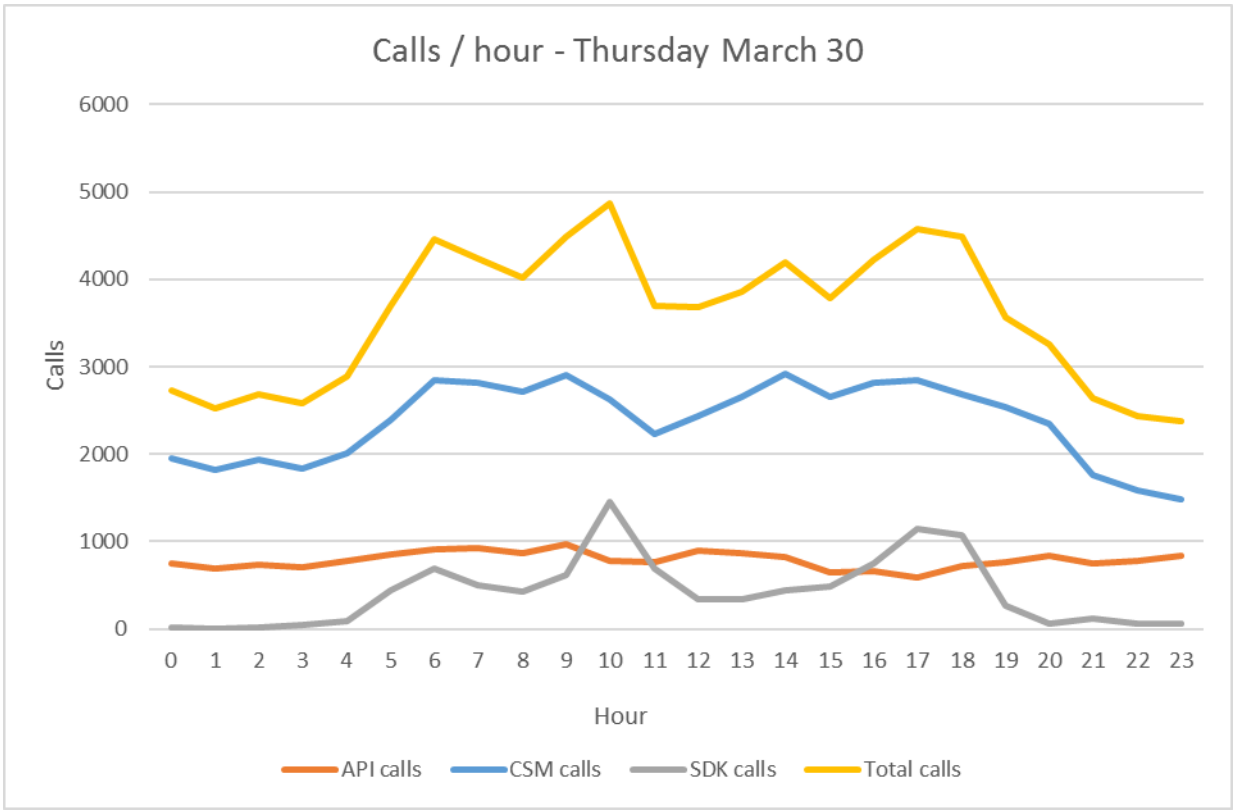

Chart 1 Calls / hour - Production monitoring

At 10AM most calls occurred: 4863 in total. Meaning that **in peak moments the three services combined handle 81 calls per minute (1.4 calls per second), in a per hour basis analysis**.

Log API unique API keys (10AM - 11 AM)	21	Use case 6, last call use case 1. cars	700	cars
Log CSM unique vins (10AM - 11 AM)	679	Use case 5: only cars (/csm call). In this log there are also a /ota calls. Does not matter too much in the comparison.		
Log SDK unique app keys (10AM - 11AM)	71	Other use cases, except 6. Phones.	71	phones

Screenshot 1 Number of cars and phones at the busiest hour - Production monitoring

At the busiest hour there are 700 unique 700 cars called the API and the CSM.
71 different phones called the SDK service.

Which calls coming from cars and which calls coming from phones are deducted from *Calls per day.txt* (in the zip file), containing information provided by the client.

The maximum load that the production environment can handle was not determined since this is not in the scope of the stress testing cases.

Scenario 1: Disregard requests from disconnected device

AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS)

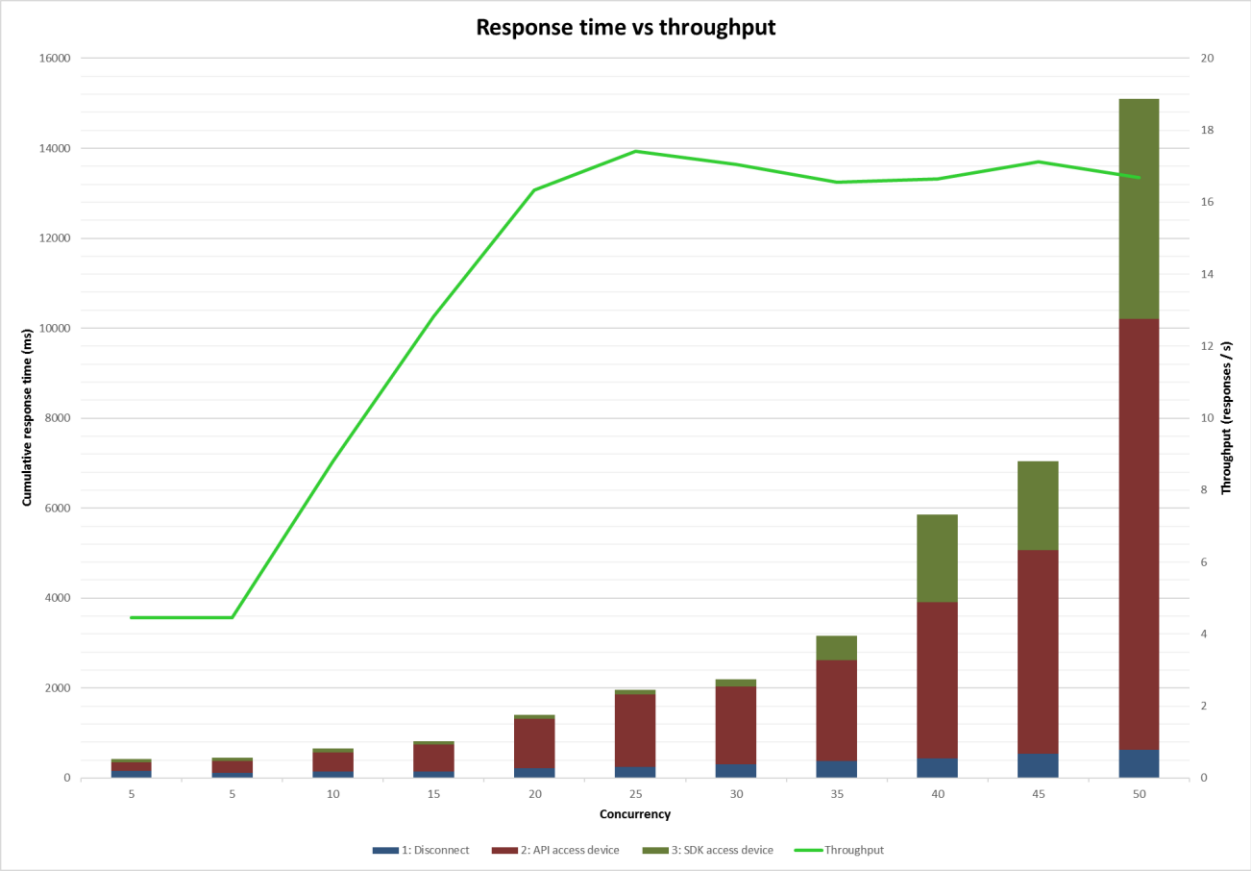

Chart 2 - Response time vs throughput - scenario 1, AWS setup 1

The throughput reaches a plateau of 16 responses / s at concurrency 20. Note how response times dramatically increase at concurrency 50: accessing the device via the API (/rest/api/v3/authenticate/accessDevice) takes on average almost 10 seconds.

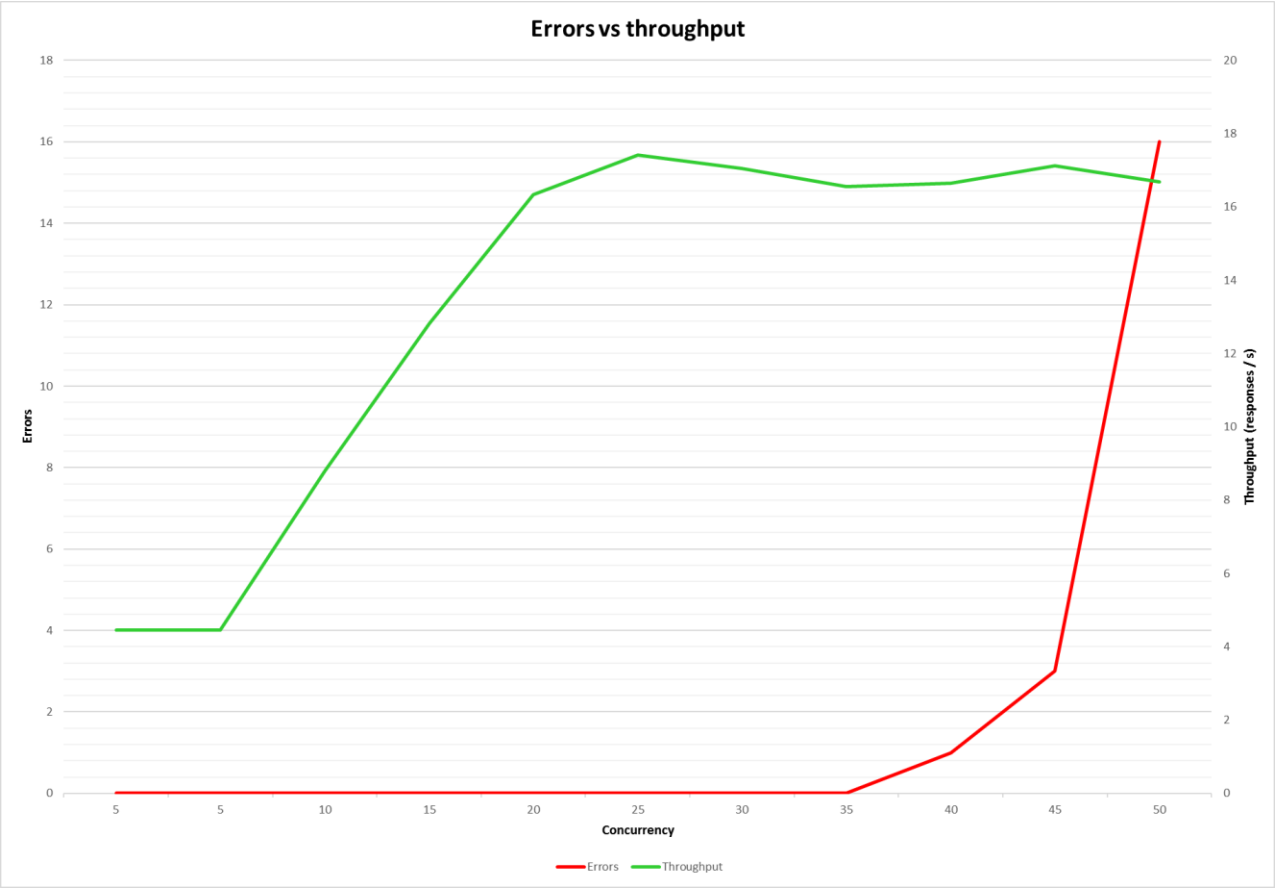

Chart 3 - Errors vs throughput - scenario 1, AWS setup 1

Starting from concurrency 35 errors start to occur. These are all 504 Gateway Timeouts.

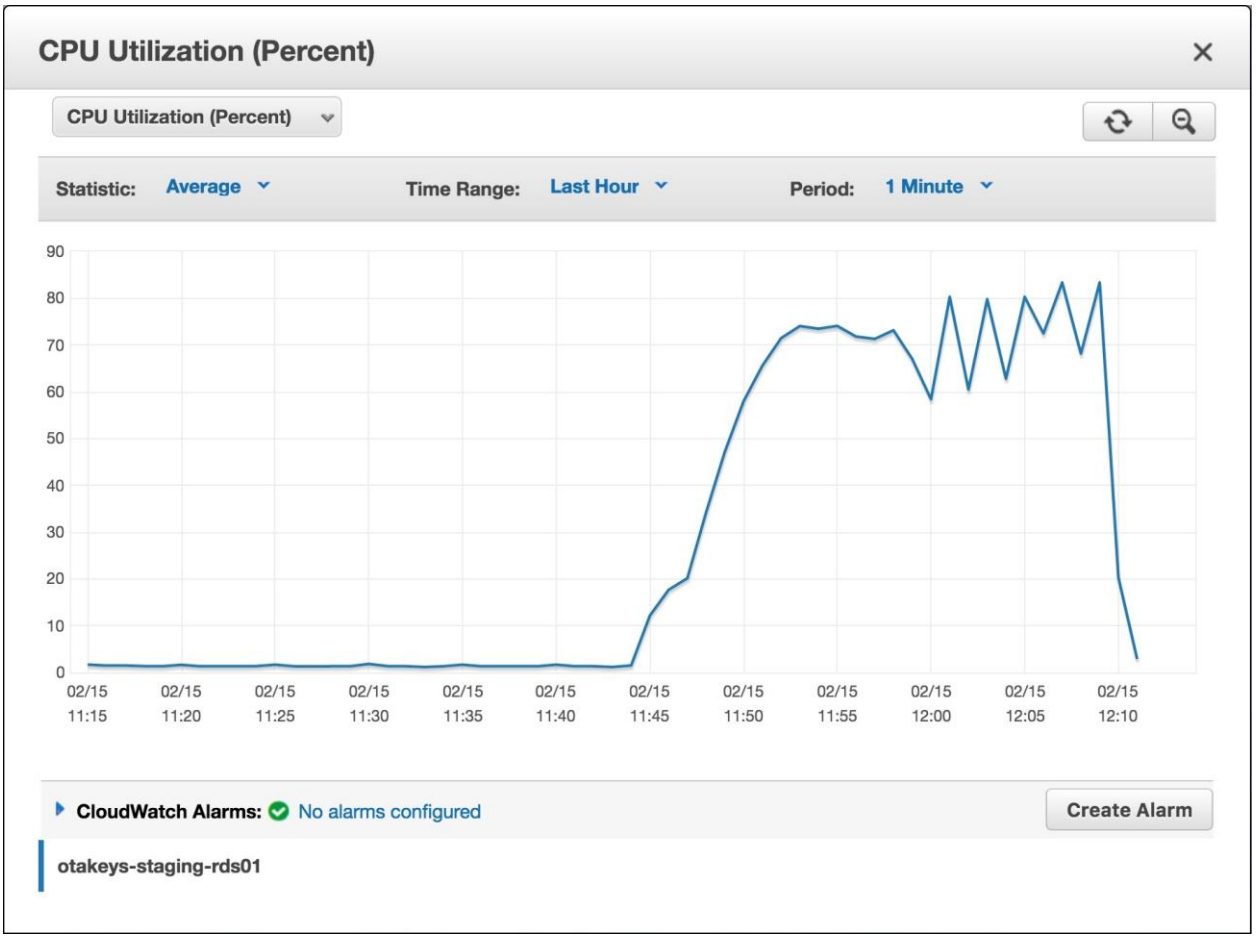

Chart 4 - RDS CPU - scenario 1, AWS setup 1

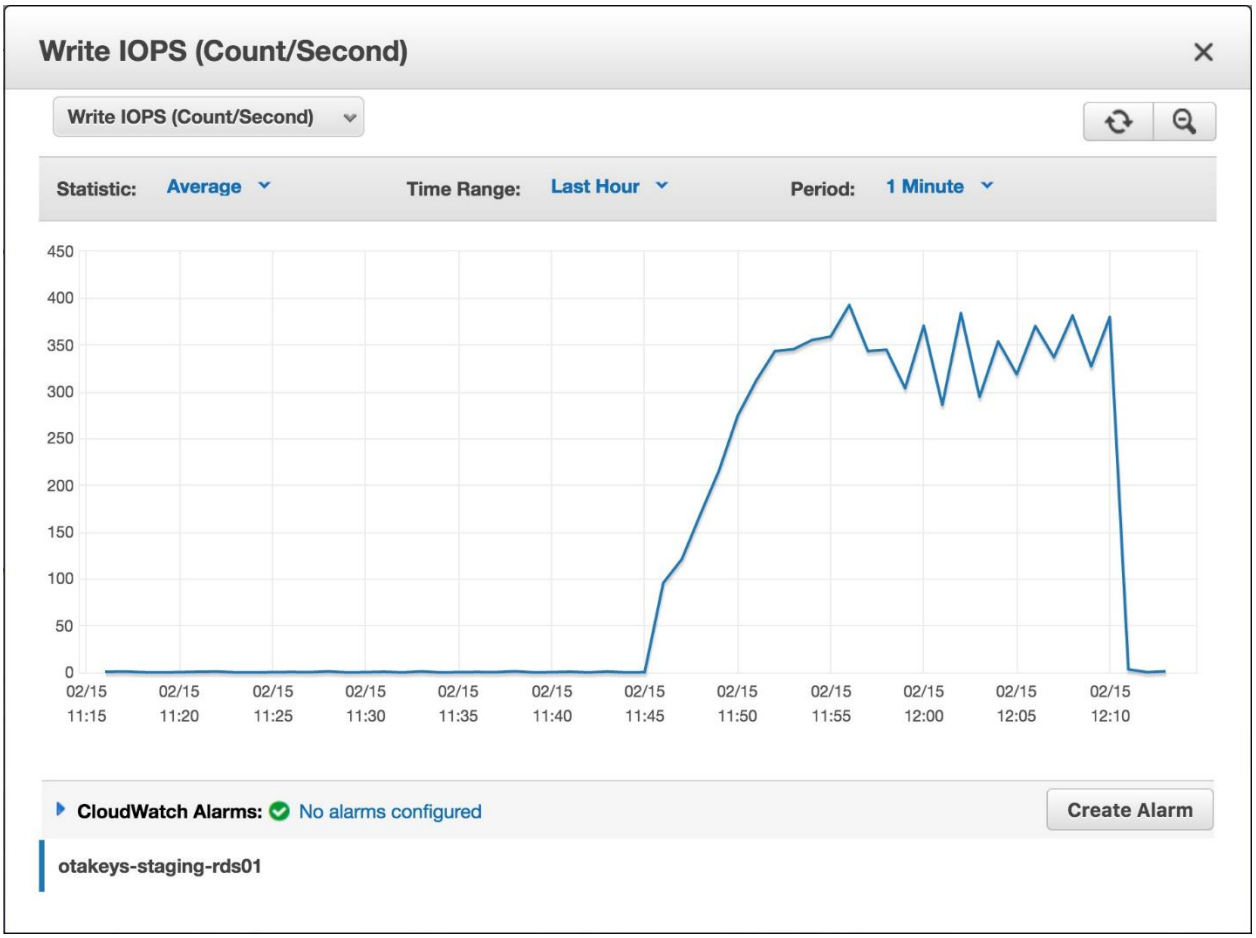

Chart 5 - RDS Write IOPS - scenario 1, AWS setup 1

RDS is clearly the bottleneck in this case.

AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS)

Use case 1 was tested 2 times this day because the throughput was lower: The first time around 12 responses per second and the second time around 10 responses per second. We expected that the throughput would be higher since the AWS setup is more beefy. No bottlenecks were detected in the monitoring.

Hereunder the charts for the second test.

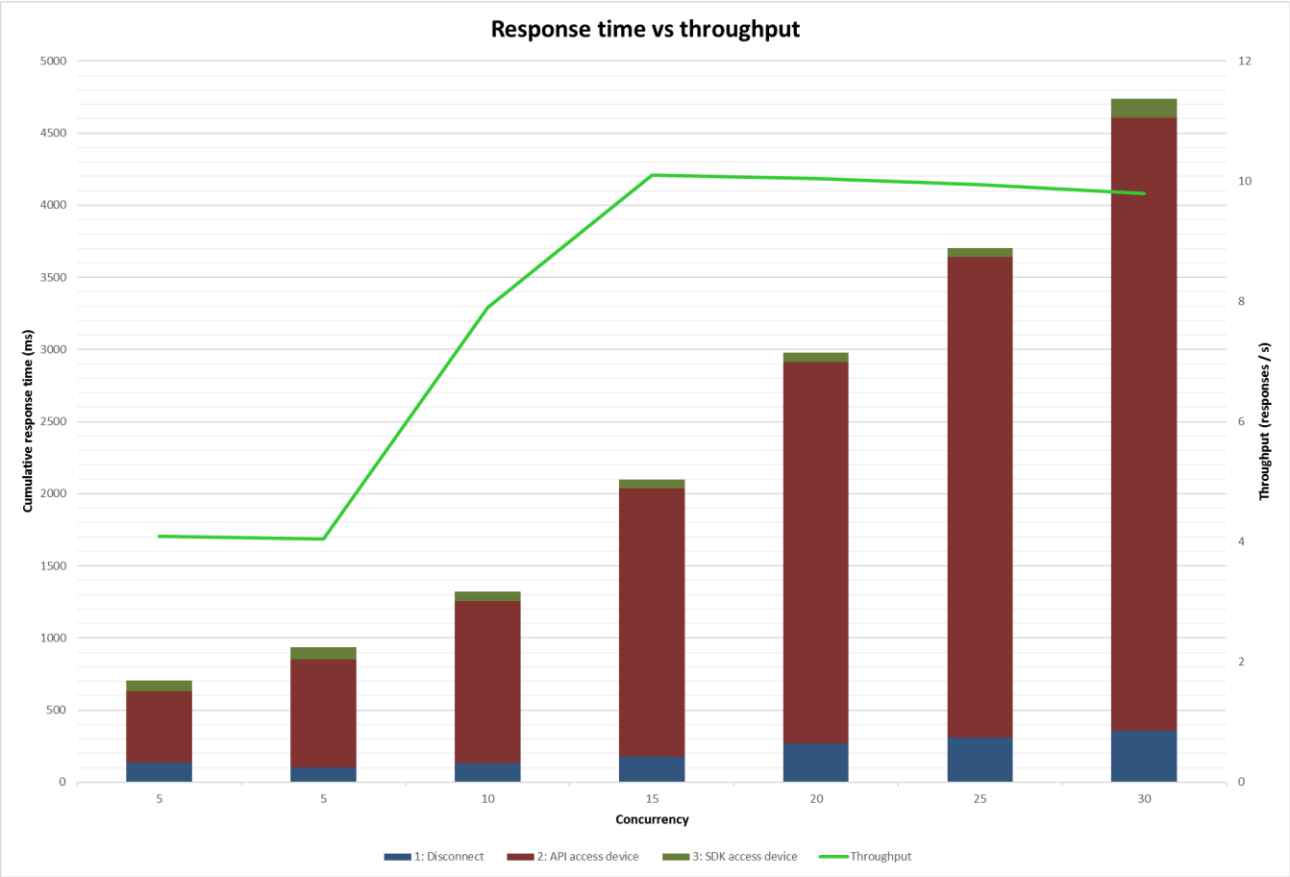

Chart 6 - Response time vs throughput - scenario 1, AWS setup 2

If you compare this chart to the previous one, you can clearly see that accessing the device via the API takes a lot longer (4 seconds instead of >2 seconds at concurrency 30). The other two calls take less time.

No errors occurred.

The problem was that devices were not parameterized in this scenario. This was addressed for the next AWS setup.

