AWS Robust Infrastructure Tutorial

Cloud Computing

Author: Anxhelo Diko

Table of contents

INTRODUCTION

We have designed and implemented a highly available and scalable virtual architecture using AWS IaaS services through Elastic Beanstalk PaaS service. We have deployed a distributed client/server and elastic application. The application is a simple dynamic blog post website , written on laravel which is a PHP framework.

In Figure 1 is shown the architecture of our infrastructure.Our EC2 instances (Web servers) can run in multiple availability zones. There is also an Elastic Load Balancer in front which load balances requests based on availability of zones and load. The application logic part (tire) is hosted in the EC2s (Web server) with Auto Scaling capability that provides the ability to scale-out or in depending on the amount of load that our application has to handle. The Database (DB) part (tire) is also separated and can be available in different zones. A synchronized mechanism is used that our DBs synchronously replicate the data to avoid a single point of failure.

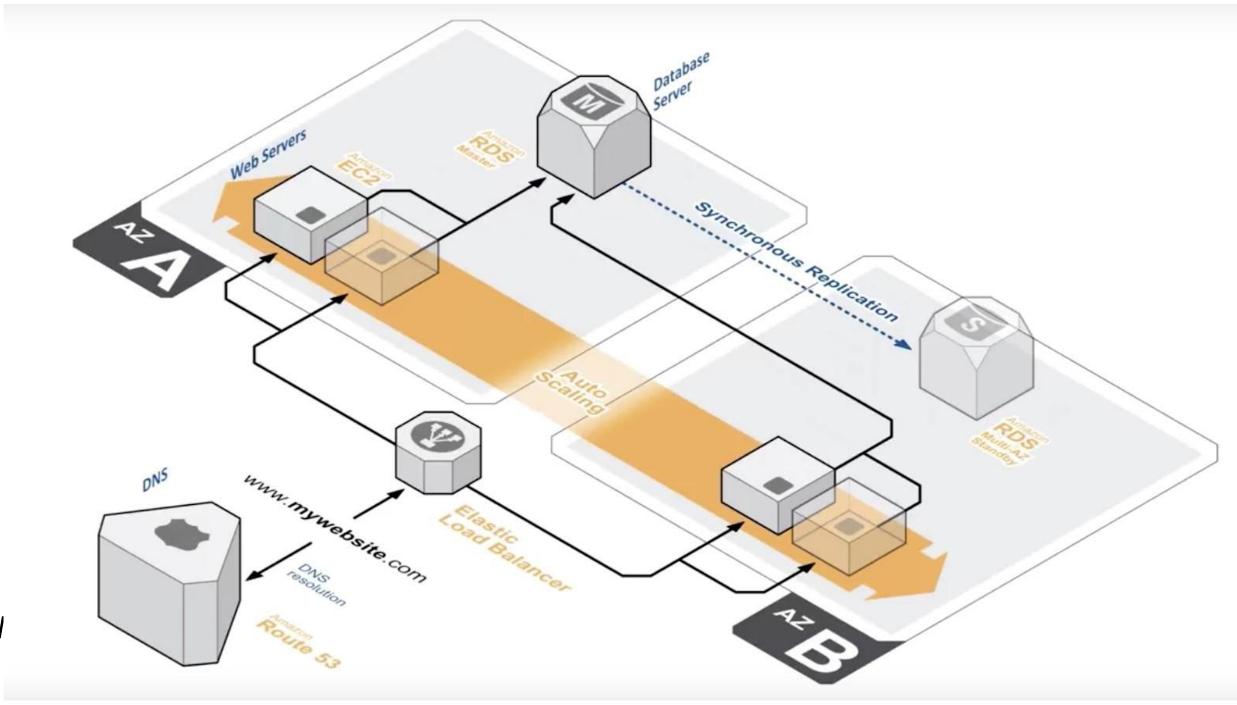


Figure 1. *Virtual*

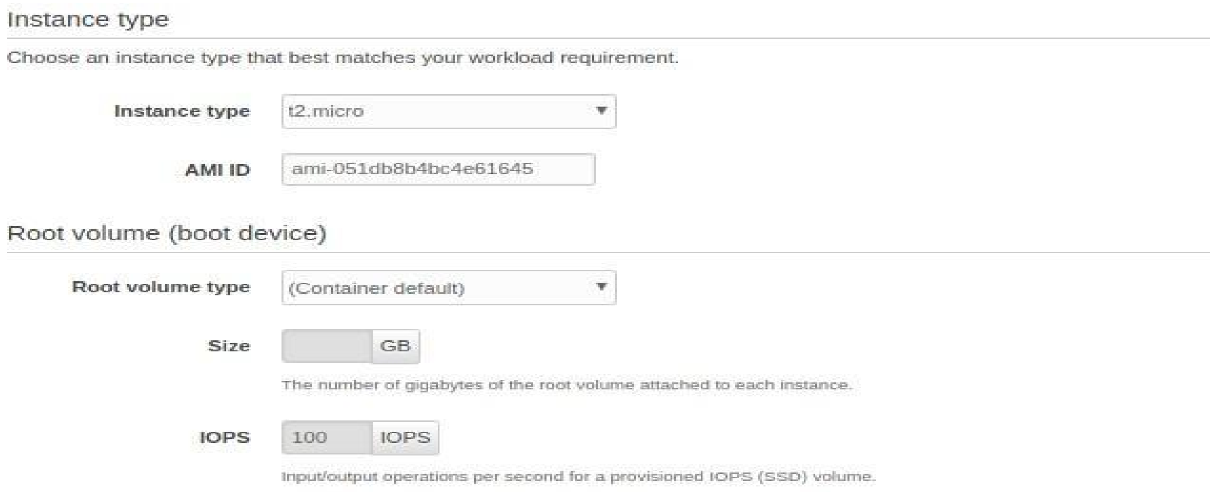
2. VIRTUAL INFRASTRUCTURE

The power of virtualisation to better manage IT capacity, provide better service levels, and streamline IT processes. The virtual infrastructure lets you share your physical resources of multiple machines across your entire infrastructure. A virtual machine lets you share the resources of a single physical computer across multiple virtual machines for maximum efficiency. Resources are shared across multiple virtual machines.

2.1 Infrastructure Components

2.1.1 EC2 instances - An EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS) infrastructure. We use t2.micro type of instances which have the features :

- Linux Operating System
- High frequency Intel Xeon processors
- Burstable CPU, governed by CPU Credits, and consistent baseline performance
- Lowest-cost general purpose instance type, and Free Tier eligible*
- Balance of compute, memory, and network resources
- up to 3.3 GHz Intel Scalable Processor



Instance type

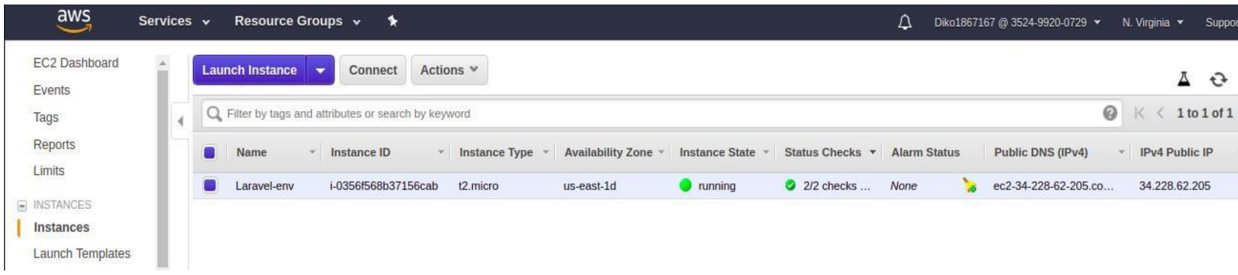
Choose an instance type that best matches your workload requirement.

Instance type	t2.micro
AMI ID	ami-051db8b4bc4e61645

Root volume (boot device)

Root volume type	(Container default)
Size	GB
	The number of gigabytes of the root volume attached to each instance.
IOPS	100 IOPS
	Input/output operations per second for a provisioned IOPS (SSD) volume.

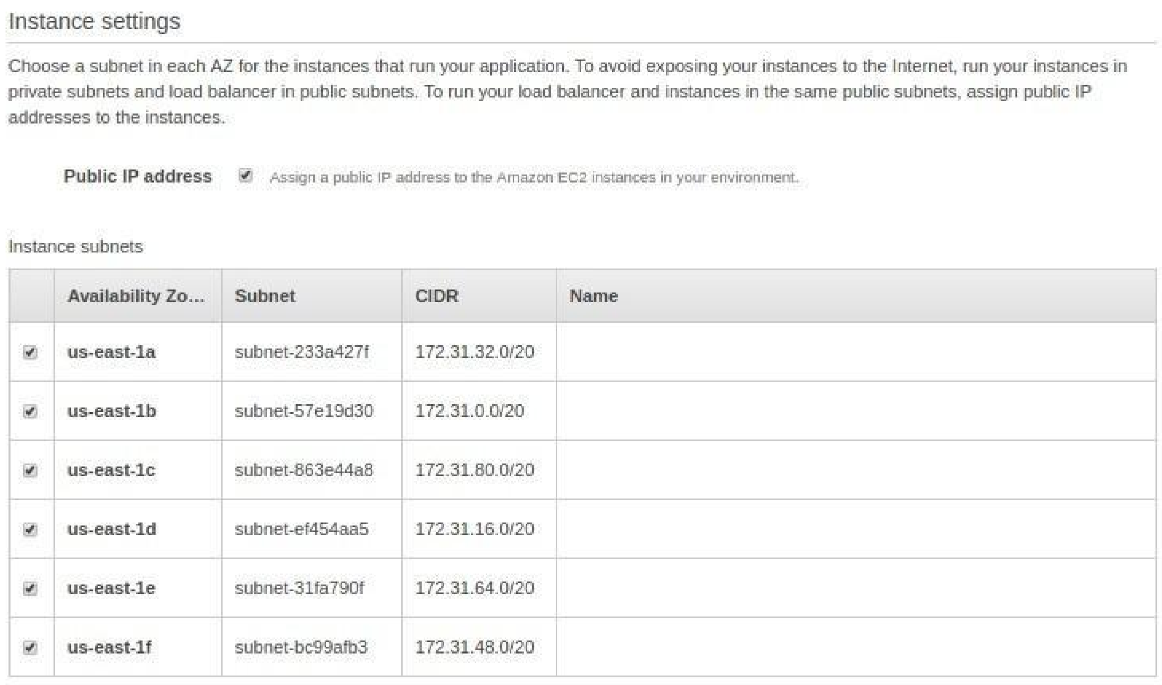
One EC2 insta



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
Laravel-env	i-0356f568b37156cab	t2.micro	us-east-1d	running	2/2 checks ...	None	ec2-34-228-62-205.co...	34.228.62.205

2.1.2 Amazon Virtual Private Cloud (Amazon VPC) We used Amazon VPC to create a secure network for our Elastic Beanstalk application and related AWS resources. When we created our environment, we chose which VPC, subnets, and security groups we used for our application instances and application load balancer.
We configured VPC based on the following requirements :
Internet Access – Instances must have access to the Internet through *Public Subnet* –

Instances have a public IP address and use an Internet Gateway to access the Internet


Instance settings

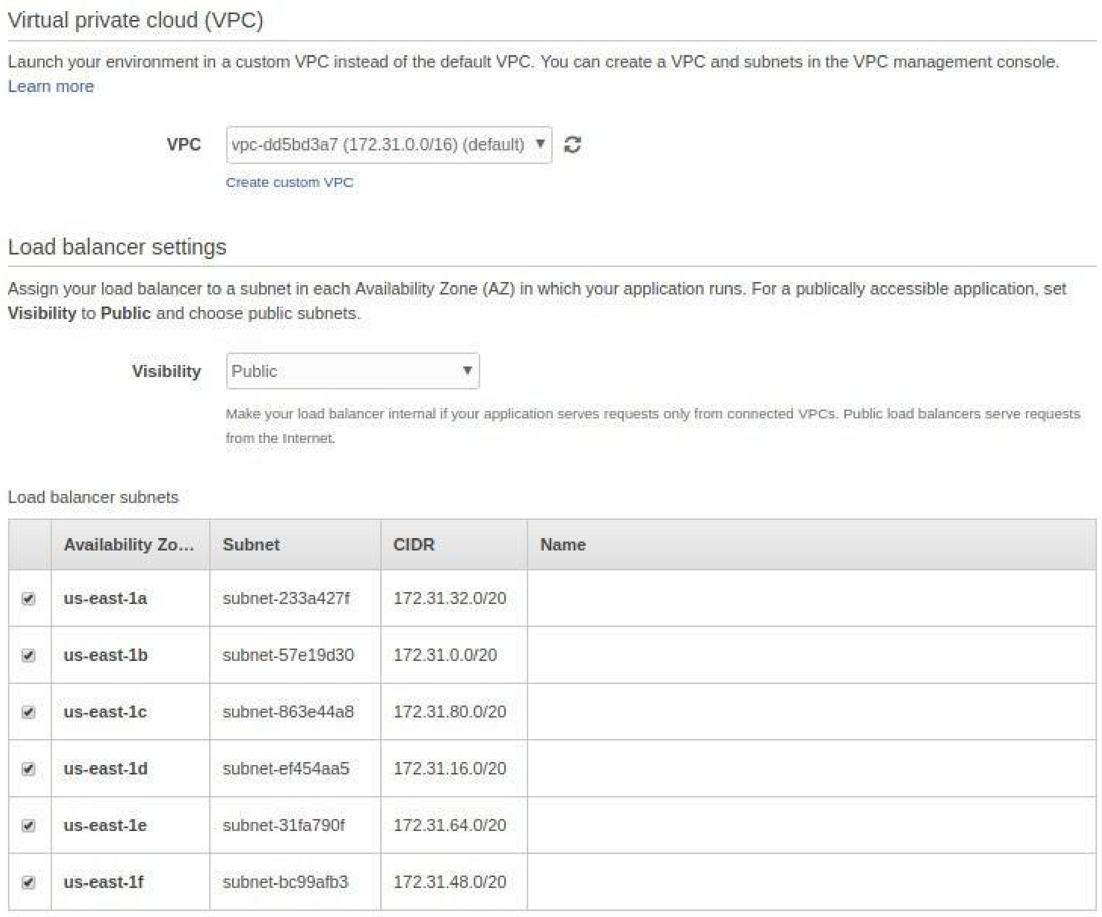
Choose a subnet in each AZ for the instances that run your application. To avoid exposing your instances to the Internet, run your instances in private subnets and load balancer in public subnets. To run your load balancer and instances in the same public subnets, assign public IP addresses to the instances.

