**InnovLab Technologies**

**Technical Architecture Document**

Author :

[Samba NDIAYE](mailto:ndiaye3.samba@gmail.com)



April 5, 2020

Table des matières

[SCOPE 3](#_Toc37091858)

[Executive Summary 3](#_Toc37091859)

[I – As-Is Architecture 4](#_Toc37091860)

[A – Physical Architecture 5](#_Toc37091861)

[1 – Diagram 5](#_Toc37091862)

[2 – Components 5](#_Toc37091863)

[B - Troubleshooting Walkthrough 6](#_Toc37091864)

[C - As-Is Architecture short term changes 14](#_Toc37091865)

[1 - Auto Scaling Definition 15](#_Toc37091866)

[2 - Application Load Balancer Migration 16](#_Toc37091867)

[D- Improvements & Enhancements 18](#_Toc37091868)

[1 – Reliability 18](#_Toc37091869)

[2 - Operational Excellence 19](#_Toc37091870)

[3 – Performance 20](#_Toc37091871)

[4 – Cost 20](#_Toc37091872)

[5 – Security 21](#_Toc37091873)

[E – Final Production Ready Architecture 24](#_Toc37091874)

[II- Target Architecture 25](#_Toc37091875)

[A - Physical Architecture 25](#_Toc37091876)

[ARTIFACTS 26](#_Toc37091877)

# SCOPE

This document is the Technical Architecture Document for the InnovLab Technologies web application solution.

I, Samba NDIAYE, freelance AWS Certified Solution Architect, has been contracted to troubleshoot, and fix the issues InnovLab Technologies, the customer, is having when deploying their web application to AWS.

InnovLab is in the process of developing a web application. They would like to deploy and run their application in the Amazon Web Services Cloud.

I will document my findings, during my consulting work for InnovLab, in regards to the current architecture of their application, the as-Is architecture; and the Target / Desired state of their solution. In this document, I shall state the driving forces behind my technological choices and the business drivers that justify my final solution. Finally, via my consulting work, InnovLab will have a state of the art solution that follows AWS Well-Architected Framework and is aligned with their business goals and objectives.

# Executive Summary

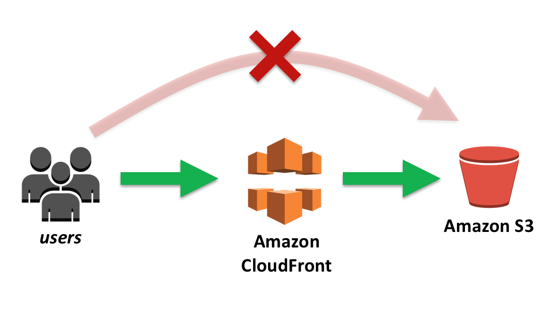
InnovLab has partly designed a static web application. The latter, when deployed to AWS Cloud, was not functional at all. After, a troubleshooting, technical assessment session, I found that:

* The Load Balancer’s targets did not include the subnet in the Availability Zone eu-central-1a.
* The security group of the EC2 instance did not allow port 80.
* The Classic Load Balancer security group did not allow port 80 either.
* The Classic Load Balancer was performing health checks on port 443 instead of port 80.

After having remediated these issues, the InnovLab web application was fully functional, via the Load Balancer DNS, and its landing page was finally accessible.

However, following AWS Architecture's best practices, I advised InnovLab to move its actual static web application to S3 Web Hosting. They will need to deploy a CloudFront + S3 solution that is cost efficient. For security purposes, InnovLab will enable OAI and prevent users from directly accessing the S3 bucket that hosts their static web application. If the need to host secure static content arises, they can leverage the usage of CloudFront signed URLs. A combination of Route53 and CloudFront will make their web app globally accessible and will increase its performance by optimizing the application latency.

Users will access InnovLab static content directly via CloudFront, as shown below:



Note the above discussion solely concerns the as-is architecture.

Soon, the InnovLab web application will include a dynamic part. They need some advice concerning their future target architecture. Based on the current trend in web application development and design, I advise them to opt for a “Modern Web Application” architecture.

A Modern Web Application, in AWS context, is Serverless. It leverages micro-service design patterns. A Modern Web Application is developed using continuous integration/continuous delivery mechanisms (CI/CD) in an agile team collaborative effort. More specifically, AWS defines a Modern Web Application as:

“*Modern applications isolate business logic, optimize reuse and iteration, and remove overhead everywhere possible. Modern apps are built using services that enable you to focus on writing code while automating infrastructure maintenance tasks*.[[1]](#footnote-1)

Following my recommendations will ensure InnovLab stays current in terms of technology: Its technology stack will be aligned with its business goals and they shall fully leverage AWS Cloud offerings. They will have their solution in an enterprise grade production environment.

The recommendations and architecture decisions in this document are implemented with conformance to the AWS Well-Architected Framework: cost, operational excellence, reliability, performance, and security

# I – As-is Architecture

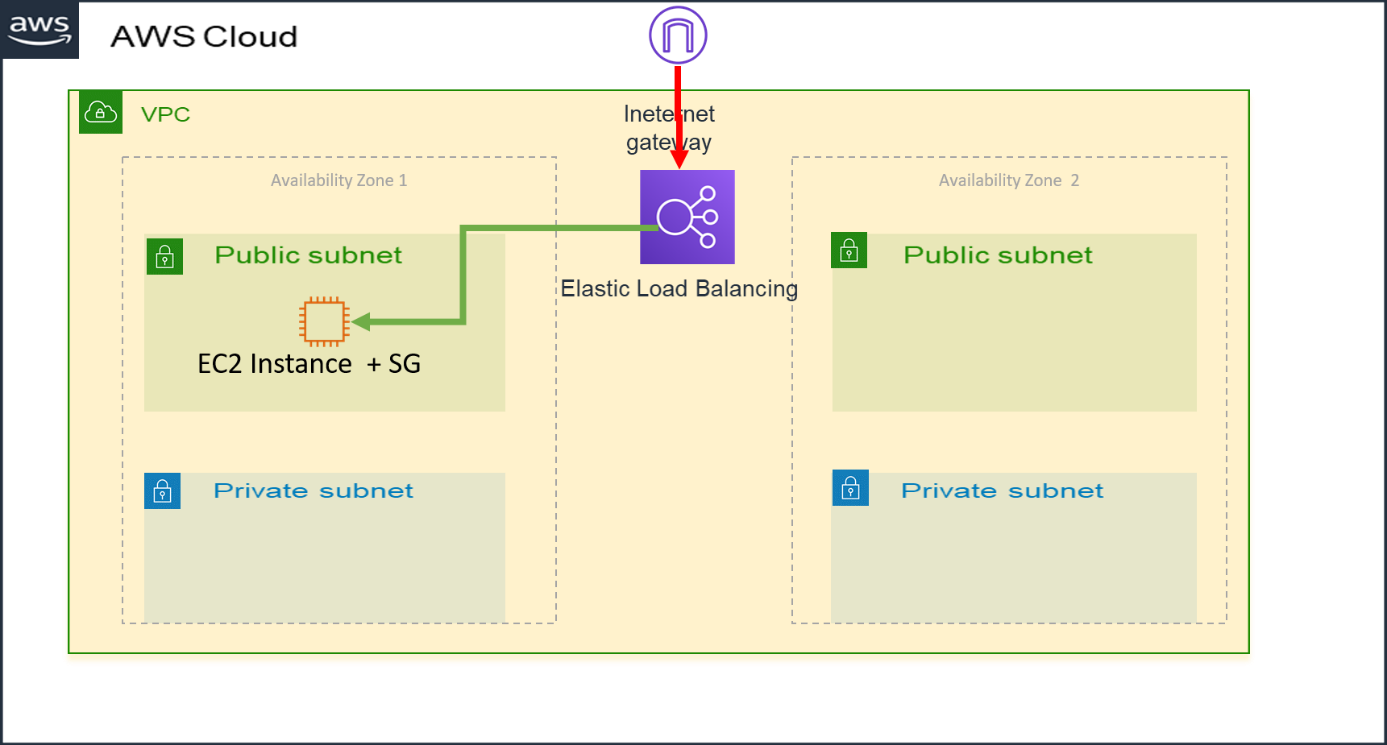
The web application InnovLab has deployed in AWS is not functional at all. One of my consulting duties consisted of troubleshooting it and making sure the homepage/landing page is accessible to users.

Let us study the physical architecture of the web app.

## A – Physical Architecture

### 1 – Diagram

Below is the physical architecture of the solution created with the CloudFormation template. It has not yet been altered. It is the starting point of our consulting work.



### 

### 2 – Components

The As-is architecture, created with the CloudFormation template, is made of the following components:

1. A Virtual Private Cloud (VPC)
2. An Internet Gateway
3. An Elastic Load balancer (ELB)
4. Two Availability zones (AZ) in the eu-central-1 region
5. A public and a private subnet in each AZ
6. An EC2 instance with a security group

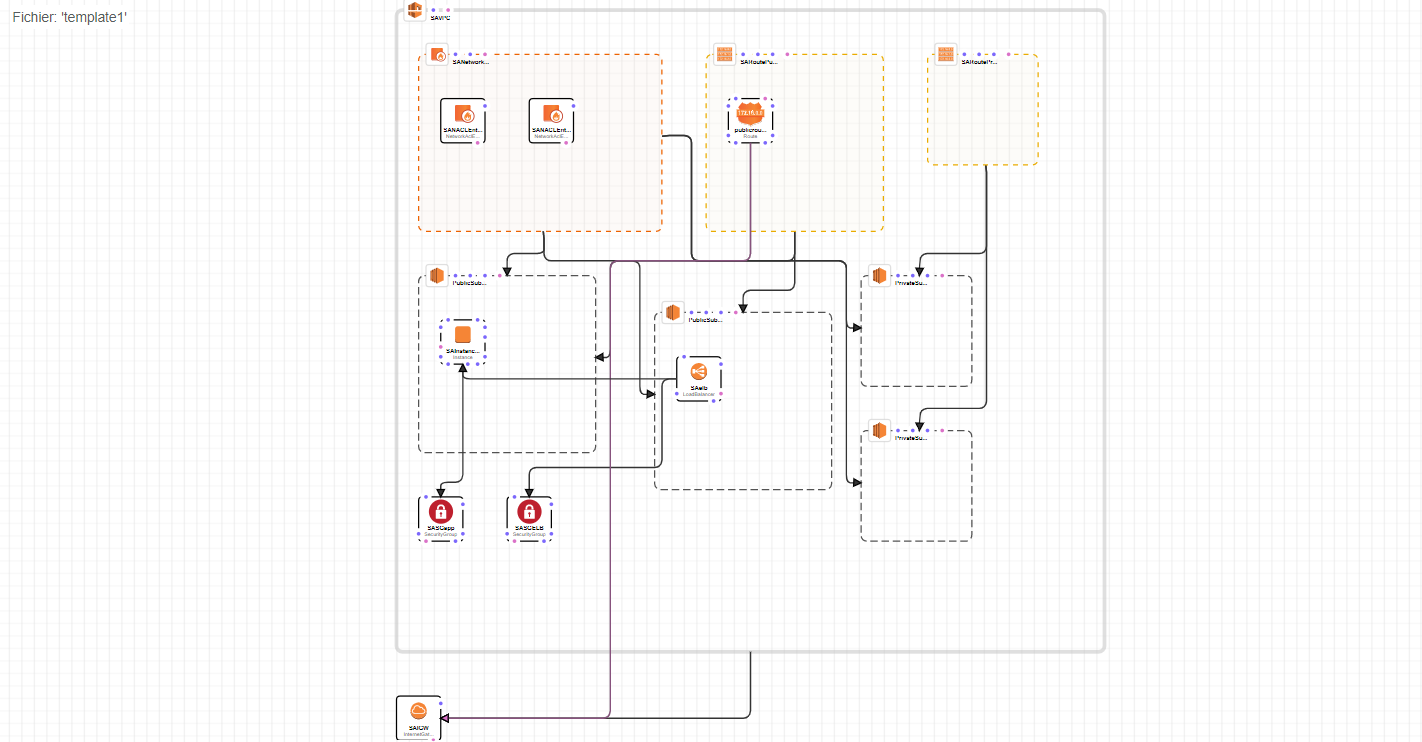
We shall realize the user requests, per the design, flow from the Internet to the Internet Gateway, then to the ELB and finally to the instance.

The next section highlights the troubleshooting steps taken to resolve the pending issues and the enhancement of the application once it is up and running.

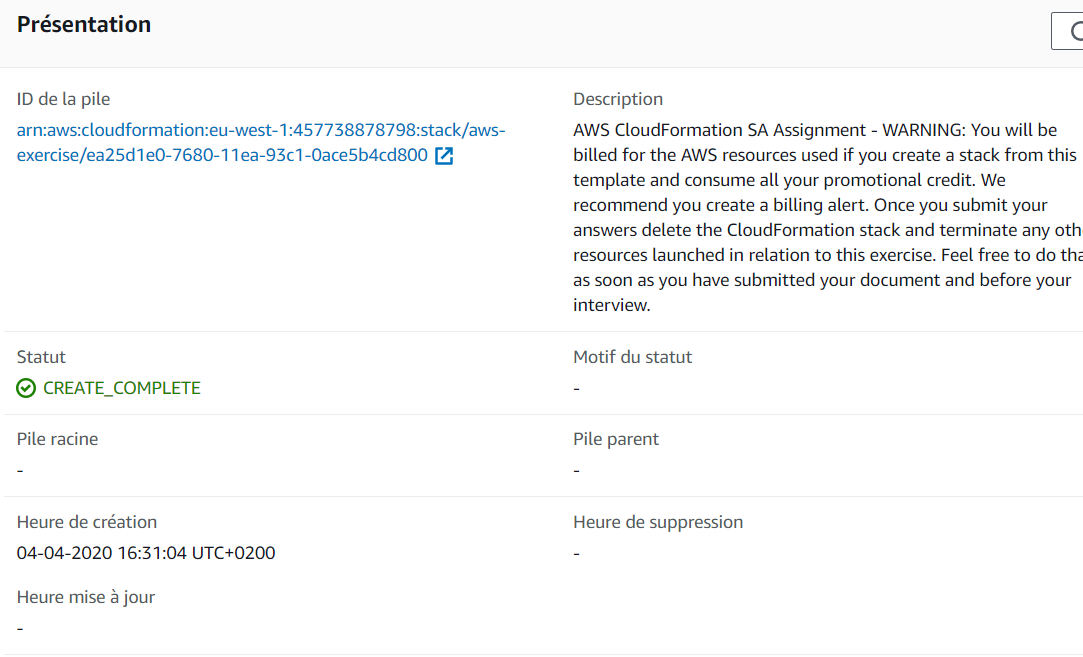
## B - Troubleshooting Walkthrough

InnovLab has provided a CloudFormation template with the architecture of their web app. I logged in the AWS Console, accessed the CloudFormation service and created a new stack using the template as a model.