AWS setup 3: ELB --> 2x c4.large + RDS Aurora

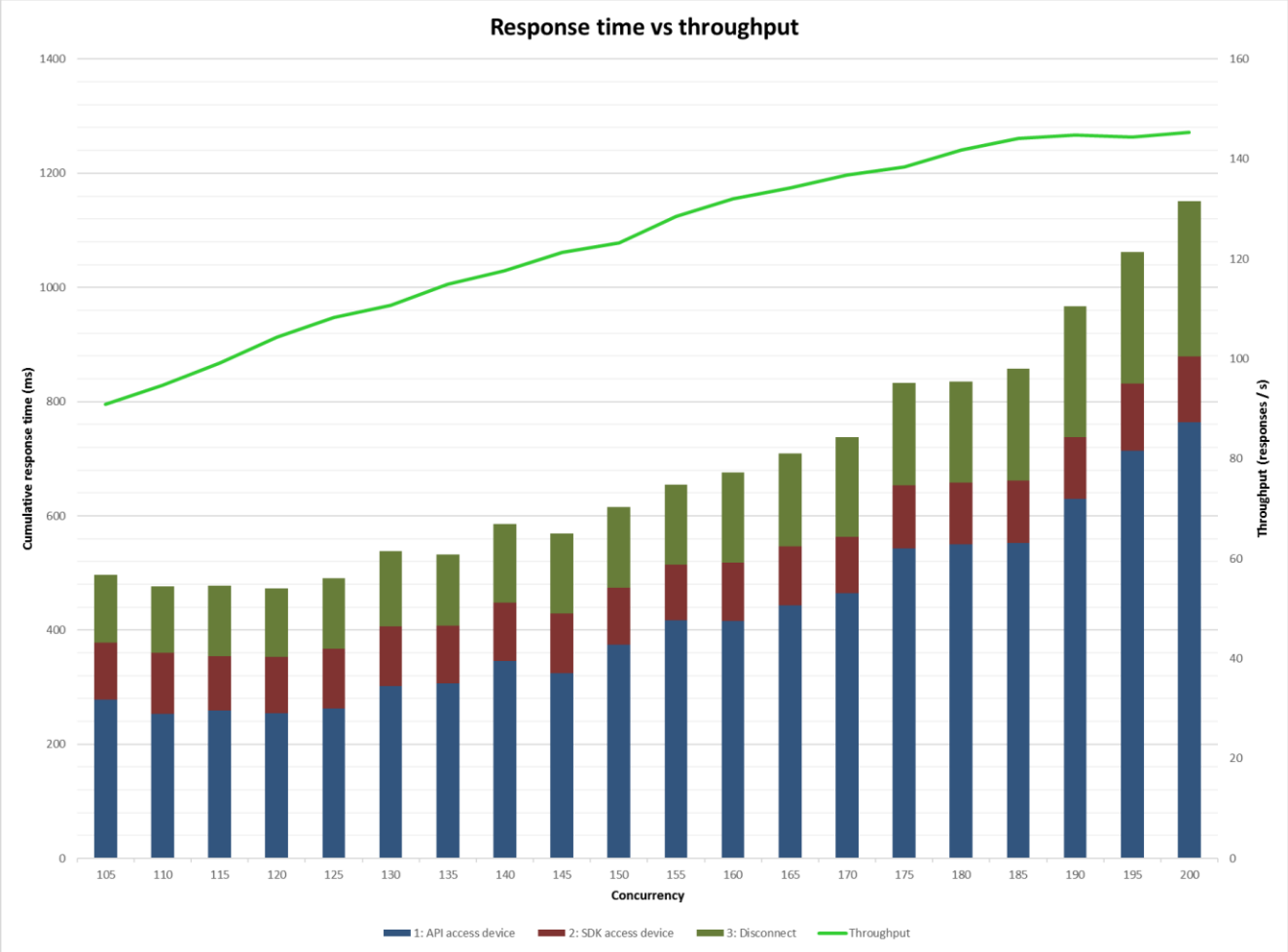

Chart 7 - Response time vs throughput - scenario 1, AWS setup 3

Throughput reaches 144 responses / s at concurrency 185. **This is 10 times better than previously**.

Please do note that vehicles where parameterized. This was not the case previously. Furthermore, as requested by the client, the disconnect call was moved to the end. So results cannot be entirely matched.

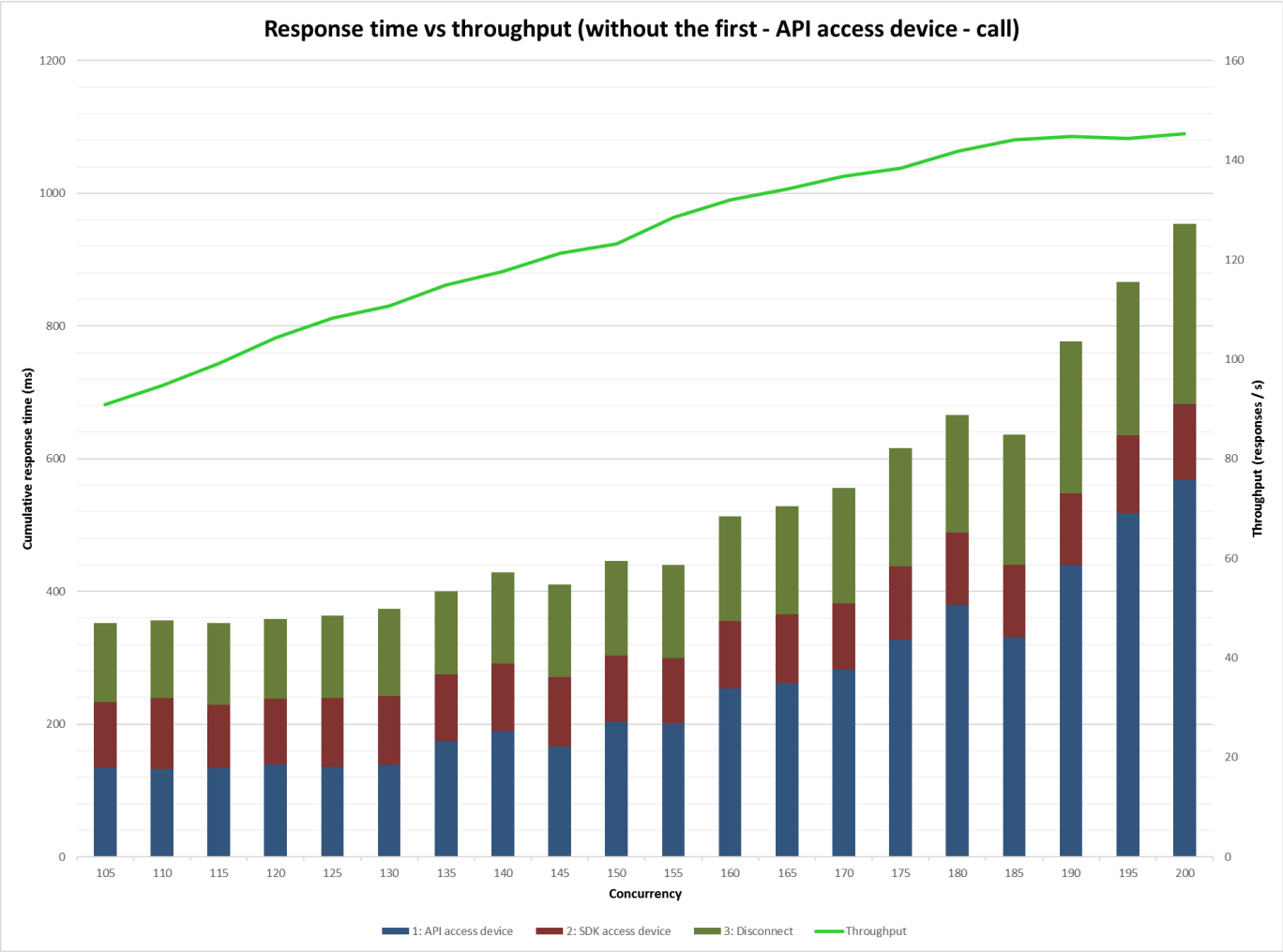

Chart 8 - Response time vs throughput (without the first - API access device - call - scenario 1, AWS setup 3

In this case the 3 calls were executed 5 times. Note that the first API access device call takes a lot longer than the other 4. Twice as long at concurrency 105.

There were no particular hardware bottlenecks. RDS CPU usage only gets over 60%. There is a fake push-client in place to enable this test. Maybe this causes a delay.

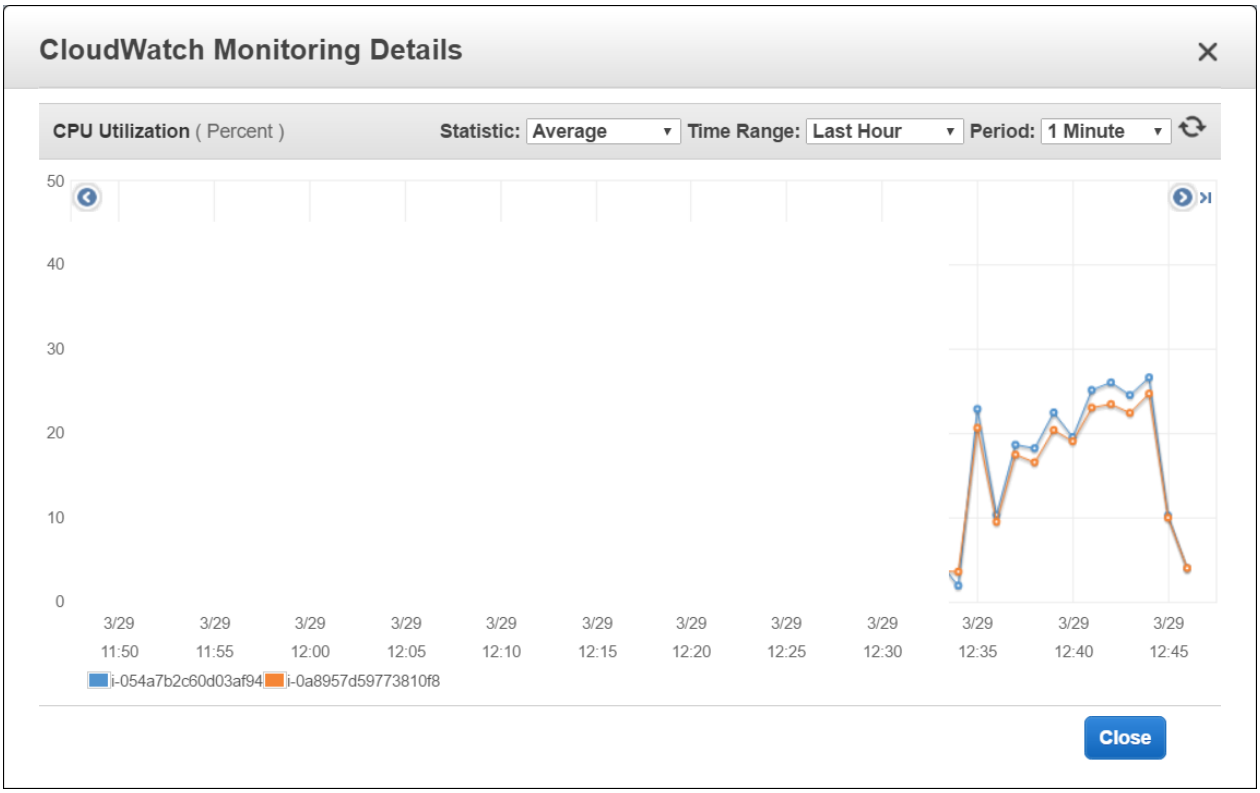

Chart 9 - EC2 CPU - scenario 1, AWS setup 3

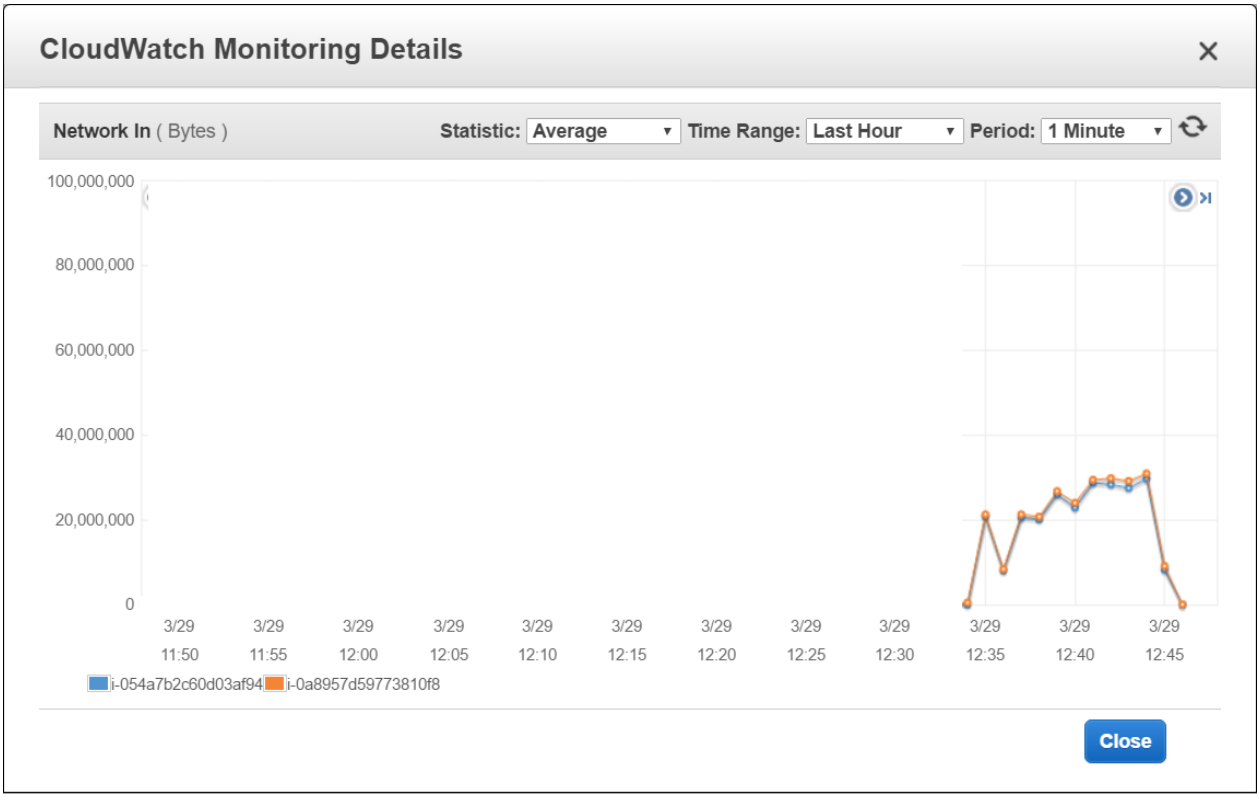

Chart 10 - EC2 Network in - scenario 1, AWS setup 3

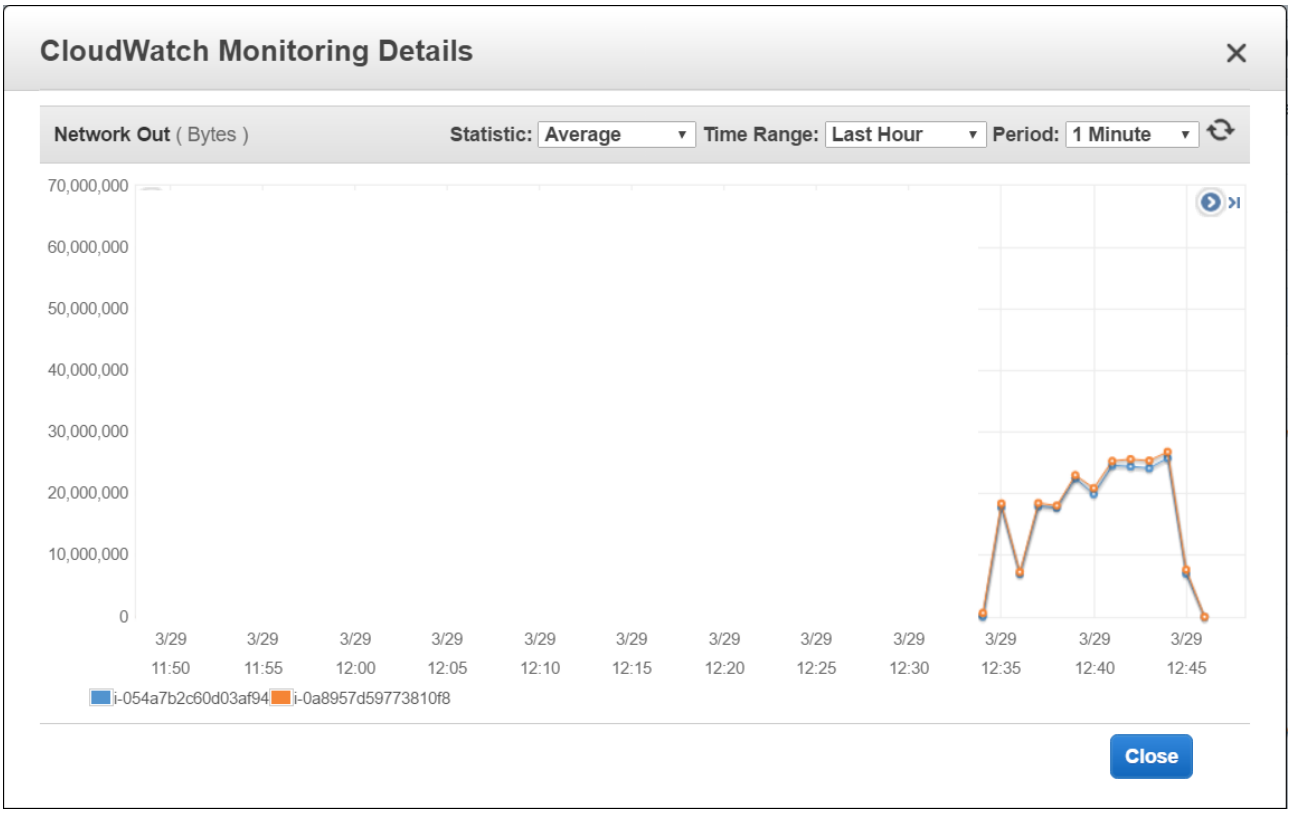

Chart 11 - EC2 Network out - scenario 1, AWS setup 3

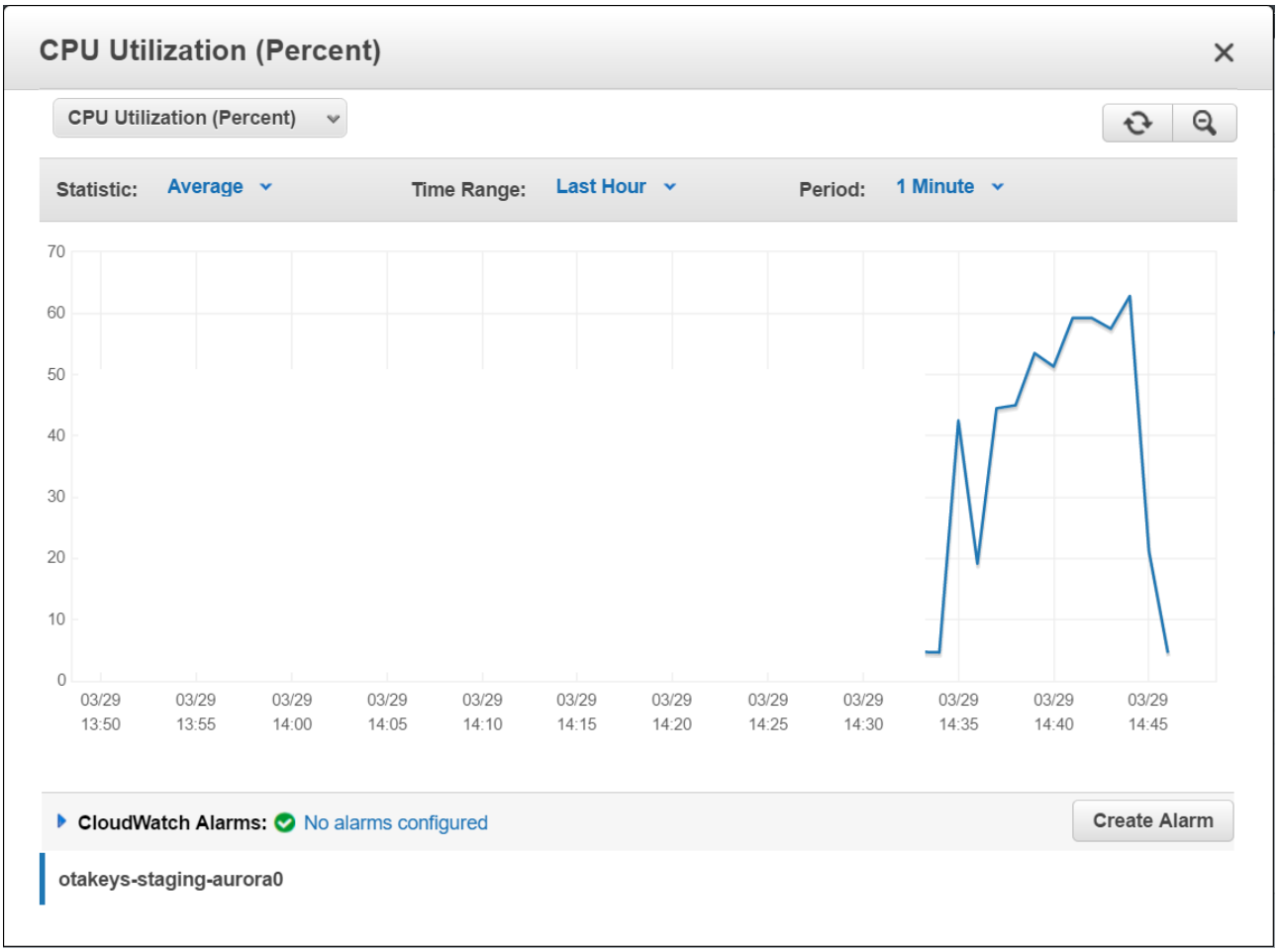

Chart 12 - RDS CPU - scenario 1, AWS setup 3

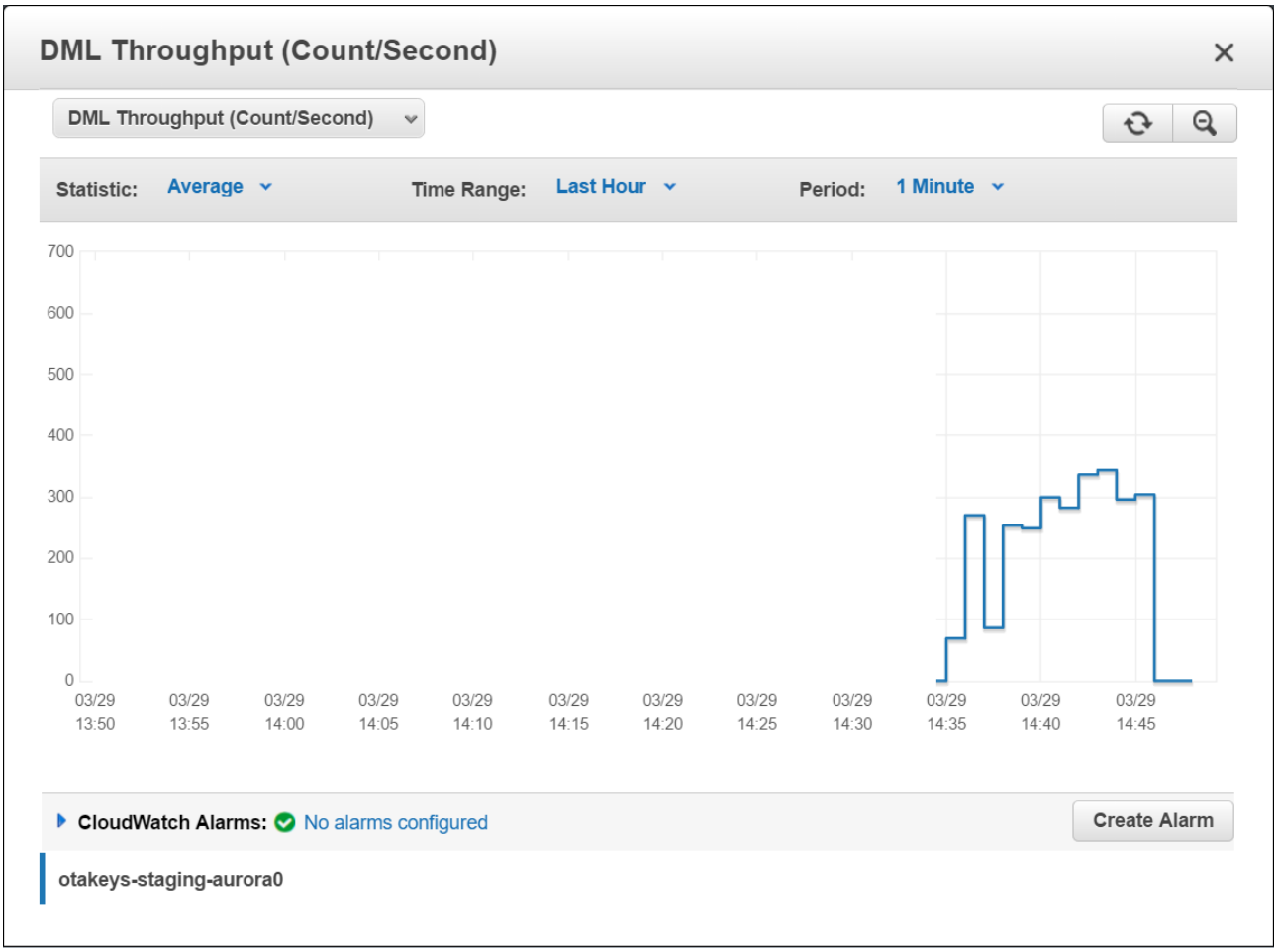

Chart 13 - RDS DML throughput - scenario 1, AWS setup 3

Scenario 2: Create a virtual key

AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS)

This one was tested 2 times. We saw a problem with the database connection pooling. After the first test the number of database connections in the application was upped from 5 to 10. In RDS they were upped 1000. The results hereunder are for the second test.