Public IP address ✓ Assign a public IP address to the Amazon EC2 instances in your environment.

Instance subnets

	Availability Zo...	Subnet	CIDR	Name
✓	us-east-1a	subnet-233a427f	172.31.32.0/20	
✓	us-east-1b	subnet-57e19d30	172.31.0.0/20	
✓	us-east-1c	subnet-863e44a8	172.31.80.0/20	
✓	us-east-1d	subnet-ef454aa5	172.31.16.0/20	
✓	us-east-1e	subnet-31fa790f	172.31.64.0/20	
✓	us-east-1f	subnet-bc99afb3	172.31.48.0/20	

VPC with Publ
We have chosen the default VPC so Elastic Beanstalk manages networking by creating a security group for the load balancer that allows traffic on port 80 from the Internet, and a security group for the application instances that allows traffic from the load balancer's security group.



Virtual private cloud (VPC)

Launch your environment in a custom VPC instead of the default VPC. You can create a VPC and subnets in the VPC management console. Learn more

VPC vpc-dd5bd3a7 (172.31.0.0/16) (default)

Create custom VPC

Load balancer settings

Assign your load balancer to a subnet in each Availability Zone (AZ) in which your application runs. For a publically accessible application, set **Visibility** to **Public** and choose public subnets.

Visibility Public

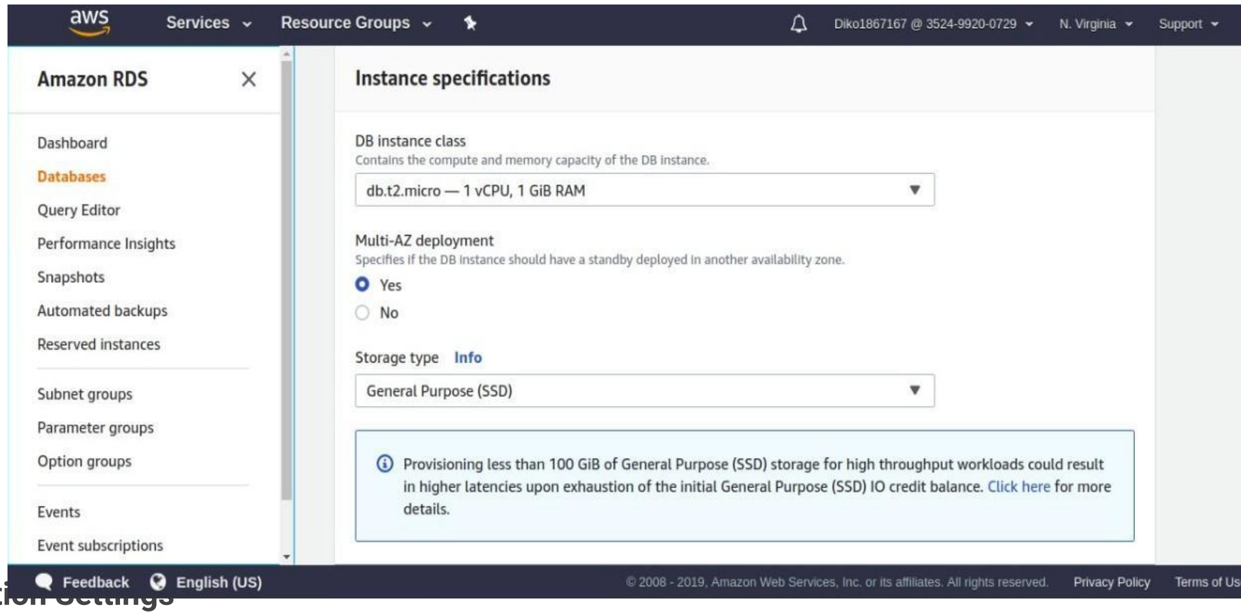
Make your load balancer internal if your application serves requests only from connected VPCs. Public load balancers serve requests from the Internet.

Load balancer subnets

	Availability Zo…	Subnet	CIDR	Name
✓	**us-east-1a**	subnet-233a427f	172.31.32.0/20	
✓	**us-east-1b**	subnet-57e19d30	172.31.0.0/20	
✓	**us-east-1c**	subnet-863e44a8	172.31.80.0/20	
✓	**us-east-1d**	subnet-ef454aa5	172.31.16.0/20	
✓	**us-east-1e**	subnet-31fa790f	172.31.64.0/20	
✓	**us-east-1f**	subnet-bc99afb3	172.31.48.0/20	

2.1.3 Fully Managed MySQL on Amazon RDS - Amazon RDS is available on several database instance types and provided us with six familiar database engines from what we chose to work with MySQL. It manages time-consuming database administration tasks including backups, software patching, monitoring, scaling and replication. The basic building block of Amazon RDS is the *DB instance*. Each DB instance runs a *DB engine*. We run our DB instance in two Availability Zones, an option called a Multi-AZ deployment. One Master RDS instance and a StandBy RDS instance

Creating an Amazon RDS StandBy Instance



Instance specifications

DB instance class
Contains the compute and memory capacity of the DB instance.
db.t2.micro — 1 vCPU, 1 GiB RAM

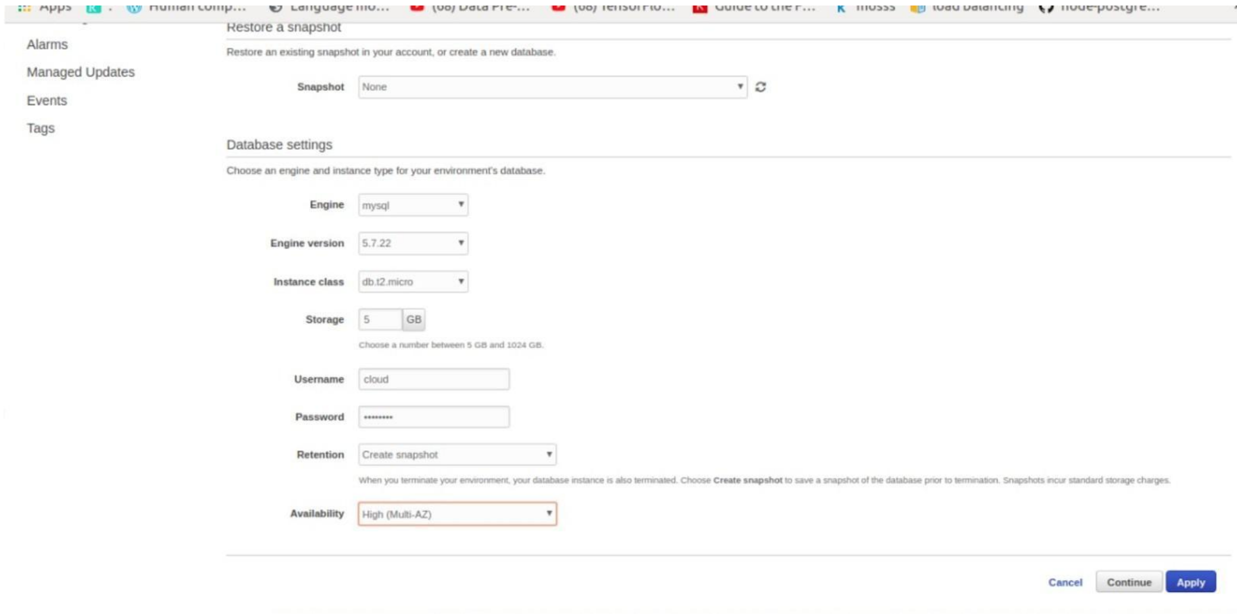
Multi-AZ deployment
Specifies if the DB instance should have a standby deployed in another availability zone.
[✓ Yes] No

Storage type Info
General Purpose (SSD)

Provisioning less than 100 GiB of General Purpose (SSD) storage for high throughput workloads could result in higher latencies upon exhaustion of the initial General Purpose (SSD) IO credit balance. Click here for more details.

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

DB Configuration Settings



Restore a snapshot
Restore an existing snapshot in your account, or create a new database.

Snapshot	None

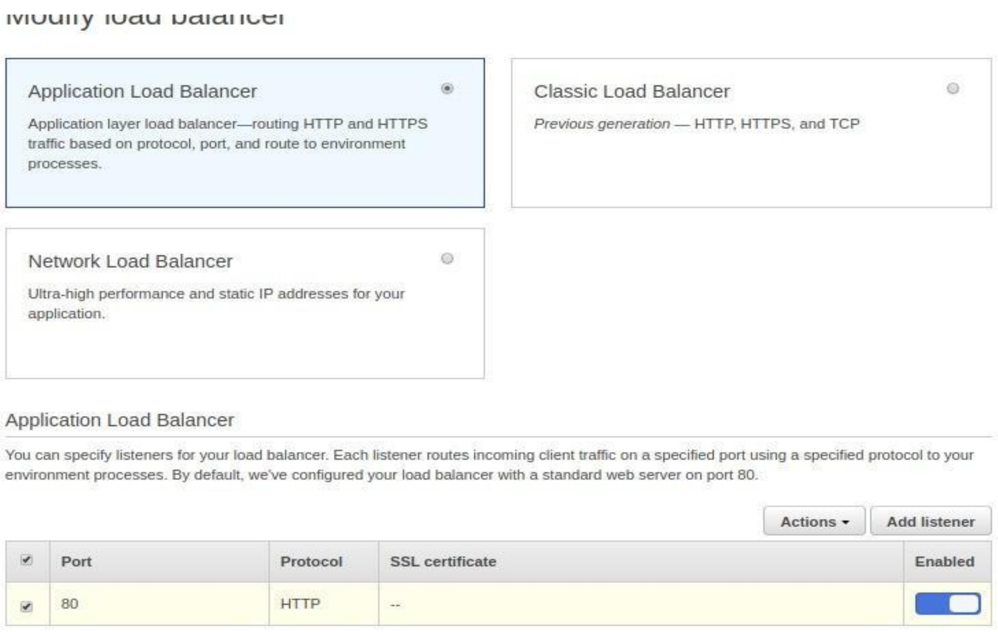
Database settings
Choose an engine and instance type for your environment's database.

Engine	mysql
Engine version	5.7.22
Instance class	db.t2.micro
Storage	5 GB
Username	cloud
Password	••••••••
Retention	Create snapshot
Availability	High (Multi-AZ)

Choose a number between 5 GB and 1024 GB.

When you terminate your environment, your database instance is also terminated. Choose **Create snapshot** to save a snapshot of the database prior to termination. Snapshots incur standard storage charges.

2.1.4 Elastic Load Balancer (Application Load Balancer) - A load balancer serves as the single point of contact for clients. Clients send requests to the load balancer, and the load balancer sends them to EC2 instances, in two or more Availability Zones.
We used *Elastic Beanstalk environment*, to configure our Application Load Balancer to direct traffic for certain paths to a different port on our web server instances.
Through the Elastic Beanstalk console we configured our Application Load Balancer's listeners (port 80), processes(default), and listener rules(default), during environment creation.


Modify load balancer

Application Load Balancer ●
Application layer load balancer—routing HTTP and HTTPS traffic based on protocol, port, and route to environment processes.

Classic Load Balancer ○
Previous generation — HTTP, HTTPS, and TCP

Network Load Balancer ○
Ultra-high performance and static IP addresses for your application.

Application Load Balancer

You can specify listeners for your load balancer. Each listener routes incoming client traffic on a specified port using a specified protocol to your environment processes. By default, we've configured your load balancer with a standard web server on port 80.

✓	Port	Protocol	SSL certificate	Enabled
✓	80	HTTP	--	●

2.2 The interaction between components

Our infrastructure, as stated above, is based on a group of components which strongly interact with each other. *Figure 1* shows how components are connected with each other and the interaction is quite intuitive.
As the requests come to our website they are first directed to the *load balancer* then load balancer redirect them to one of the running *EC2 (Web server) instances* where our application logic is running. If there are http put/push requests or delete then the app communicates with our *RDS instances* to edit the data that are found there or to add new data.