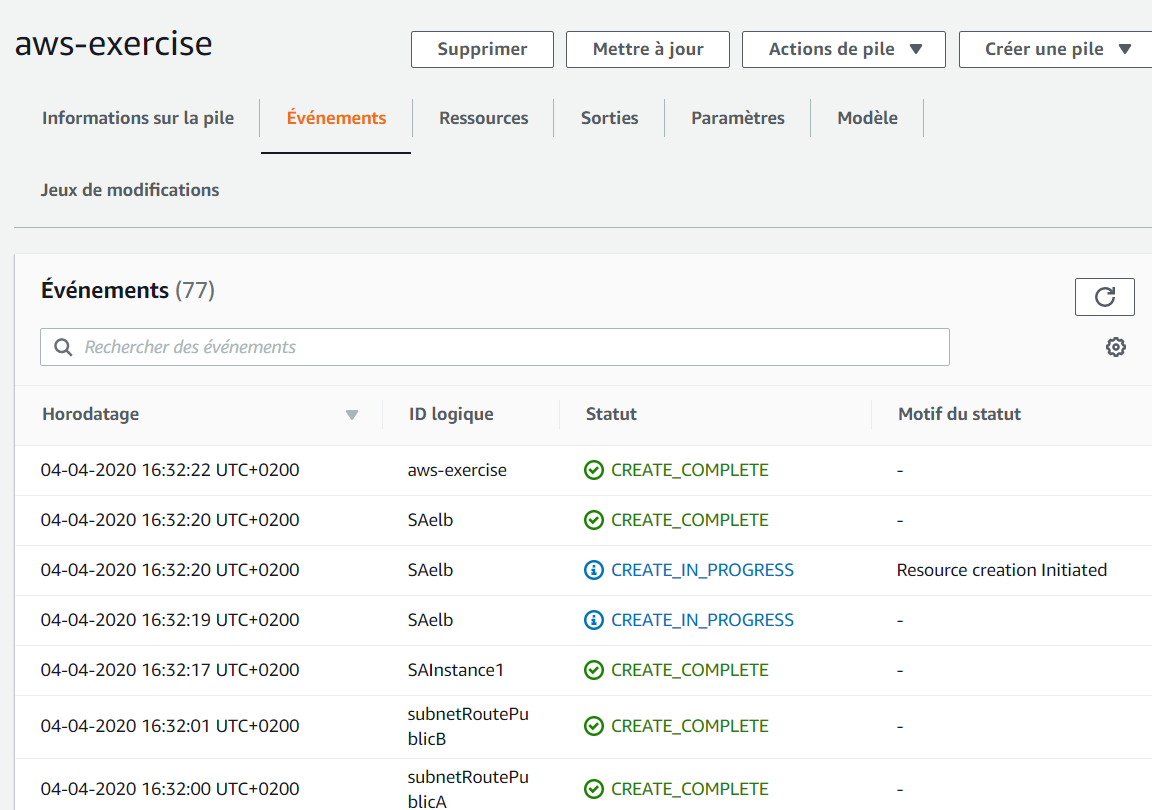
The figure below shows the architecture of the web application in CloudFormation Designer.



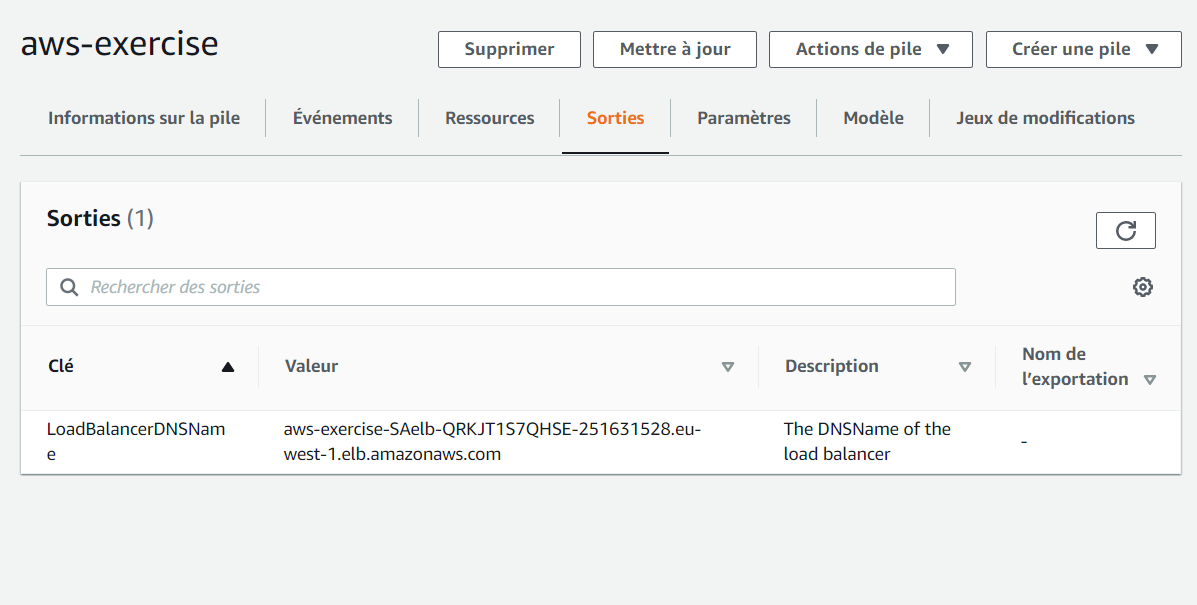
I proceed with the validation of the Cloudformation Stack; follow with the creation of the latter and finally the Stack is successfully created in AWS.



The CloudFormation Stack logs shows the events leading to its creation and its status.



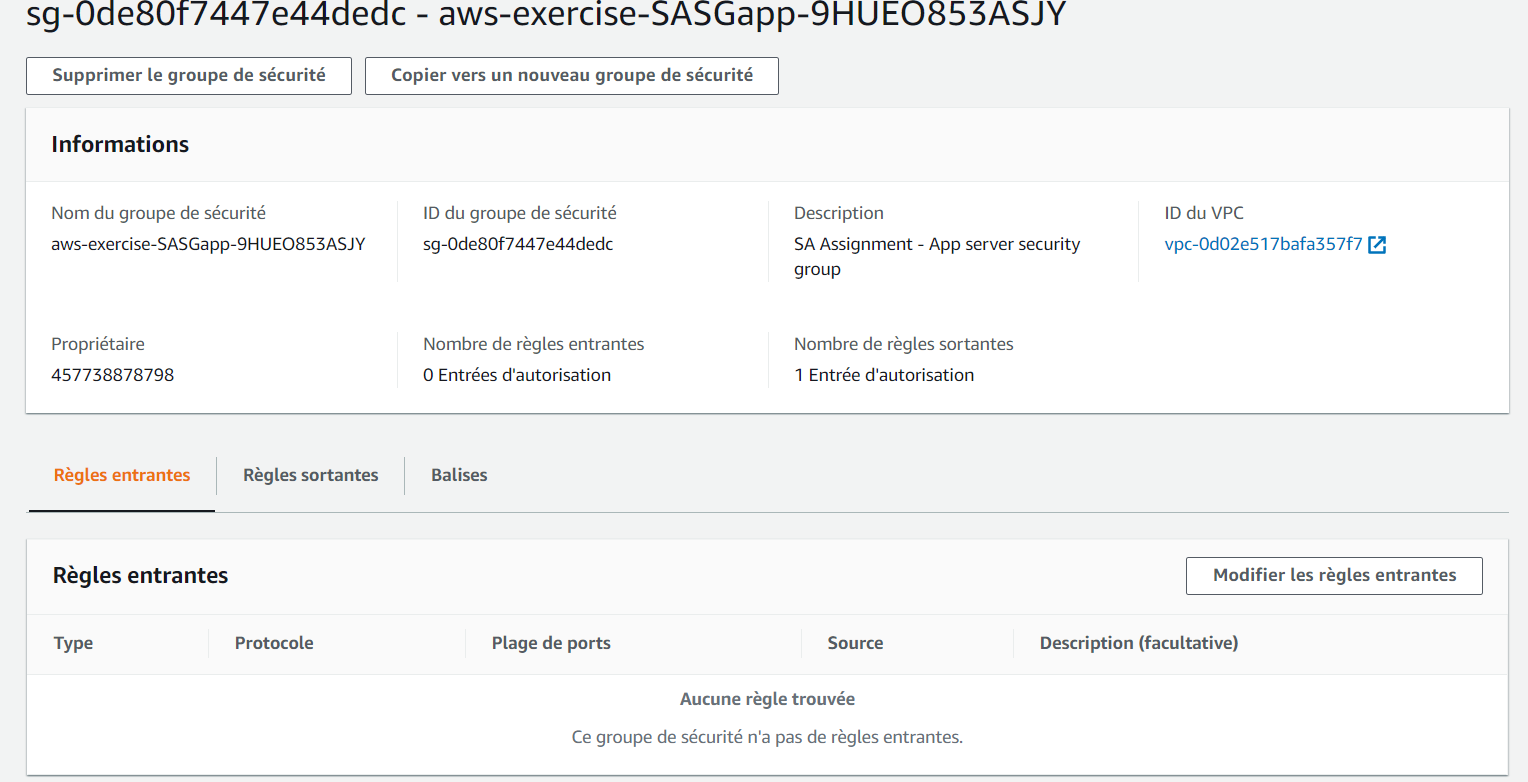
Next, I look up the content of the stack output to search for the Application-landing page Uniform Resource locator (URL). I shall test to see if it is running or not. The URL in question is the DNS of the classic Load Balancer.



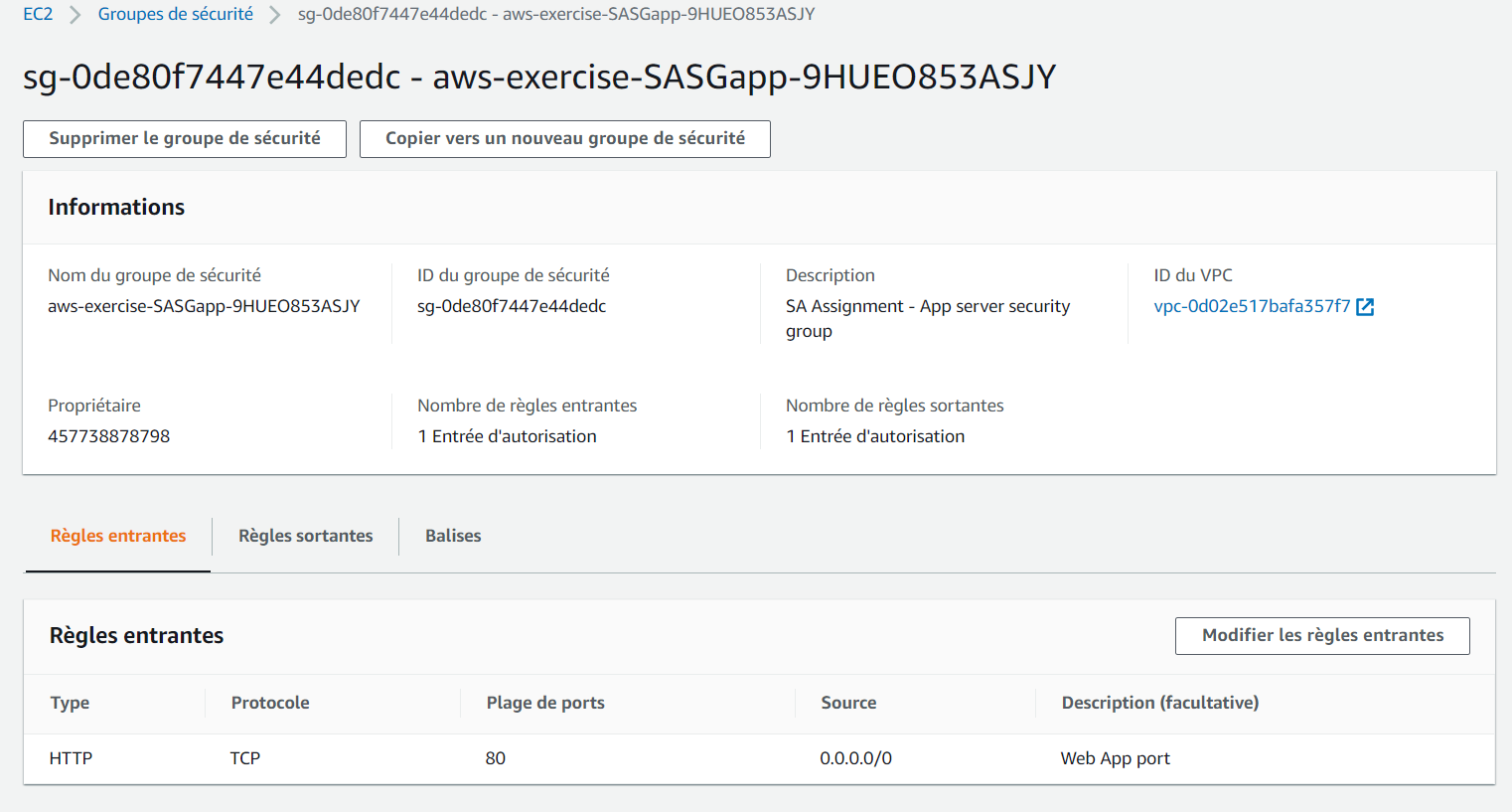
When access via a browser, the application is not responding: the content is not being displayed.

Common web application troubleshooting techniques teach us that such behavior is primarily due to blocked, not opened communication ports; notably port 80.

I decide to head to the EC2 service and corroborate port 80 of the instance is configured in the corresponding security group. It is not.