Important to note: To simulate authentication the server-side waits between 150 and 4500 milliseconds before returning the response.

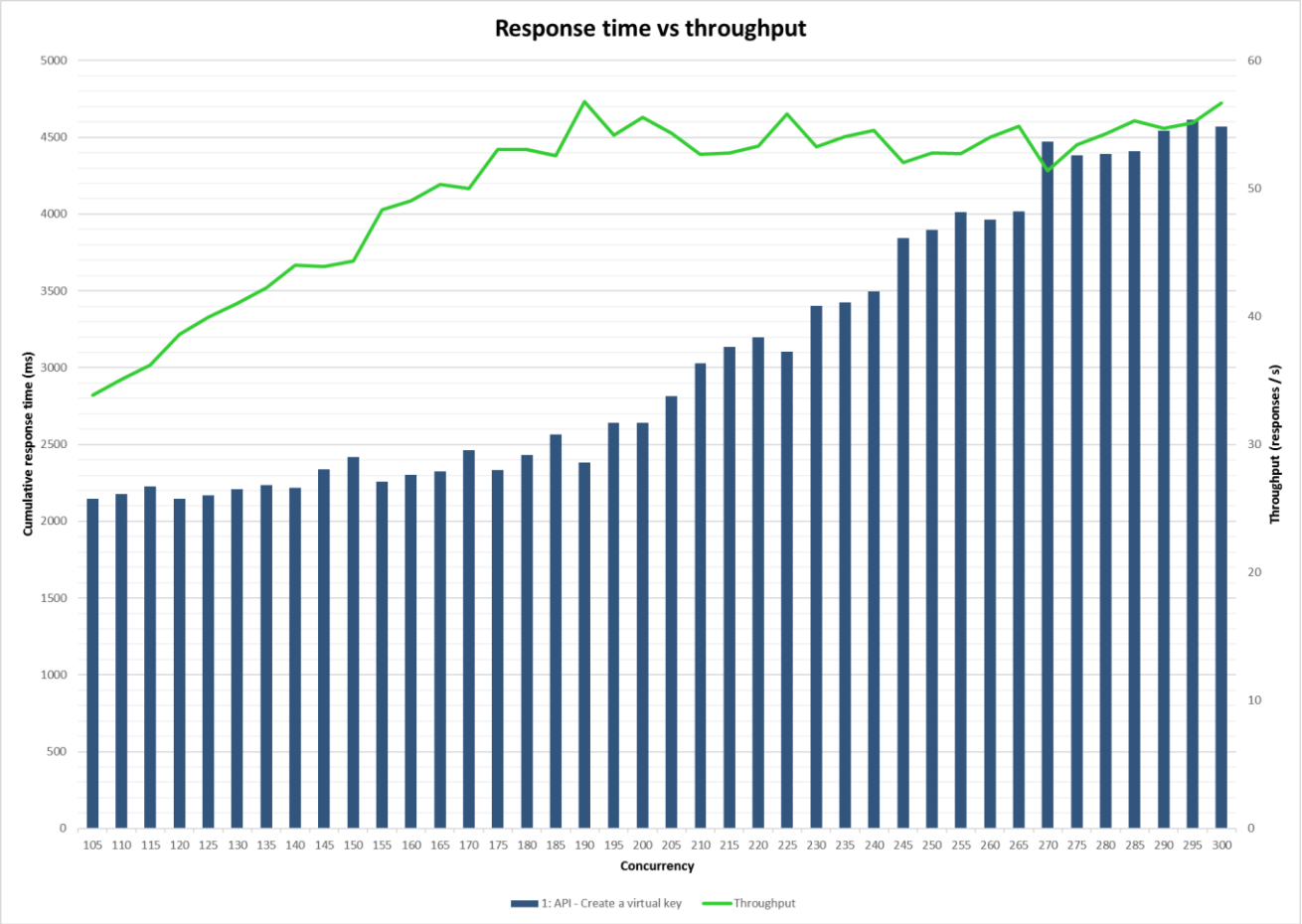

Chart 14 - Response time vs throughput - scenario 2, AWS setup 1

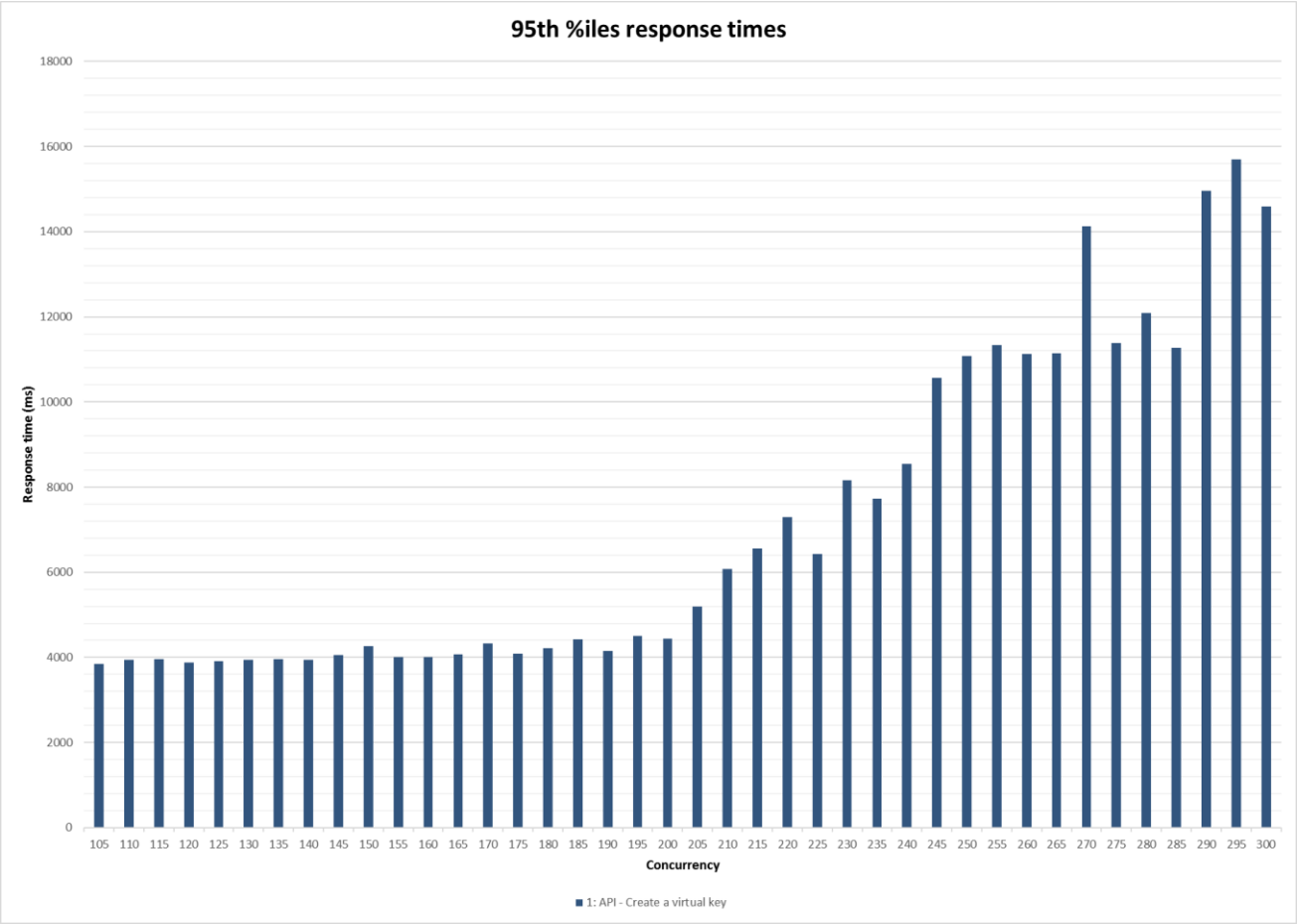

Chart 15 - 95th %iles response times - scenario 2, AWS setup 1

If maximum response times are around 4 seconds this means that the server can handle the requests without problems, in this specific case.

Around concurrency 200 the application is saturated.

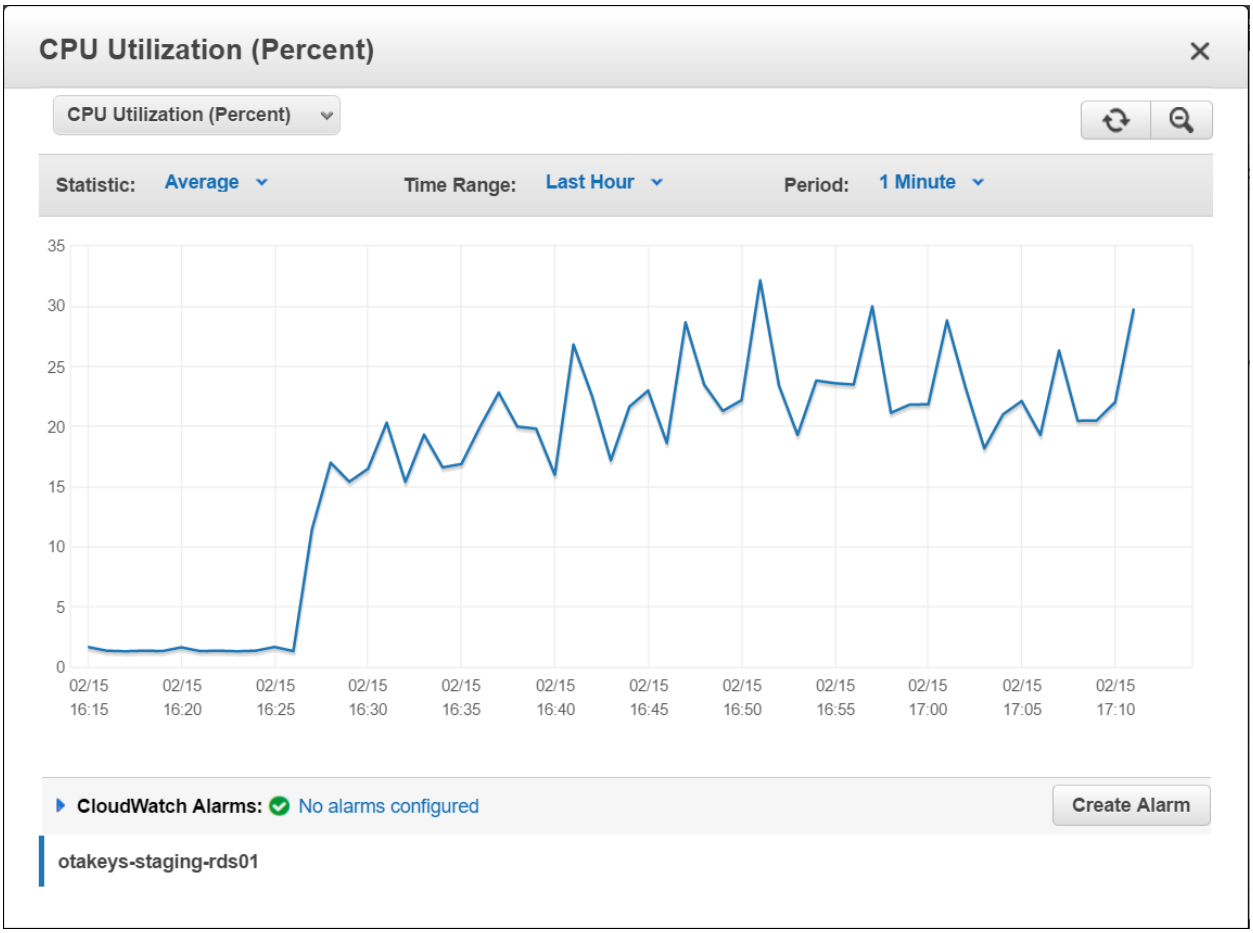

Chart 16 - RDS CPU - scenario 2, AWS setup 1

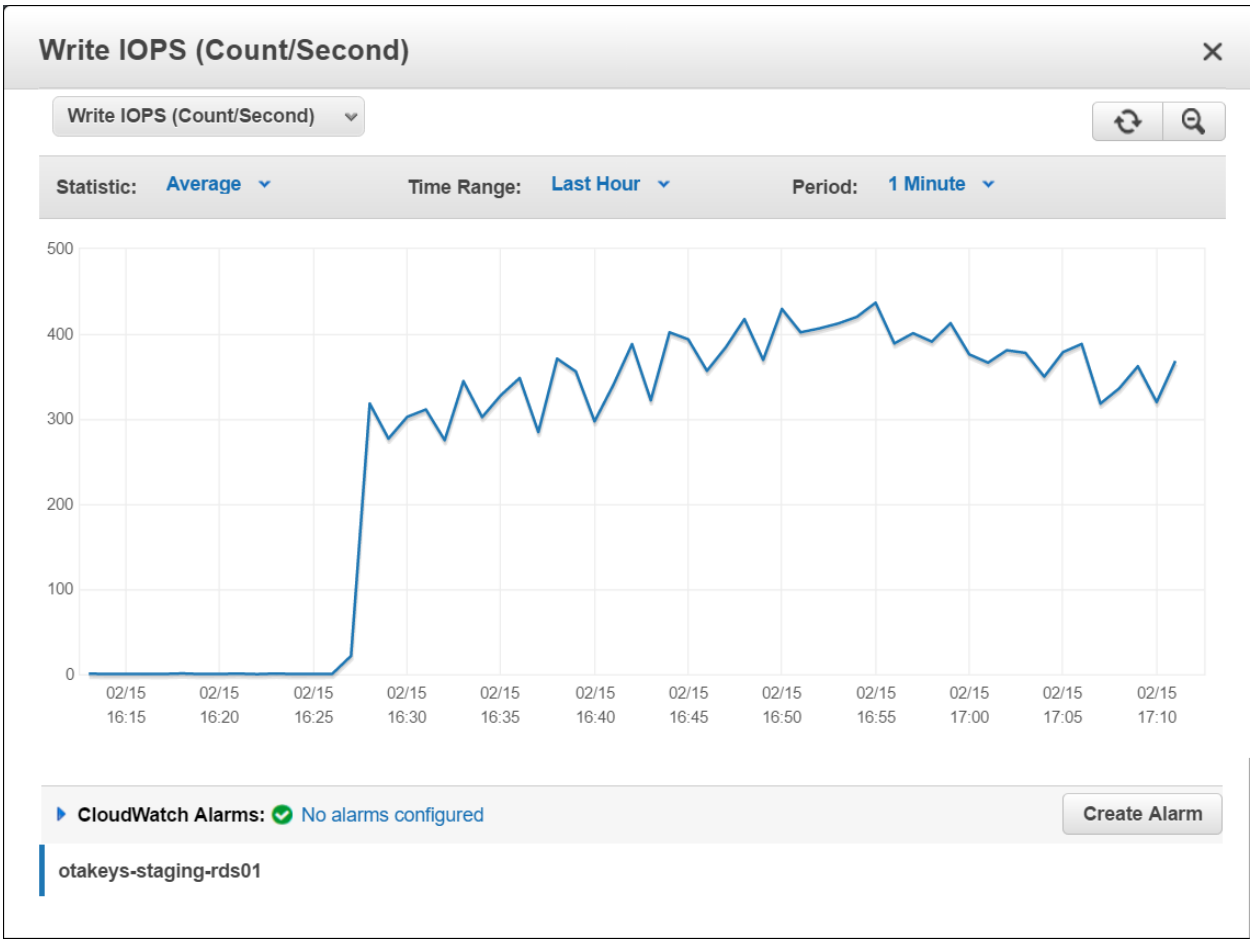

Chart 17 - RDS Write IOPS - scenario 2, AWS setup 1

RDS is at its Write IOPS limit. Below the relevant API EC2 instances monitors.

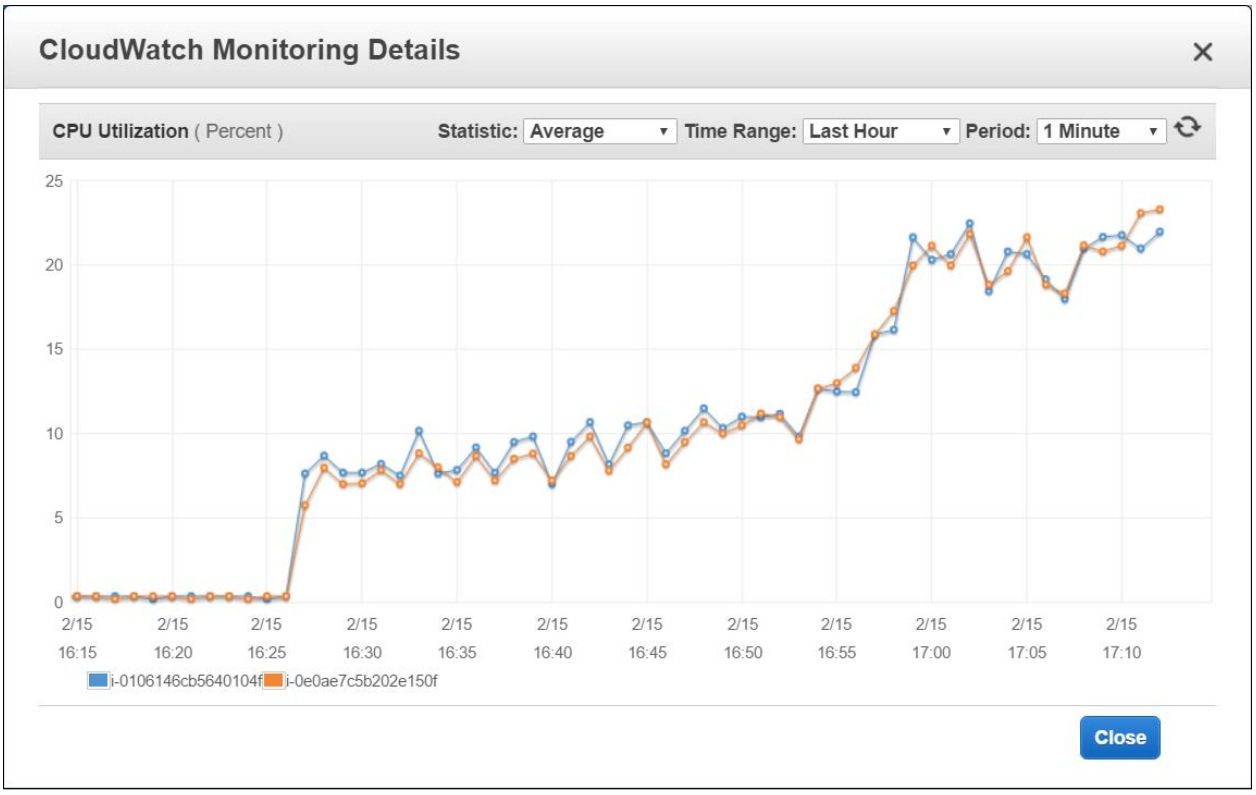

Chart 18 - EC2 CPU - scenario 2, AWS setup 1

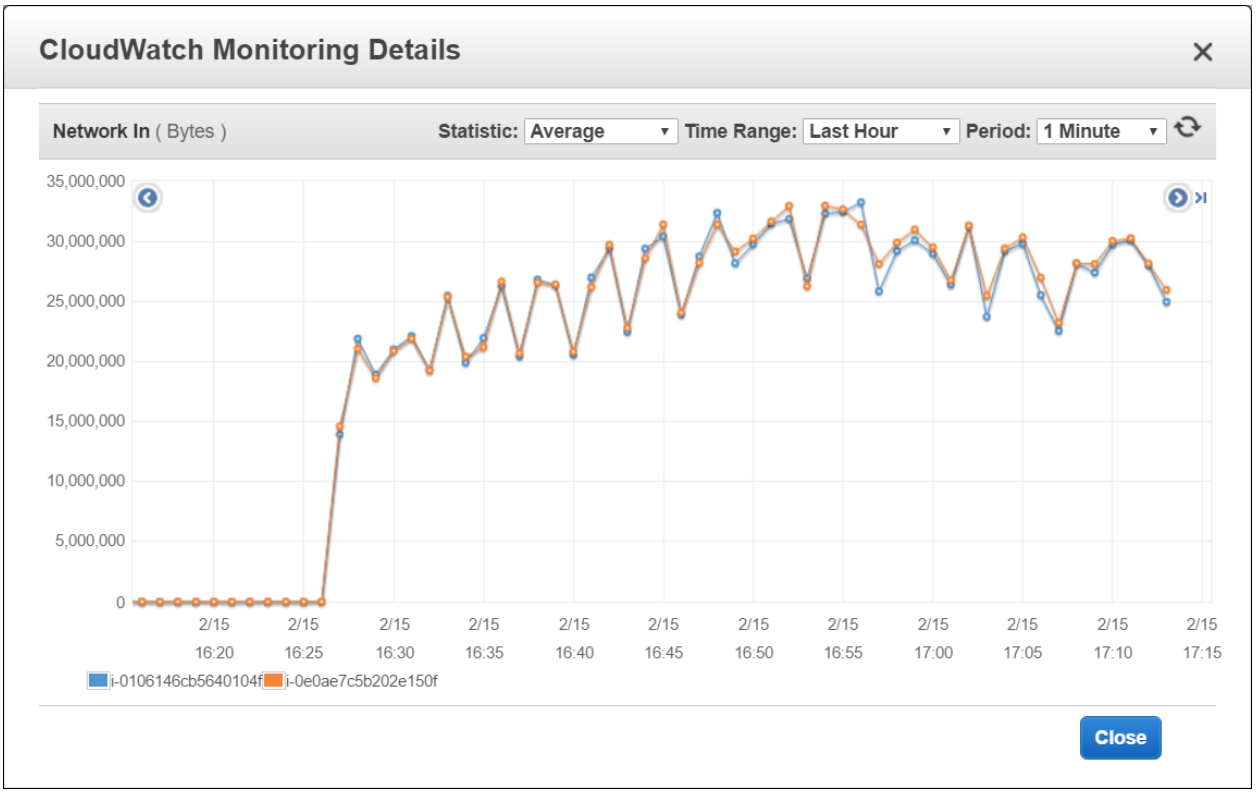

Chart 19 - EC2 Network in - scenario 2, AWS setup 1

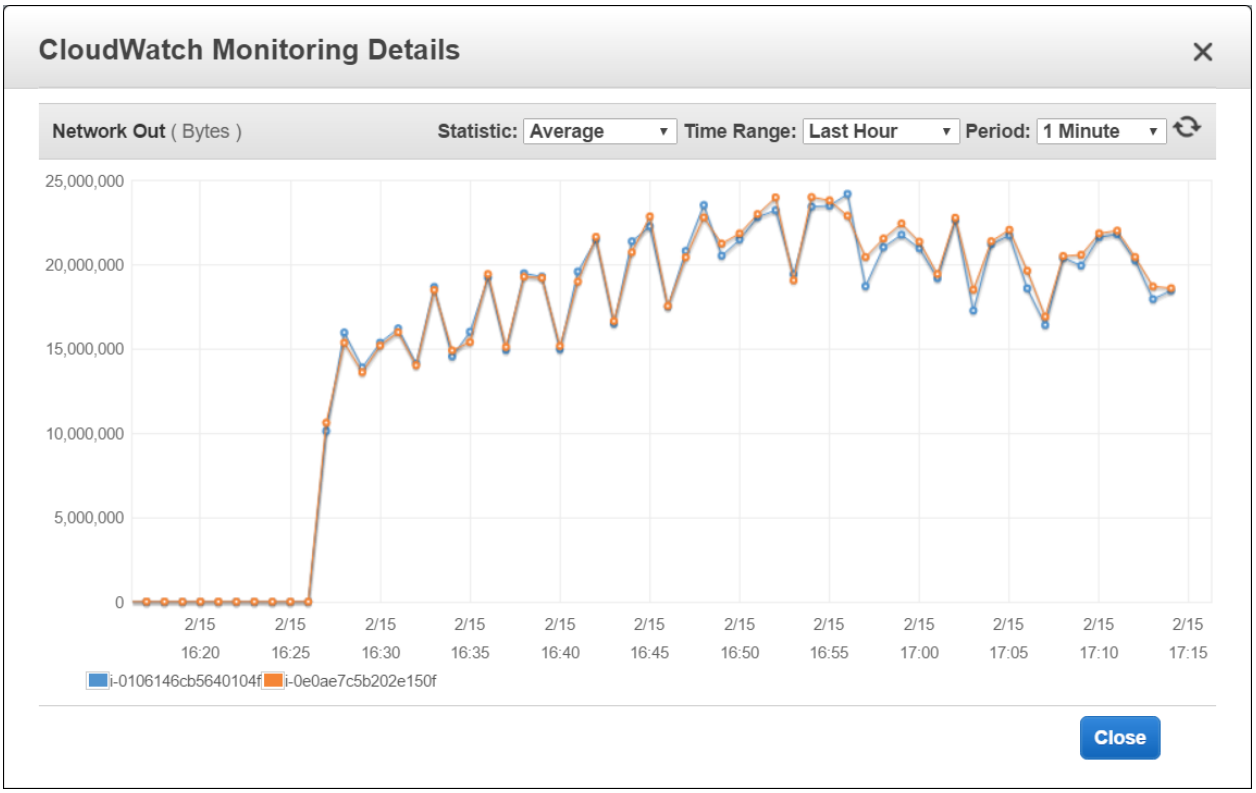

Chart 20 - EC2 Network out - scenario 2, AWS setup 1

AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS)