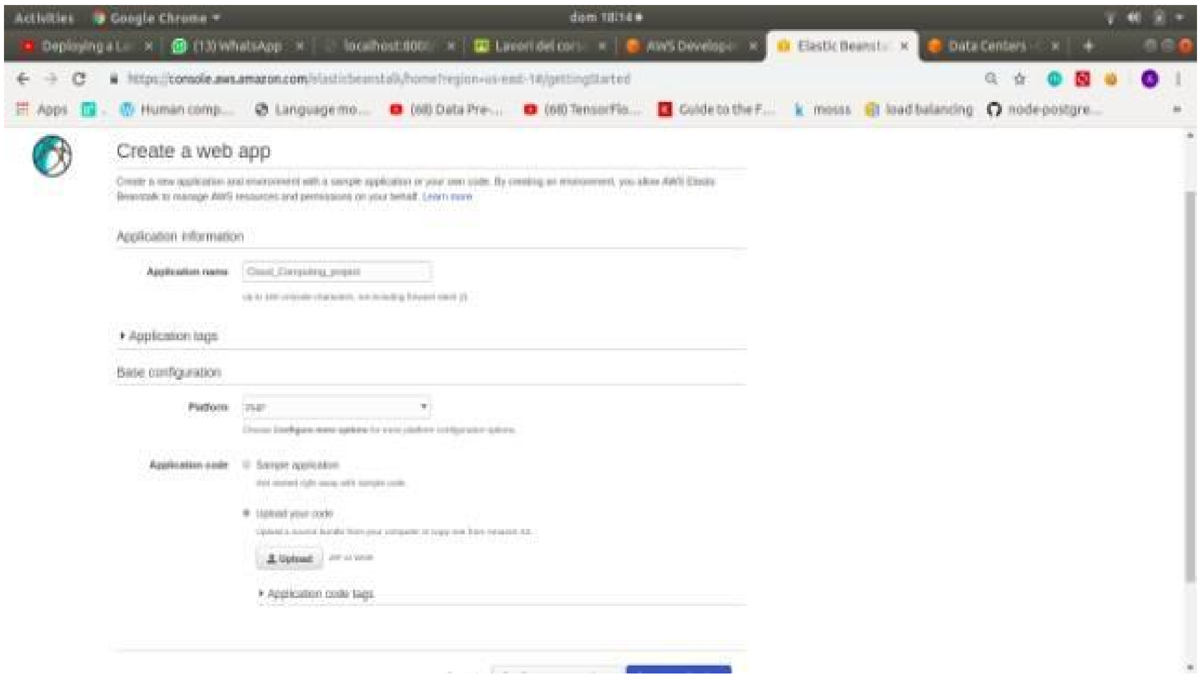
There are five other components/processes that take place: auto *scaling*, *synchronous replication, network, security group* and there is even a hidden component *CloudWatch* , without whom many of the characteristics of our infrastructure would not be possible.
Auto scaling policy we have followed is of a simple scaling type. It is configured with some upper threshold and lower threshold for scaling our infrastructure. To be able to monitor overall scaling, is needed a constant monitoring of the scaling metric plus an alarm which is configured so when the thresholds are reached, the alarm breaches and the scaling operations are executed as needed.
CloudWatch is the component which shows us the statistics about the metrics it is monitoring, including even our auto-scaling metric. It has a really important role in this component interaction.
Synchronous replication comes handy every time we modify components of our DB. It replicates its newly added data to some selected instance in some different availability zones so if our current DB goes down or even the current availability zone goes down our app will still be live and with all its data, so we have high availability.
The network (VPC) is configured to assign public ip`s to our instances and selected the different subnets we can use, this allows us to operate in different availability zones inside the Region.
Meanwhile the *security group* works as a firewall to select in-coming and out-going traffic in and from our app.

3 Application Design and Deployment

To deploy our app we used *elastic beanstalk*, that is a PaaS which makes it easy to monitor and manage EC2 instances and RDS instances that are part of our virtual infrastructure. We created a Three-tier, Client/Server system architecture. The main reason why we decided for this kind of architecture is because our web application uses a relational database. We had to separate the application logic from data storage, since relational databases can not be scalable as they are created to run on a single machine, so the scalability of our application depends only on the scalability of the web server.

3.1 Application components
We developed our application based on the MVC web development framework, its main components are views, controllers and models. The main focus of this project is the model and its configuration which is very crucial for making our app scalable.
Modeling on MVC deals with the *data model, business logic* and *DB configuration*.
All our models are based on relational databases so we had to find the best way to make our app elastic. And is only one way with relational databases, a three-tier Client/Server architecture for separating the data tire from application logic.

3.2 Deployment
For deploying our application we selected the platform and the option to upload your code. The platform selection specifies the environment that was built inside the EC2 instances that we used as web server and the type of EC2 instances.


Create a web app

Application information

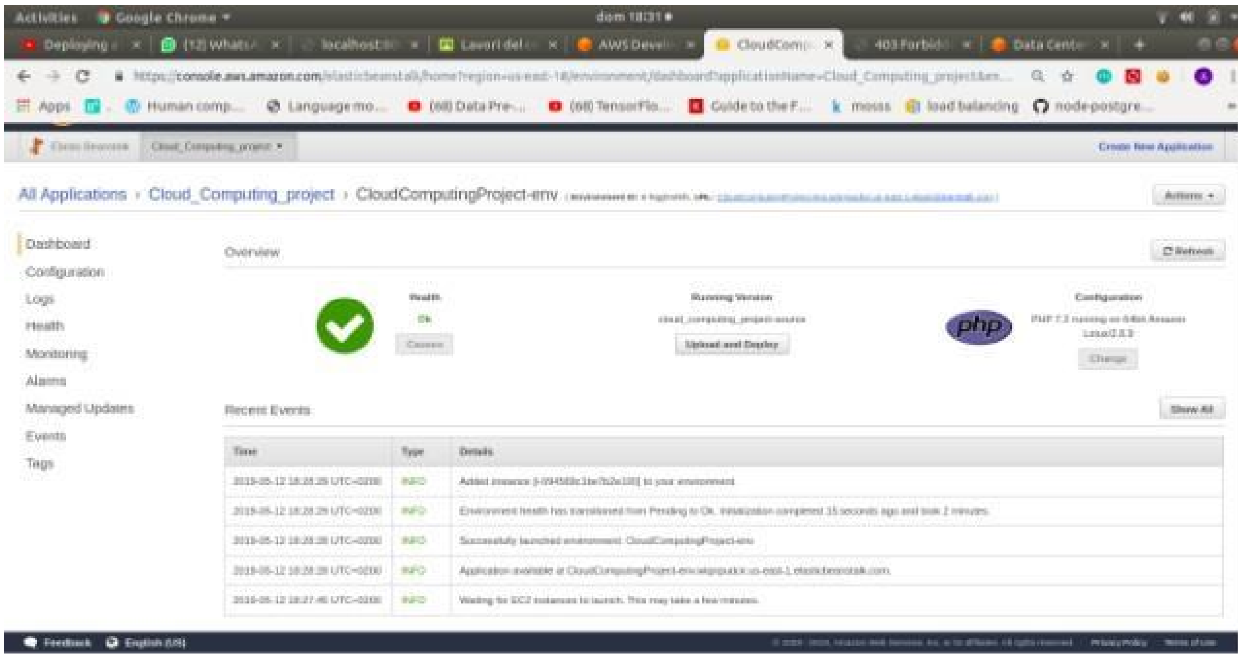
Application name: Cloud_Computing_project

Base configuration

Platform: PHP

Application code: Sample application / [✓ Upload your code]

The result after app deployment



All Applications › Cloud_Computing_project › CloudComputingProject-env

Recent Events

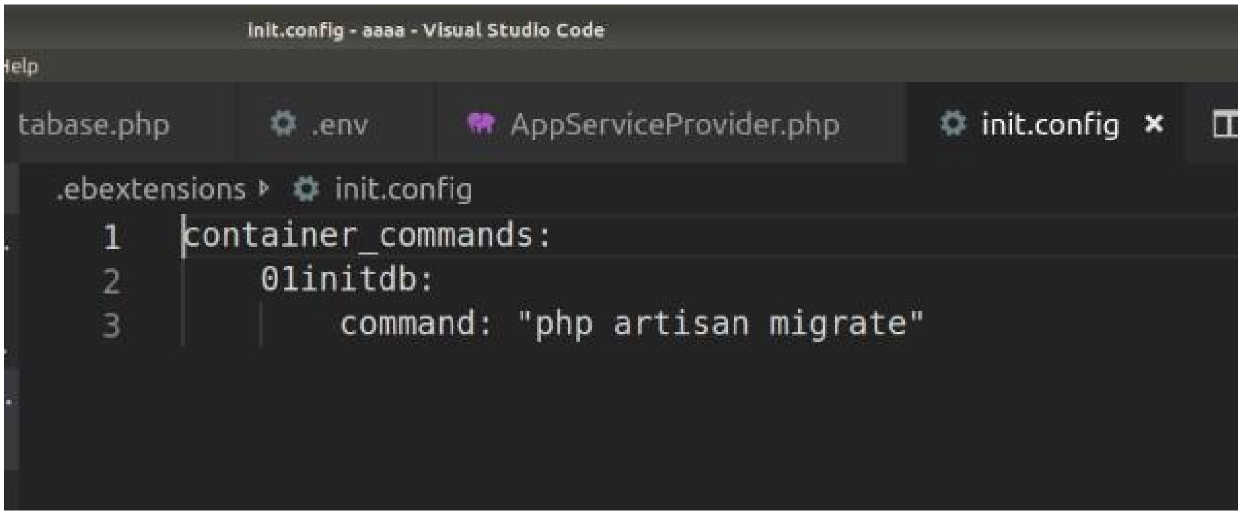
Time	Type	Details
2019-05-12 18:28:29 UTC+0200	INFO	Added instance [i-094568c1be7b2e100] to your environment.
2019-05-12 18:28:29 UTC+0200	INFO	Environment health has transitioned from Pending to Ok. Initialization completed 15 seconds ago and took 2 minutes.
2019-05-12 18:28:28 UTC+0200	INFO	Successfully launched environment: CloudComputingProject-env
2019-05-12 18:28:28 UTC+0200	INFO	Application available at CloudComputingProject-env.wkpipukx.us-east-1.elasticbeanstalk.com.
2019-05-12 18:27:46 UTC+0200	INFO	Waiting for EC2 instances to launch. This may take a few minutes.

The website was uploaded with success. What the Beanstalk did was:

- It created a security group (based on our instructions and rules)
- It created EC2 instances (the type we selected and with the needed configuration to deploy our website)
- It assigned an elastic IP to our website (we configured the network for this part)
- It tested if the EC2 instances were available
- It finished the Deployment

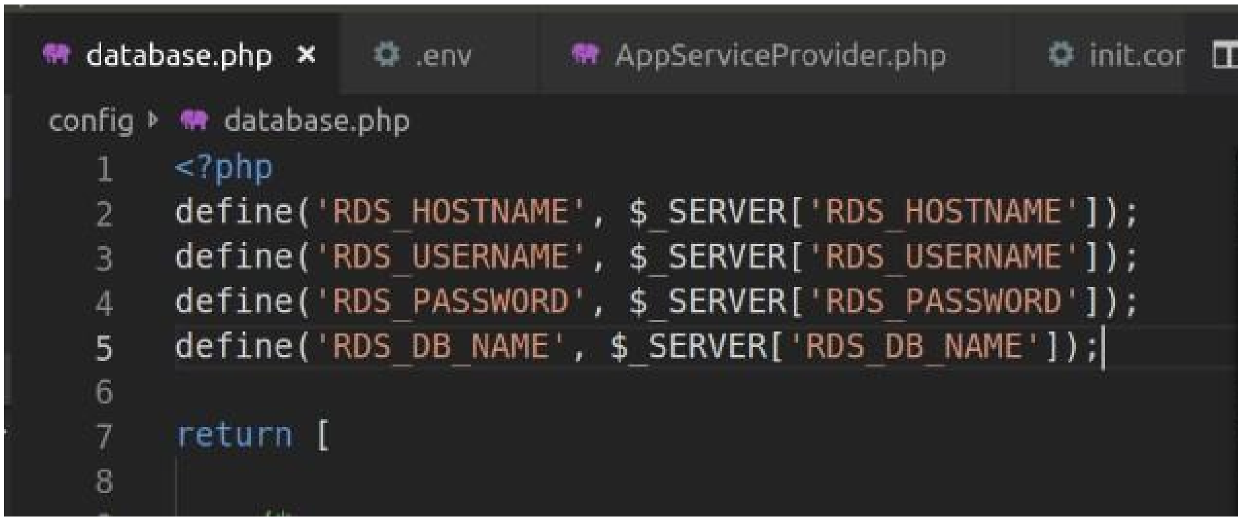
It was not easy to deploy because now the response to our request to access the website with the url provided by amazon responded with : **FORBIDEN**
This happened because the web servers (installed in EC2 instances) needed some configurations. To fix that problem :

- We defined the root directory which says where the requests should be directed.
- We created some global variables inside our database configuration in order for aws (RDS databases) to read and use them as credentials.
- After creating those variables we created a hidden folder named .*ebextension* and inside it an *init.config* file with the commands that need to be executed on our machines. Our web server connected to the RDS databases and did actions like “php artisan migrate” which created all the specified tables in our model to our RDS instance.



```
container_commands:
    01initdb:
        command: "php artisan migrate"
```

Fig. : Init.config the file which contains the commands for creating db tables



```
<?php
define('RDS_HOSTNAME', $_SERVER['RDS_HOSTNAME']);
define('RDS_USERNAME', $_SERVER['RDS_USERNAME']);
define('RDS_PASSWORD', $_SERVER['RDS_PASSWORD']);
define('RDS_DB_NAME', $_SERVER['RDS_DB_NAME']);

return [

```

Fig: DB_config for creating the global variables

So now we created separate servers for our application logic tire (EC2) and data tire (RDS) and also configured the communication between them. The communication was settled from the global variables we declared on our DB configuration which made possible taking the credentials from the RDS instance and starting the communication.