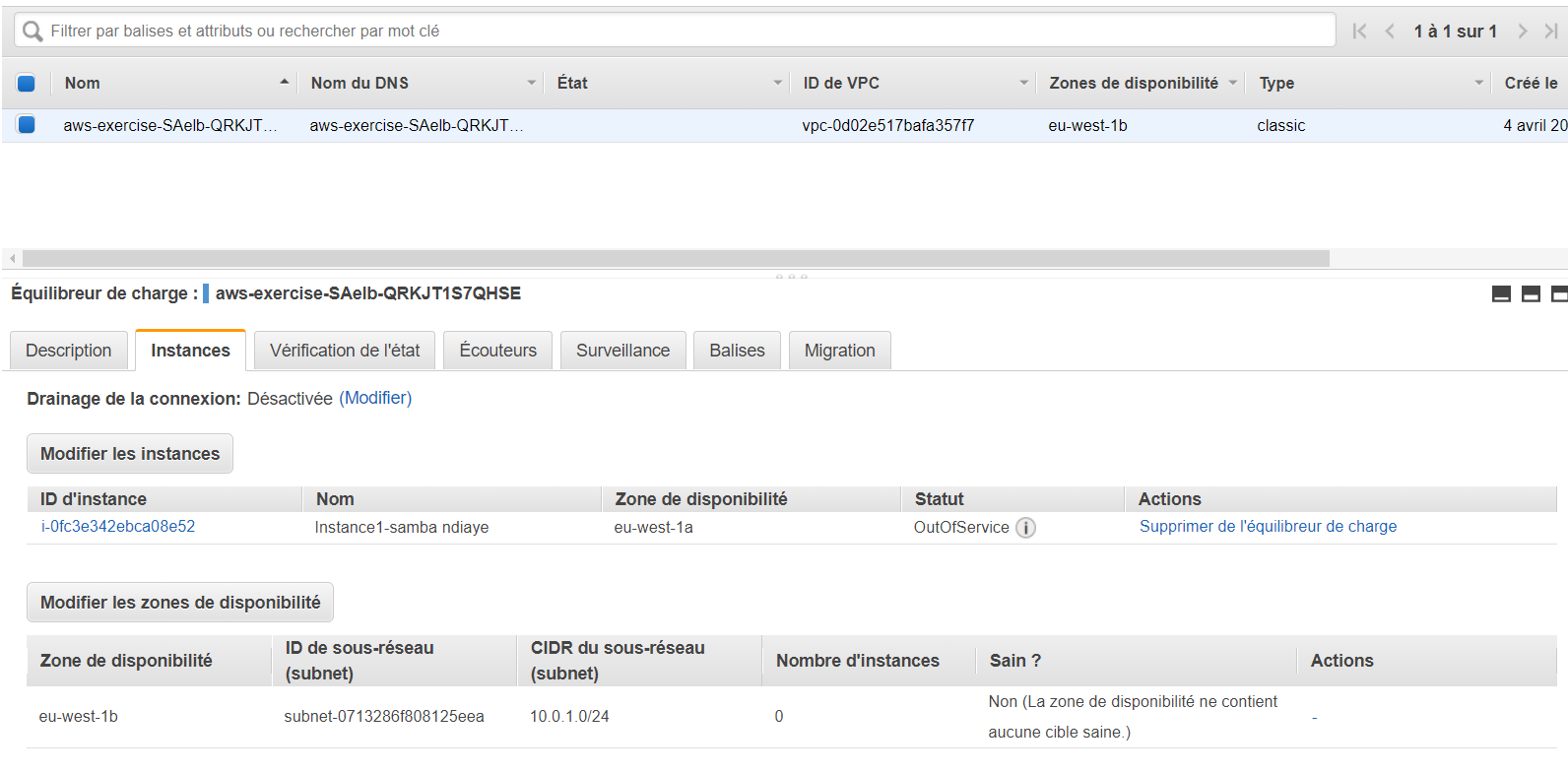
I create an allow rule (incoming) to allow access to the instance web port 80.



I return to the browser and test the application but to no avail. I can continue my troubleshooting session.

The requests flow from the instance to the Classic Load Balancer and finally to the user that initiated it. The instance is allowing the traffic. Is the Load Balancer doing the same?

To answer this question, I proceed to the Classic Load Balancer management interface and verify its state.

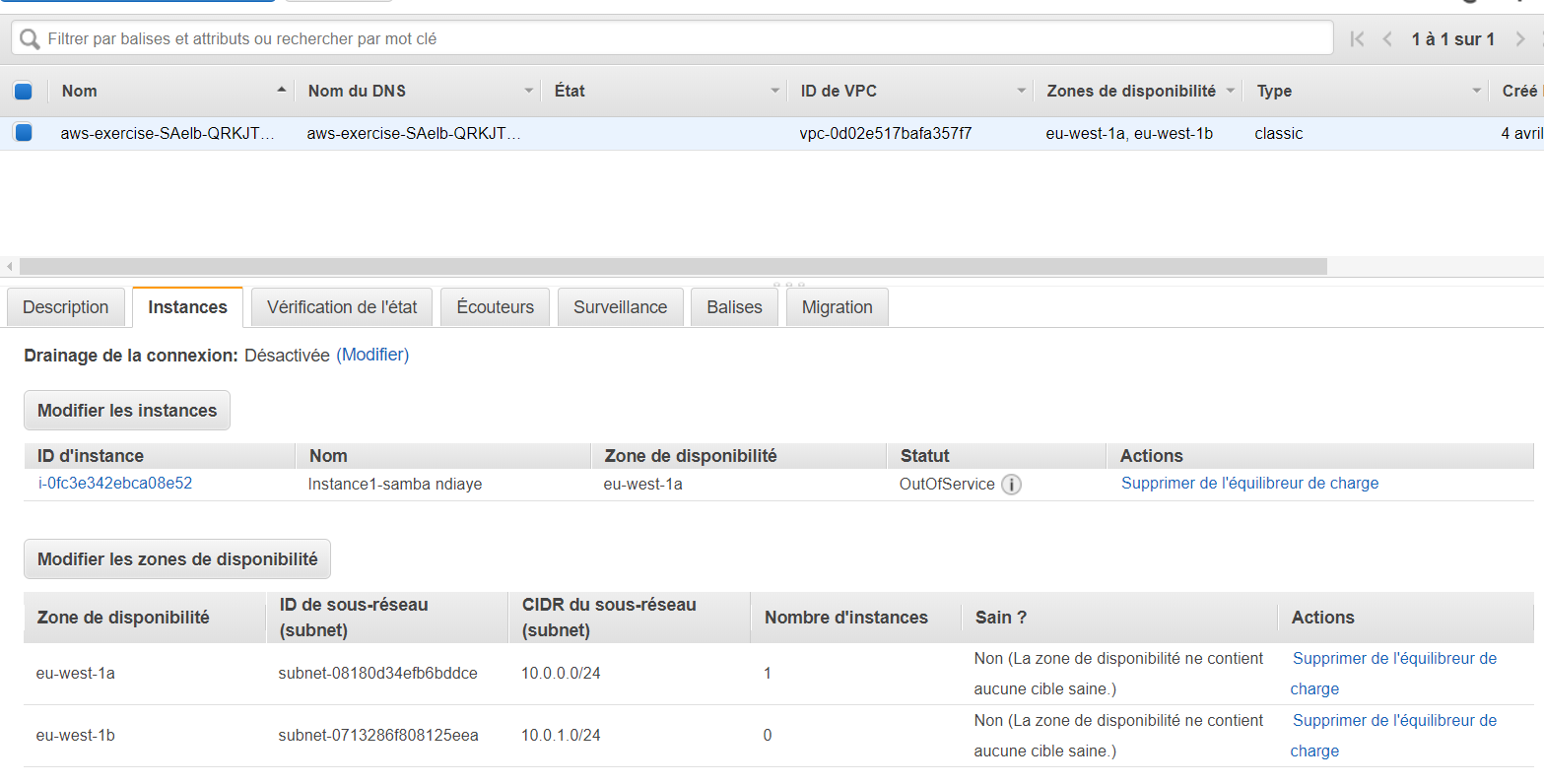


I check the instance tab of the load balancer: The EC2 instance is out of service. The reason: “instance is in the EC2 availability zone for which load balancer is not configured to route traffic to”.

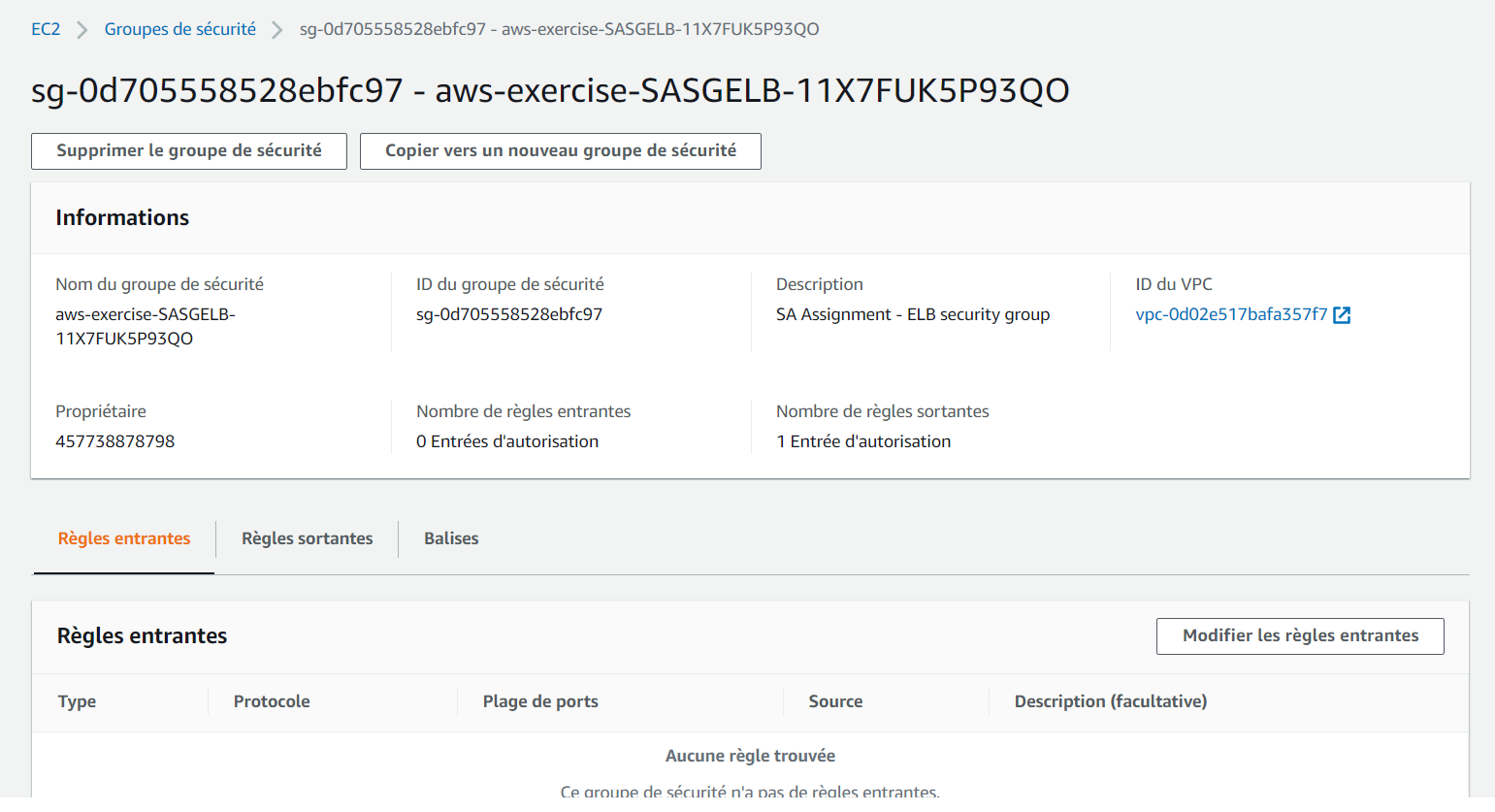
I remediate this issue by adding the subnet where the EC2 instance is deployed (eu-central-1a AZ) to the load balancer available subnets.



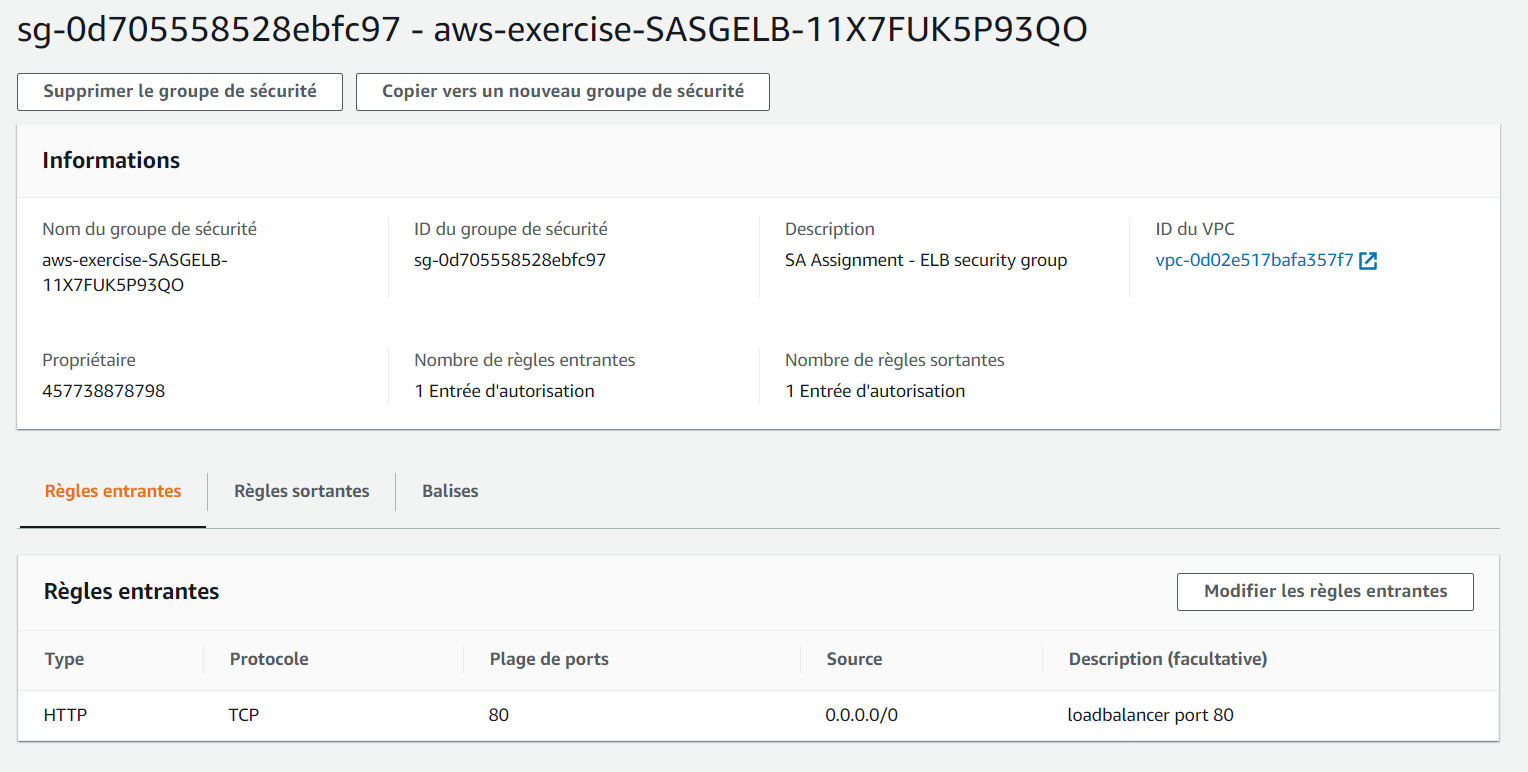
Performing these tasks (above) leads to the Load Balancer routing traffic to all subnets.



Next, I verify the security group configuration of the ELB. There is no entry at all. That partly explain why our application is not responding.