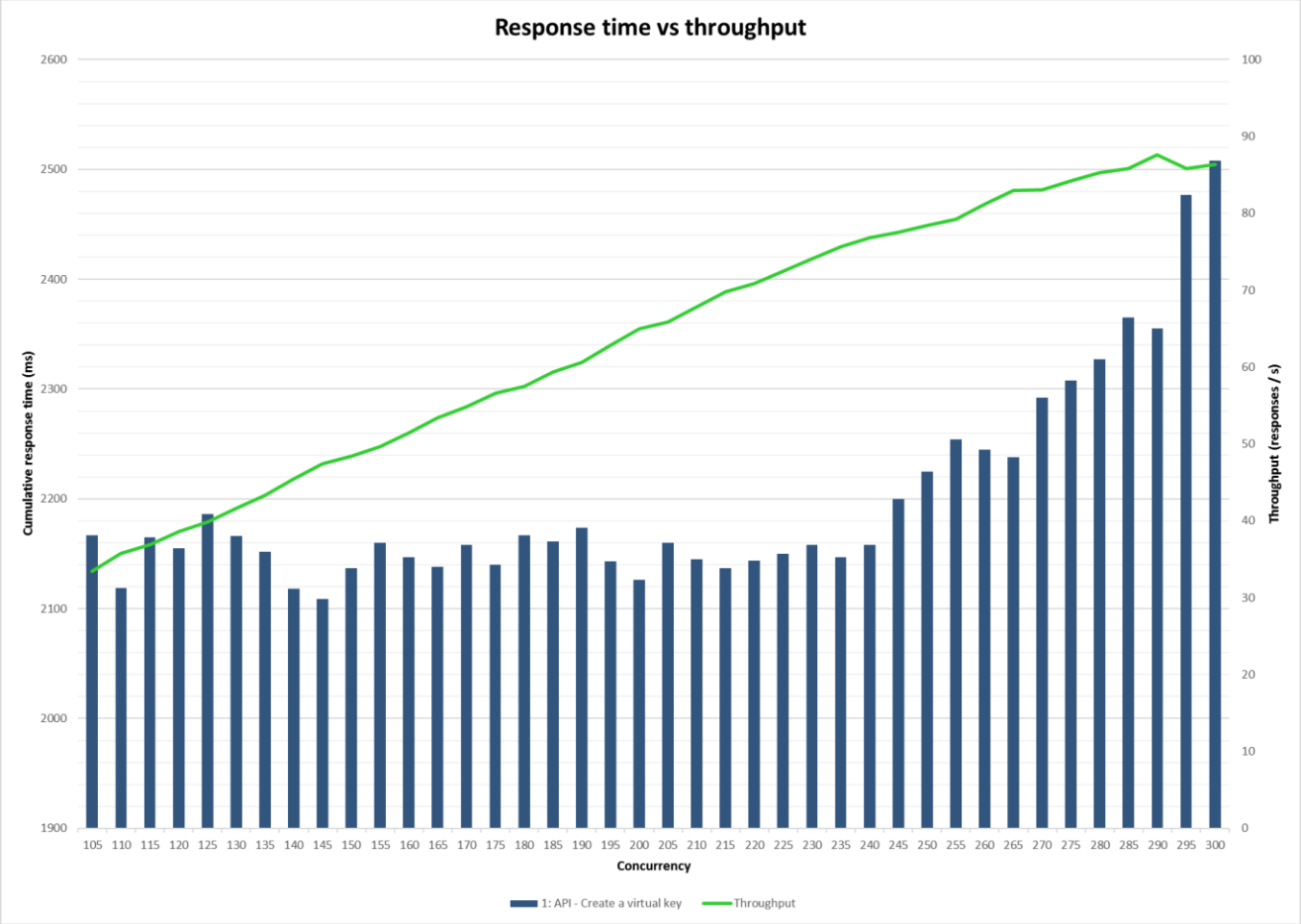

Chart 21 - Response time vs throughput - scenario 2, AWS setup 2

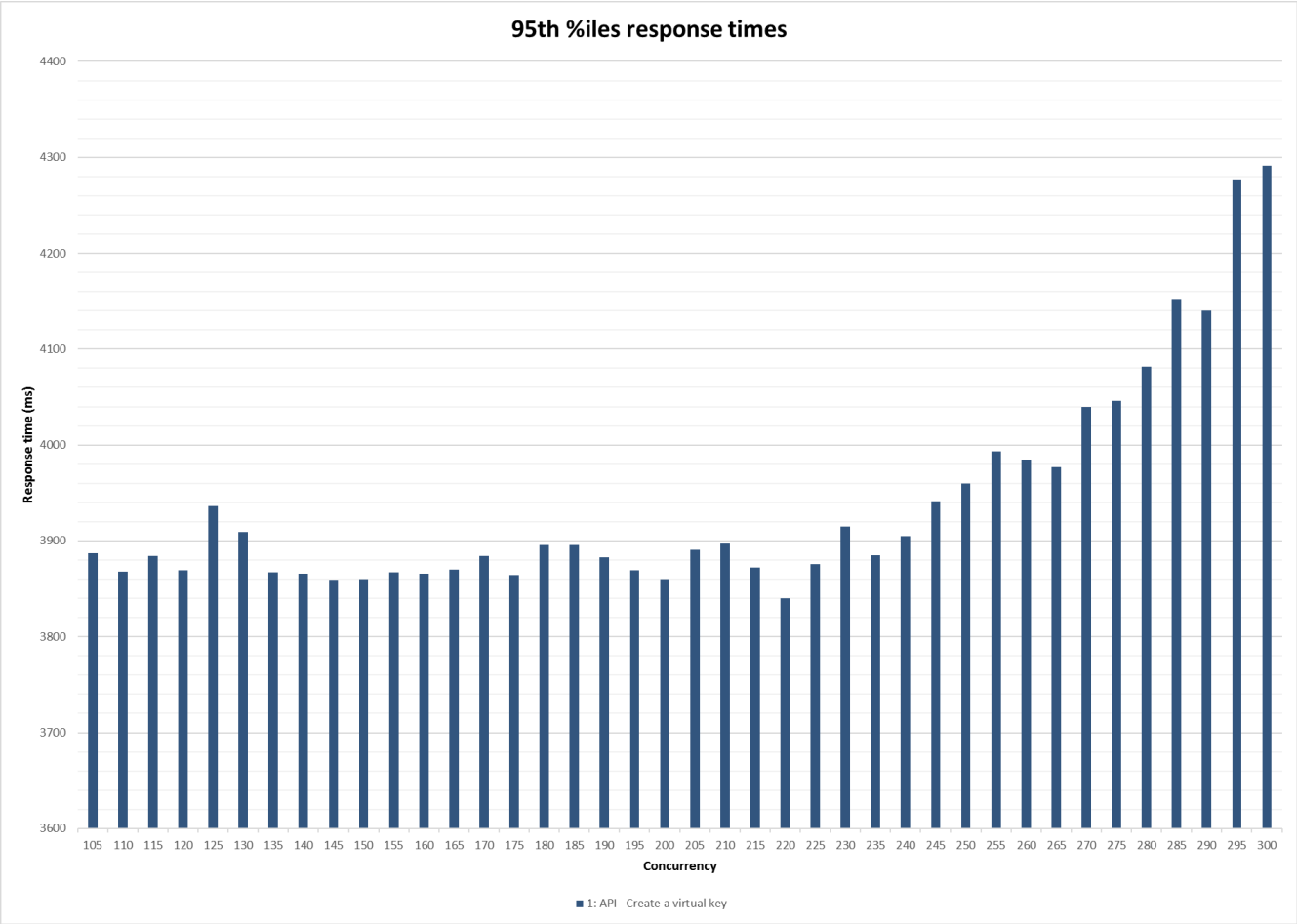

Chart 22 - 95%iles response times - scenario 2, AWS setup 2

If maximum response times are around 4 seconds this means that the server can handle the requests without problems, in this specific case.

Around concurrency 265 the application is saturated.

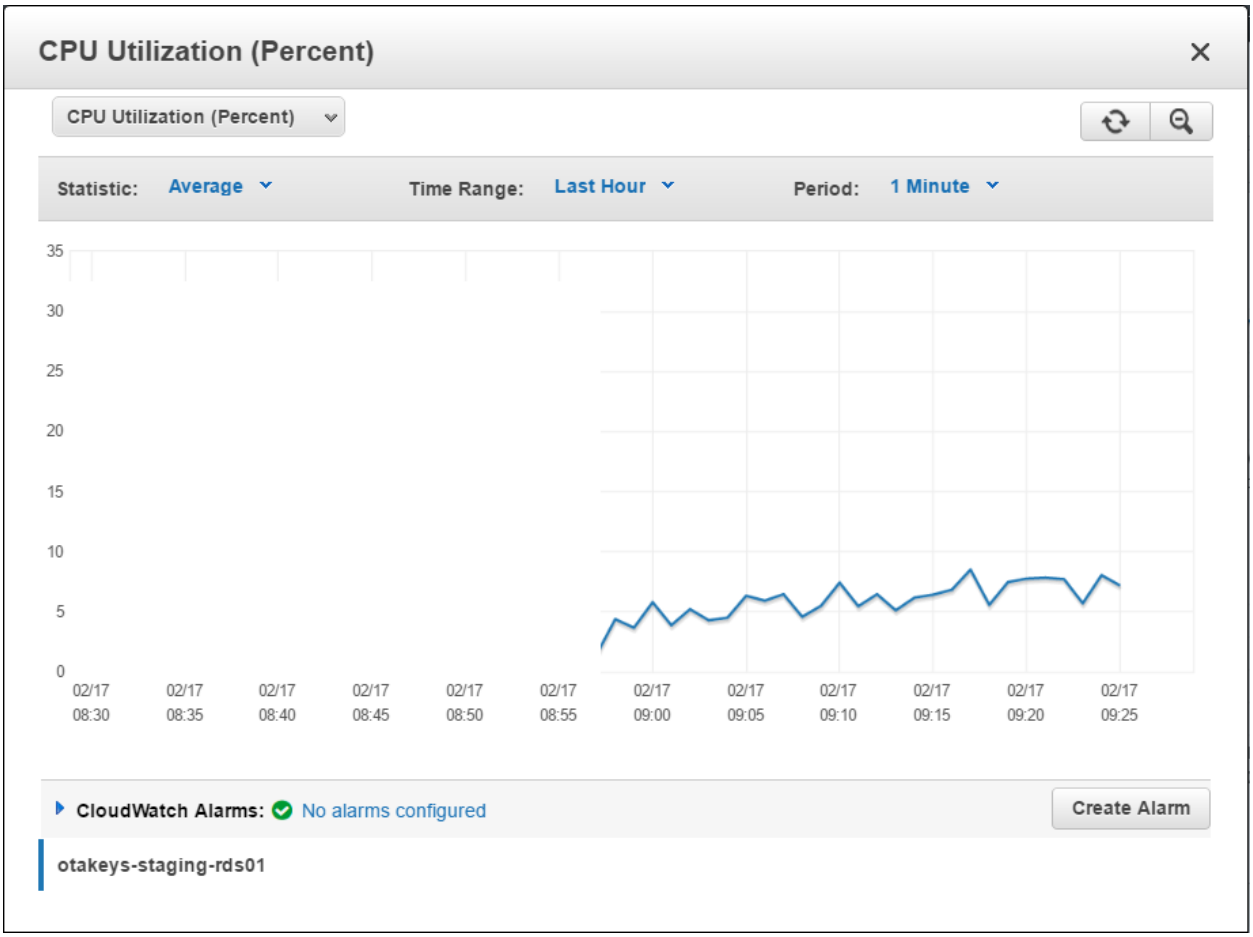

Chart 23 - RDS CPU - scenario 2, AWS setup 2

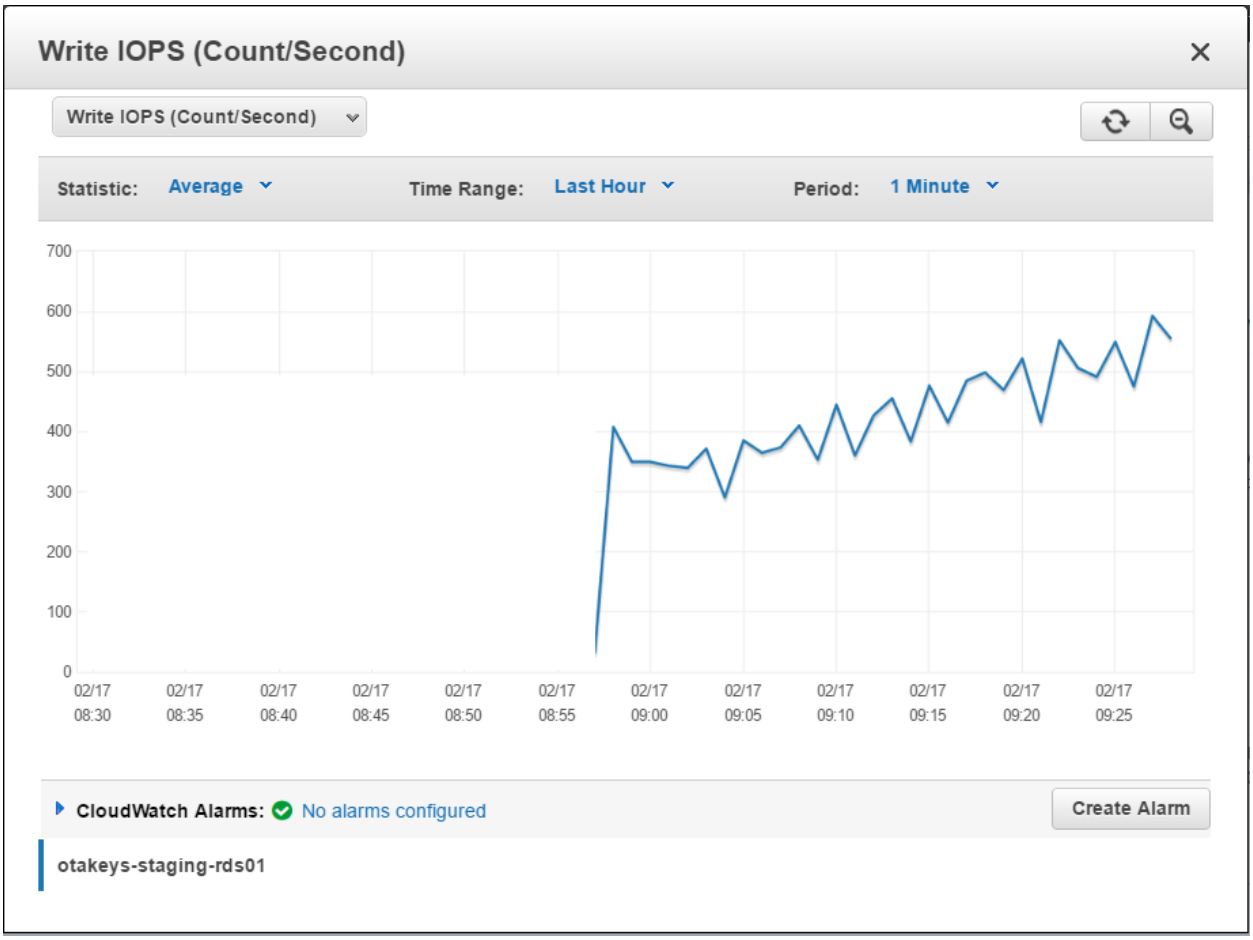

Chart 24 - RDS Write IOPS - scenario 2, AWS setup 2

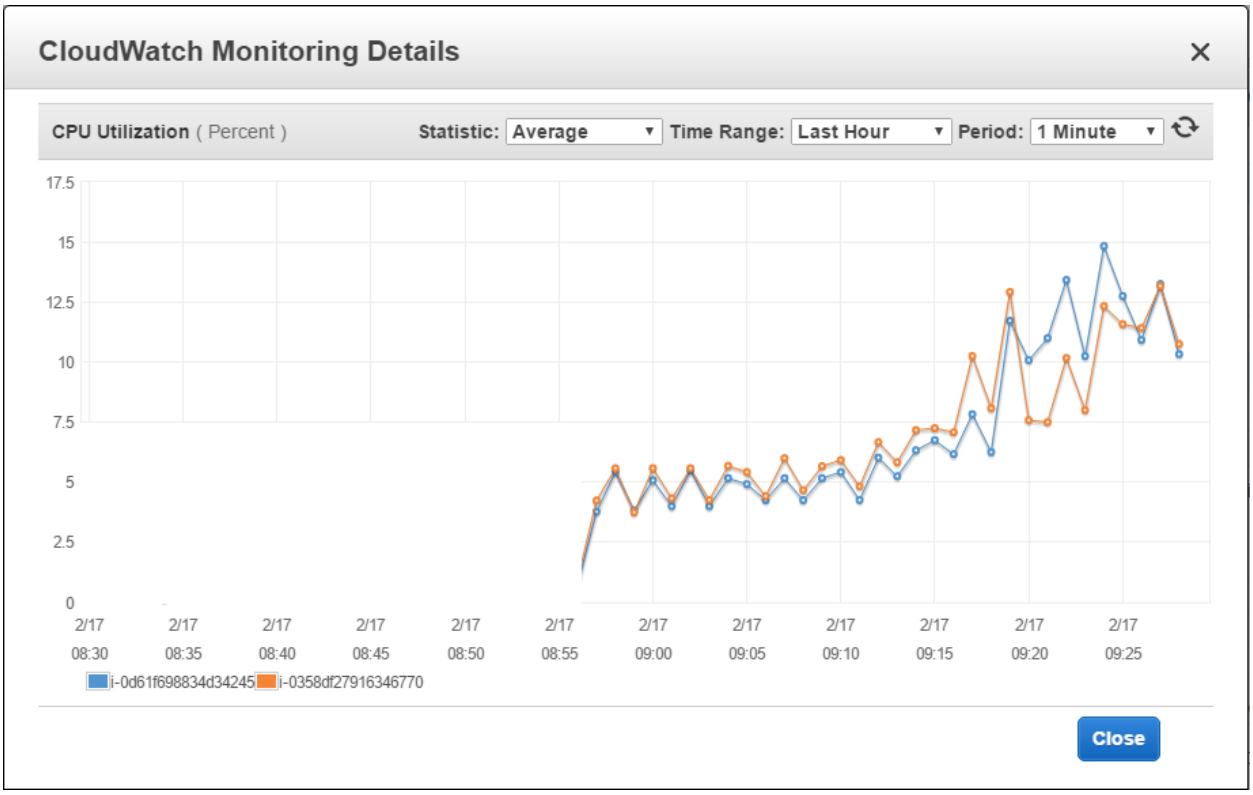

Chart 25 - EC2 CPU - scenario 2, AWS setup 2

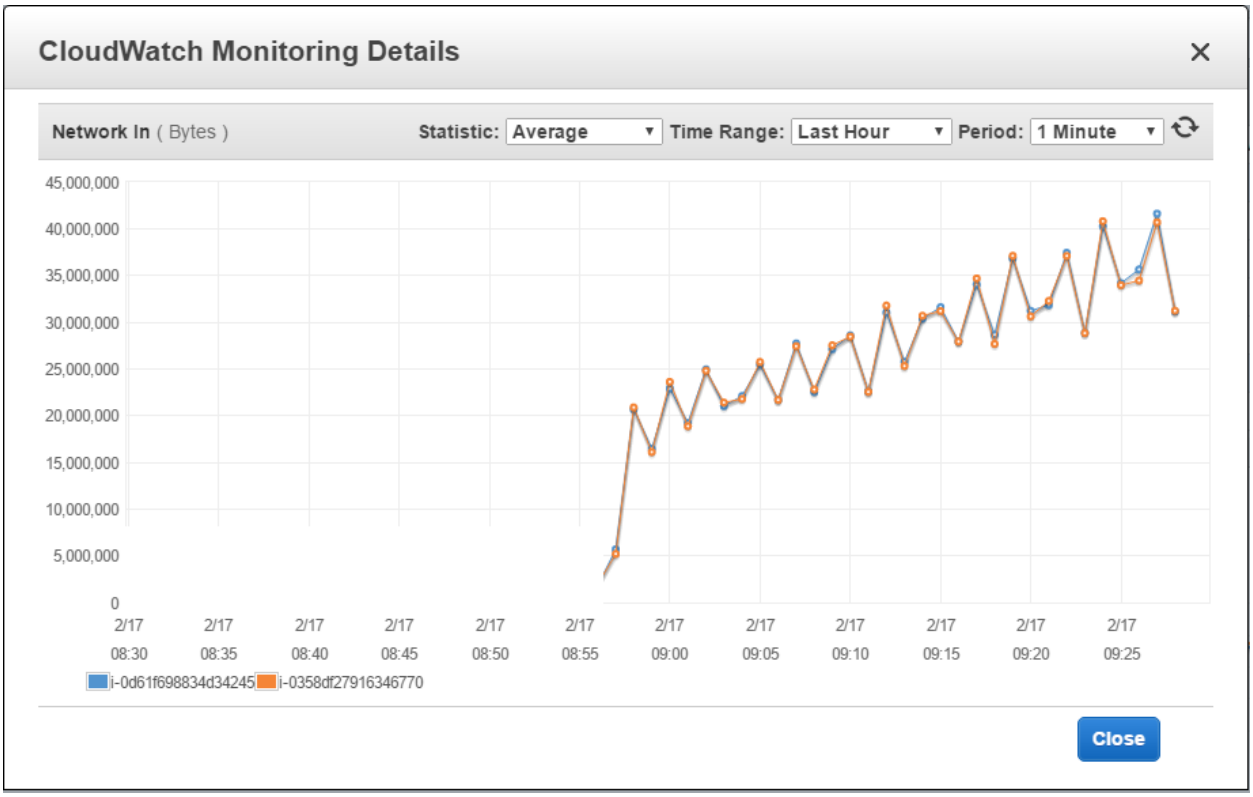

Chart 26 - EC2 Network in - scenario 2, AWS setup 2

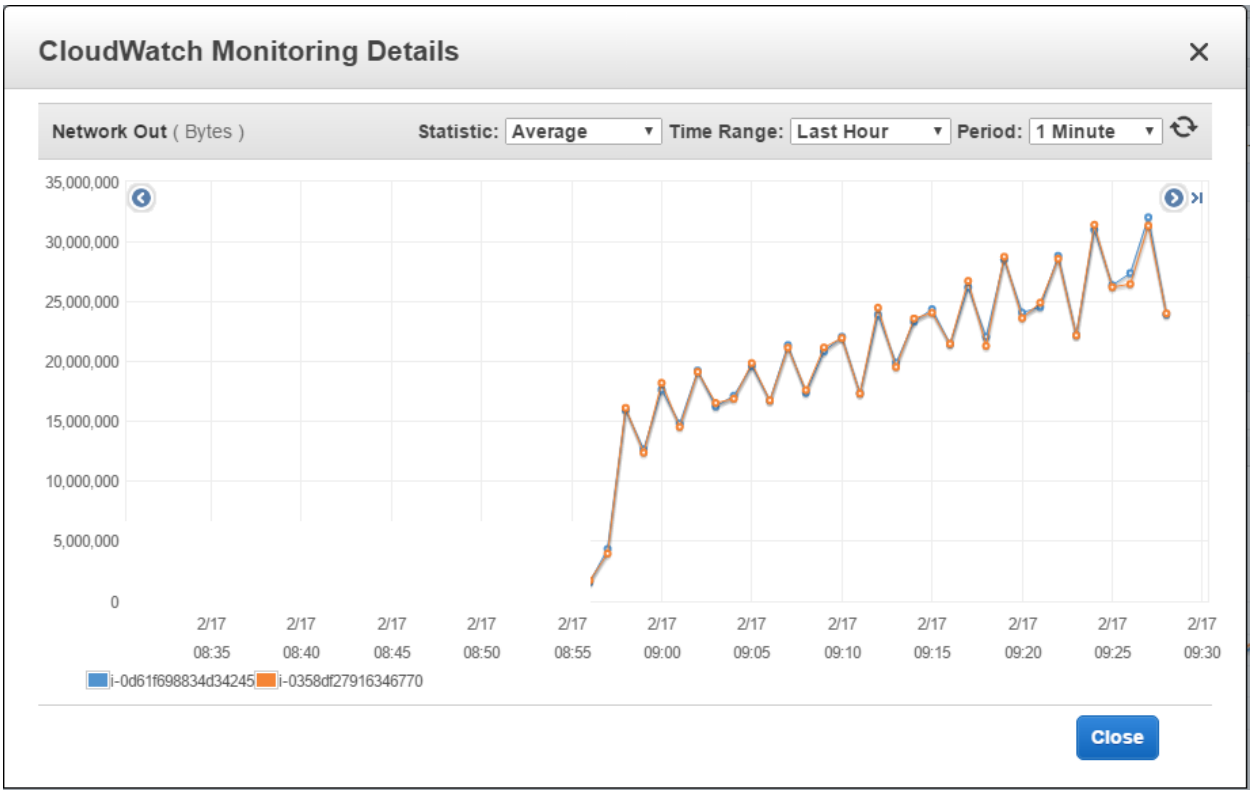

Chart 27 - EC2 Network out - scenario 2, AWS setup 2

AWS setup 3: ELB --> 2x c4.large + RDS Aurora

Important to note: To simulate authentication the server-side waits between 150 and 4500 milliseconds before returning the response.