At that moment our app became *elastic*, it could run in different instances because all the launched instances (EC2 web servers) communicated with the same database, and so we had consistency.

4 Project Requirements and Test-Cases

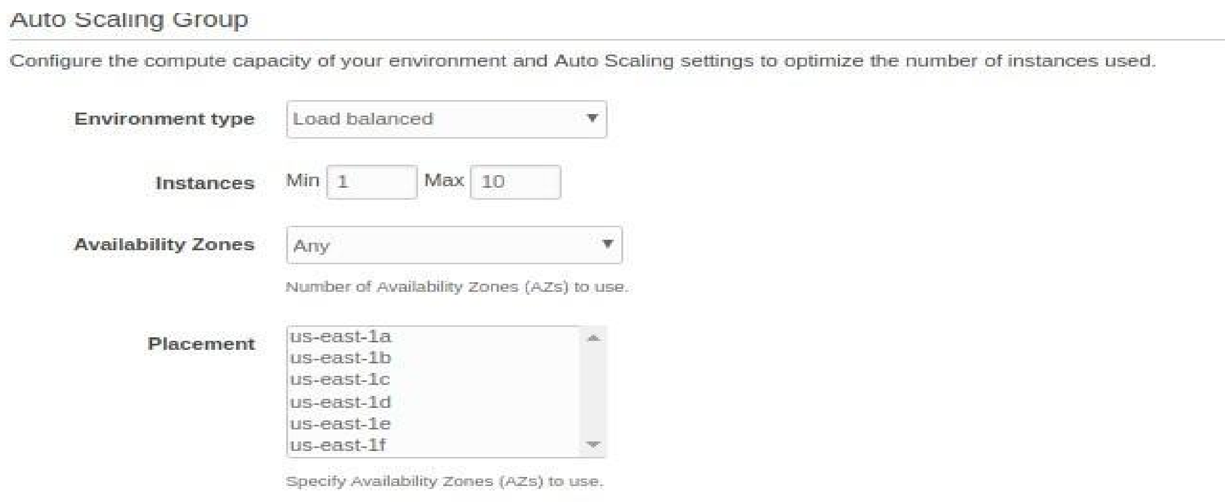
4.1 Properties of our infrastructure and configurations

4.1.1 Autoscaling

Scaling is the ability to scale-in/out for elastically and efficiently handling the workload changes. Auto means that this is done automatically from the infrastructure based on the rules and configurations we have done.

Configurations

To give our infrastructure the ability to autoscale we started with basic configurations.



Auto Scaling Group

Configure the compute capacity of your environment and Auto Scaling settings to optimize the number of instances used.

Environment type: Load balanced

Instances: Min 1 Max 10

Availability Zones: Any

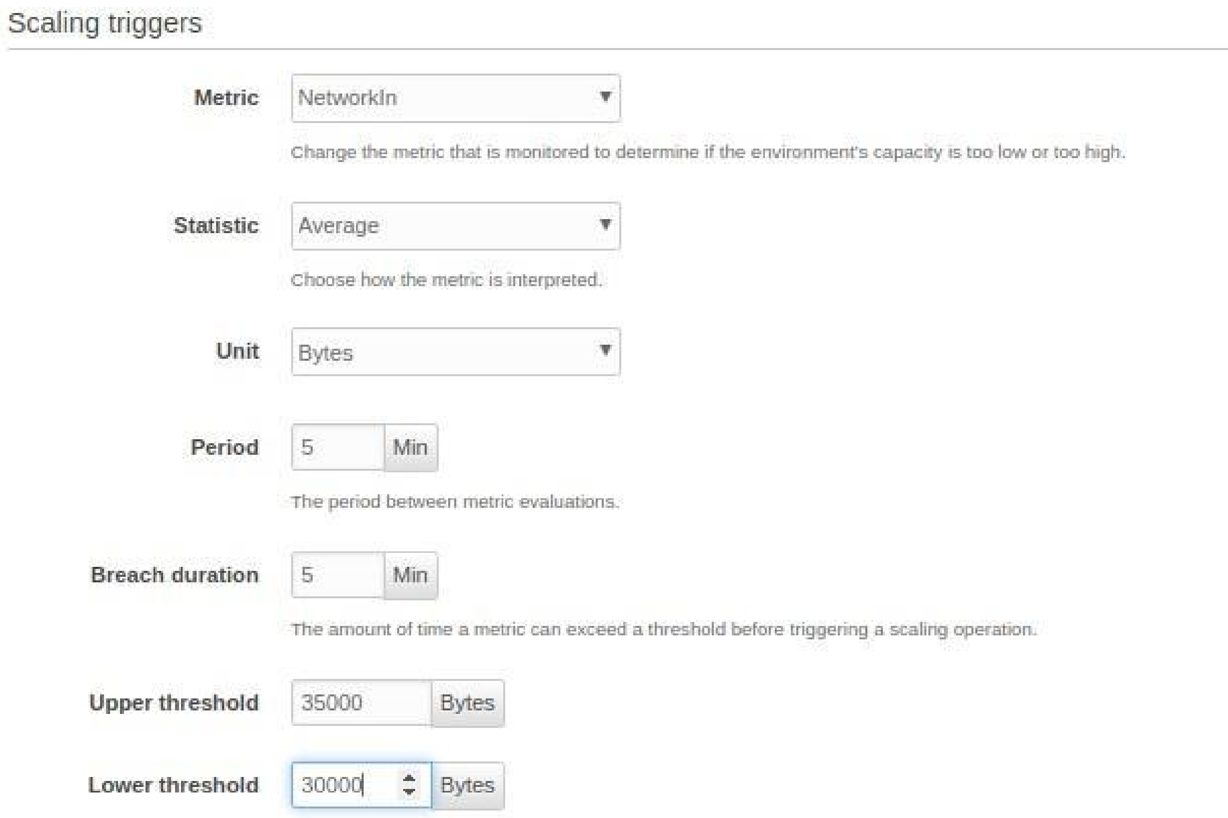
Number of Availability Zones (AZs) to use.

Placement: us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, us-east-1f

Specify Availability Zones (AZs) to use.

1. *Setting*
   have auto scaling properties.
   a. Setting the number of desired machines, choosing the minimum and the maximum number of machines running.
   b. Selecting the availability zones where we can launch our instance. We selected all, so we can scale in different availability zones (the above picture). The property of having multiple availability zones is related even with the load balancer, this way he has the ability to distribute the load in different availability zones.
2. *Autoscaling policies* (scaling triggers).
   a. First we choose the metric. We have gone with the NetworkIn (the network in our app since the load balancer we use is application type, so the listener listens only traffic directed to our application). We went with this metric for test purposes because we the main idea was to use request count for

counting the number of requests but the request handling did not happen in real time so our calculations for the scaling policy could not be accurate for designing the right stressing plan with jmeter to test our infrastructure properties. We use bytes as a unit and the statistic is the a



Scaling triggers

Metric: NetworkIn

Change the metric that is monitored to determine if the environment's capacity is too low or too high.

Statistic: Average

Choose how the metric is interpreted.

Unit: Bytes

Period: 5 Min

The period between metric evaluations.

Breach duration: 5 Min

The amount of time a metric can exceed a threshold before triggering a scaling operation.

Upper threshold: 35000 Bytes

Lower threshold: 30000 Bytes

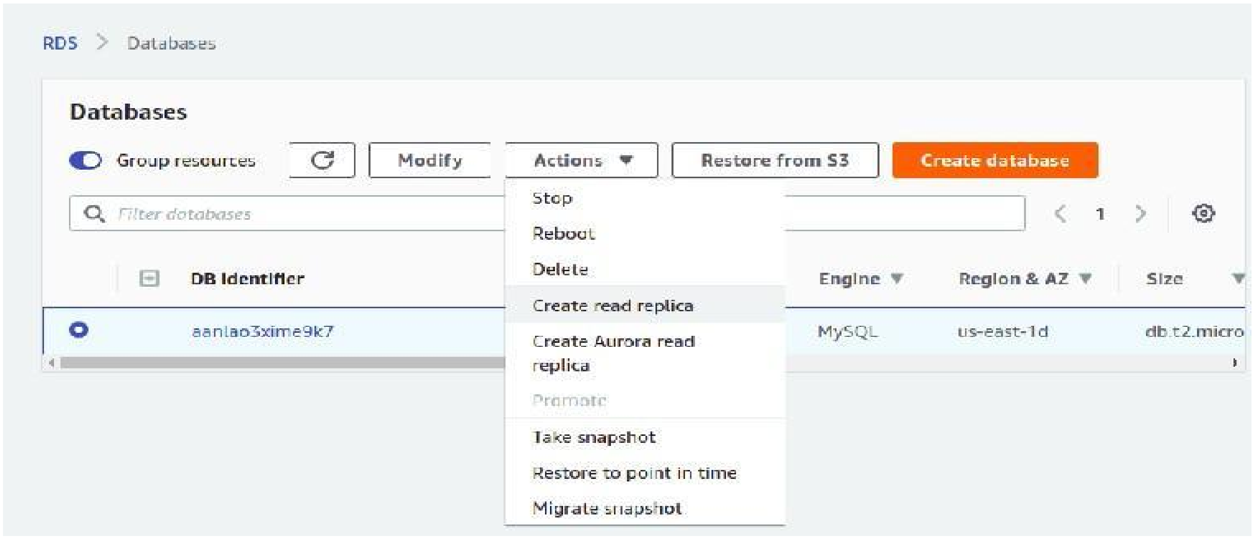
Scaling policy

Now we set as upper threshold 35000 bytes and as lower threshold 30000 bytes so when the 35000 threshold is met we give 5 minutes time to see if the threshold continues (Breach duration) and then a scale out action is done by adding one instance. We evaluate the metric every 5 minutes. Same thing happens even with the scale out action, only that now is evaluated with the lower threshold and one instance is removed if the conditions are met.

Load balancer

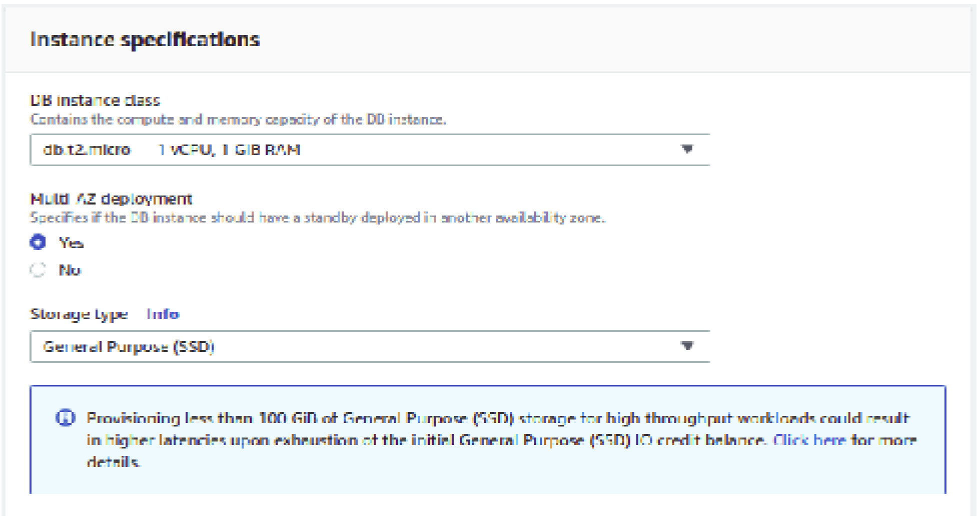
Load balancer properties are said in the second chapter, what we did in the configuration is that we gave him the property to distribute in different availability zones and configure the listener to be the port 80 of HTTP protocol.

RDS databases
To set up our app we have used RDS databases as our data tier (aws works with tiers) with instances of type mysql as the one defined in our db configuration in our app and in the second chapter. Another configuration or action that we have to do regarding our database is to activate synchronous replica for our RDS databases and the action is shown in the following picture.



DB identifier	Engine	Region & AZ	Size
aanlao3xime9k7	MySQL	us-east-1d	db.t2.micro

Read replicas in Amazon RDS for MySQL, provide a complementary availability mechanism to Amazon RDS Multi-AZ Deployments. You can promote a read replica if the source DB instance fails. This functionality complements the synchronous replication, automatic failure detection, and failover provided with Multi-AZ deployments.