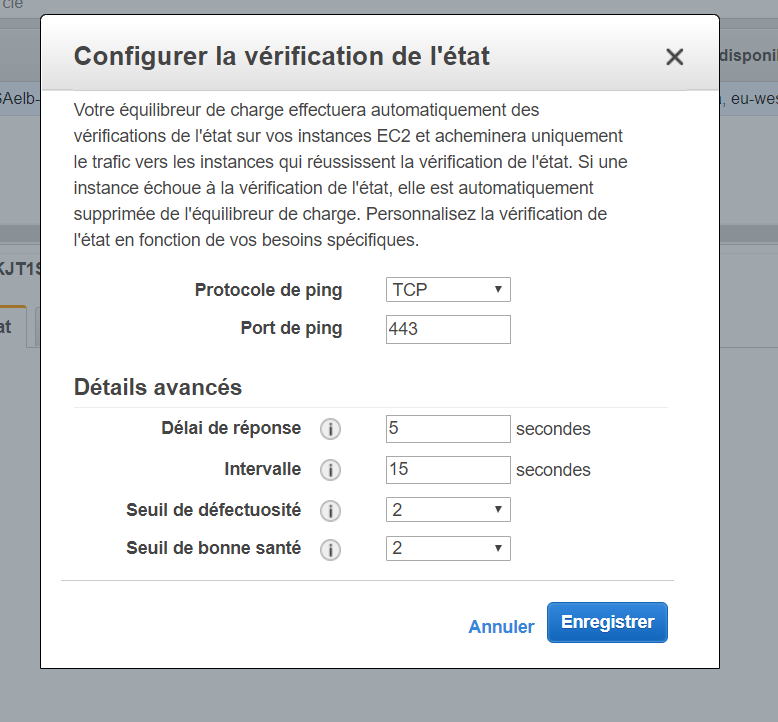


I add an incoming traffic entry, allowing port 80, to the ELB security group.

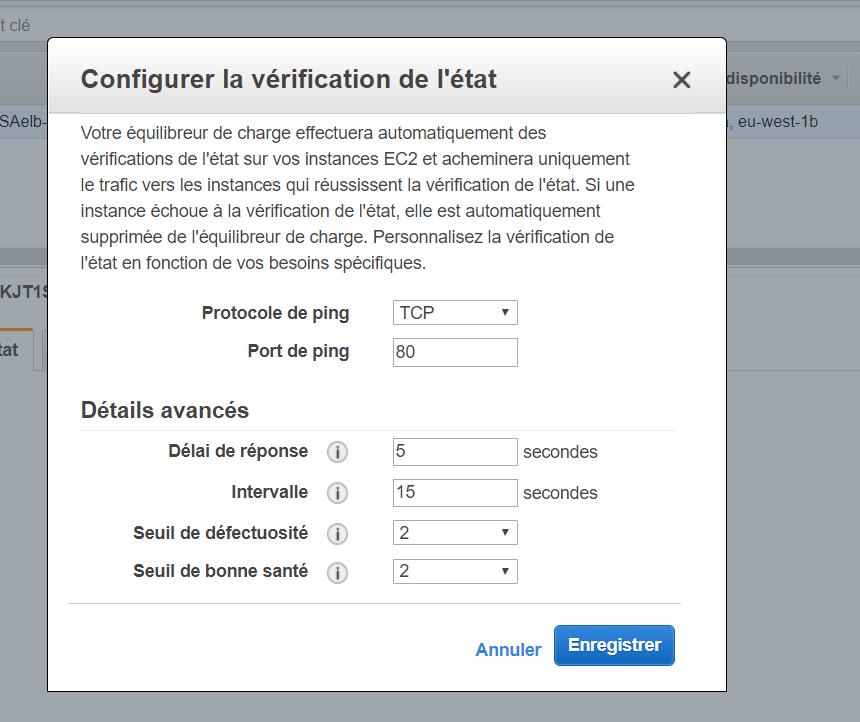


The ELB still displays an error message: Instance still out of service reason: “failed at least the Unhealthy Threshold number of health checks consecutively”.

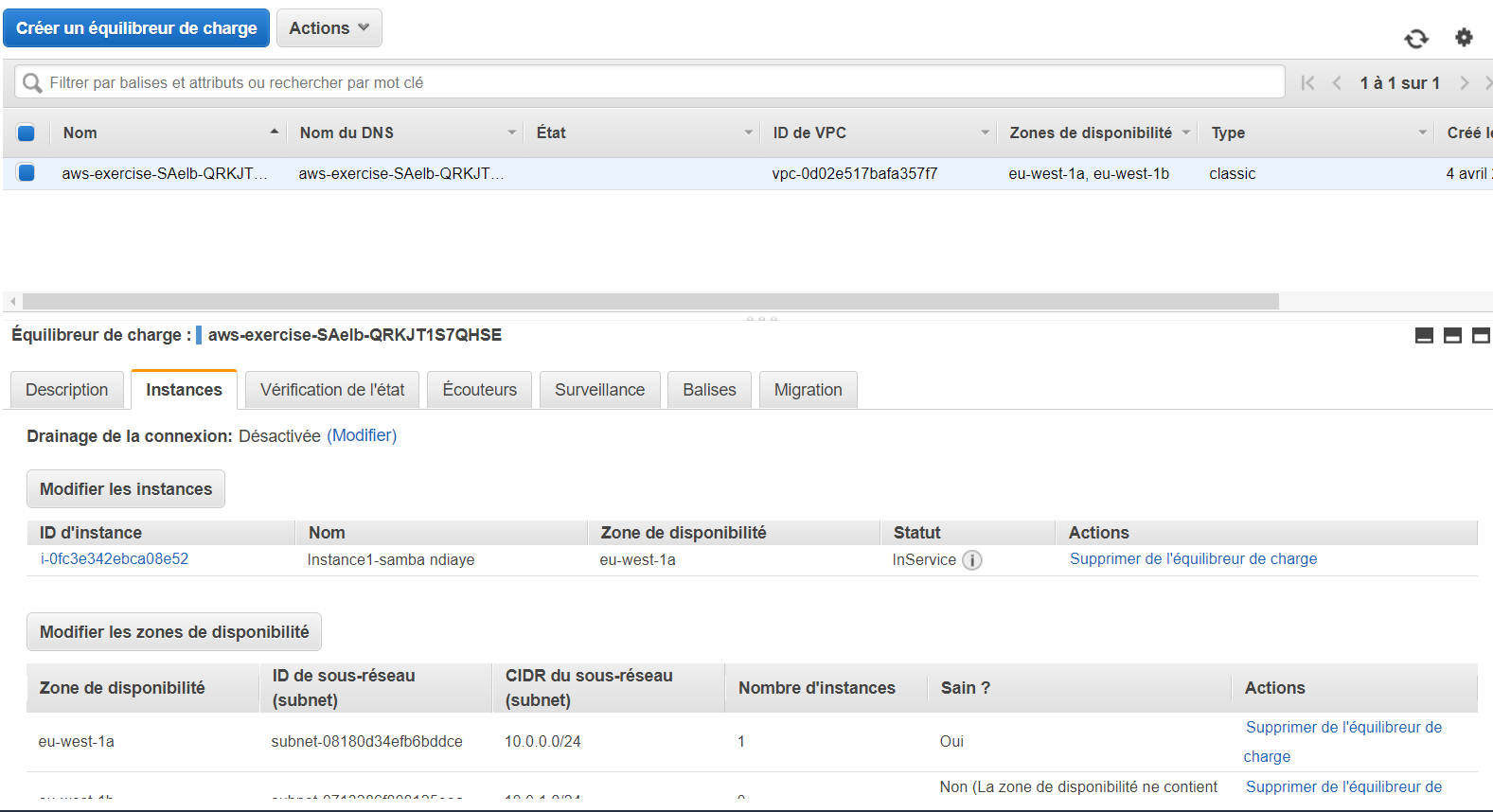
I decide to go along and verify the health check configuration of the ELB. The ELB, as presently configured, performs health checks on port 443.



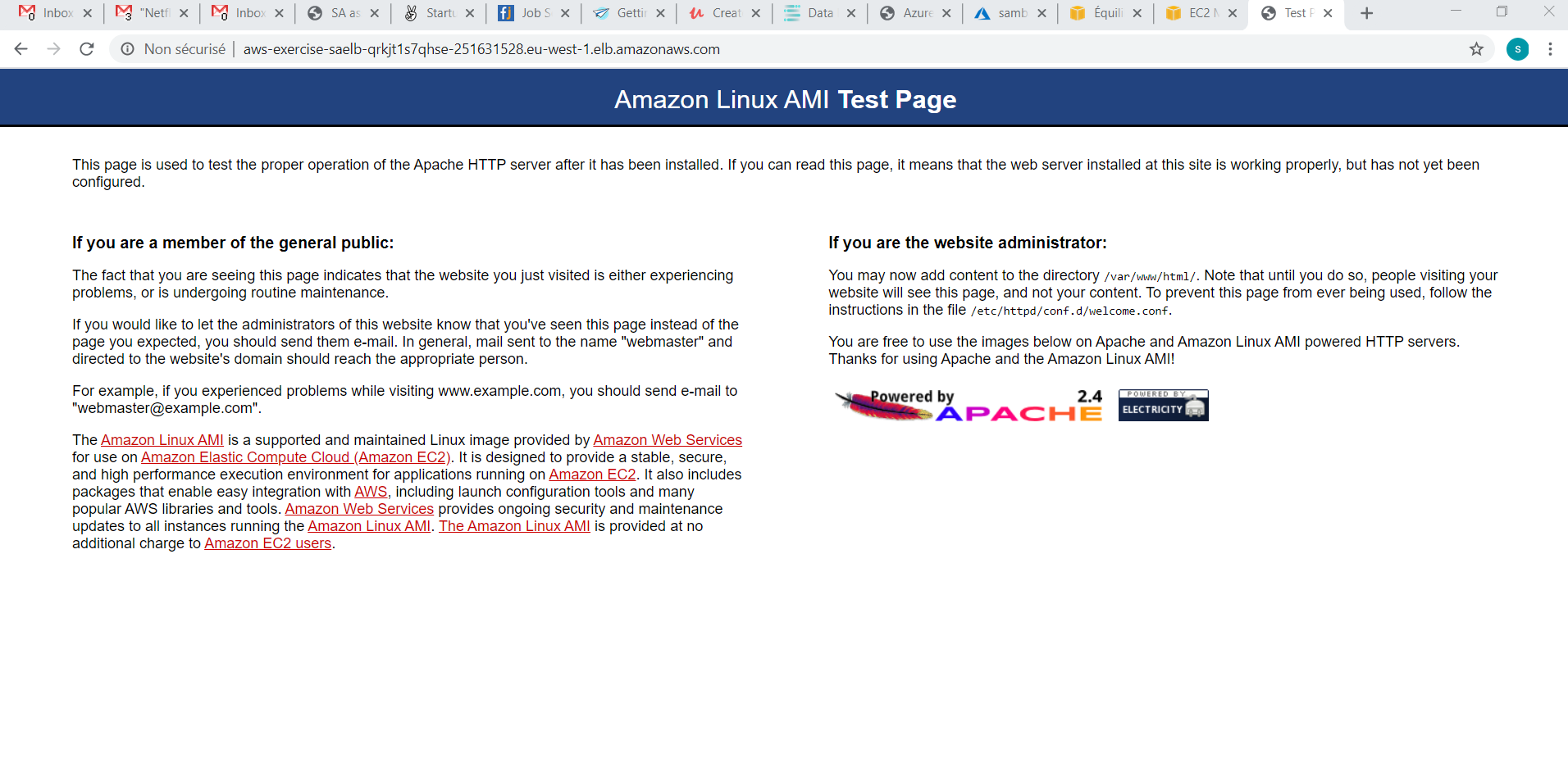
The ELB security group does not allow port 443. Therefore, I update the ELB configuration (ping port) to port 80.



Now, the message “instance is in-service” is displayed.



That concludes our troubleshooting session. The application is live and is showing its content.



## C - As-is Architecture short term changes

The application, even though functional, is not yet ready for production use. It is not highly available because the “at least two EC2 instances” redundancy requirement is not met. Amazon Web Services has deprecated the Classic Load Balancer that does not offer all the capabilities of an Application Load balancer.

To classify the application as “production ready”, we shall perform the following major changes:

1. Migrating from a Classic to an Application Load Balancer (ALB)
2. Setting up Auto Scaling
3. Deploying Bastion hosts, for operations team to connect and perform administrative duties, in the public subnets. This requires defining a private SSH keys during instance creation and opening port 22 on the instance security group.
4. Wrapping those bastion hosts in an ASG.
5. Deploying two NAT Gateways; one in each public subnet. The NAT Gateway will allow for instances in the private subnet to access the internet (for software updates).

The following topic highlights two of the most crucial additional changes: Auto Scaling setup and Application Load Balancer Migration.

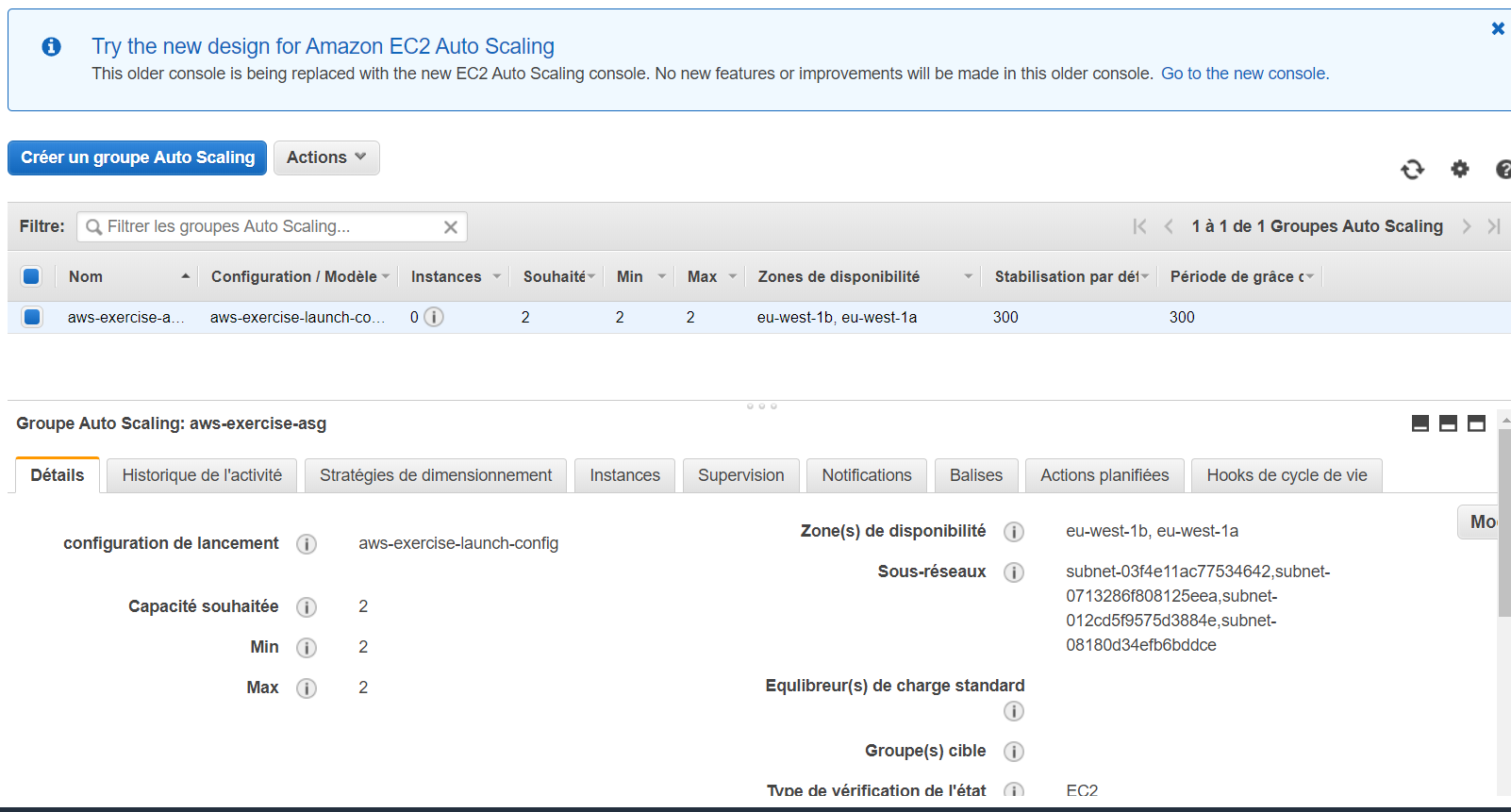
### 1 - Auto Scaling Definition

To meet the scalability and, to some extent, part of the performance pillar, the application must be scalable. In AWS, an Auto Scaling Group allows for the increase or decrease of the number of instance based on chosen, pre-defined metrics.