Concurrency	Average response time	95th %ile response time	Throughput	Errors
300	2720	4733	81.59	30

Screenshot 2 - Results - scenario 2, AWS setup 3

Creating a virtual key at concurrency 300 is not a problem. Max response times are slightly above 4.5 seconds.

There were some errors. All the results of a key Id that could not be fetched from a response. Meaning that the expected response was not returned.

FYI, Extracted like this:

```
if(relUrl.EndsWith("/key/create")){
  string response = DownloadResponse(out length, false);
_keyId = GetStringBetween(response, "\"result\":{\"key\":{\"id\":", ",\"extId\""); }
}
```

No hardware bottlenecks were detected: We maybe could get good results with even higher concurrencies.

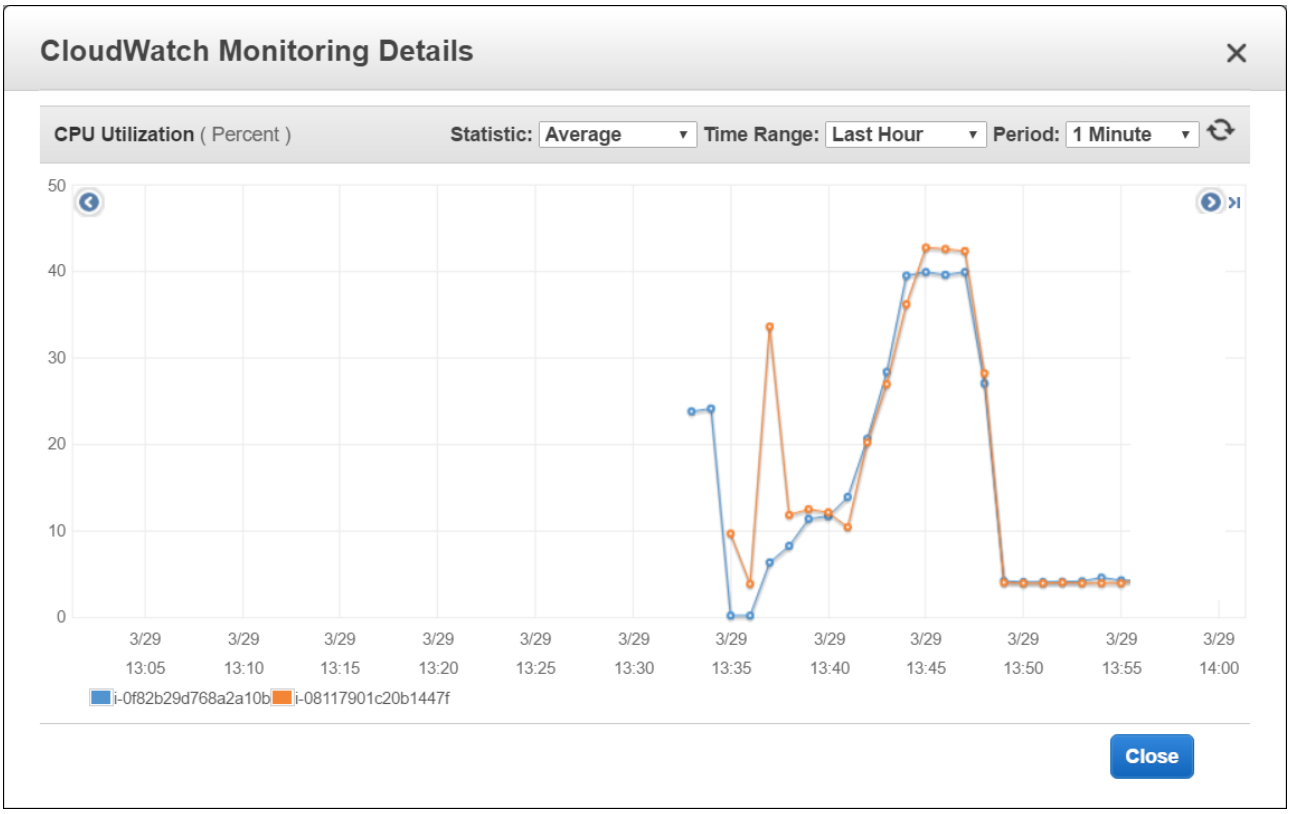

Chart 28 - EC2 CPU- scenario 2, AWS setup 3

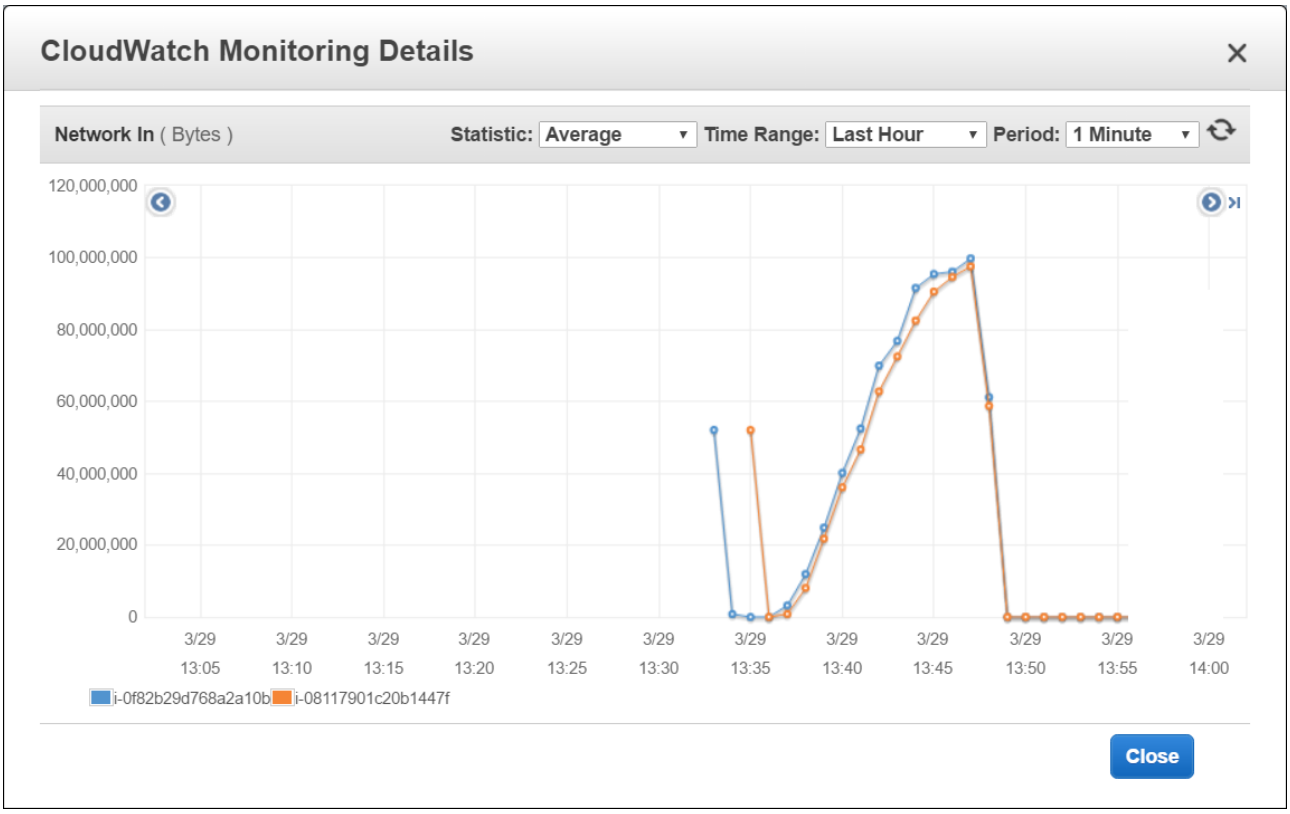

Chart 29 - EC2 Network in - scenario 2, AWS setup 3

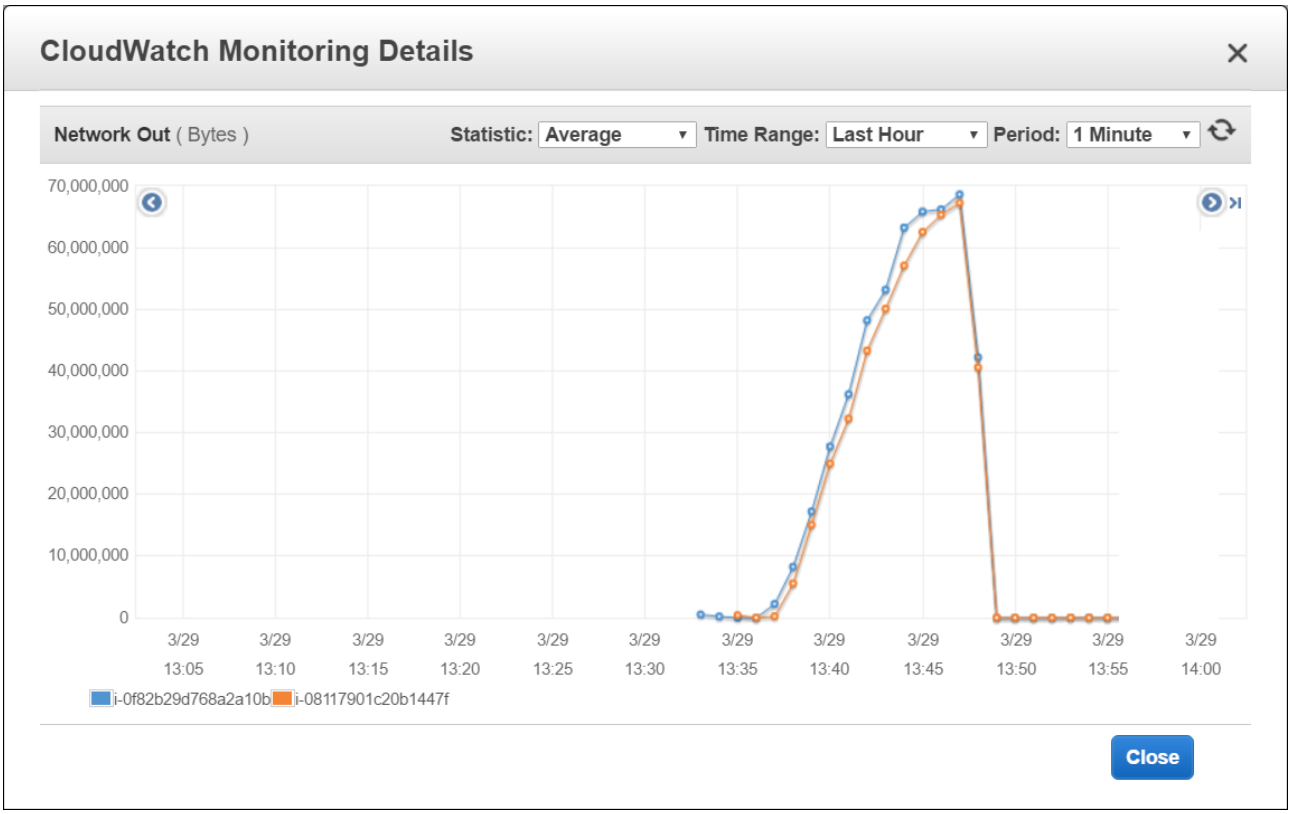

Chart 30 - EC2 Network out - scenario 2, AWS setup 3

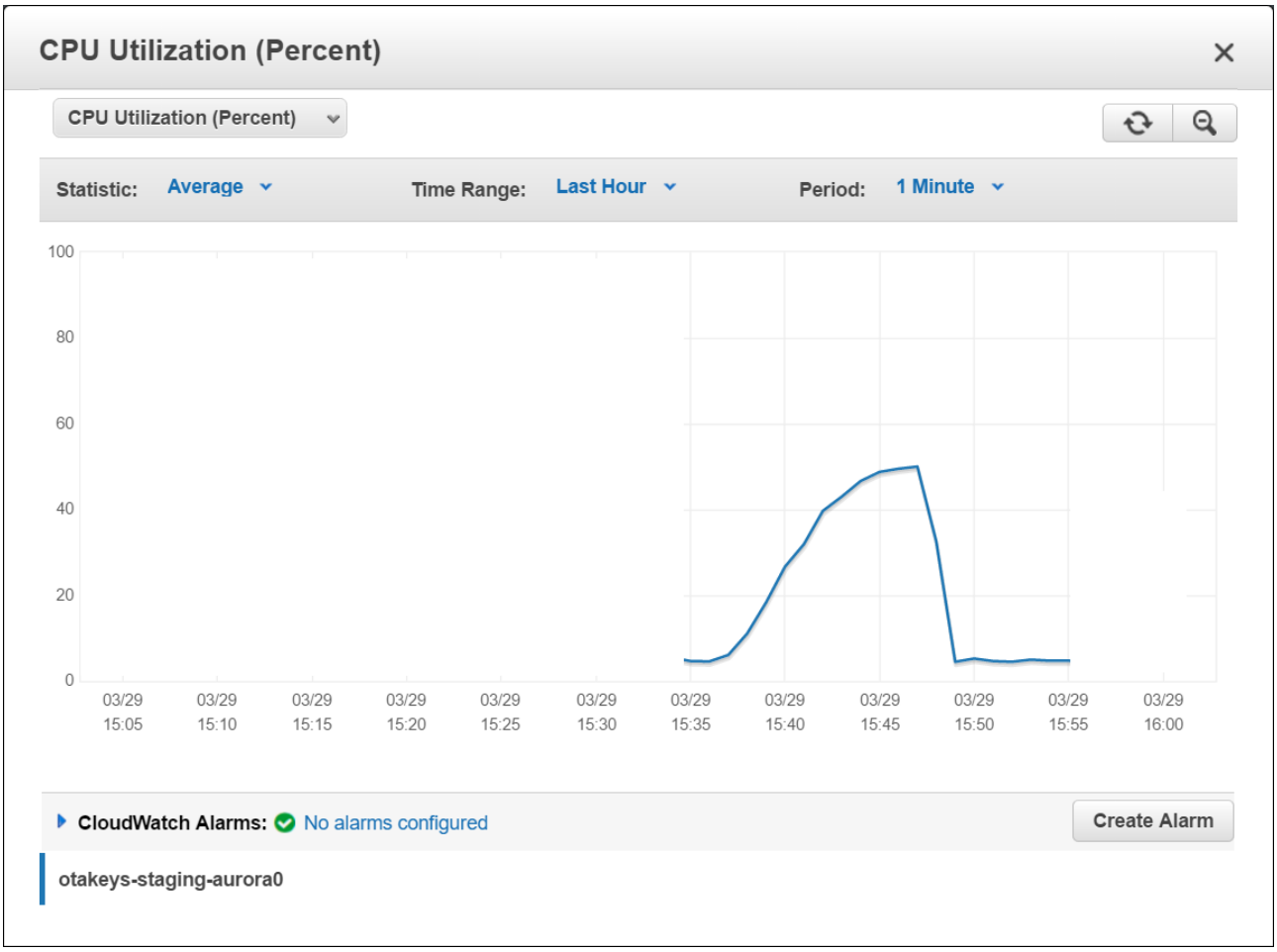

Chart 31 - RDS CPU - scenario 2, AWS setup 3

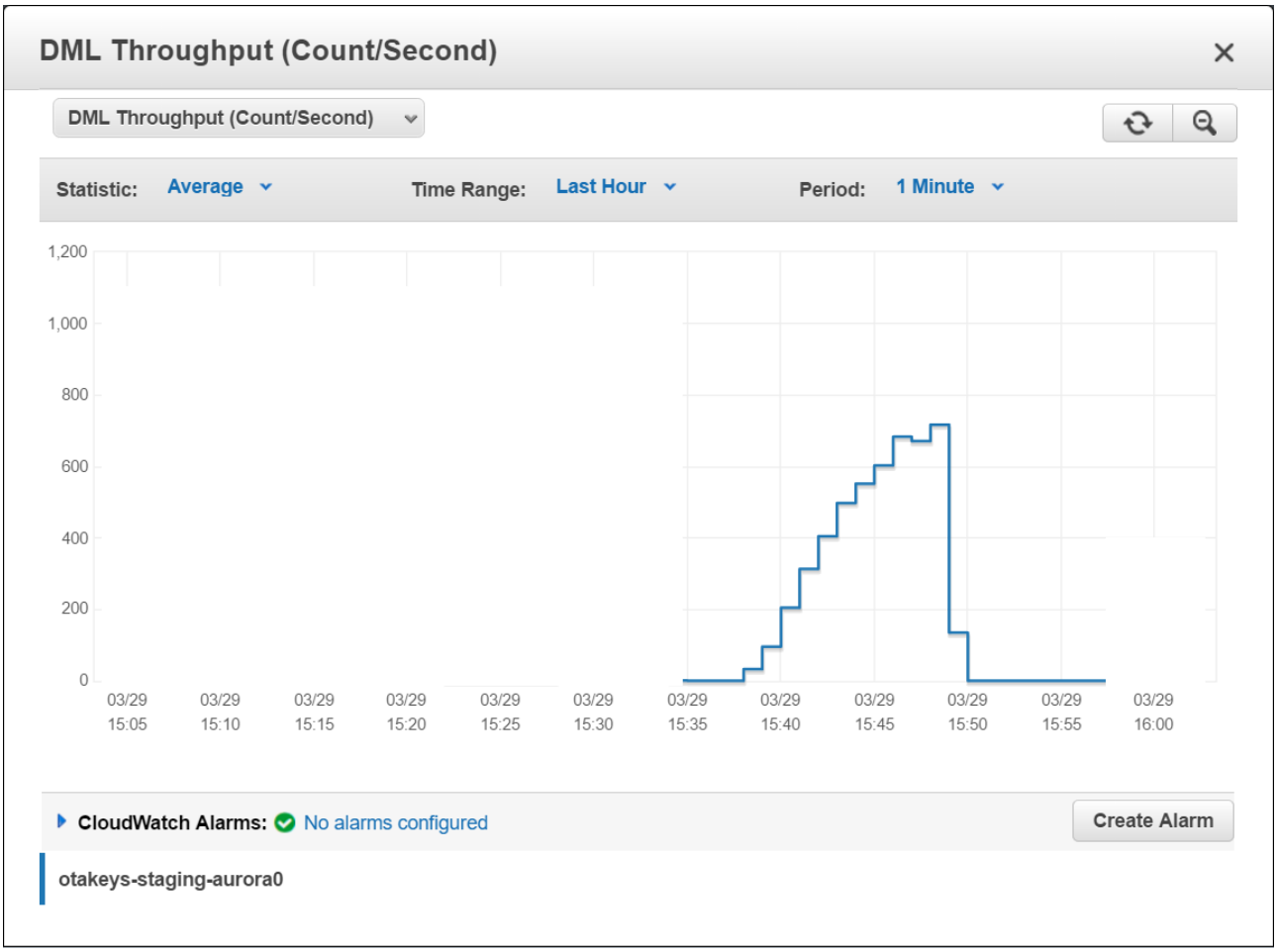

Chart 32 - RDS DML throughput - scenario 2, AWS setup 3

Scenario 4: Get keys, create a key, post synthesis and unlock doors

AWS setup 1: 2 x t2.small for API + db.t2.small RDS MySQL with 100GB gp2 (300 base IOPS)

Keep in mind that behavior is simulated server-side (250 – 4000 milliseconds delay) on 2 calls.

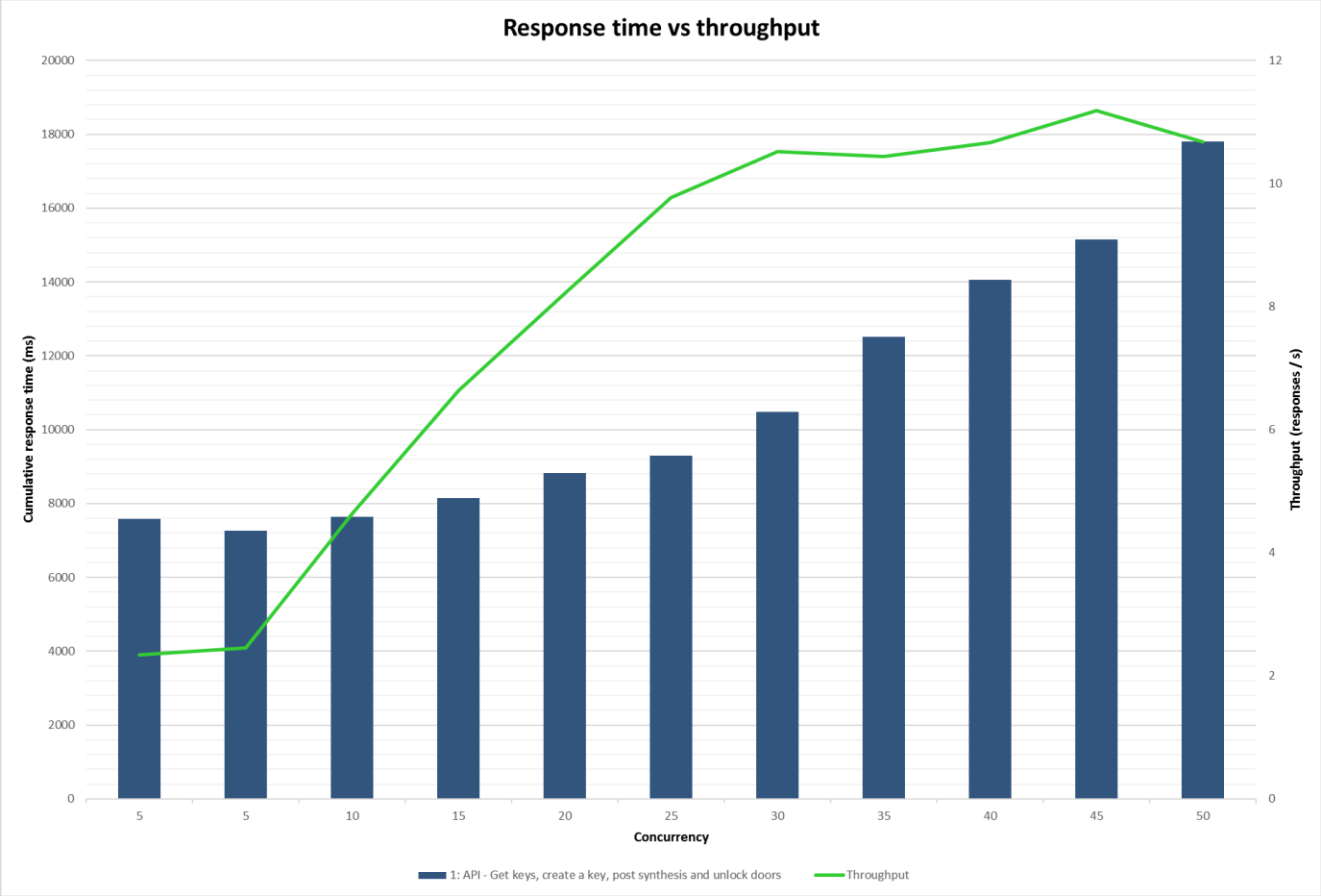

Chart 33 - Response time vs throughput - scenario 4, AWS setup 1

Throughput reaches its plateau at concurrency 30. From the start it takes on average 8 seconds to execute the requests in the user action combined. Hereunder the response times of the requests.

	A	C	D	F
1	Concurrency	Request	Avg. response time (ms)	95th percentile of the response times (ms)
10	50	GET•/rest/sdk/v3/keys••••appli	7381	14358
11	50	POST•/rest/sdk/v3/key/restitut	4024	8916
12	50	POST•/rest/sdk/v3/key/create•	3127	7425
13	50	POST•/rest/sdk/v3/synthesis••{	3279	7072
42	10	POST•/rest/sdk/v3/key/restitut	2223	4107
43	10	POST•/rest/sdk/v3/synthesis••{	2299	3950
44	10	POST•/rest/sdk/v3/key/create•	2274	3895
45	10	GET•/rest/sdk/v3/keys••••appli	851	2712

Screenshot 3 - Response times at concurrency 50 - scenario 4, AWS setup 1

If the user action takes on average around 2.5 seconds and maximum 4.5 seconds that means that the application has no problem handling the requests.