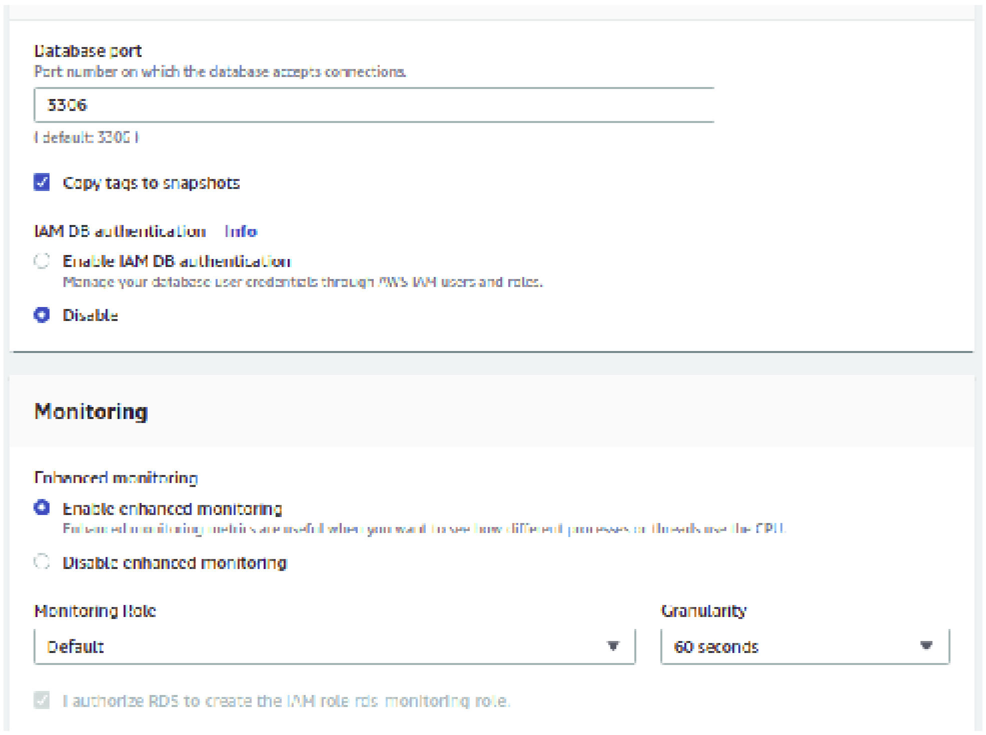

Instance specifications

DB instance class
Contains the compute and memory capacity of the DB instance.
db.t2.micro — 1 vCPU, 1 GiB RAM

Multi-AZ deployment
Specifies if the DB instance should have a standby deployed in another availability zone.
[✓ Yes] No

Storage type Info
General Purpose (SSD)

Provisioning less than 100 GiB of General Purpose (SSD) storage for high throughput workloads could result in higher latencies upon exhaustion of the initial General Purpose (SSD) IO credit balance. Click here for more details.



Database port
Port number on which the database accepts connections.

5306

(default: 3306)

[✓] Copy tags to snapshots

IAM DB authentication Info

Enable IAM DB authentication
Manage your database user credentials through AWS IAM users and roles.

[✓ Disable]

Monitoring

Enhanced monitoring

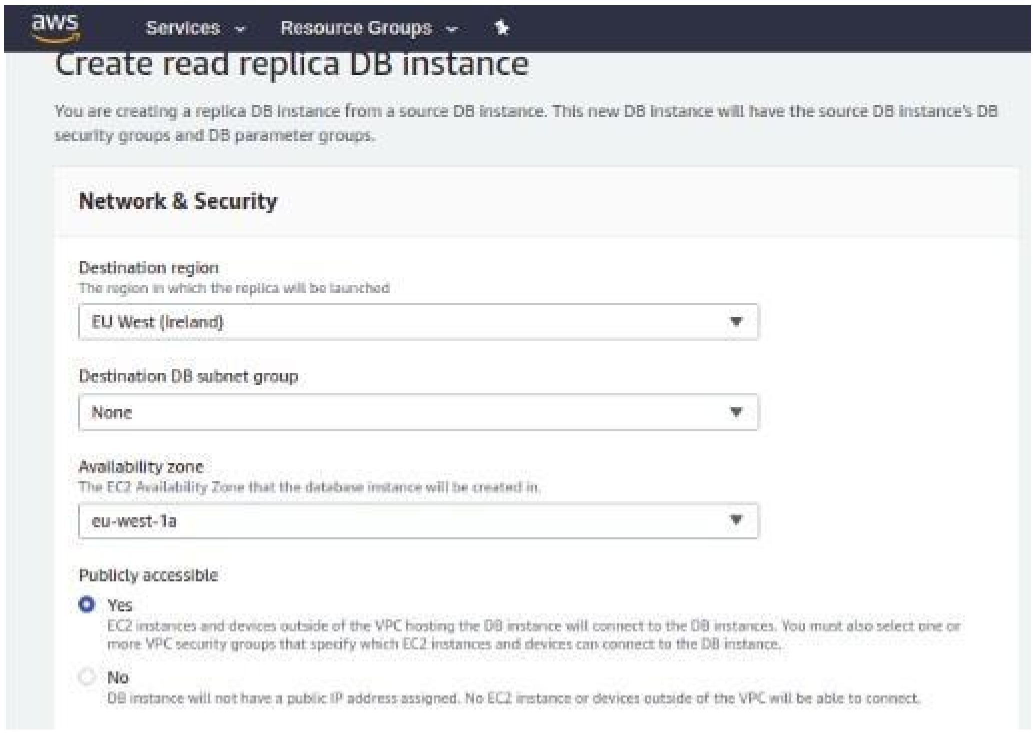
[✓ Enable enhanced monitoring]
Enhanced monitoring metrics are useful when you want to see how different processes or threads use the CPU.

Disable enhanced monitoring

Monitoring Role: Default

Granularity: 60 seconds

[✓] I authorize RDS to create the IAM role rds-monitoring-role.



Create read replica DB instance

You are creating a replica DB instance from a source DB instance. This new DB instance will have the source DB instance's DB security groups and DB parameter groups.

Network & Security

Destination region
The region in which the replica will be launched

EU West (Ireland)

Destination DB subnet group

None

Availability zone
The EC2 Availability Zone that the database instance will be created in.

eu-west-1a

Publicly accessible

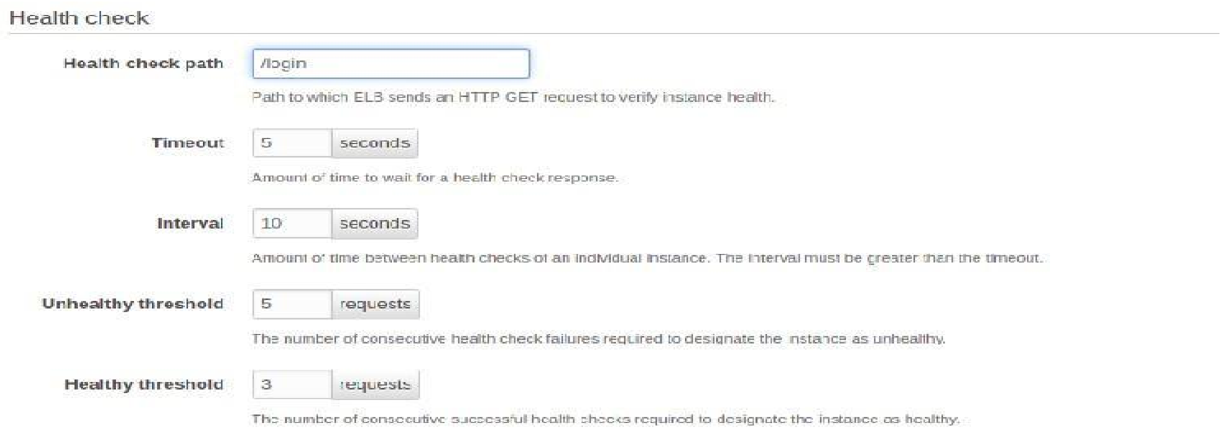
[✓ Yes]
EC2 instances and devices outside of the VPC hosting the DB instance will connect to the DB instances. You must also select one or more VPC security groups that specify which EC2 instances and devices can connect to the DB instance.

No
DB instance will not have a public IP address assigned. No EC2 instance or devices outside of the VPC will be able to connect.

4.1.2 Health check (self healing configuration)

Another strong point of a high availability infrastructure is self healing the instances. This in aws is done by the load balancer. The load balancer ping the machines in some url at some port for a certain number of times (as specified in the policy) and if which machine (ec2 instance) does not respond after this request threshold it runs a new instance to replace it automatically, so the infrastructure self heals itself.

In the following picture is shown our health check configuration. To do the health check the load balancer will ping on the /login path our website with a http get request. It ping every instance every 10 seconds and for the health check response it waits 5 seconds otherwise the health check is said to be negative (instance not healthy). For an instance to be declared unhealthy we have set the unhealthy threshold equal to 5, so five consecutive unhealthy reports (failure) for one instance means that the instance will be replaced (according to our policy that we will show). Meanwhile an instance is healthy if it responds three times consecutively to the load balancer ping request (healthy threshold equal to three).


Health check

Health check path: /login
Path to which ELB sends an HTTP GET request to verify instance health.

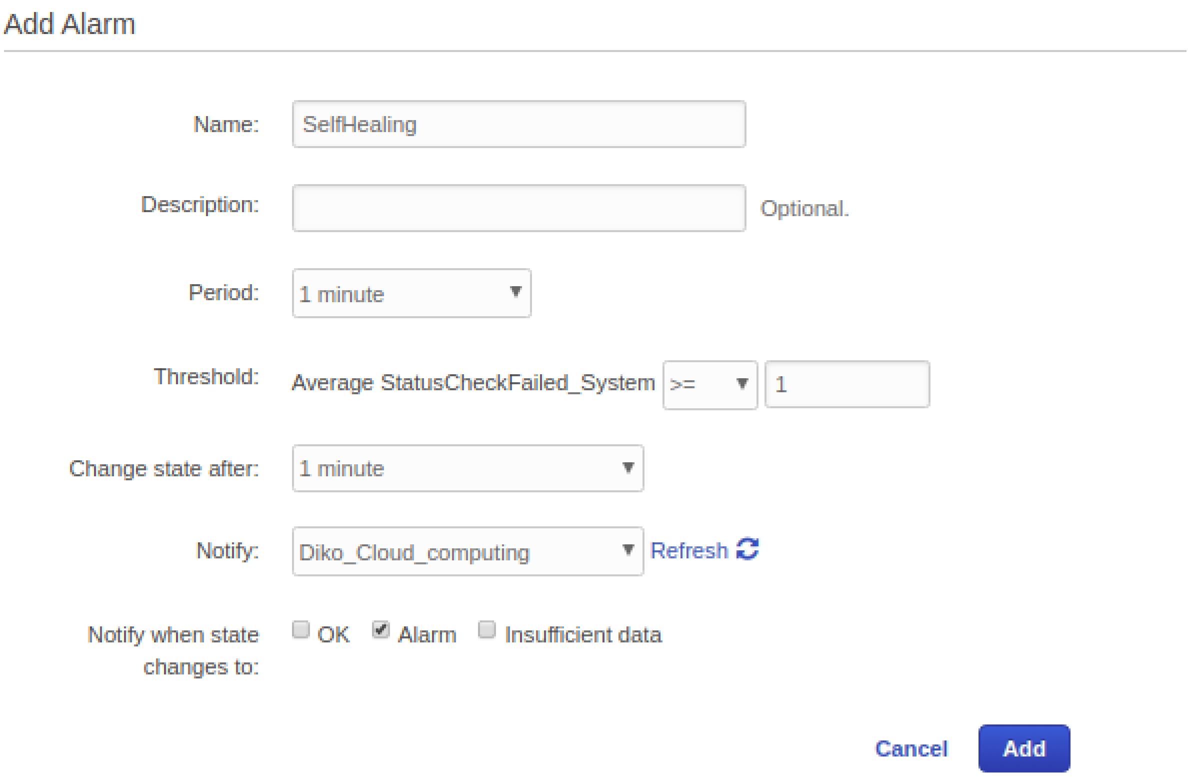
Timeout: 5 seconds
Amount of time to wait for a health check response.

Interval: 10 seconds
Amount of time between health checks of an individual instance. The interval must be greater than the timeout.

Unhealthy threshold: 5 requests
The number of consecutive health check failures required to designate the instance as unhealthy.

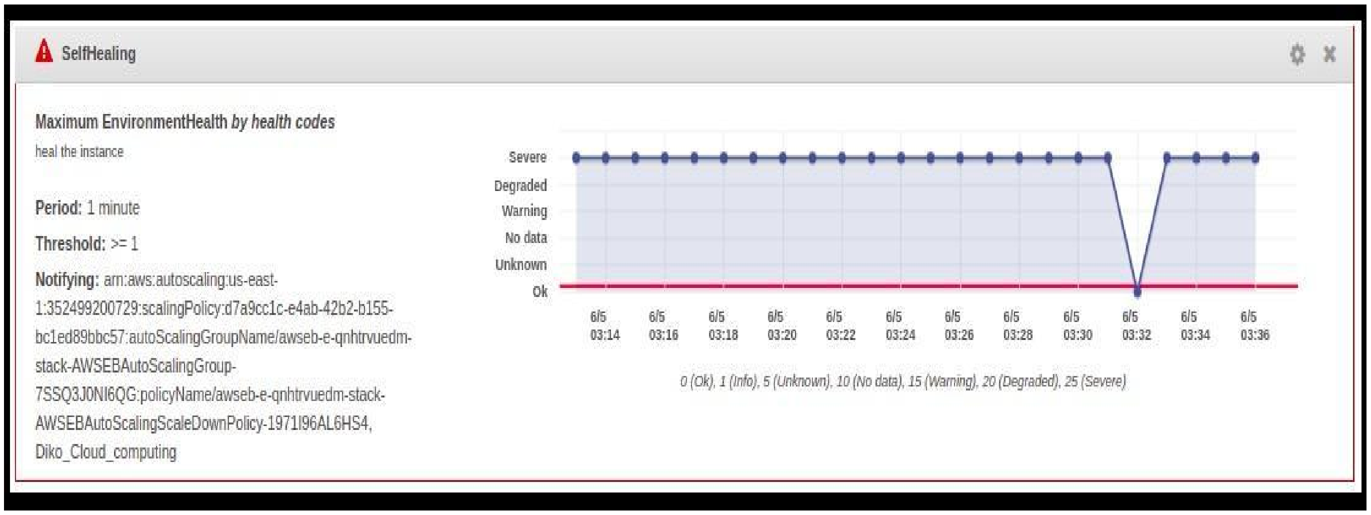
Healthy threshold: 3 requests
The number of consecutive successful health checks required to designate the instance as healthy.