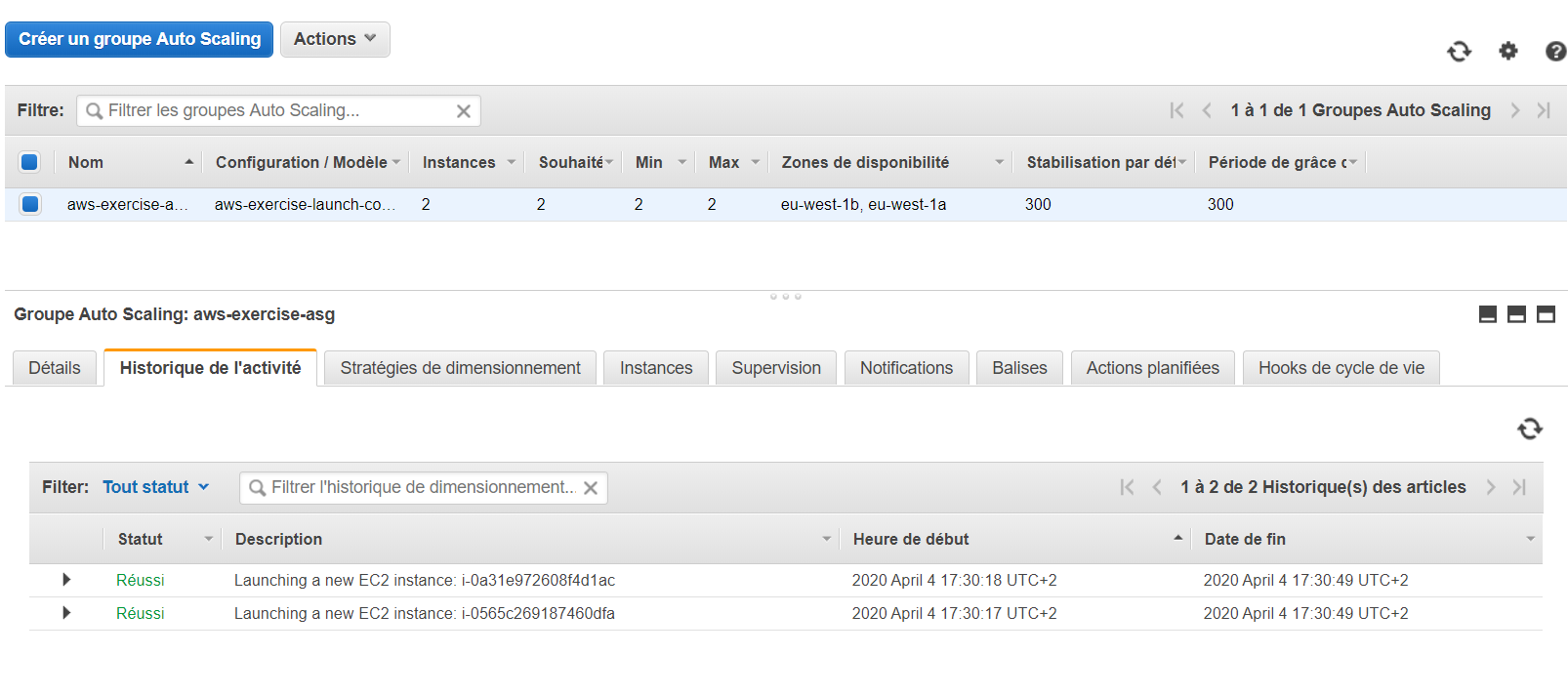
AWS defines an Auto Scaling Group (ASG) as:

“*AWS Auto Scaling monitors your applications and automatically adjusts capacity to maintain steady, predictable performance at the lowest possible cost. Using AWS Auto Scaling, it is easy to setup application scaling for multiple resources across multiple services in minutes. The service provides a simple, powerful user interface that lets you build scaling plans for resources including* [*Amazon EC2*](https://aws.amazon.com/ec2/) *instances and Spot Fleets,* [*Amazon ECS*](https://aws.amazon.com/ecs/) *tasks,* [*Amazon DynamoDB*](https://aws.amazon.com/dynamodb/) *tables and indexes, and* [*Amazon Aurora*](https://aws.amazon.com/aurora/) *Replicas. AWS Auto Scaling makes scaling simple with recommendations that allow you to optimize performance, costs, or balance between them. If you are already using* [*Amazon EC2 Auto Scaling*](https://aws.amazon.com/ec2/autoscaling/) *to dynamically scale your Amazon EC2 instances, you can now combine it with AWS Auto Scaling to scale additional resources for other AWS services. With AWS Auto Scaling, your applications always have the right resources at the right time*.”[[2]](#footnote-2)

I will modify the application to use an ASG in this phase. I create a launch configuration (with the initial EC2 instance‘s AMI) and a definition of the minimum number of instance in the Launch Configuration. I choose two (2) minimum instances.



An EC2 instance is running on each AZ and the ALB distributes the load accordingly. I remove the original instance since it is not needed anymore.



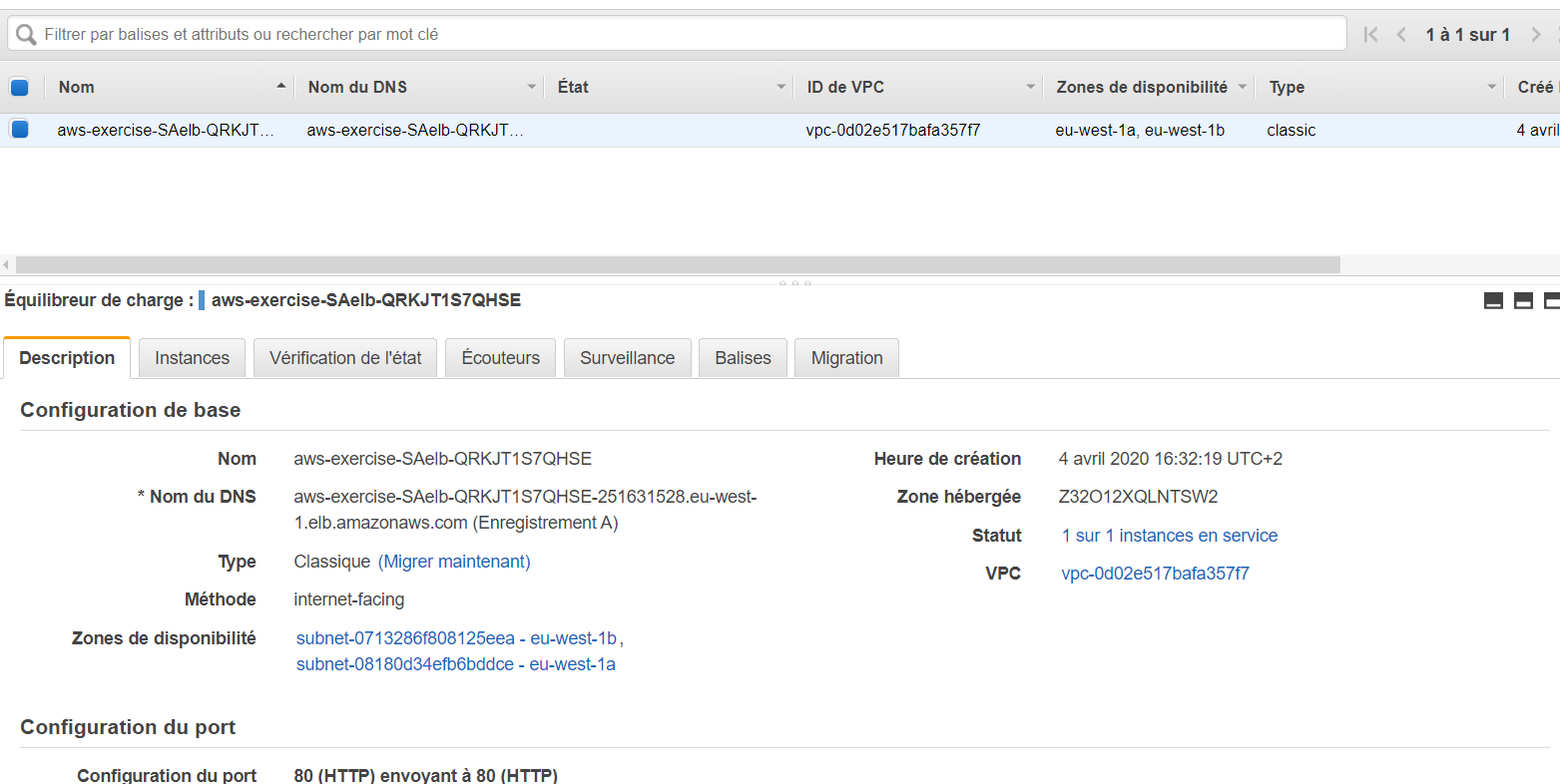
### 2 - Application Load Balancer Migration

The application, as stated before is architected with a Classic Load Balancer. Amazon recommends using either a Network or an Application Load balancer instead. We need to migrate the Classic Load Balancer to an Application one.

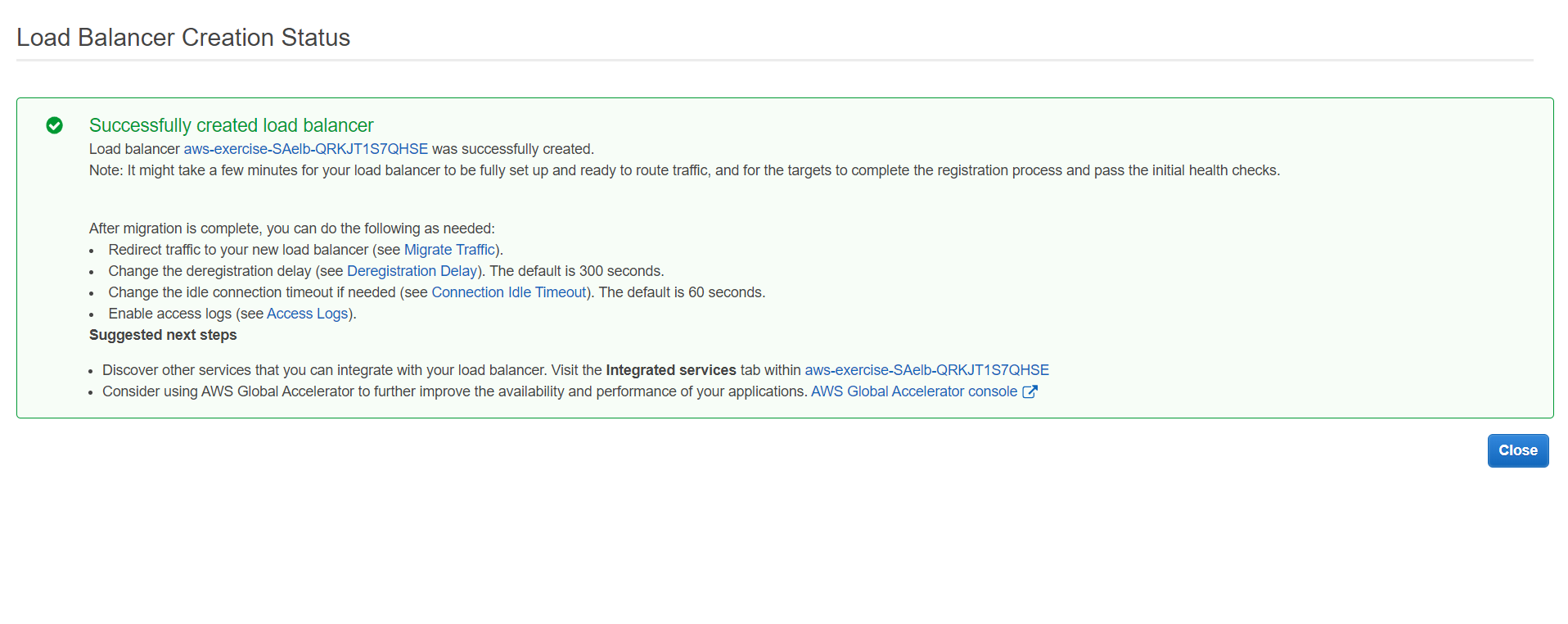
AWS defines an Application Load Balancer as:

“*An Application Load Balancer functions at the application layer, the seventh layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it evaluates the listener rules in priority order to determine which rule to apply, and then selects a target from the target group for the rule action. You can configure listener rules to route requests to different target groups based on the content of the application traffic. Routing is performed independently for each target group, even when a target is registered with multiple target groups. You can configure the routing algorithm used at the target group level. The default routing algorithm is round robin; alternatively, you can specify the least outstanding requests routing algorithm*.”[[3]](#footnote-3)

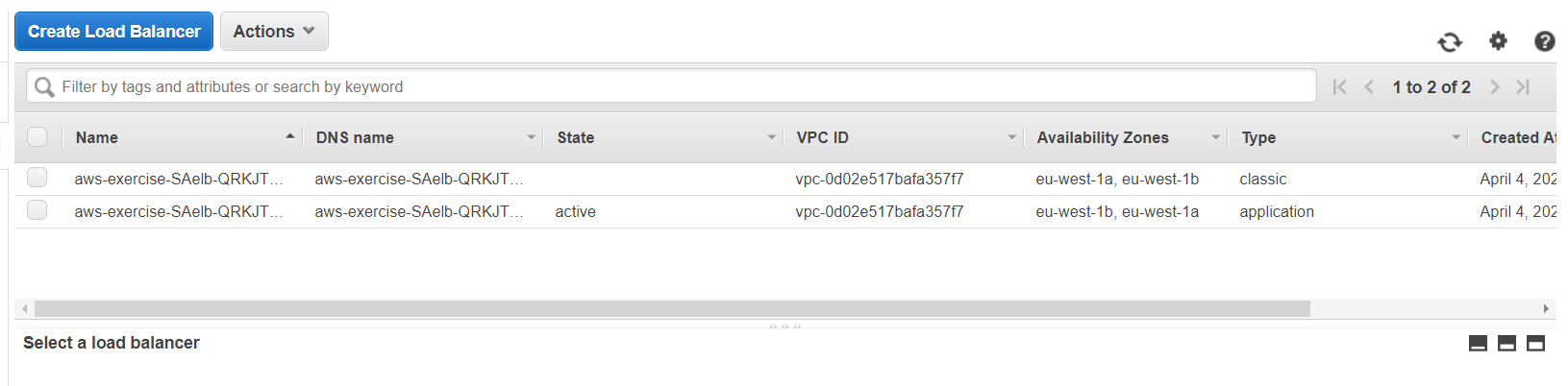
The figure below shows the option to migrate in the Aws Console Load Balancer service page.