Note that at concurrency 50 getting all keys take a very long time, doubling the response time from the start.

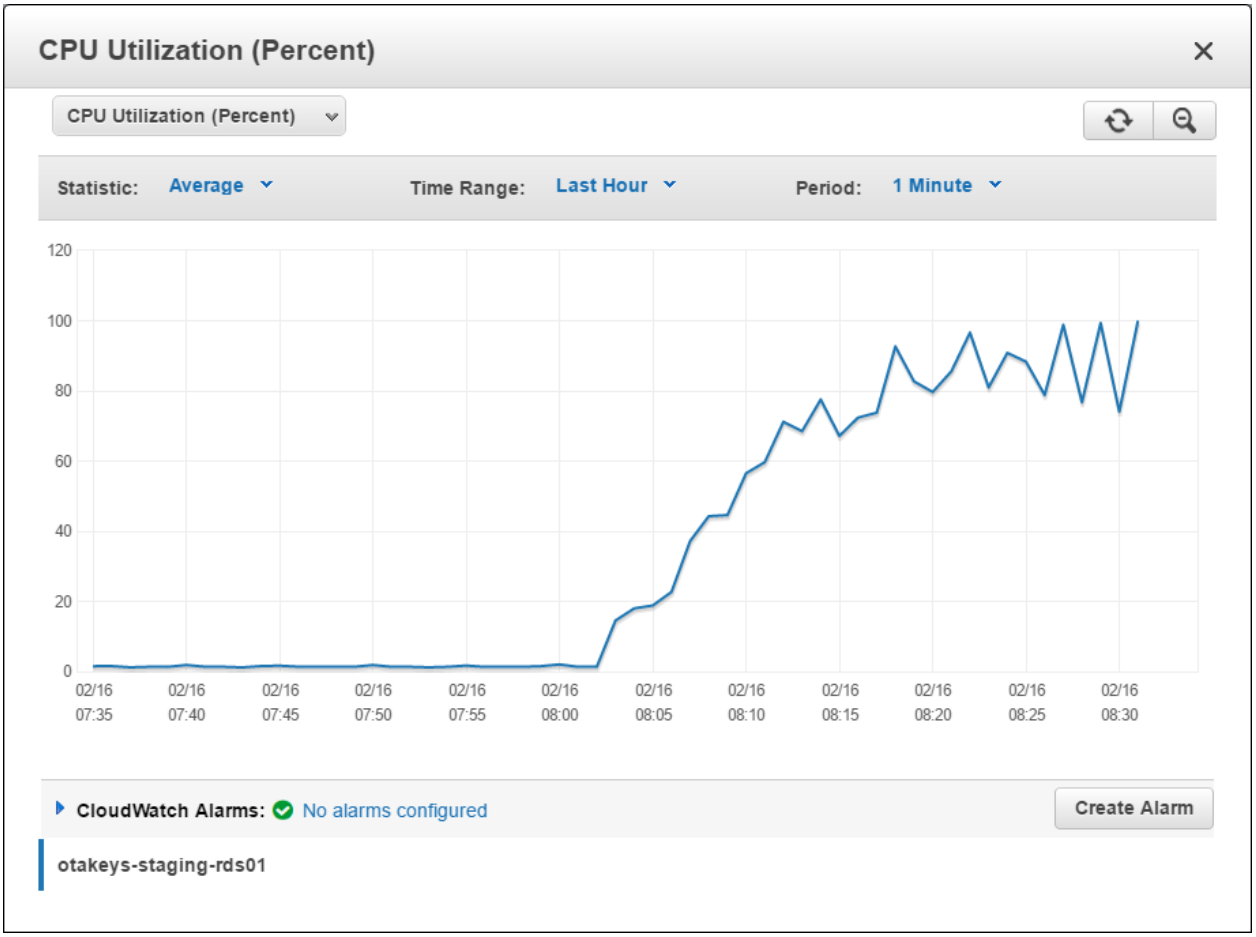

Chart 34 - RDS CPU - scenario 4, AWS setup 1

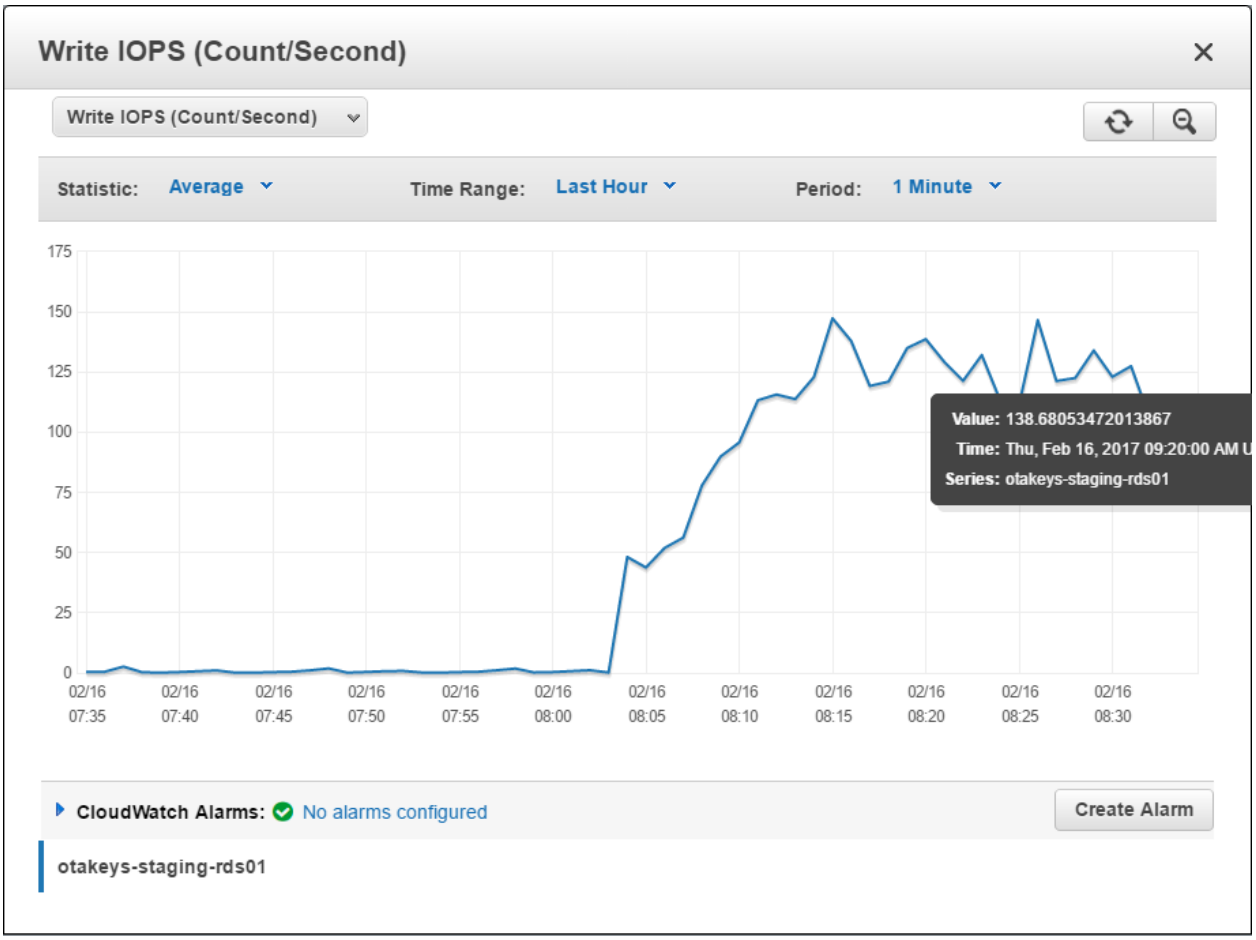

Chart 35 - RDS Write IOPS- scenario 4, AWS setup 1

RDS uses all available processing power. No problems detected at the API EC2 instances.

AWS setup 2: ELB --> 2x c4.large + RDS MySQL db.r3.xlarge 1TB gp2 (3000 base IOPS)

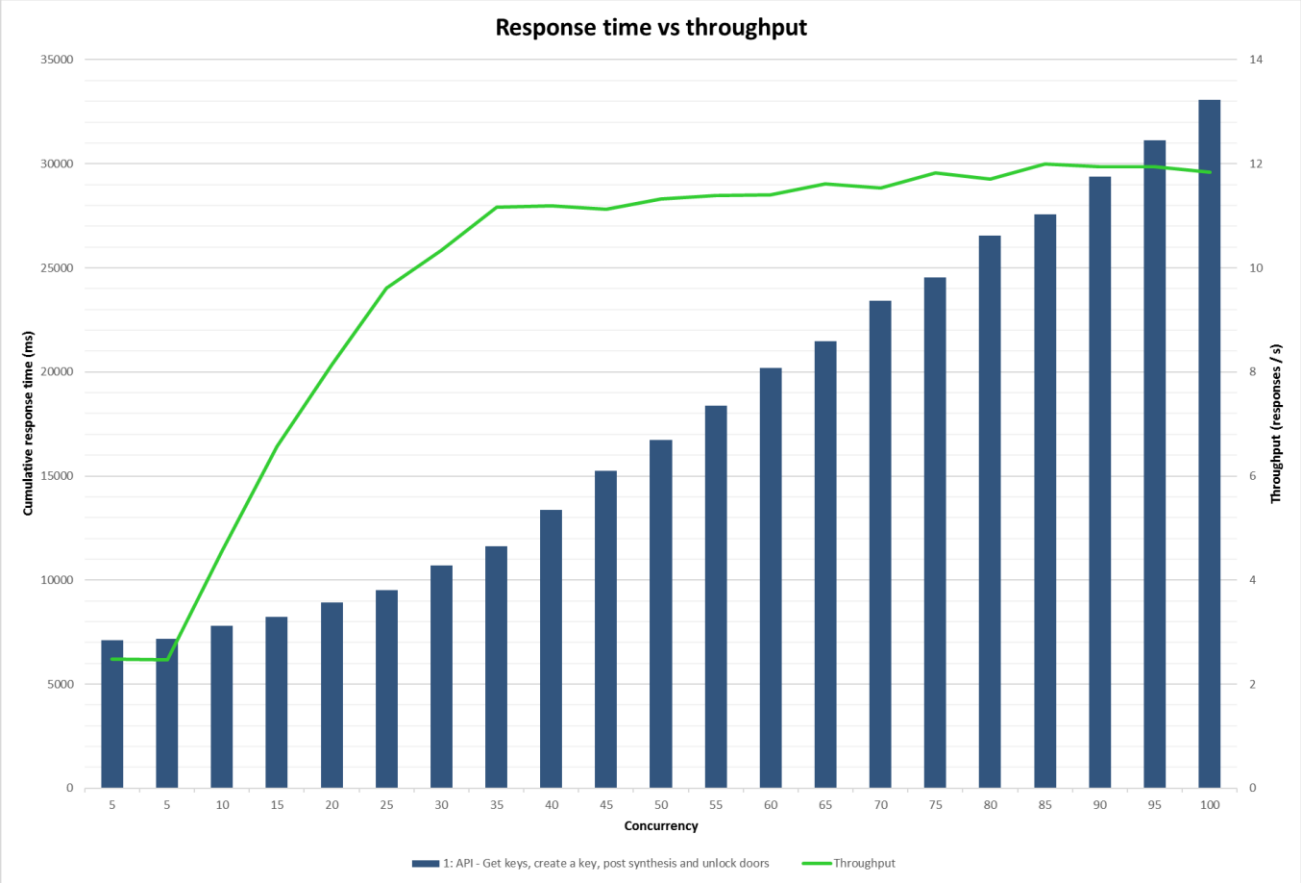

Chart 36 - Response times vs throughput - scenario 4, AWS setup 2

The results are slightly better than the previous ones. At concurrency 5 we have a response time for the complete user action around 7.1 seconds instead of around 7.5 seconds. Throughput reaches a plateau at concurrency 35 and increases slightly until the test is finished (because of the more beefy RDS).

	A	C	D	E	F
1	Concurrency	Request	Avg. response time (ms)	Max. response time (ms)	95th percentile of the response times (ms)
10	50	GET•/rest/sdk/v3/keys••••applic	7703	16271	15270
11	50	POST•/rest/sdk/v3/key/create••	3030	10386	7272
12	50	POST•/rest/sdk/v3/key/restitute	3226	10159	6115
13	50	POST•/rest/sdk/v3/synthesis••{	2767	10052	5763
42	10	POST•/rest/sdk/v3/synthesis••{	2208	4049	3978
43	10	POST•/rest/sdk/v3/key/create••	2150	4031	3869
44	10	POST•/rest/sdk/v3/key/restitute	2260	4168	3834
45	10	GET•/rest/sdk/v3/keys••••applic	1178	3017	2974

Screenshot 4 - Response times at concurrency 50 - scenario 4, AWS setup 2

If creating a key (and the POST restitute and synthesis as well, I suspect) take on average around 2.5 seconds and maximum 4.5 seconds that means that the application has no problem handling the requests.

Note that getting all keys takes longer than previously.

Chart 37 - Errors vs throughput - scenario 4, AWS setup 2

There were some errors: 504 Gateway Timeout.

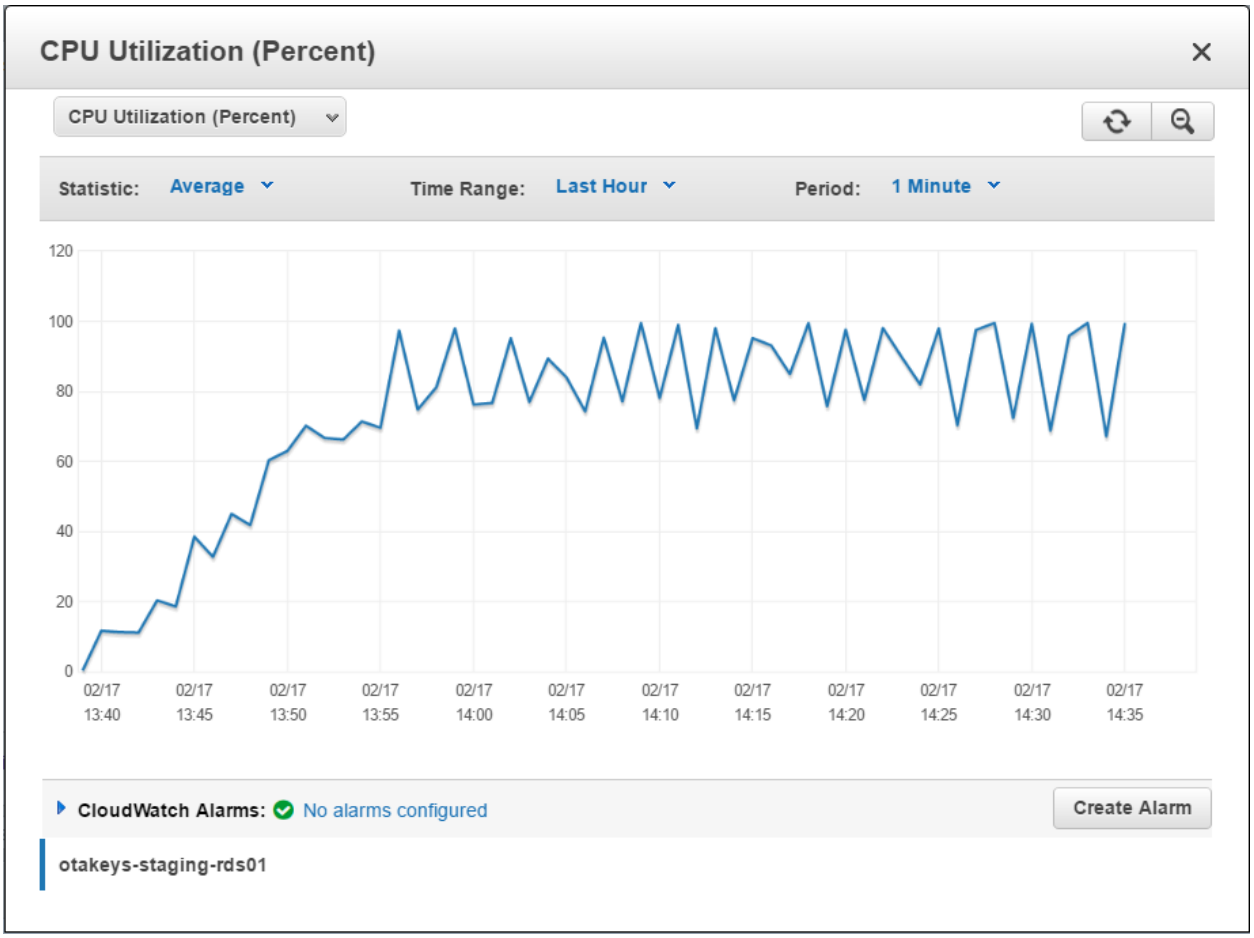

Chart 38 - RDS CPU - scenario 4, AWS setup 2

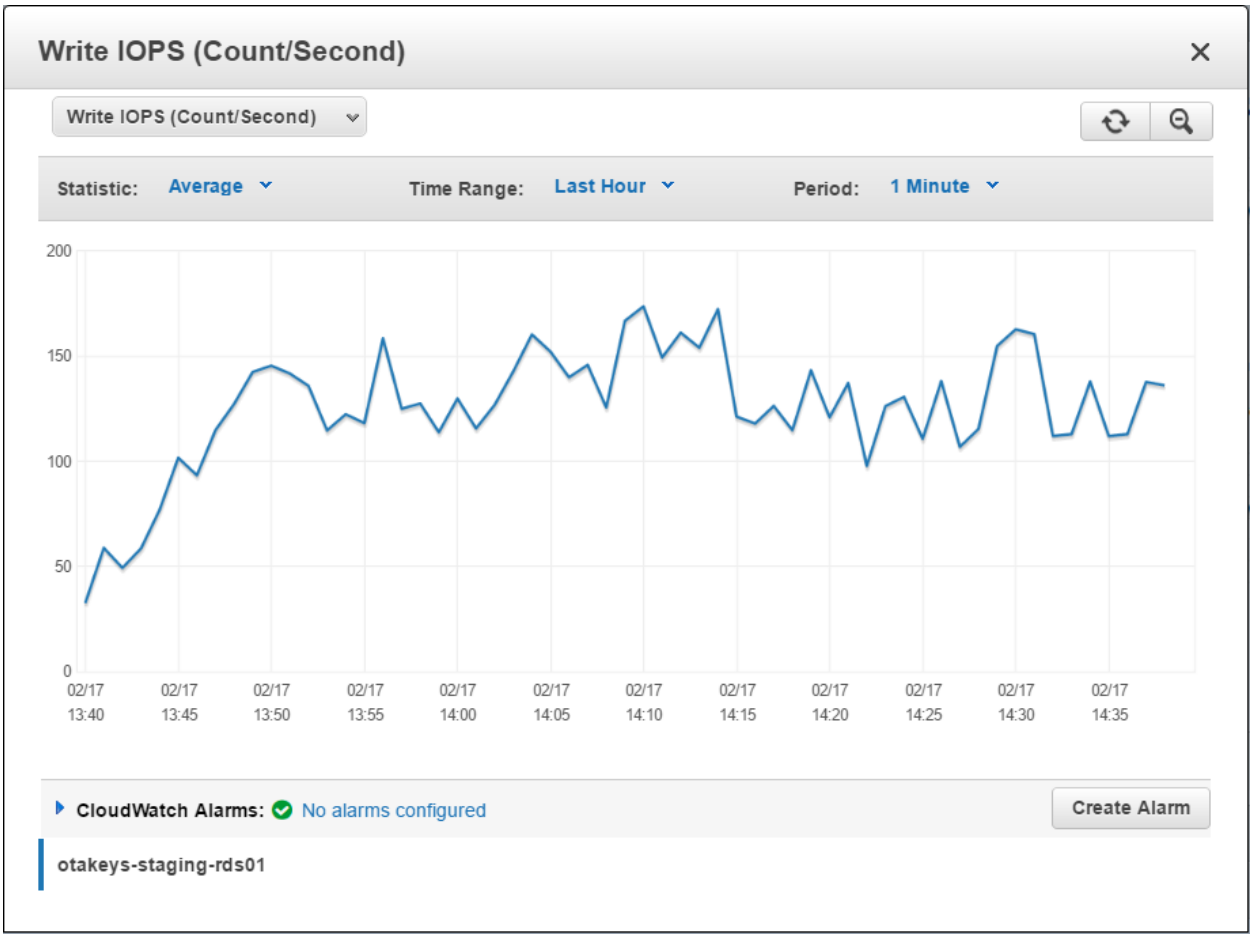

Chart 39 - RDS Write IOPS - scenario 4, AWS setup 2

Even though the AWS configuration is more beefy than previously RDS still poses a bottleneck in terms of processing power.

```
user@       :~$ pt-query-digest "mysql-slowquery (1).log"

# 3.6s user time, 10ms system time, 35.23M rss, 104.10M vsz
# Current date: Mon Feb 20 09:44:16 2017
# Hostname: GravPOC
# Files: mysql-slowquery (1).log
# Overall: 7.29k total, 1 unique, 2.46 QPS, 16.81x concurrency _____________
# Time range: 2017-02-17T14:00:00 to 2017-02-17T14:49:22
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time         49782s      1s     12s      7s      8s      1s      7s
# Lock time             2s   147us    12ms   301us   348us   247us   273us
# Rows sent         28.49k       4       4       4       4       0       4
# Rows examine       2.61G 371.63k 378.84k 375.23k 362.29k       0 362.29k
# Query size        19.03M   2.67k   2.67k   2.67k   2.62k       0   2.62k

# Profile
# Rank Query ID           Response time     Calls R/Call V/M   Item
# ==== ================== ================= ===== ====== ===== ===========
#    1 0xA59C8E004DA547AC 49781.7662 100.0%  7293 6.8260  0.19 SELECT virtual_key vehicle access_device

# Query 1: 2.46 QPS, 16.81x concurrency, ID 0xA59C8E004DA547AC at byte 11765414
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.19
# Time range: 2017-02-17T14:00:00 to 2017-02-17T14:49:22
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count        100    7293
# Exec time    100  49782s      1s     12s      7s      8s      1s      7s
# Lock time    100      2s   147us    12ms   301us   348us   247us   273us
# Rows sent    100  28.49k       4       4       4       4       0       4
# Rows examine 100   2.61G 371.63k 378.84k 375.23k 362.29k       0 362.29k
# Query size   100  19.03M   2.67k   2.67k   2.67k   2.62k       0   2.62k
# String:
# Databases    otakeys_staging
# Hosts        10.11.21.183 (3689/50%)... 1 more
# Users        ota_user
# Query_time distribution
#   1us
#  10us
# 100us
#   1ms
#  10ms
# 100ms
#    1s  ################################################################
#  10s+  #
# Tables
#    SHOW TABLE STATUS FROM `otakeys_staging` LIKE 'virtual_key'\G
#    SHOW CREATE TABLE `otakeys_staging`.`virtual_key`\G
#    SHOW TABLE STATUS FROM `otakeys_staging` LIKE 'vehicle'\G
#    SHOW CREATE TABLE `otakeys_staging`.`vehicle`\G
#    SHOW TABLE STATUS FROM `otakeys_staging` LIKE 'access_device'\G
#    SHOW CREATE TABLE `otakeys_staging`.`access_device`\G
# EXPLAIN /*!50100 PARTITIONS*/
/* named HQL query VirtualKey.getEligibleKeysByAccessDeviceId */ select virtualkey0_.id as id1_83_0_, ve
ter2_83_0_, virtualkey0_.counter_end as counter_3_83_0_, virtualkey0_.deletion_date as deletion4_83_0_,
ualkey0_.key_args as key_args8_83_0_, virtualkey0_.key_sensitive_args as key_sens9_83_0_, virtualkey0_.m
ualkey0_.restitute as restitu13_83_0_, virtualkey0_.security_code as securit14_83_0_, virtualkey0_.start
nd2_71_1_, vehicle1_.deletion_date as deletion3_71_1_, vehicle1_.enabled as enabled4_71_1_, vehicle1_.en
t_id8_71_1_, vehicle1_.finition as finition9_71_1_, vehicle1_.horse_power as horse_p10_71_1_, vehicle1_.
_1_, vehicle1_.tank_capacity as tank_ca15_71_1_, vehicle1_.transmission as transmi16_71_1_, vehicle1_.ve
nt12_31_2_, accessdevi2_.app_key as app_key2_31_2_, accessdevi2_.authenticated as authenti3_31_2_, acces
_.push_access_id as push_acl3_31_2_, accessdevi2_.push_device_id as push_dev7_31_2_, accessdevi2_.revoke
```

Screenshot 5 - Slow query - scenario 4, AWS setup 2

When looking at the slow query log we can match the get keys call to the MySQL query: this query takes maximum (95th %ile) 8 seconds to execute.

This was corrected by the client for the next AWS setup. Furthermore, is it necessary to get all keys in order to do this scenario correctly?