This is not enough to provide a self healing infrastructure and is somehow complicated to do it but we came out with a solution as following. First thing we created an alarm from elastic beanstalk monitoring session based on the SystemCheck_Failure which monitors if we have failures in our instances and if yes it alarms.



Add Alarm

Name:	SelfHealing
Description:	Optional.
Period:	1 minute
Threshold:	Average StatusCheckFailed_System >= 1
Change state after:	1 minute
Notify:	Diko_Cloud_computing
Notify when state changes to:	OK [✓ Alarm] Insufficient data



SelfHealing

Maximum EnvironmentHealth ***by health codes***

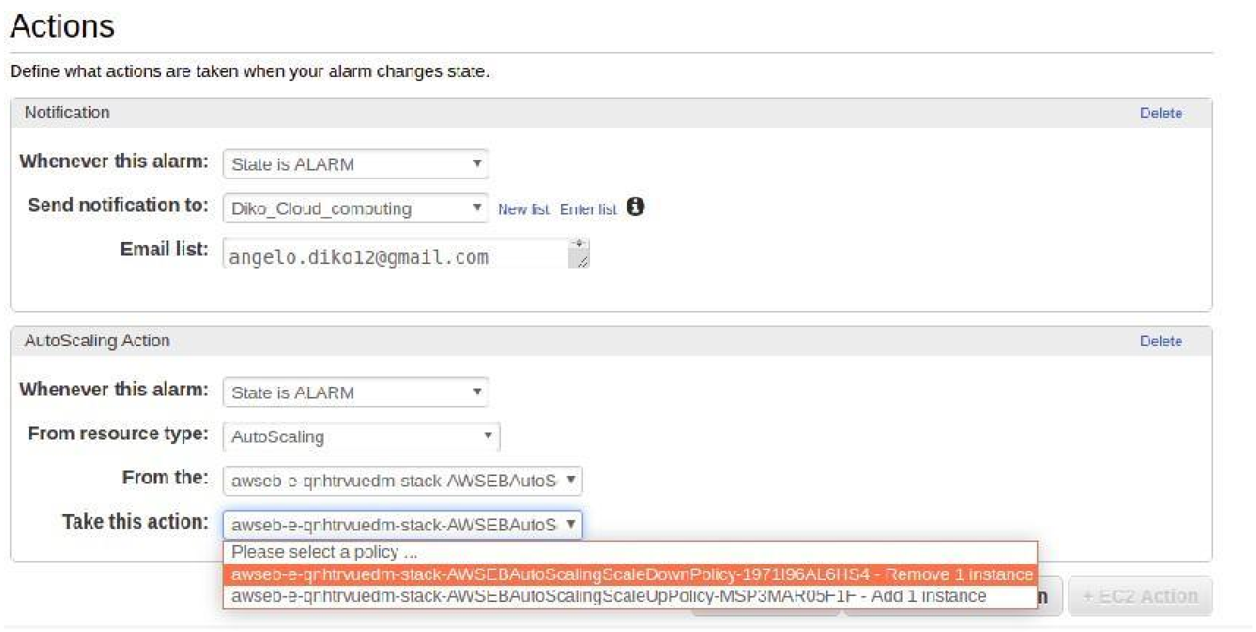
heal the instance

Period: 1 minute

Threshold: >= 1

Notifying: arn:aws:autoscaling:us-east-1:352499200729:scalingPolicy:d7a9cc1c-e4ab-42b2-b155-bc1ed89bbc57:autoScalingGroupName/awseb-e-qnhtrvuedm-stack-AWSEBAutoScalingGroup-7SSQ3J0NI6QG:policyName/awseb-e-qnhtrvuedm-stack-AWSEBAutoScalingScaleDownPolicy-1971I96AL6HS4, Diko_Cloud_computing

This is the alarm created in beanstalk. Well still is not enough because till here we are just are able to know if an instance fails but can t take actions on beanstalk, to make it capable to take actions we need to further modify it through CloudWatch.



Actions

Define what actions are taken when your alarm changes state.

Notification

Whenever this alarm: State is ALARM

Send notification to: Diko_Cloud_computing

Email list: angelo.diko12@gmail.com

AutoScaling Action

Whenever this alarm: State is ALARM

From resource type: AutoScaling

From the: awseb-e-qrhtrvuedm-stack-AWSEBAutoS

Take this action: awseb-e-qrhtrvuedm-stack-AWSEBAutoS

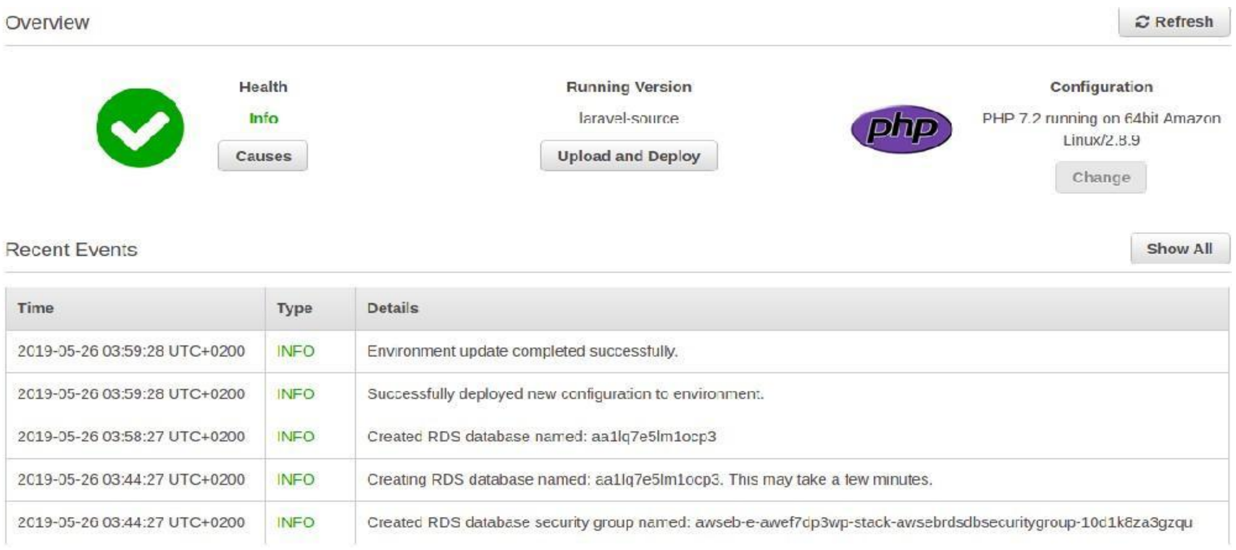
Please select a policy ...
awseb-e-qrhtrvuedm-stack-AWSEBAutoScalingScaleDownPolicy-1971I96AL6HS4 - Remove 1 instance
awseb-e-qrhtrvuedm-stack-AWSEBAutoScalingScaleUpPolicy-MSP3MAR05F1F - Add 1 instance

The picture above shows the modification we did to the alarm we created from beanstalk using CloudWatch -> Alarms -> Modify and setting the action remove instance when the alarm breaches. Why remove, because if we remove an instance the autoscaling group will replace it with another one. How? Because when we remove the instance the autoscaling policy comes in and compares the number of instances with the desired number based on the load, if it is less it will add another one. We will prove this property later on.

4.2 Test-Cases

We used jmeter tool to stress our website to prove the properties of our infrastructure as stated above except for the self healing we used some different approach.

This was our initial state: one RDS running instance and one EC2 (in the picture are shown successfully the last steps of the creation which is the creation of RDS because its configuration should be done separately from its dashboard).

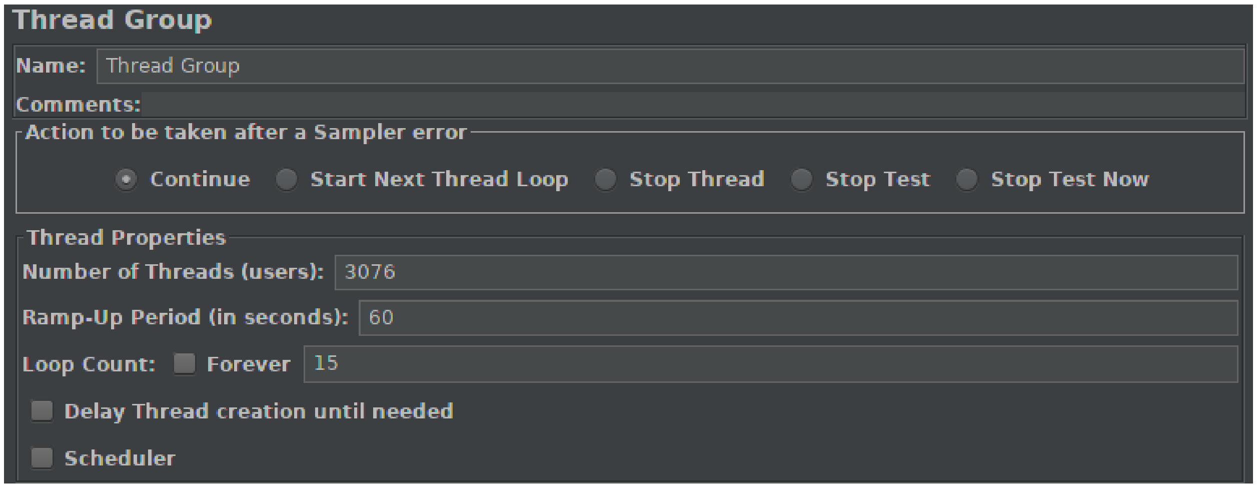


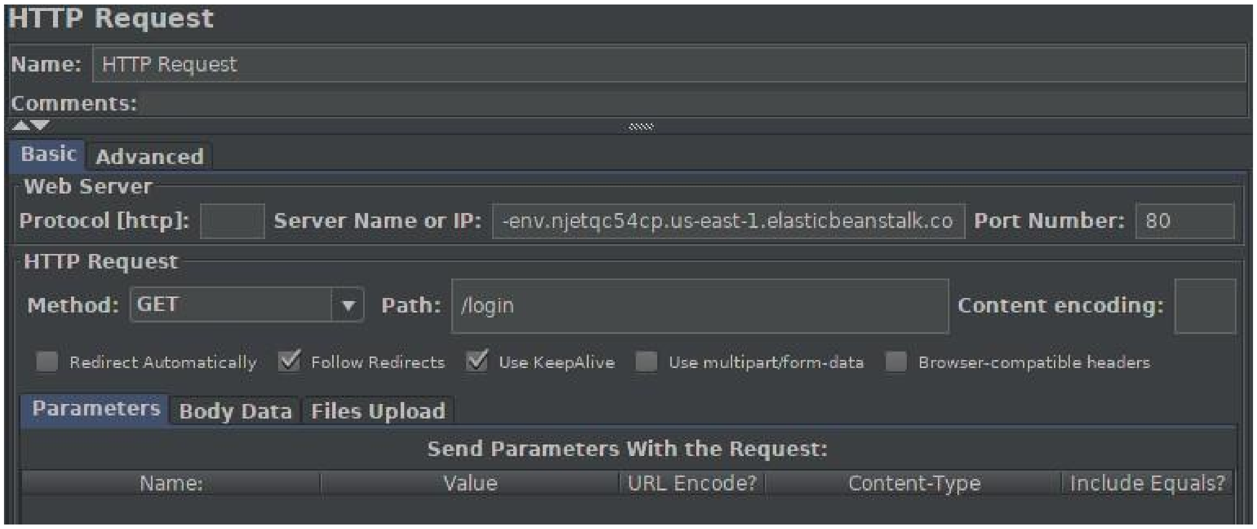
Time	Type	Details
2019-05-26 03:59:28 UTC+0200	INFO	Environment update completed successfully.
2019-05-26 03:59:28 UTC+0200	INFO	Successfully deployed new configuration to environment.
2019-05-26 03:58:27 UTC+0200	INFO	Created RDS database named: aa1lq7e5lm1ocp3
2019-05-26 03:44:27 UTC+0200	INFO	Creating RDS database named: aa1lq7e5lm1ocp3. This may take a few minutes.
2019-05-26 03:44:27 UTC+0200	INFO	Created RDS database security group named: awseb-e-awef7dp3wp-stack-awsebrdsdbsecuritygroup-10d1k8za3gzqu

App Dashboard
We need to prove our infrastructure properties so the plan was to stress our app. Because we dealt with network_in traffic (our metric in the autoscaling policy) we had to take in consideration how big a *http get package* sent from *Jmeter.* It actually is 4069 bytes (4kb). We did the calculations on how much he had to send so we can lunch up to 6 instances. After that we decrease the load to see our scale in action.
Our plan was as follows:
We sent around 9 requests per second for at least 10 minutes to see our scaling policy in action, but because unfortunately the requests sent to aws are not sent handled exactly when we send them, they got distributed in time which we do not know why or how, our test sent a huge amount of request to scale out up to 6 instances and then we decreased the load it to scale in.