I launch the ALB migration wizard, follow all the steps in the wizard and the migration completes successfully



I verify the application is now using an ALB: ALB is active.



I test the application by going to the ALB DNS page and confirm it is running. I proceed and delete the Classic Load Balancer.

That is the conclusion of the troubleshooting phase. All major, blocking issues are removed.

## D- Improvements & Enhancements

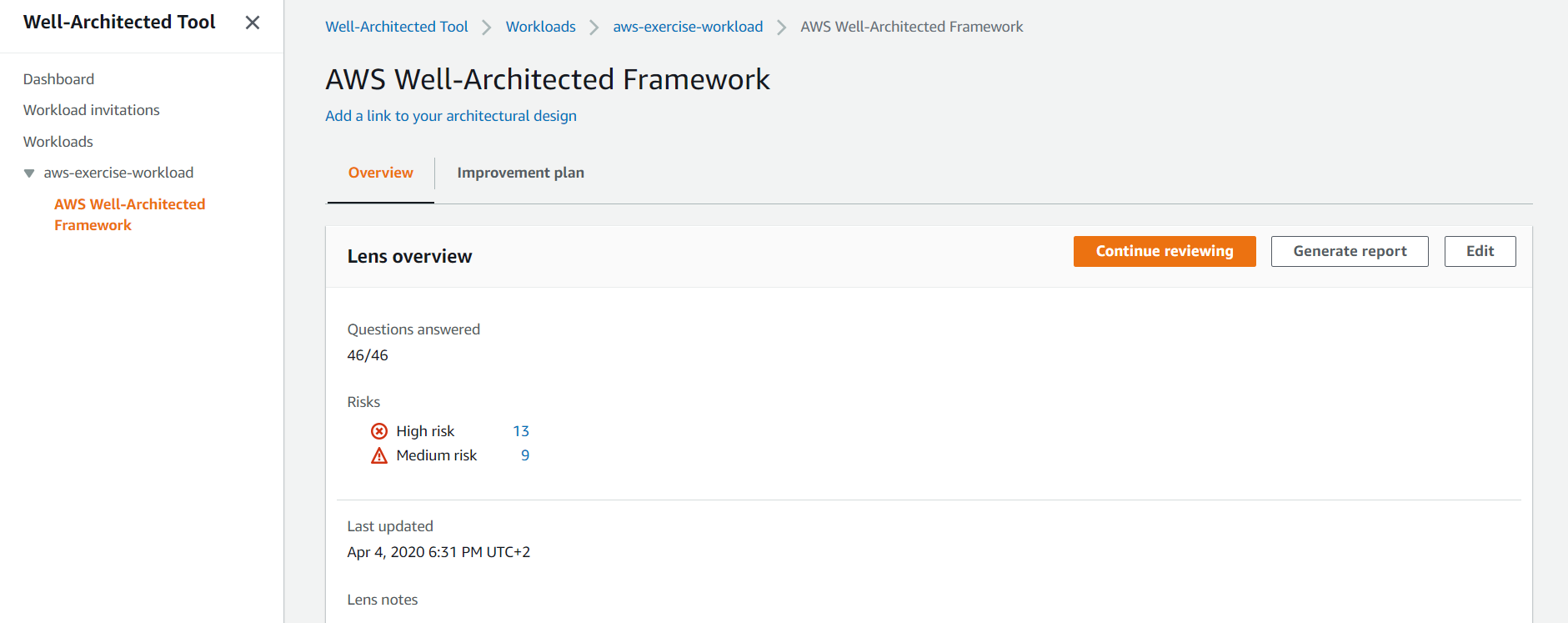
This section covers the short-term improvements and enhancements to the as-is architecture of the InnovLab web application. It consists of a series of design/implementation patterns & best practices commonly shared by AWS professionals.

These changes ensure the application adheres to AWS Well-Architected Framework principles.

“*The* [***AWS Well-Architected Framework***](https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf) *has been developed to help cloud architects build secure, high-performing, resilient, and efficient infrastructure for their applications. Based on five pillars — operational excellence, security, reliability, performance efficiency, and cost optimization — the Framework provides a consistent approach for customers and partners to evaluate architectures, and implement designs that will scale over time.”[[4]](#footnote-4)*

These non-functional requirements, coupled with the functional ones, justify the business value of deploying the application in AWS.

I have run the AWS Well-Architected Tool, answered all the questions and downloaded the final report. That did allow me to identify risk and take mitigation actions.



### 1 – Reliability

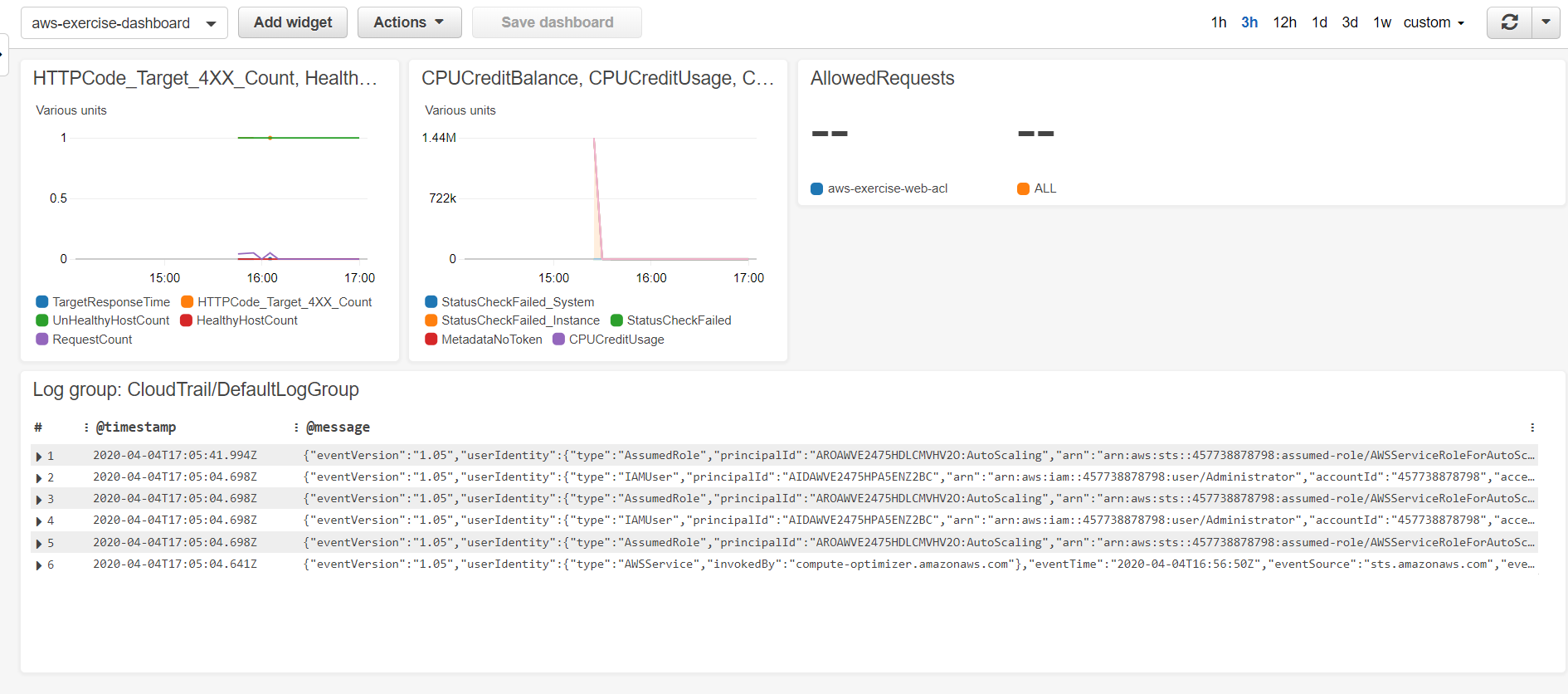
The Reliability pillar includes the ability of a system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues.

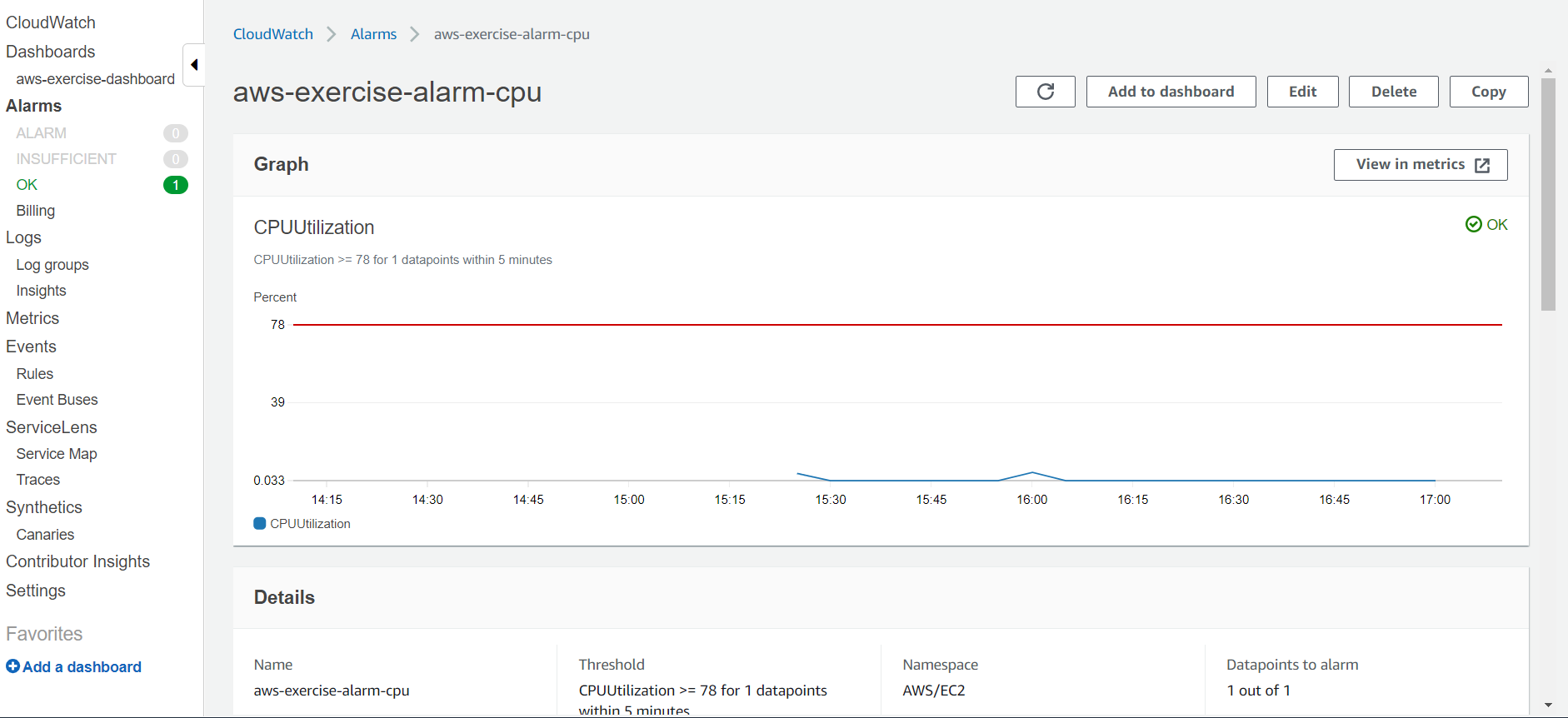
The reliability pillar is satisfied since the web app is deployed in two availability zones (high availability) and an auto-scaling group has been defined (scalability).

### 2 - Operational Excellence

The Operational Excellence pillar includes the ability to run and monitor systems to deliver business value and to continually improve supporting processes and procedures.

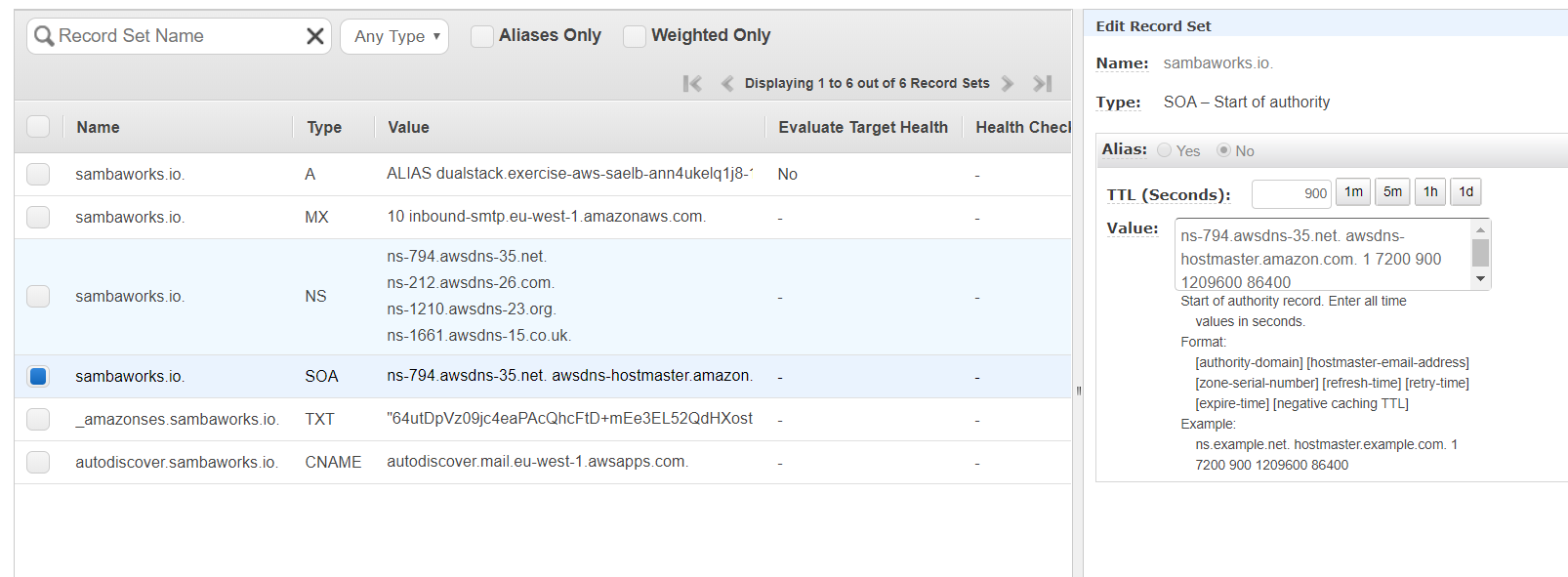
To satisfy this pillar, I have configured CloudWatch to collect metrics, record events via logs and designed Dashboards for InnovLab operations team.