AWS setup 3: ELB --> 2x c4.large + RDS Aurora

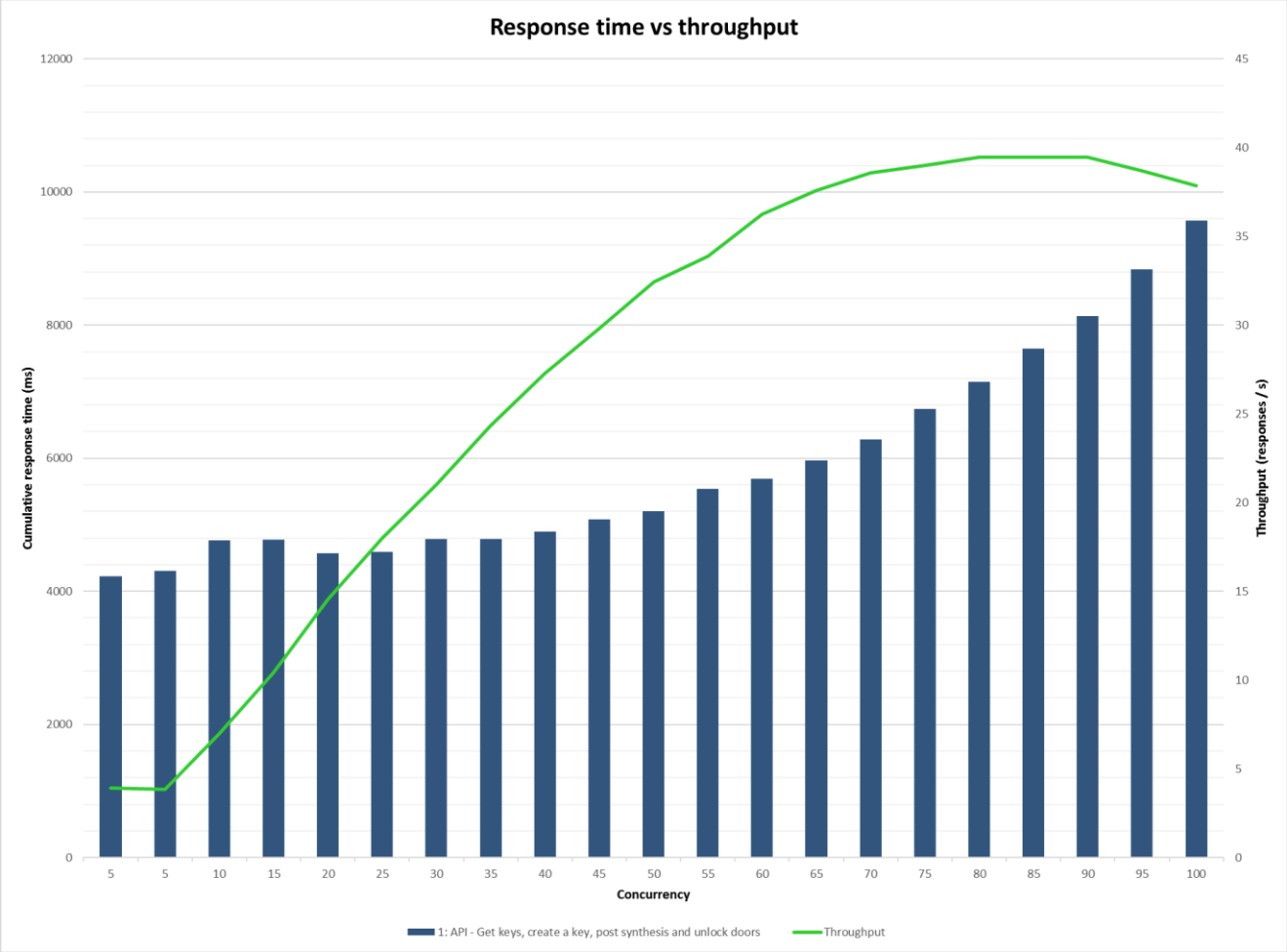

Chart 40 - Response time vs throughput - scenario 4, AWS setup 3

There is a server side delay on 2 calls (250ms to 4000ms) to mimic real-world behavior. We see that the application is much faster now with this scenario. Mainly fixing the 'get all keys SQL request' is responsible for this.

Concurrency	Request	Avg. response time (ms)	Max. response time (ms)	95th percentile of the response times (ms)
50	GET /rest/sdk/v3/keys••••applicatior	83	341	254
50	POST /rest/sdk/v3/key/create••{	2244	4472	4050
50	POST /rest/sdk/v3/synthesis••{	428	1379	914
50	POST /rest/sdk/v3/key/restitute••{	2453	4712	4220

Getting keys takes now 83ms on average instead of 7700ms before.

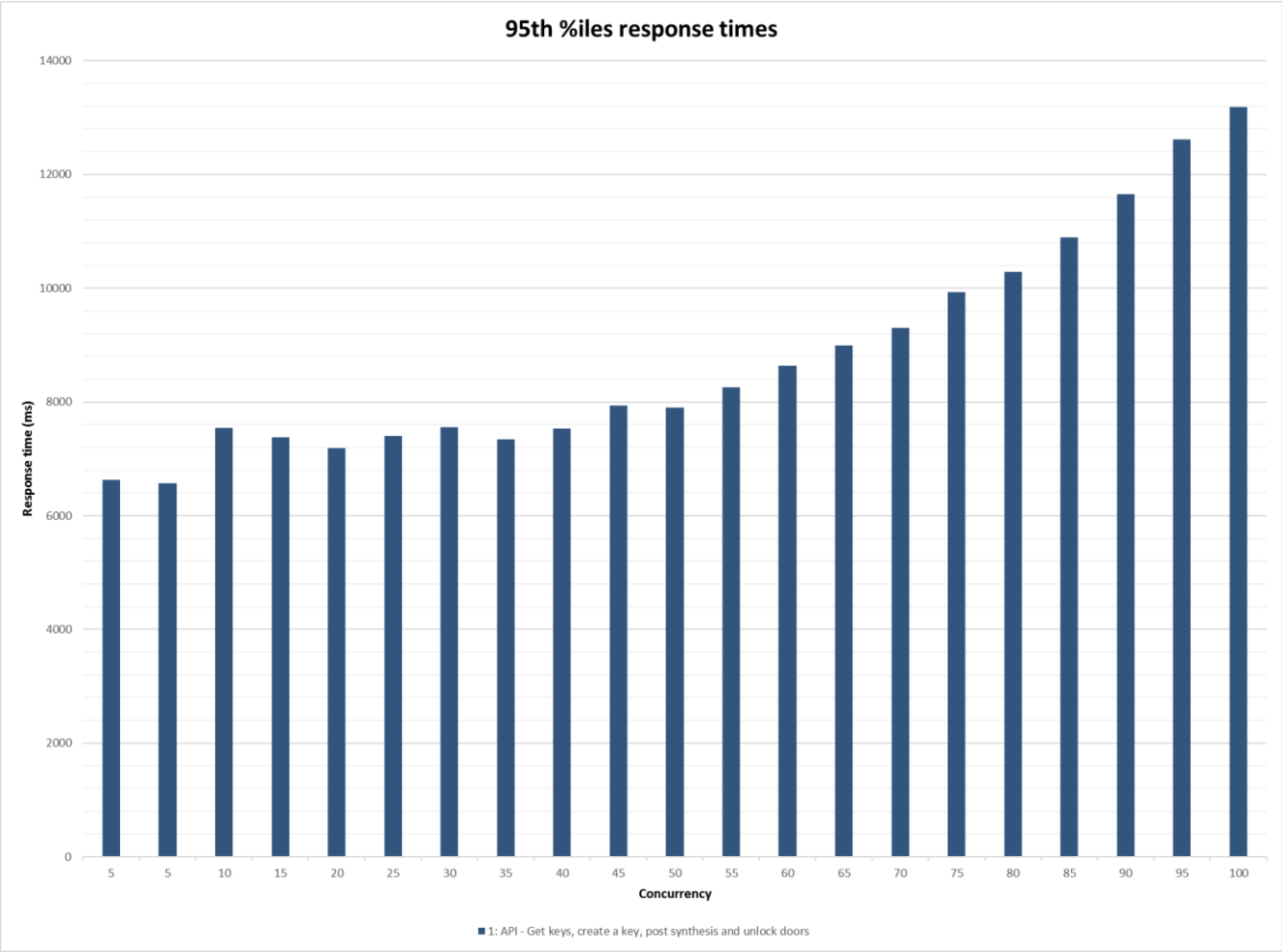

Chart 41 - 95th %iles response times - scenario 4, AWS setup 3

Since 2 calls have a combined server-side delay of maximum 8 seconds we can state that if the measured maximum response time is not higher than that 8 seconds the application can handle the load.

The application meets its saturation point at concurrency 60-65, making it twice as fast as before.

Chart 42 - Errors vs throughput - scenario 4, AWS setup 3

There were some errors. All the results of a key Id that could not be fetched from a response. Meaning that the expected response was not returned.

FYI, Extracted like this:

```
if(relUrl.EndsWith("/key/create")){
  string response = DownloadResponse(out length, false);
_keyId = GetStringBetween(response, "\"result\":{\"key\":{\"id\":", ",\"extId\""); }
}
```

As you can see further on RDS CPU usage poses a bottleneck.