First: Scale out
Jmeter plan : 9 request in sec per instance * 6 (number of instances we want) * 60 (one minute) and will repeat this action 15 times (15 minutes) to see the actions because the metrics are not shown in real time in CloudWatch, maybe it happens because we use free sources. But because as we said the requests are distributed in time we run this test several times.

Note: Jmeter components we used are *Thread groups*, *Listener* (Http Request) and *view Results* to see the execution of the plan

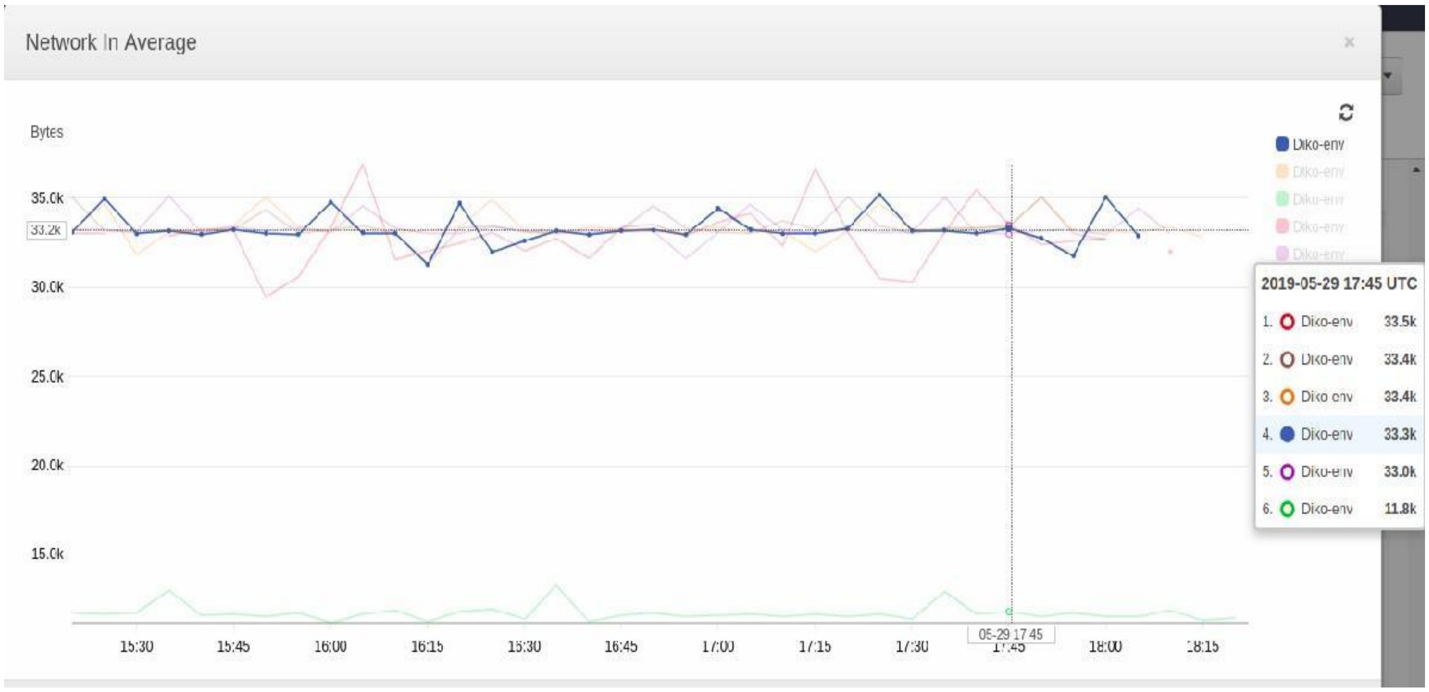

Thread Group
Name: Thread Group
Comments:
Action to be taken after a Sampler error
[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now
Thread Properties
Number of Threads (users): 3076
Ramp-Up Period (in seconds): 60
Loop Count: Forever 15
Delay Thread creation until needed
Scheduler


HTTP Request
Name: HTTP Request
Comments:
Basic Advanced
Web Server
Protocol [http]: Server Name or IP: -env.njetqc54cp.us-east-1.elasticbeanstalk.co Port Number: 80
HTTP Request
Method: GET Path: /login Content encoding:
Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] Use multipart/form-data Browser-compatible headers
Parameters Body Data Files Upload
Send Parameters With the Request:

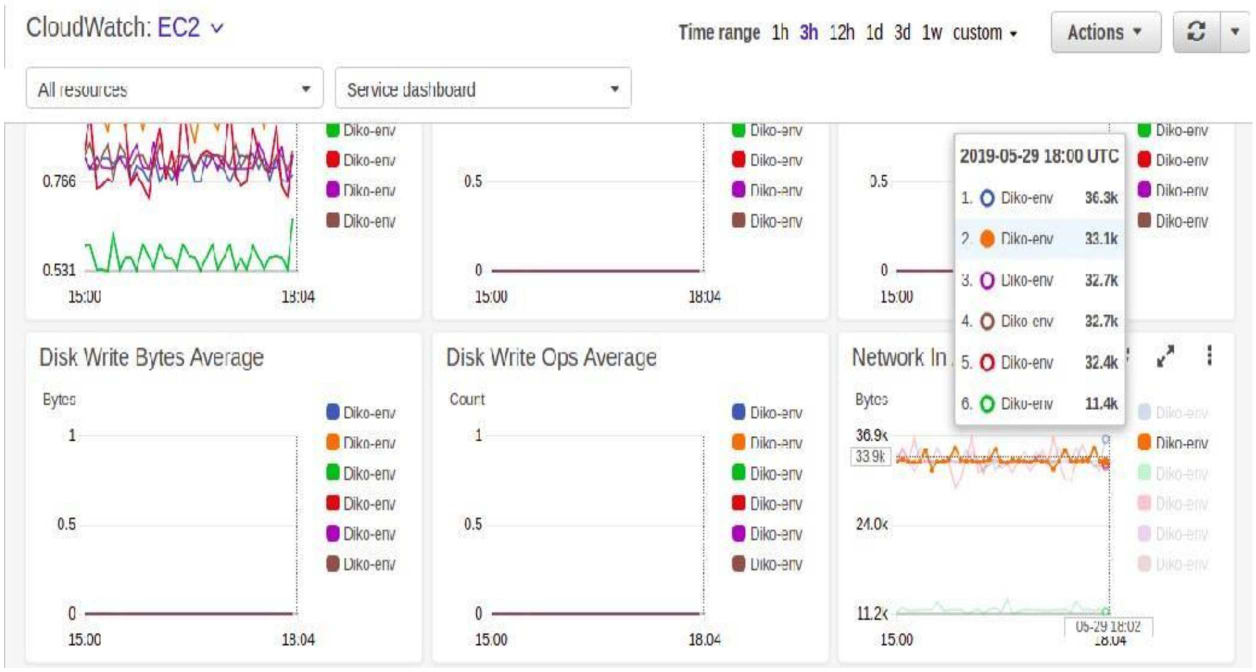
Name:	Value	URL Encode?	Content-Type	Include Equals?

The above two pictures show the plan and the http request configuration. Basically we are sending http (get) request in port 80 the url (IP) on /login path. With that much load we expected to lunch up to 6 instances and keep them running for some time because we did run the test several times.

The results

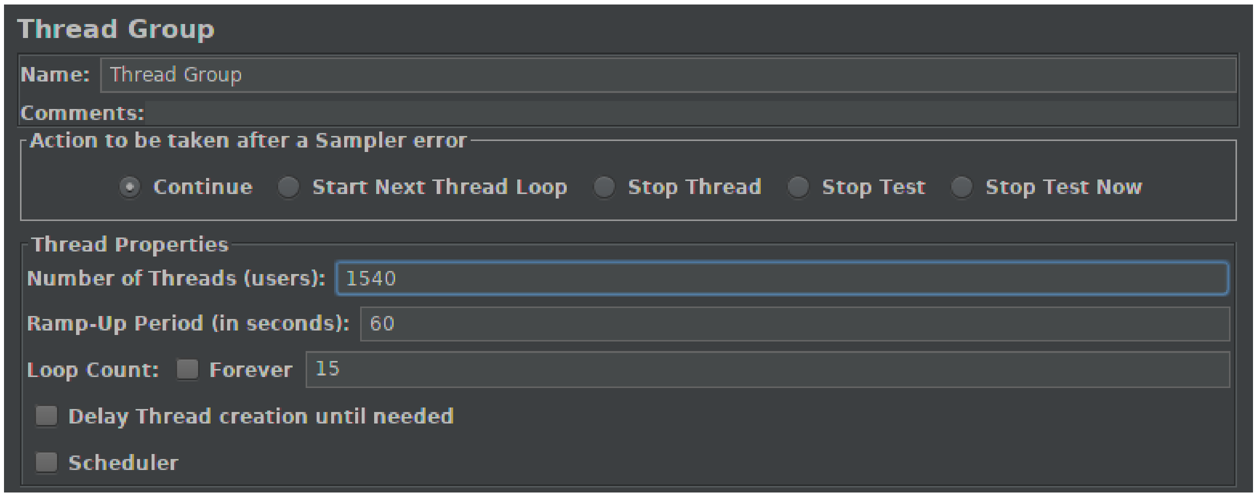


Since the test began there were 6 instances up and the load between them is distributed. We had a problem, why the last instance is with 11.9 kb, why is it not terminated?
The reason is simple, if that instance is turned out the load will be distributed to the other instances and then the upper threshold will be reached. The scaling action will again add another instance. To avoid this process of constantly adding and removing instances the infrastructure keeps this instance functional with a small amount of load, and we can see that it is kept during all the time (green line in the picture).



In the picture above we see some of the metrics used. We are just ping-ing to login page, so no read/write or other operations have CPUUsage (top left metric) smaller than 1%. Network IN is the bottom right metric, while network OUT and StatuCheckFailed_System are not shown in the picture.The scale out and the load balancer worked properly.

Second : Scale in.
Immediately after the plan execution for scale out in Jmeter finished, we created a new plan with half of the load like in the following picture.


Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 1540

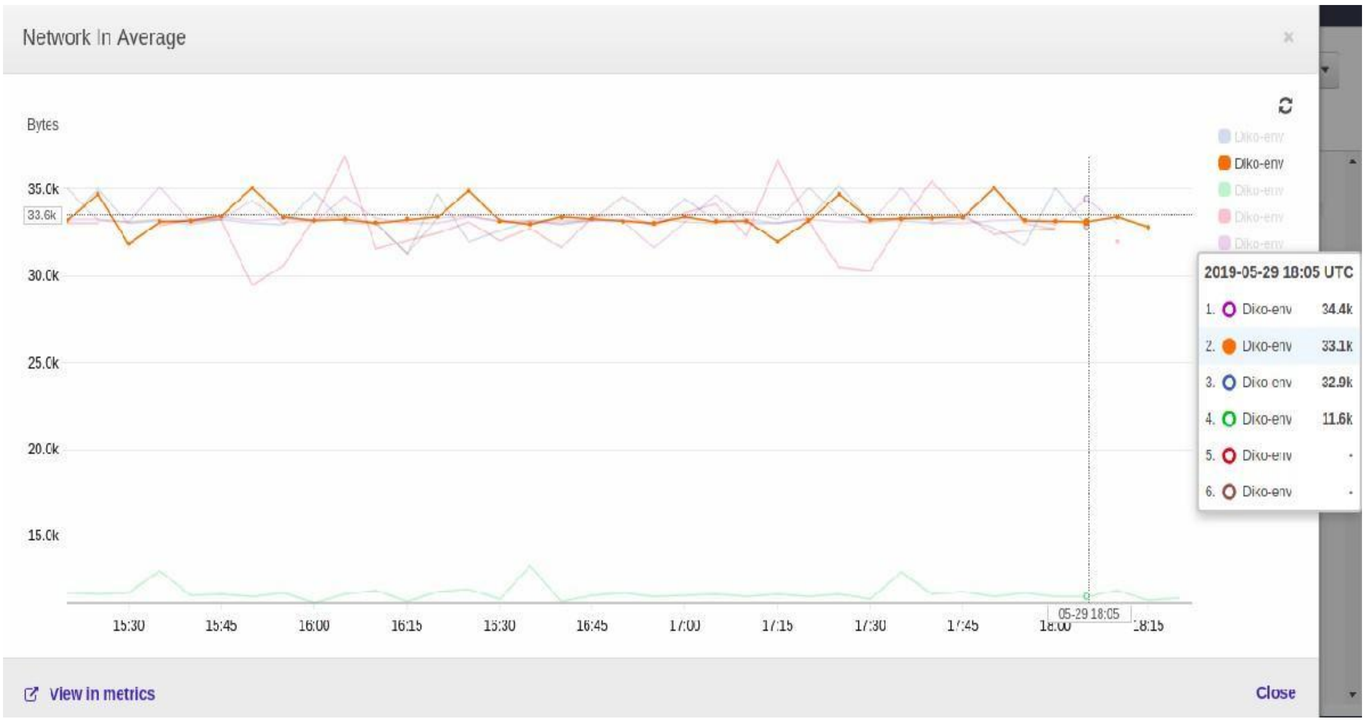
Ramp-Up Period (in seconds): 60

Loop Count: Forever 15

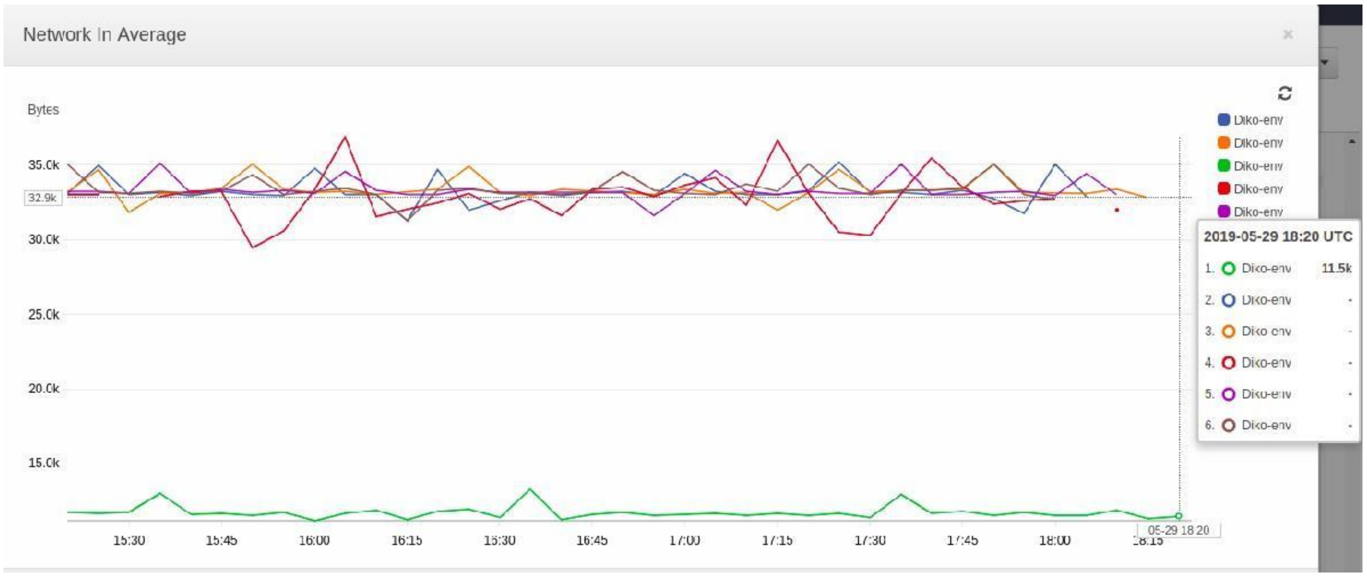
Delay Thread creation until needed

Scheduler

The results



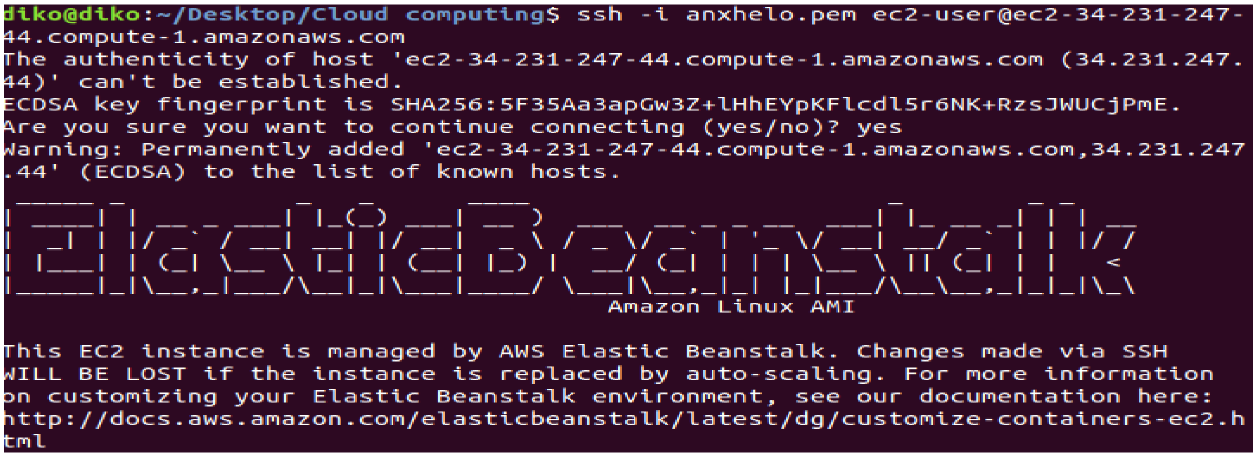
The number of instances was not reduced two times, instead two instances were terminated and we are in the same conditions as before. One instance was kept with a small load for the very same reason. We run this scale in the test just once so in the timeline period it should be shown only for 15 minutes, and after that we had only one running instance, the green line in the bottom or the one in the following picture.



Up till now we have shown that our infrastructure can :

- *scale horizontally (automatically)*
- *can load balance to instances inside the same datacenter*
- *can lunch VMs (EC2 instances) in different availability zones*.

Time to prove the *self healing ability.*
First we will connect to the instance that is up by ssh.

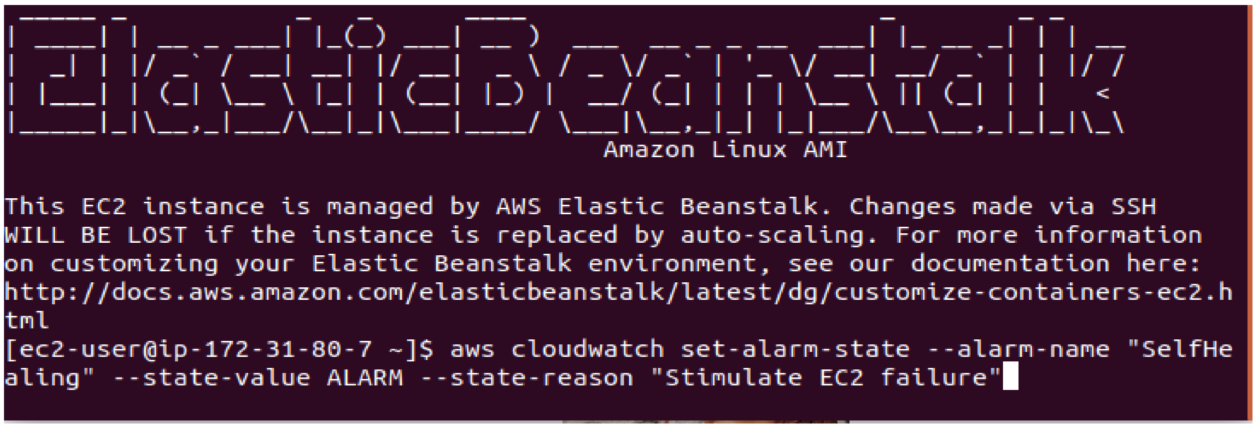


```
diko@diko:~/Desktop/Cloud computing$ ssh -i anxhelo.pem ec2-user@ec2-34-231-247-
44.compute-1.amazonaws.com
The authenticity of host 'ec2-34-231-247-44.compute-1.amazonaws.com (34.231.247.
44)' can't be established.
ECDSA key fingerprint is SHA256:5F35Aa3apGw3Z+lHhEYpKFlcdl5r6NK+RzsJWUCjPmE.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-231-247-44.compute-1.amazonaws.com,34.231.247
.44' (ECDSA) to the list of known hosts.
                                                   Amazon Linux AMI

This EC2 instance is managed by AWS Elastic Beanstalk. Changes made via SSH
WILL BE LOST if the instance is replaced by auto-scaling. For more information
on customizing your Elastic Beanstalk environment, see our documentation here:
http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-containers-ec2.h
tml
```

We connected. To prove the self healing ability we have to fake the hardware failure of our running EC2 instance by causing or stimulating HW failure from the terminal with the following command.

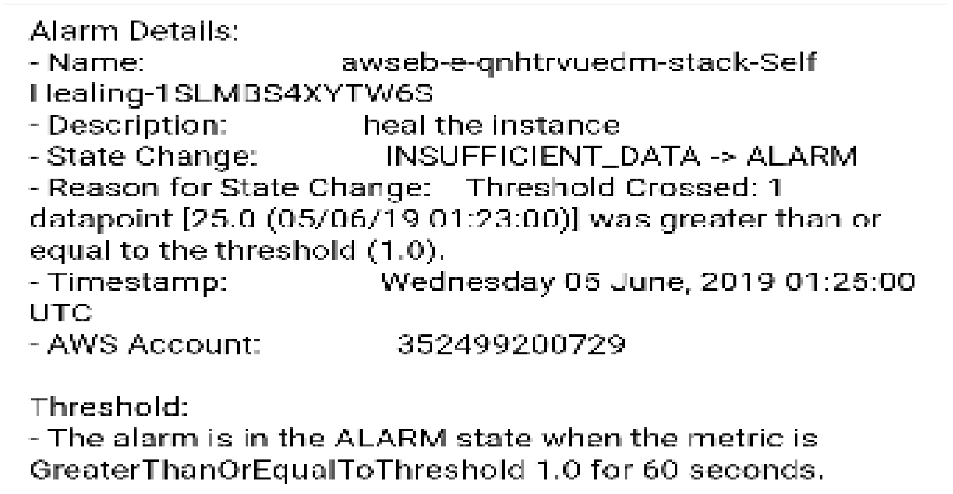
aws cloudwatch setalarmstate \ *"this command says that we are changing alarm state"*
alarmname "SelfHealing" \ *"With this command we give the name of the alarm to change"*
statevalue ALARM \ *"This sets the alarm state to ALARM"*
statereason "Stimulate EC2 failure"\ *"It needs a reason for this to be executed"*



```
                                                   Amazon Linux AMI

This EC2 instance is managed by AWS Elastic Beanstalk. Changes made via SSH
WILL BE LOST if the instance is replaced by auto-scaling. For more information
on customizing your Elastic Beanstalk environment, see our documentation here:
http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-containers-ec2.h
tml
[ec2-user@ip-172-31-80-7 ~]$ aws cloudwatch set-alarm-state --alarm-name "SelfHe
aling" --state-value ALARM --state-reason "Stimulate EC2 failure"
```

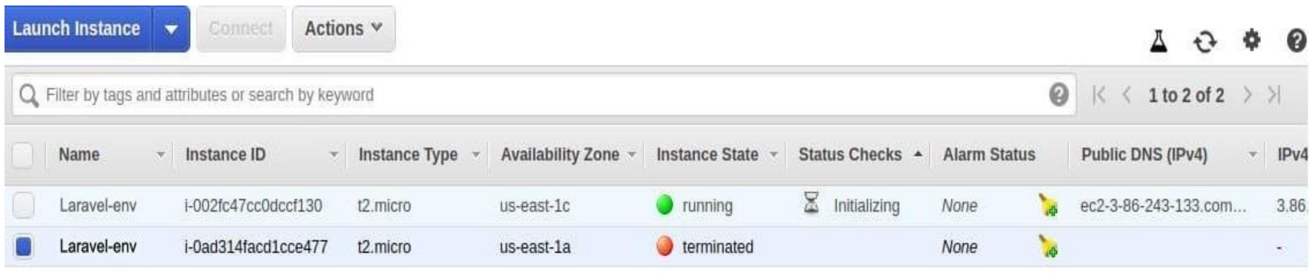
This command terminated the instance which was active and because we have configured it to send notifications when the alarm breaches, the following email arrived to us:



```
Alarm Details:
- Name:                     awseb-e-qnhtrvuedm-stack-Self
Healing-1SLMBS4XYTW6S
- Description:              heal the instance
- State Change:             INSUFFICIENT_DATA -> ALARM
- Reason for State Change:  Threshold Crossed: 1
datapoint [25.0 (05/06/19 01:23:00)] was greater than or
equal to the threshold (1.0).
- Timestamp:                Wednesday 05 June, 2019 01:25:00
UTC
- AWS Account:              352499200729

Threshold:
- The alarm is in the ALARM state when the metric is
GreaterThanOrEqualToThreshold 1.0 for 60 seconds.
```

The situation in our EC2 dashboard was like follows:



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4
Laravel-env	i-002fc47cc0dccf130	t2.micro	us-east-1c	running	Initializing	None	ec2-3-86-243-133.com...	3.86.
Laravel-env	i-0ad314facd1cce477	t2.micro	us-east-1a	terminated		None		-

So one instance was terminated and the other one was being initialized.

Self healing ability worked perfectly. In this picture is shown that both instances, the one which was over and the one which is initializing, are in different availability zones.

Conlcusion

- Our infrastructure was built using both IaaS and PaaS (Beanstalk)
- A three tier Client/Serve, is able to load balance in machines running in the same datacenter (Region), lunch machines in different availability zones, autoscales and has the property of self healing.
- The application is capable of running simultaneously in different instances and being consistent and highly available.
- Our database is synchronously replicated to avoid failure and this way it double powers up the availability of our application.

REFERENCES

https://docs.aws.amazon.com/