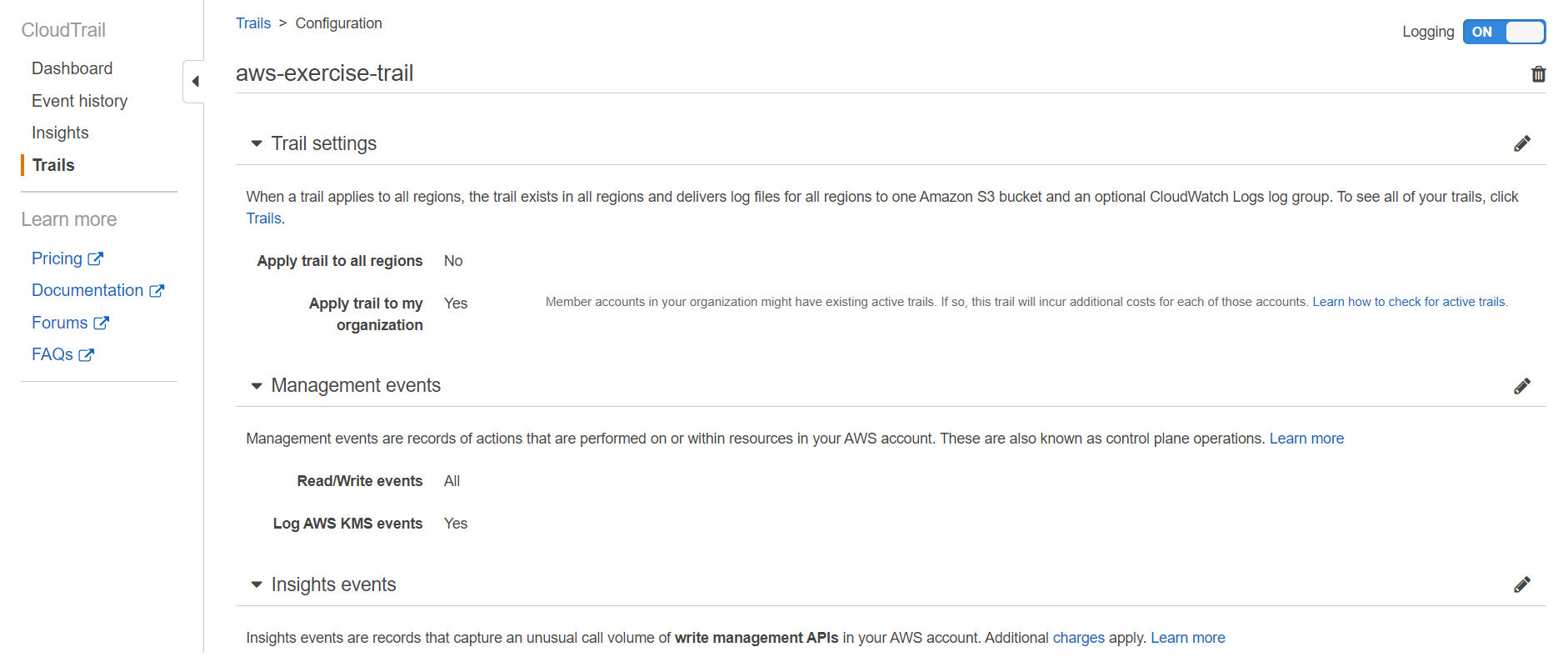


A CloudTrail trail is also available for auditing purposes.

I have registered a domain name, SambaWorks.io’ with Route53 and configured it, still in Route 53, with a Simple routed policy: Create an Alias in the Hosted Zone SambaWorks.io pointing to the ALB.



The application is now available at SambaWorks.io.



### 3 – Performance

The Performance Efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve.

I assume InnovLab dynamic application uses some kind of persistence layer: a Database. I have created an AuroraDB instance replicated in the second AZ. I have chosen AuroraDB because it is managed by AWS, freeing customer’s resources. AuroraDB is available and durable. It offers higher security, is highly scalable, compatible with MySQL & PostgreSQL, and comes with complete management.

In-memory data caching can be one of the most effective strategies to improve the overall application performance and to reduce database costs. In this implementation, I chose a deployment of ElastiCache in the private subnets where Aurora is located. Increased performance and reduced cost are the advantages gained from using ElastiCache.

For the time being, the application is able to handle the load with its scaling and high availability capabilities. I did run the AWS Compute Optimizer tool to gather insights and plan for future improvements.

### 4 – Cost

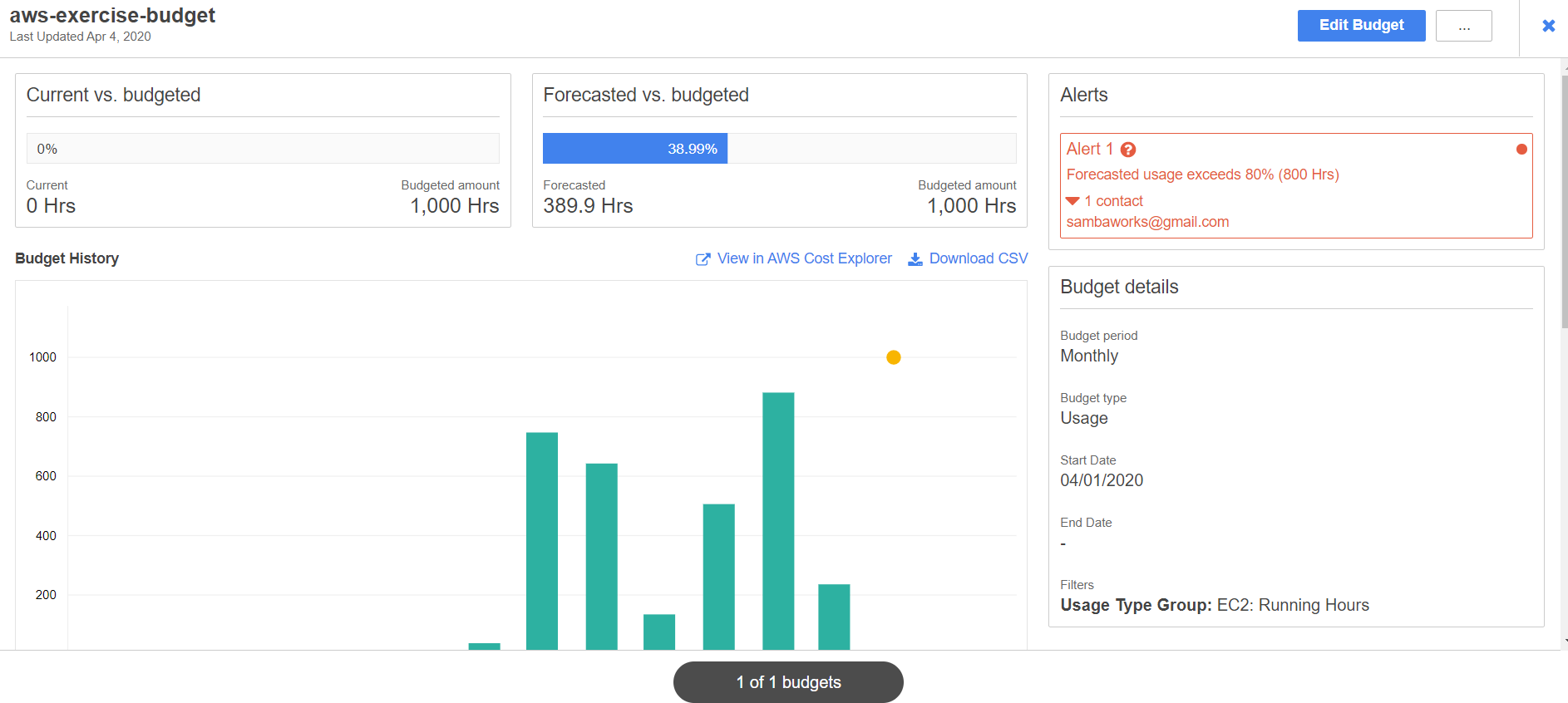
The Cost Optimization pillar includes the ability to run systems to deliver business value at the lowest price point.

I have experimented with the AWS Cost Explorer tool and generated reports based on resources, instance types or category. I have also downloaded the cost & usage report. I will provide all these cost related artifacts to InnovLab for future planning and optimization. During my work, I took the habit to annotate, to tag resources so their cost can be managed with more granularity (cost allocation tags).

The AWS-Generated Cost Allocation Tags results are available to InnovLab for future use.

I advised the customer to consider using Savings Plans. AWS Savings Plans are a flexible pricing model that offer low prices on AWS usage, in exchange for a commitment to a consistent amount of usage (measured in $/hour) for a 1- or 3-year term.

A strategy to purchase reserved instances, based on previsions, is more advantageous that using on demand instances. The savings could amount to about 70%.



### 5 – Security

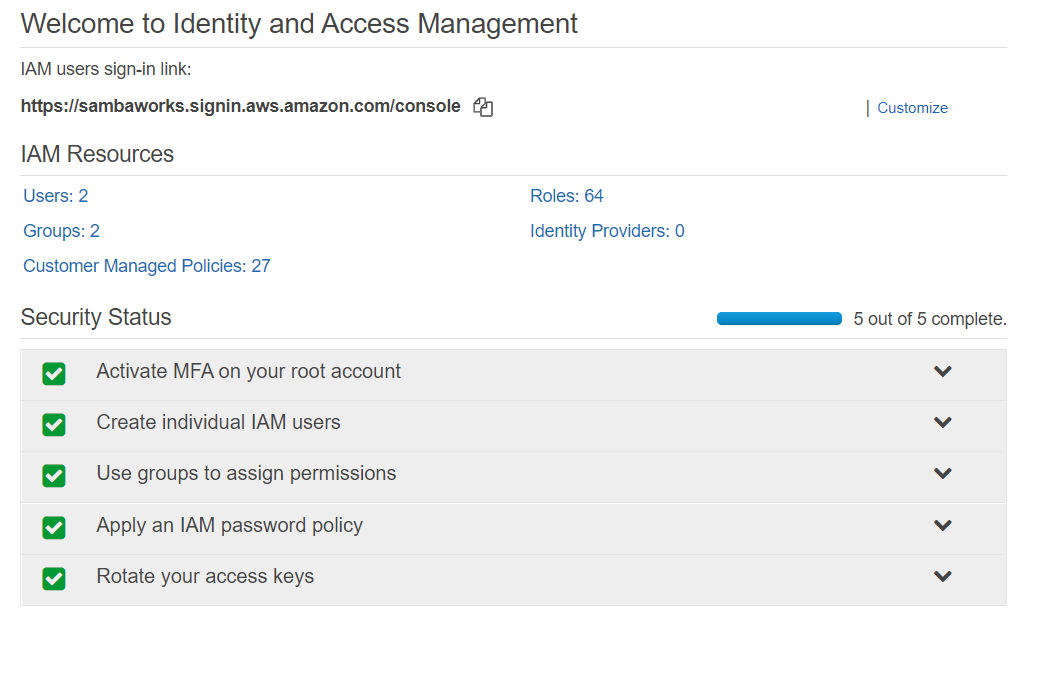
The Security pillar includes the ability to protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies.

By design, the application is already secure: VPC, NACLs and Security Groups etc…

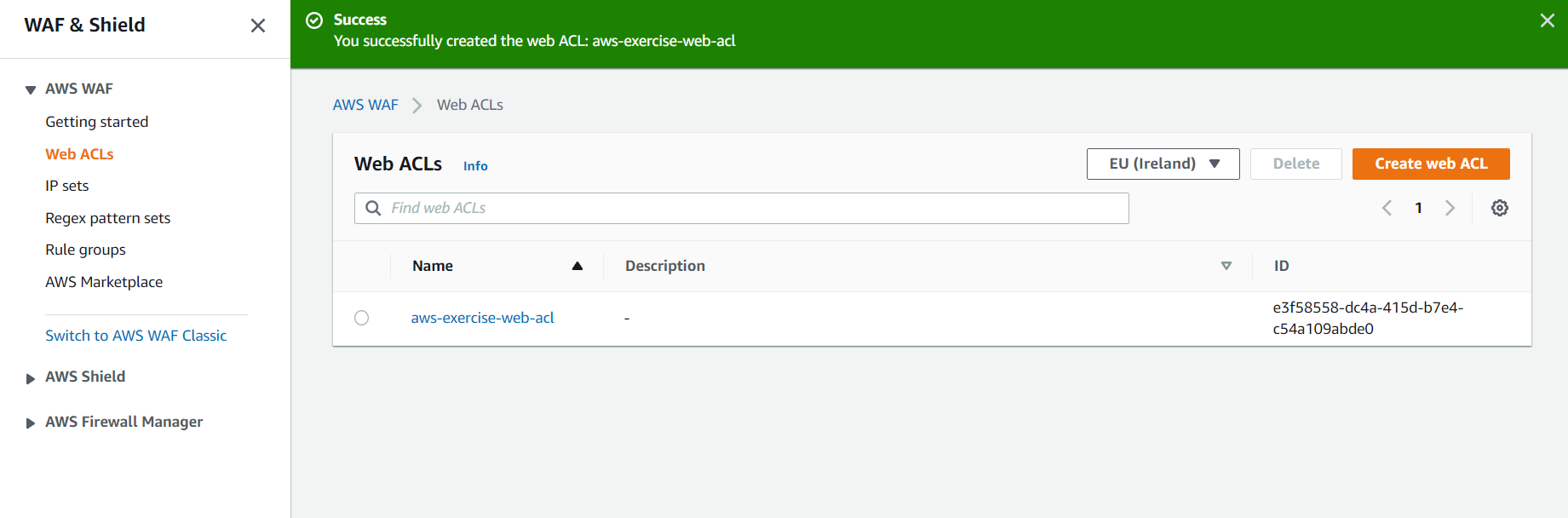
I ran the AWS trusted advisor and downloaded the corresponding report. I created an assessment template and triggered a run of AWS Inspector.