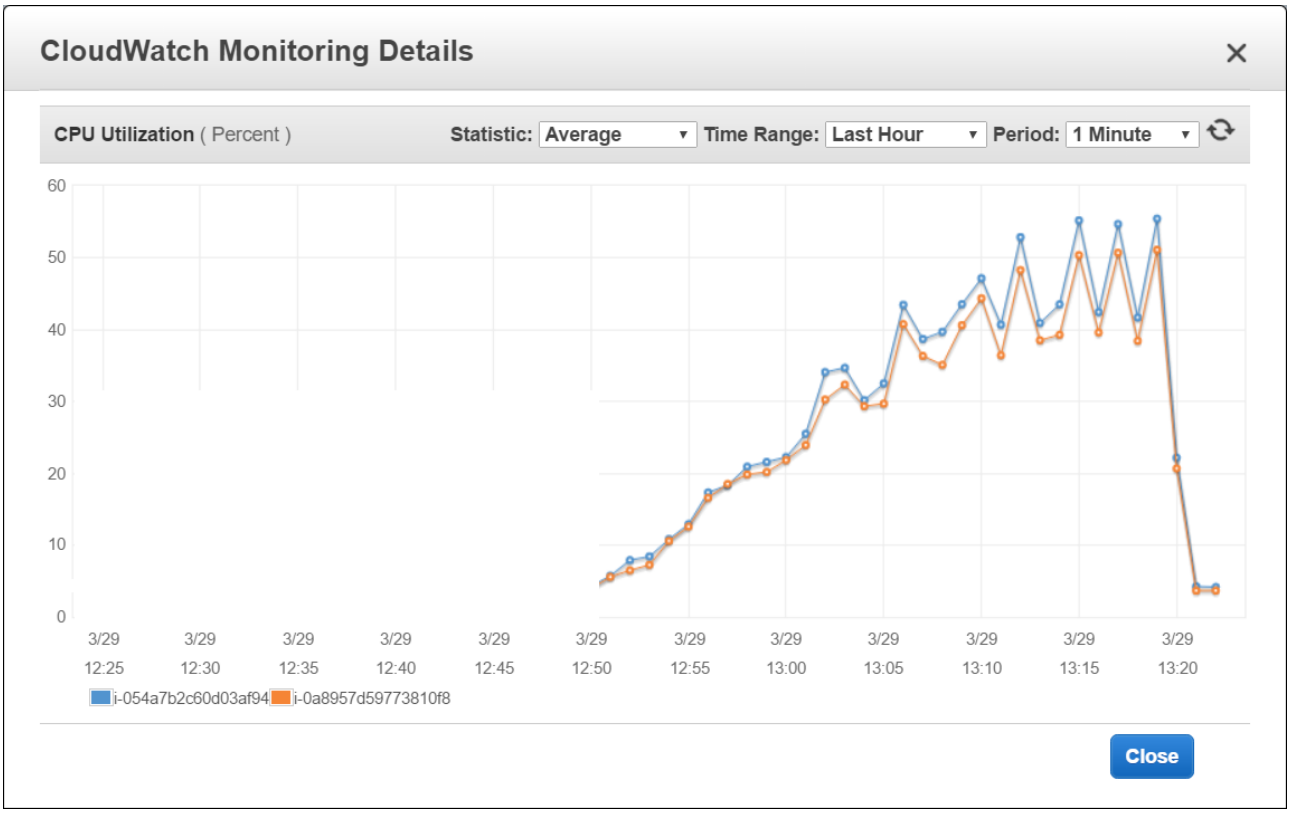

Chart 43 - EC2 CPU - scenario 4, AWS setup 3

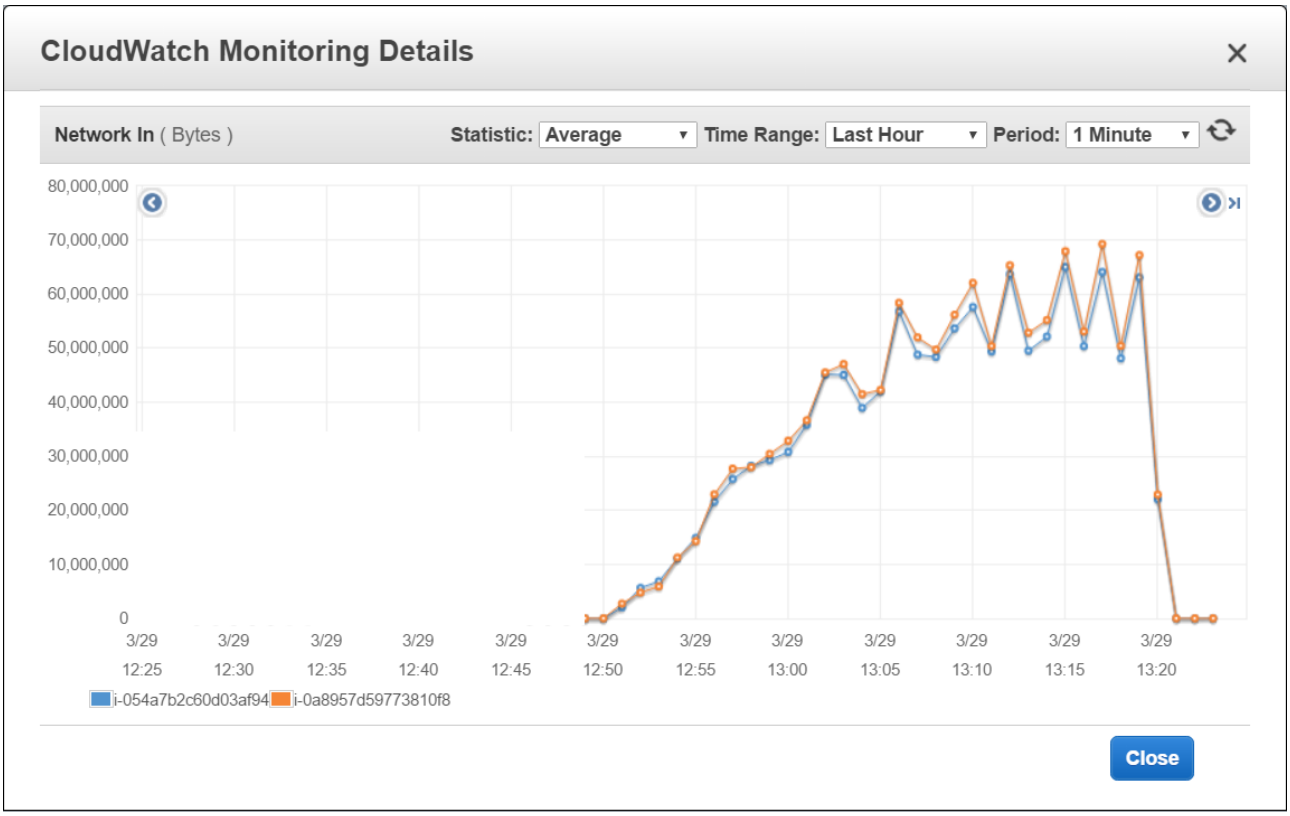

Chart 44 - EC2 Network in - scenario 4, AWS setup 3

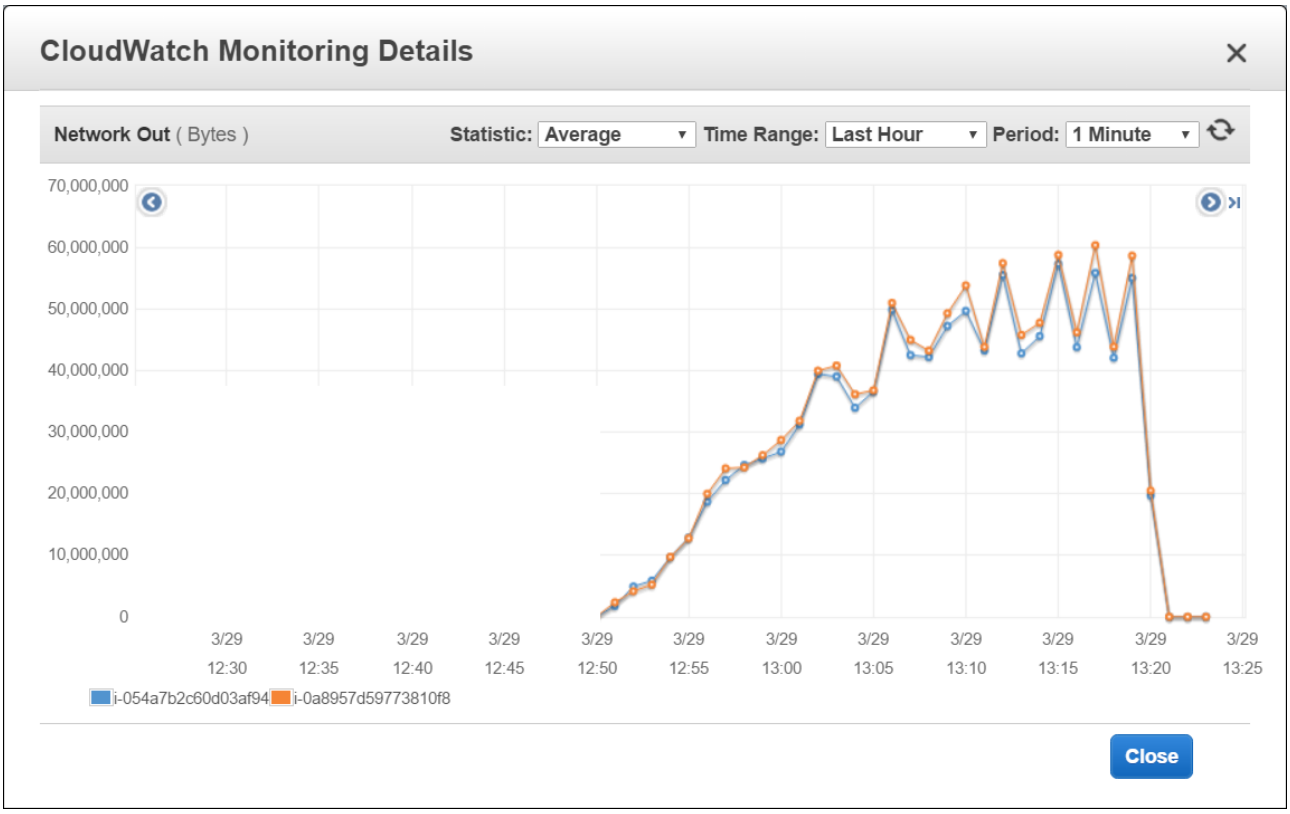

Chart 45 - EC2 Network out - scenario 4, AWS setup 3

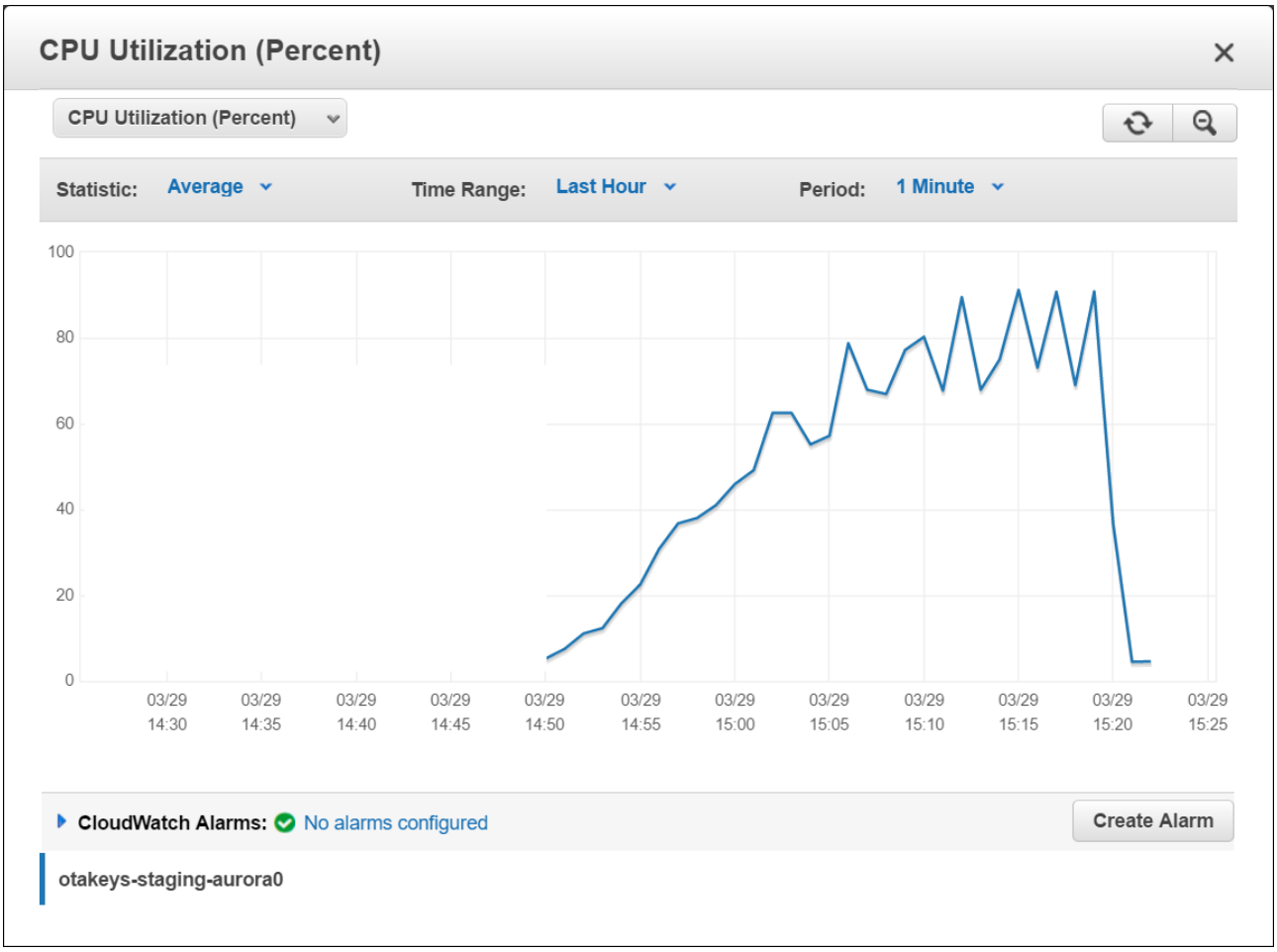

Chart 46 - RDS CPU - scenario 4, AWS setup 3

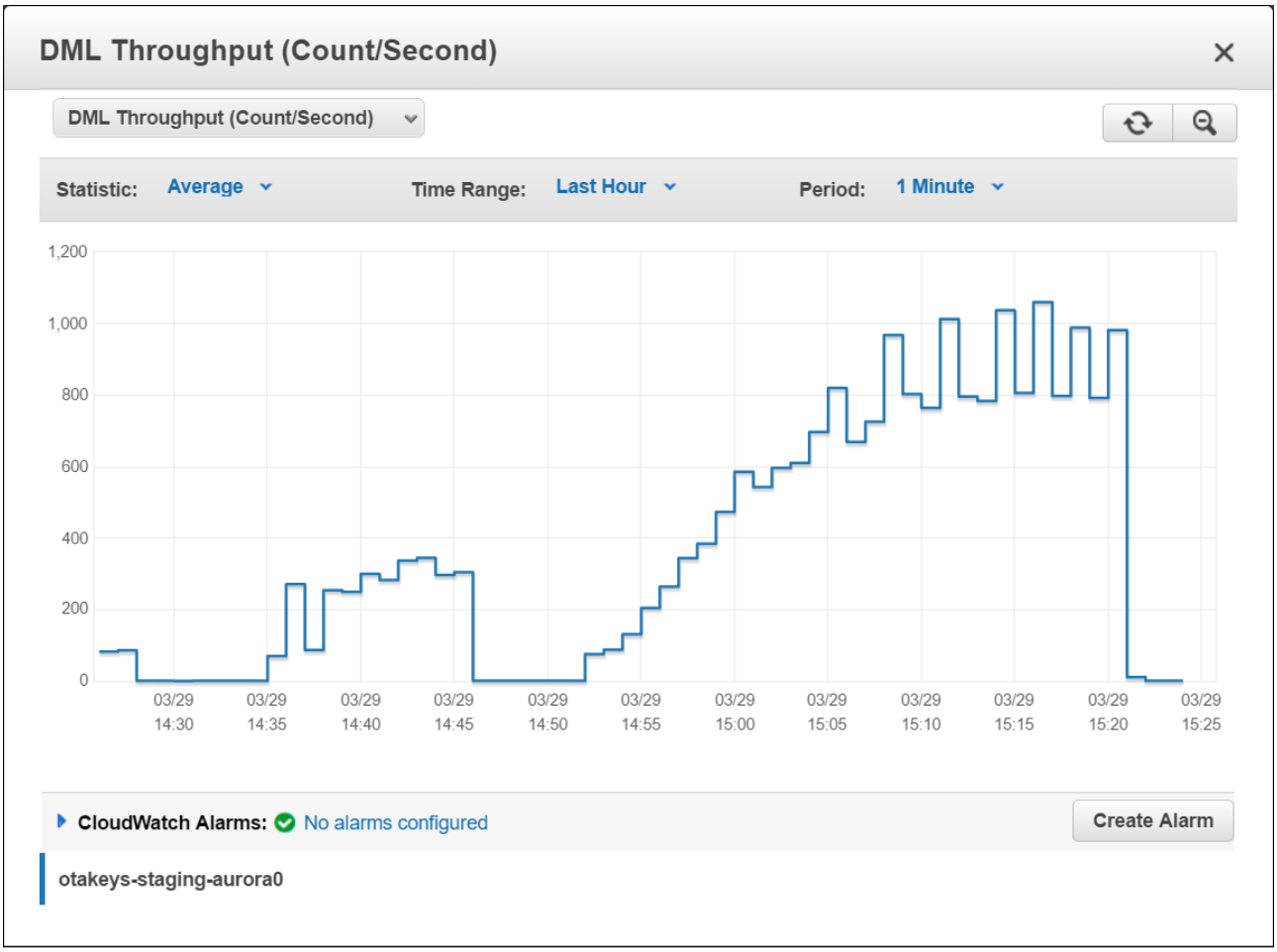

Chart 47 - RDS DML throughput - scenario 4, AWS setup 3

Scenario 5: Post synthesis

AWS setup 3: ELB --> 2x c4.large + RDS Aurora

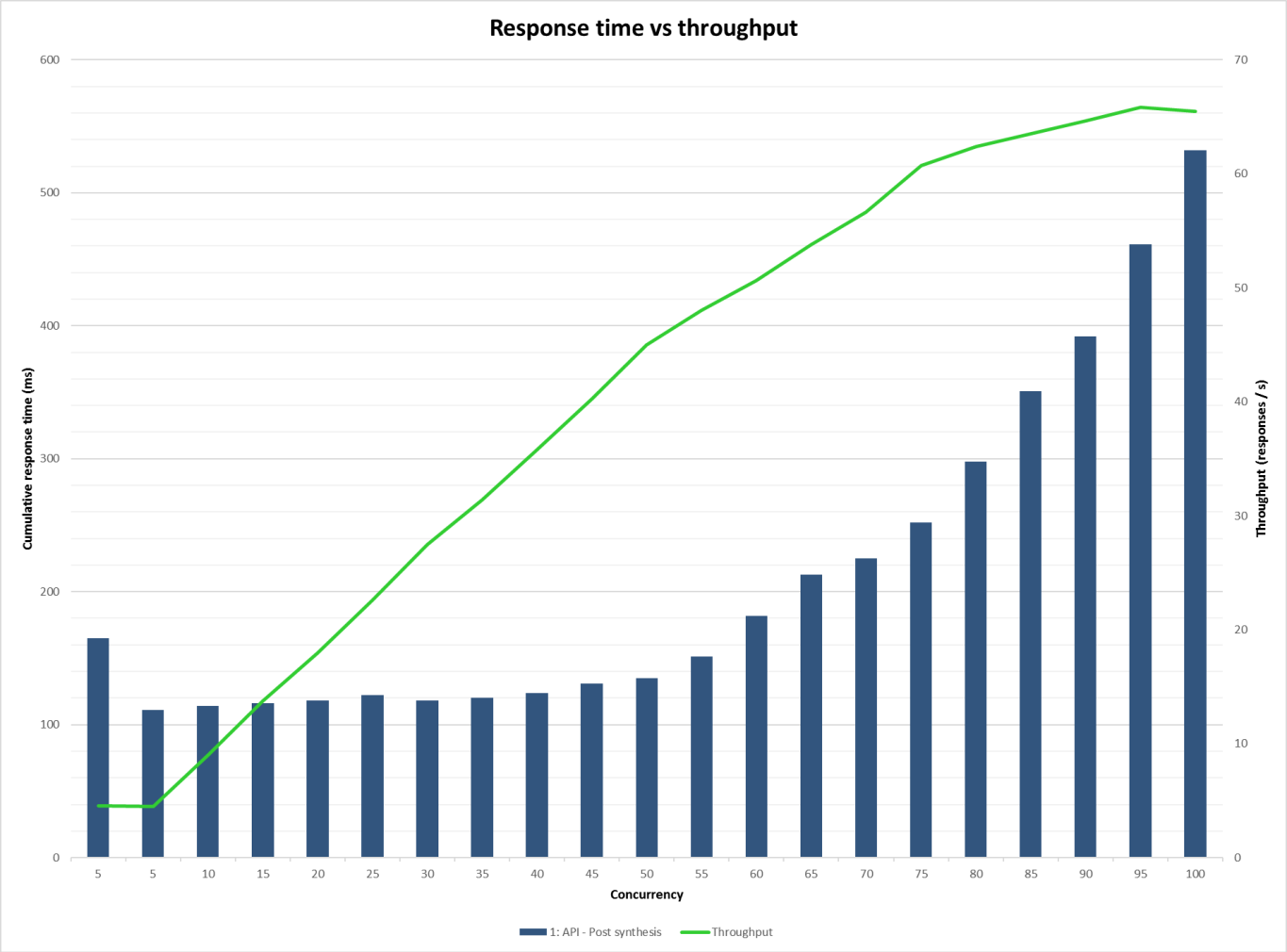

Chart 48 - Response times vs user actions - scenario 5, AWS setup 3

Posting a synthesis reaches its plateau at concurrency 95-100, with a throughput of 65 responses per second. On average, it takes around 460ms at concurrency 95 to get a response back.

No bottlenecks where seen in the monitors or the slow query log. We could try testing with higher concurrencies.

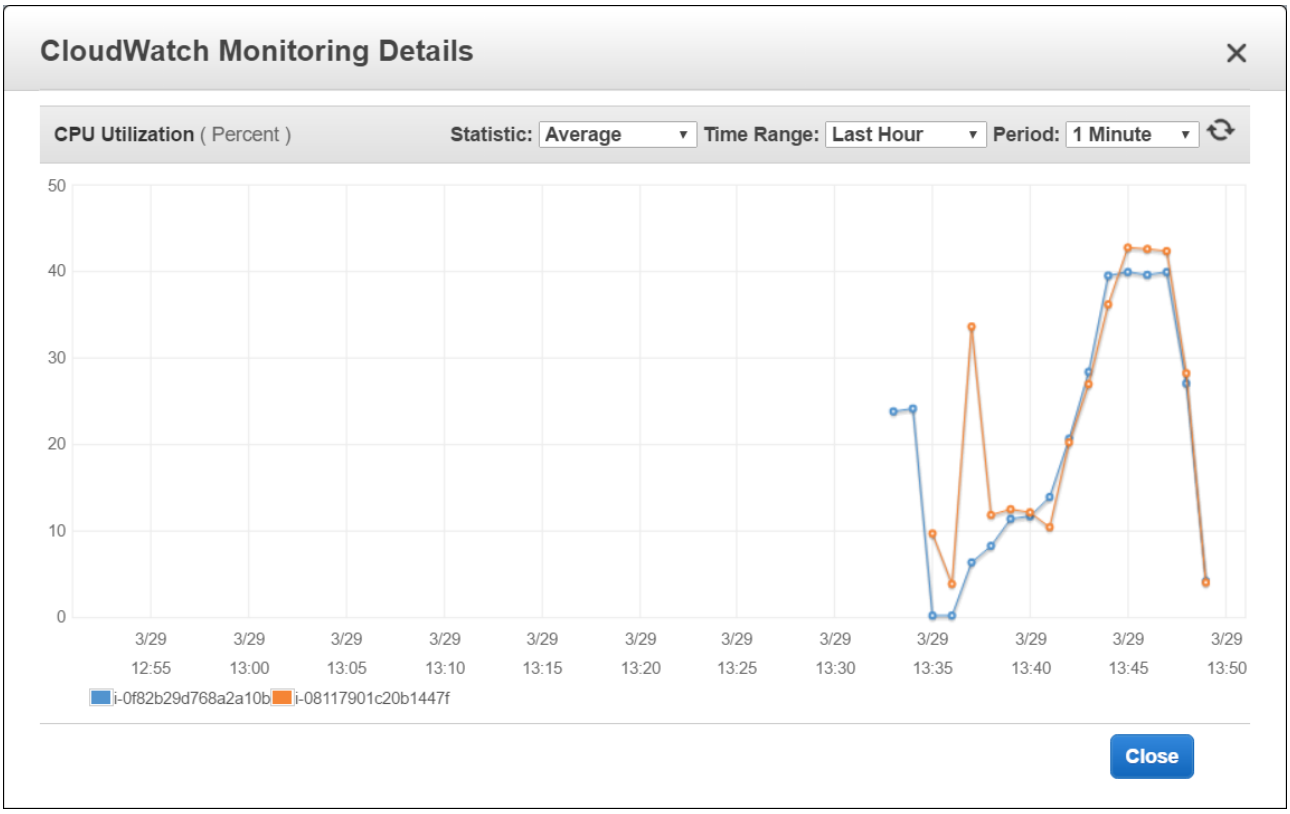

Chart 49 - EC2 CPU - scenario 5, AWS setup 3

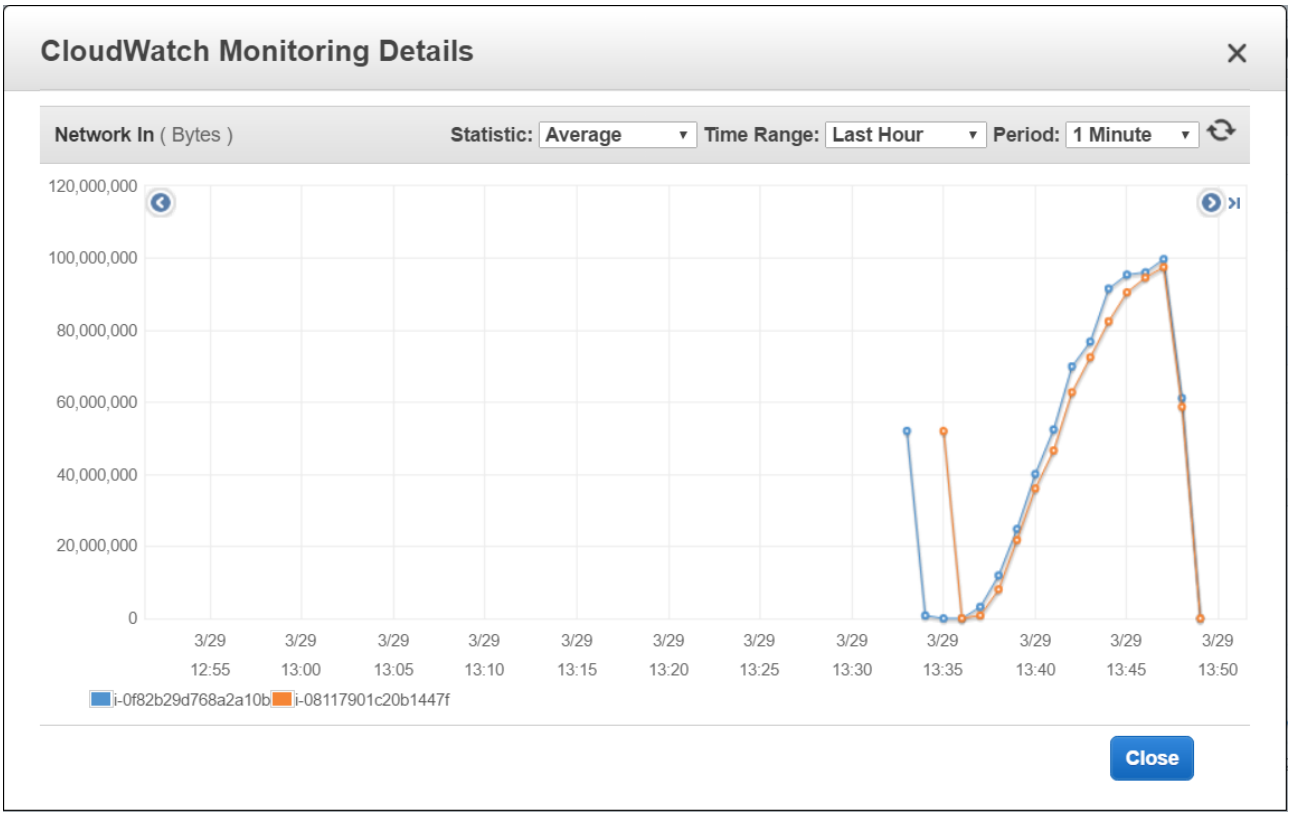

Chart 50 - EC2 Network in - scenario 5, AWS setup 3

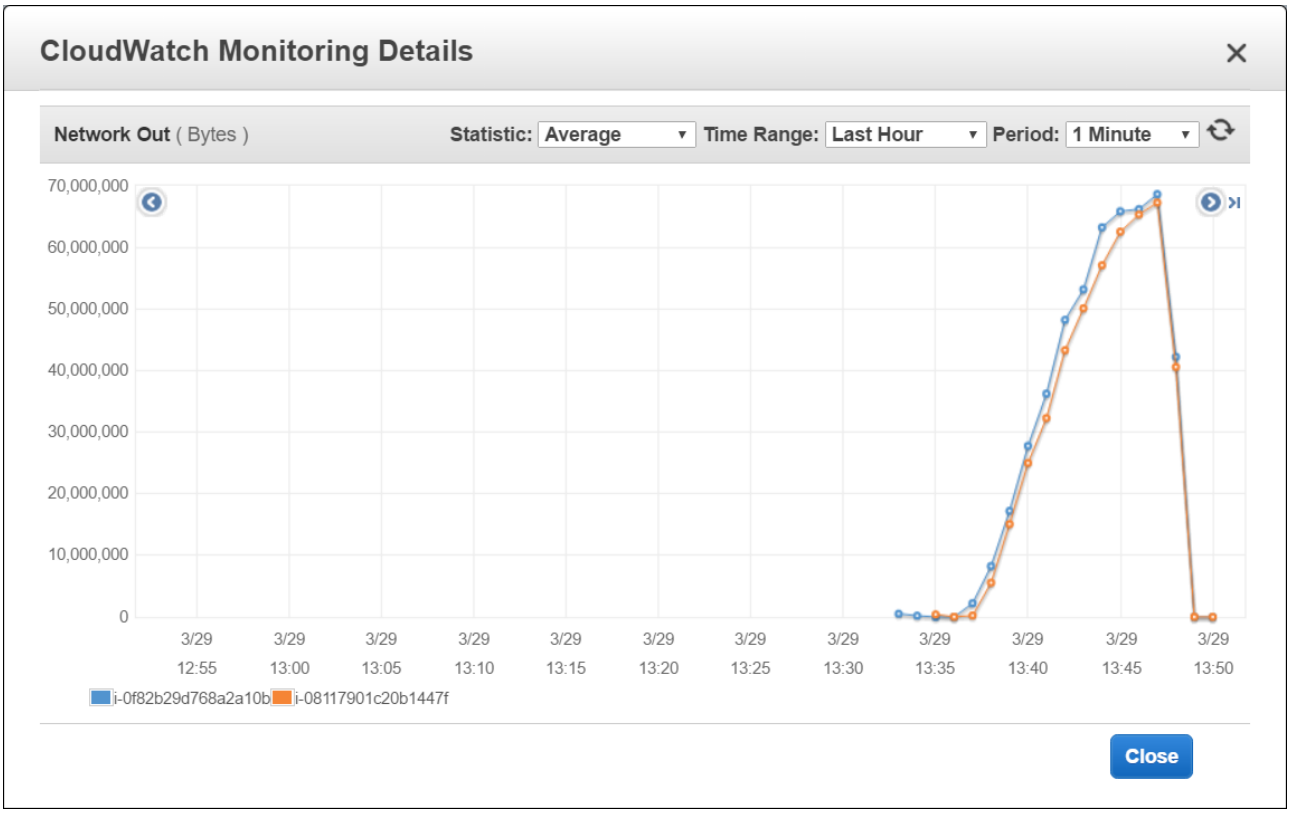

Chart 51 - EC2 Network out - scenario 5, AWS setup 3

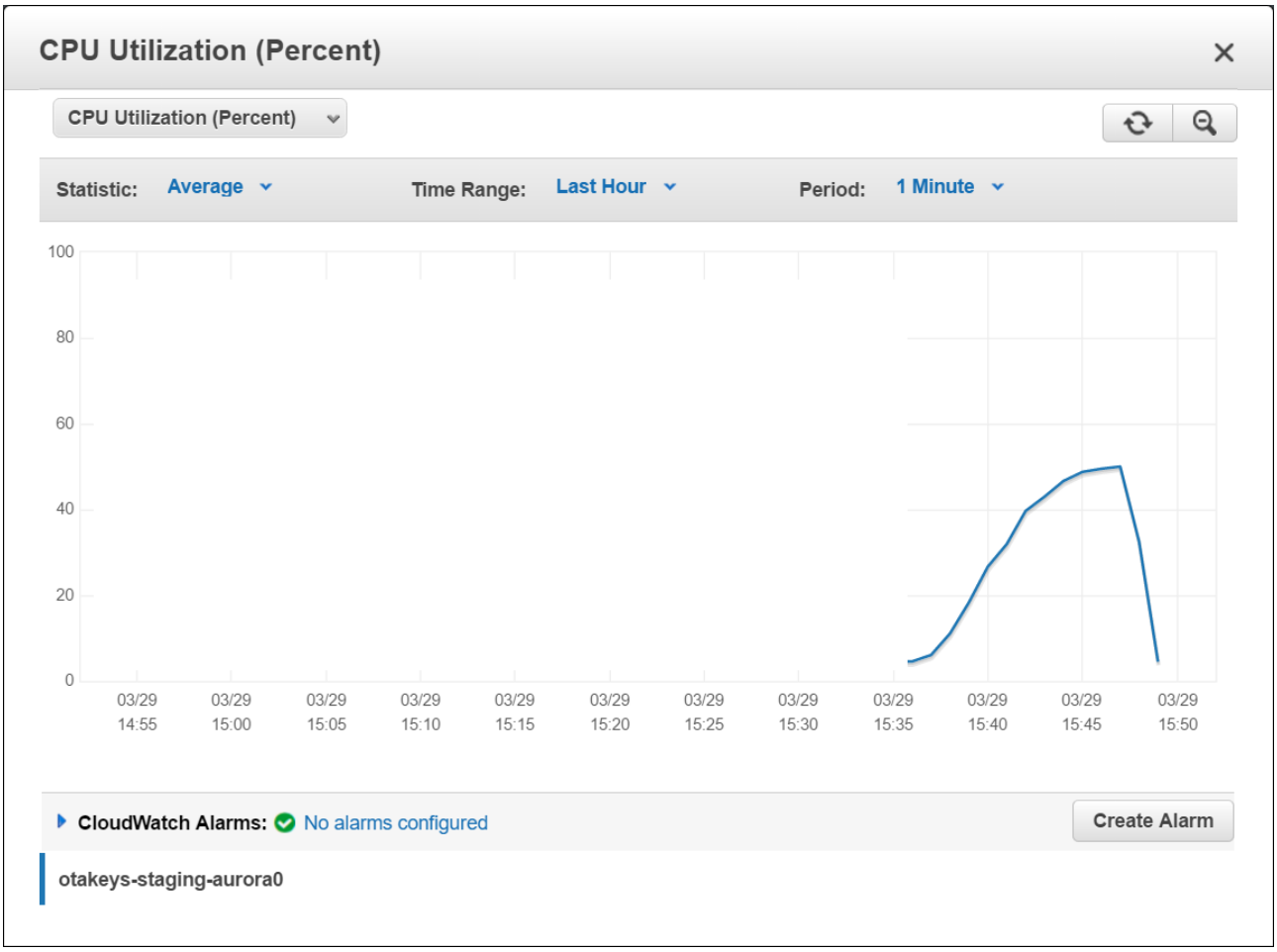

Chart 52 - RDS CPU - scenario 5, AWS setup 3

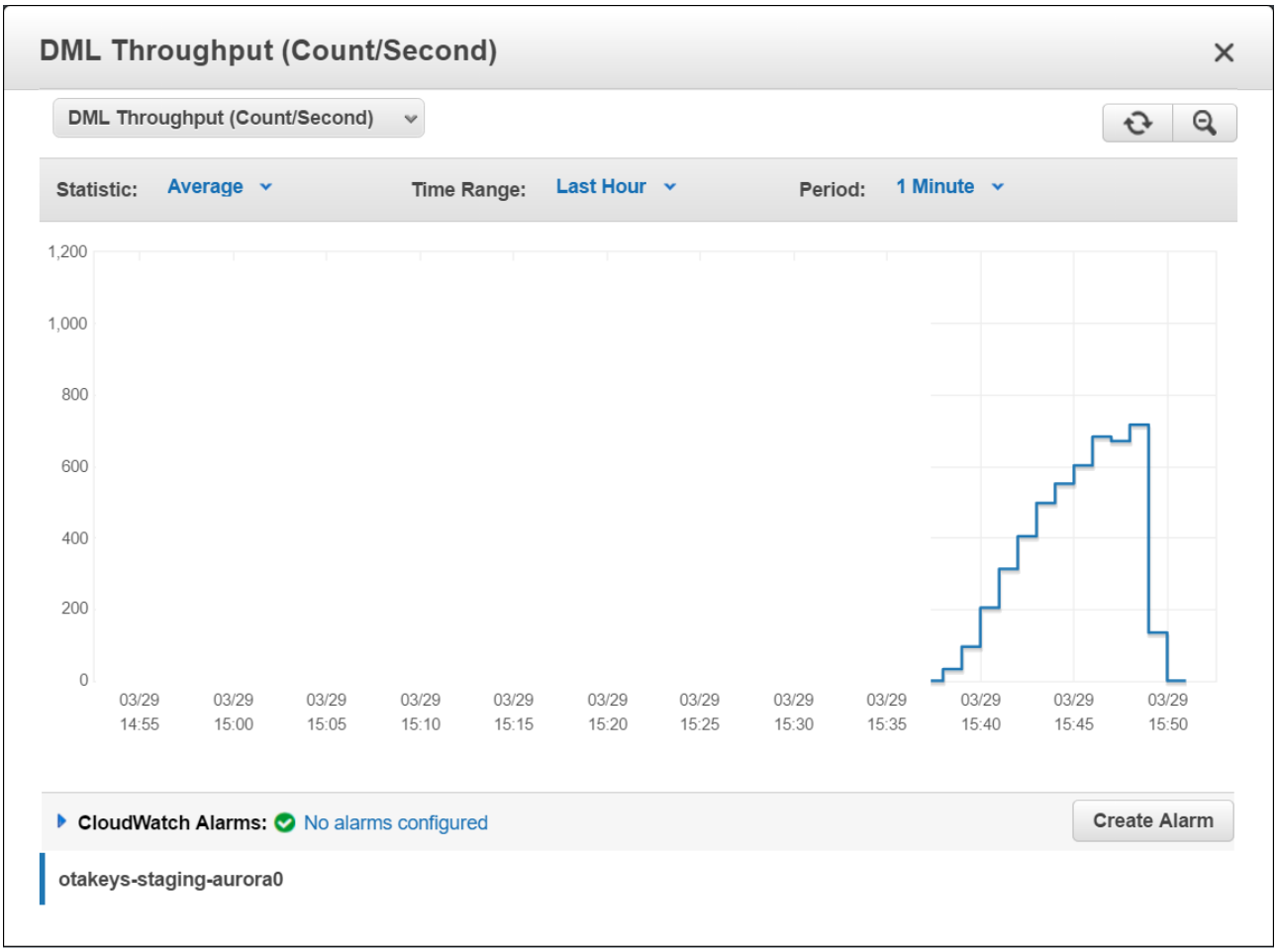

Chart 53 - RDS DML throughput - scenario 5, AWS setup 3