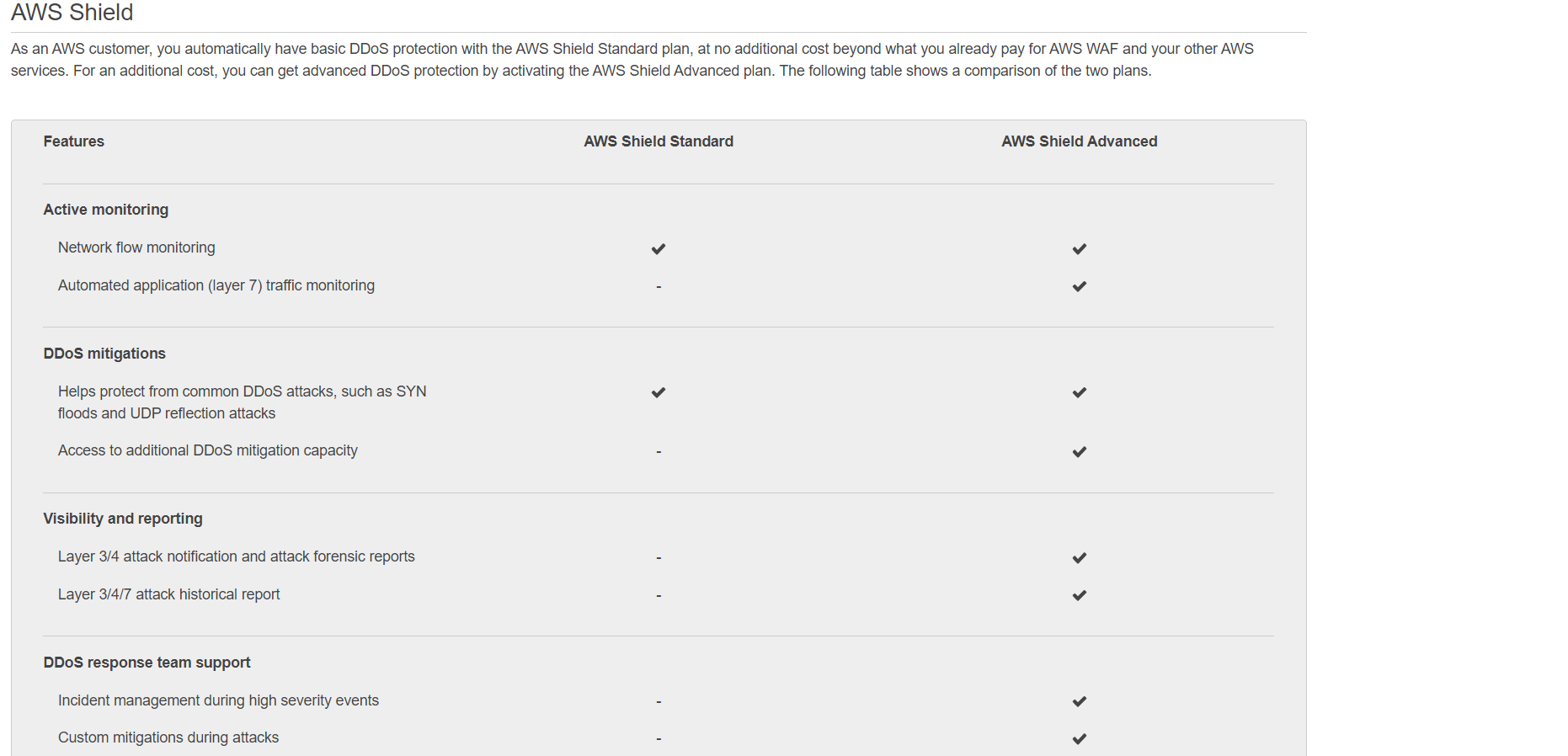
The below IAM, best practices have been implemented:

* Activate MFA on root account
* Create individual IAM users
* Use groups to assign permissions
* Apply an IAM password policy
* Rotate access keys

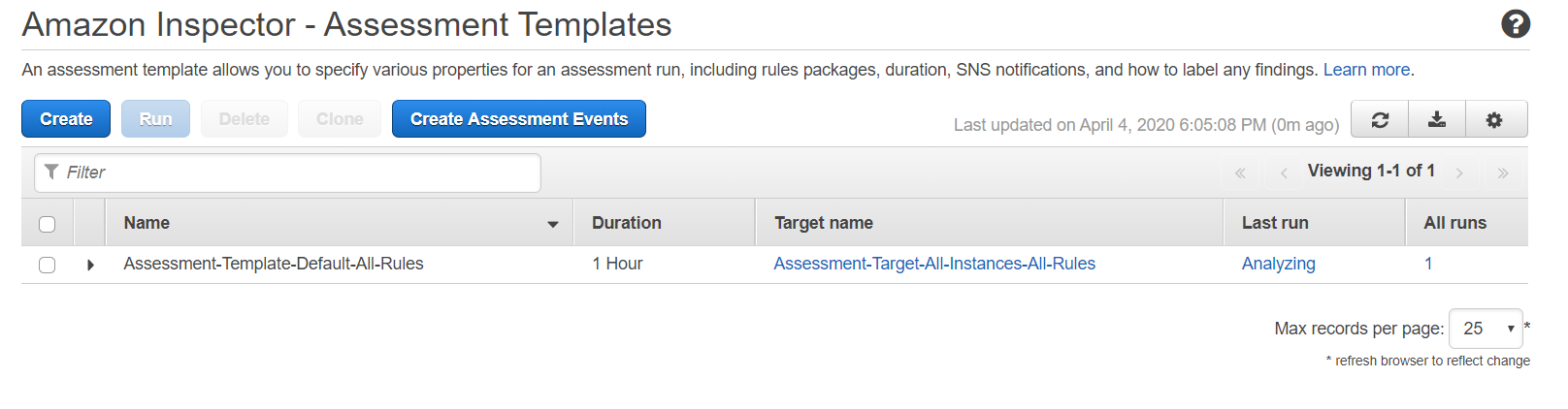


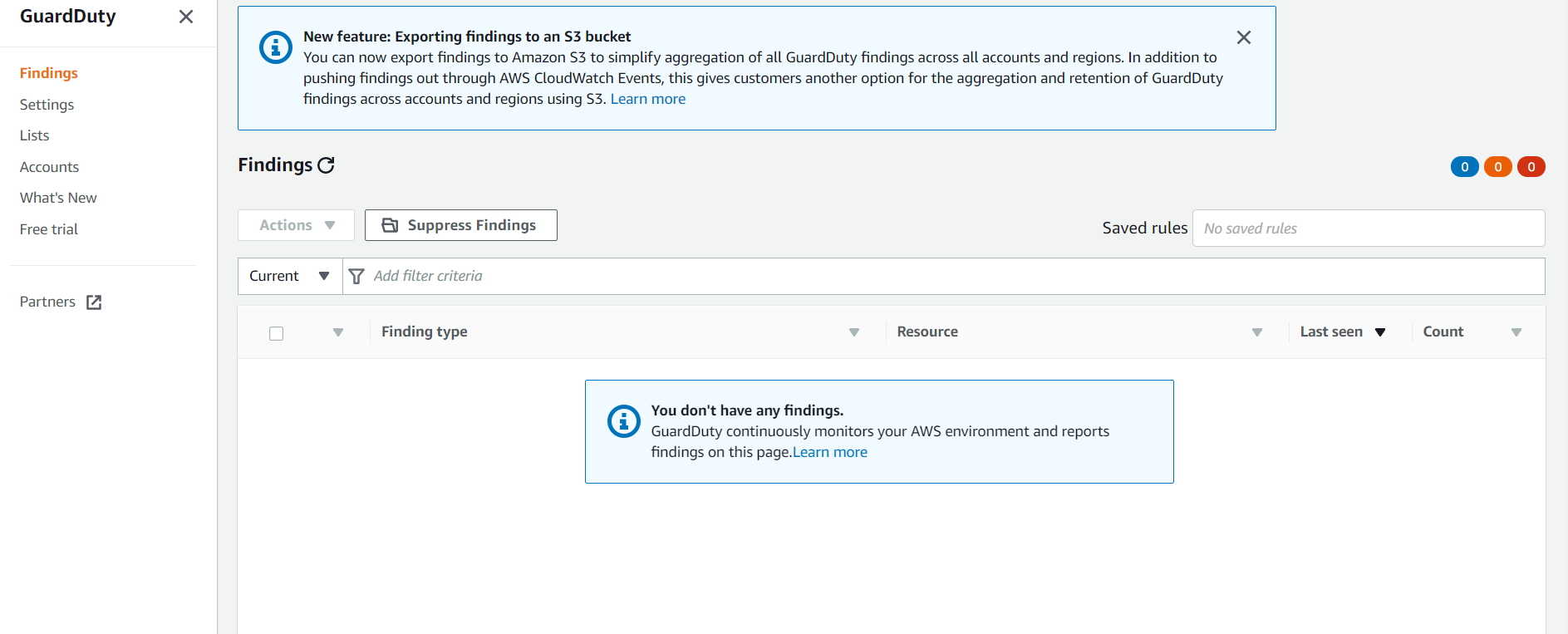
Web application in AWS are secured, at the ALB level, via the use of WAF. My solution does enabled WAF and AWS Shield.





Finally, I conclude the security implementation with a run of AWS Inspector and Guard Duty.





## E – Final Production Ready Architecture

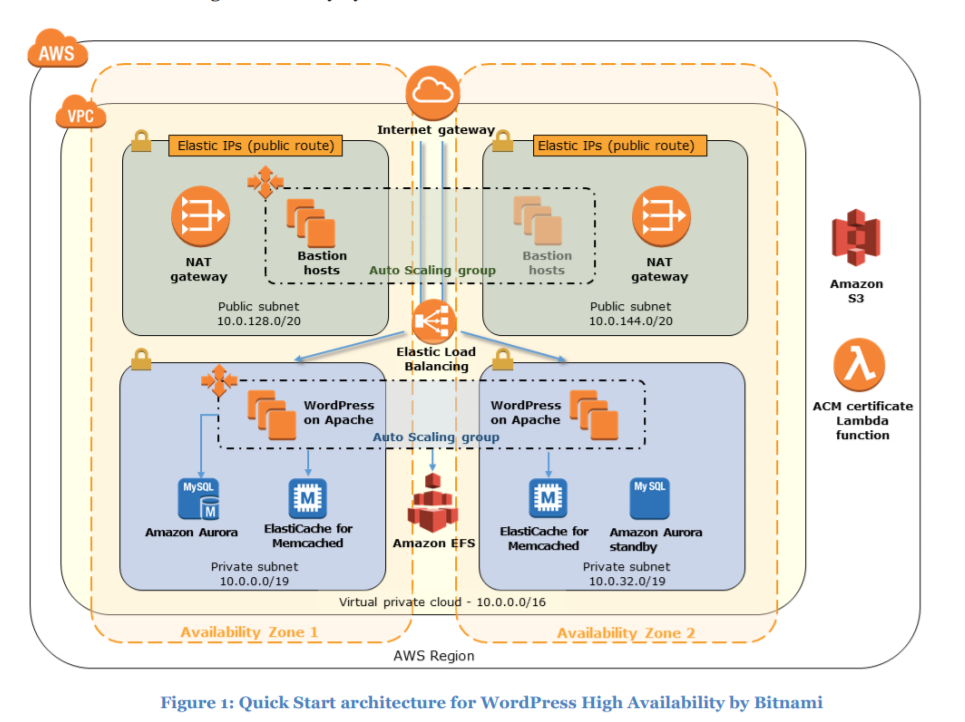
Once all the above changes, improvements & enhancements are applied, we shall have the architecture depicted in the diagram below. I call it the production ready architecture because it has all the components of a typical “AWS Web Application Reference” except for a shared file system. A shared file system is not needed at this point. InnovLab will make its future web app stateless so they can auto scale it easily. AWS EFS is a service that provide NFS type file system for Linux based hosts.

The Final production ready architecture is very similar to the “Quick start architecture for WordPress high availability by Bitnamy” (depicted below) with the sole exceptions:

* WordPress is replaced by the InnovLab Web Application
* The InnovLab Web Application is deployed in the public subnets
* There is no internal load balancer
* EFS is not implemented

**“sa-exercise-cloudformation-template.txt[[5]](#footnote-5)”** is a CloudFormation template generated by running CloudFormer to export the final web application built in my AWS environment. InnovLab will use it a starting point of their implementation work. I have provided a basic shell script that will help create the stack.

Information about the WordPress Bitnamy architecture are found [here](mailto:https://aws-quickstart.s3.amazonaws.com/quickstart-bitnami-wordpress/doc/wordpress-high-availability-by-bitnami-on-the-aws-cloud.pdf)[[6]](#footnote-6).



# II- Target Architecture

The target architecture for InnovLab should be available, have operational excellence built in and be reliable and cost effective. InnovLab has just started running workloads in the Cloud; they do not have enough in-house expertise and know how. InnovLab, as most startups, do not have unlimited financial resources. They need to react rapidly to ever changing requirements and be very agile.

Taking all the above into considerations, I advise InnovLab to opt for a Serverless architecture.

“The Serverless-computing model allows you to build and run applications and services without having to worry about infrastructure or servers. It eliminates infrastructure management tasks such as server provisioning, patching, operating system maintenance, scaling, and capacity provisioning.”[[7]](#footnote-7)

There does exist, in AWS Documentation, a Serverless reference architecture that InnovLab will leverage for its target architecture: [Building A Modern Web Application](https://aws.amazon.com/fr/getting-started/projects/build-modern-app-fargate-lambda-dynamodb-python/)

I advise for the use of a “Micro service architecture pattern”. The Benefits of Micro services are:

* Easier to Build and Maintain Apps. The key principle of micro services is simplicity.
* Organized Around Business Capabilities
* Improved Productivity and Speed
* Flexibility in Using Technologies and Scalability
* Autonomous, Cross-functional Teams

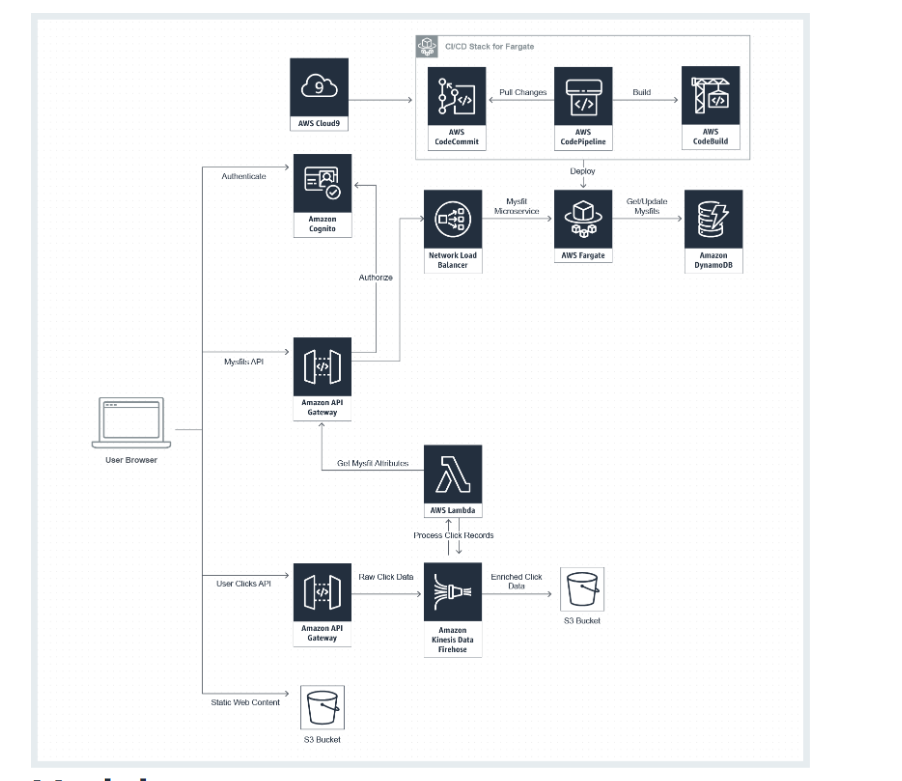
The following section details InnovLab physical target architecture (based on the “Modern Web Application” reference architecture).

## A - Physical Architecture

“[Modern applications](https://aws.amazon.com/fr/modern-apps/)[[8]](#footnote-8) isolate business logic, optimize reuse and iteration, and remove overhead everywhere possible. Modern apps are built using services that enable you to focus on writing code while automating infrastructure maintenance tasks.”[[9]](#footnote-9)

The architecture diagram below is a reference model. InnovLab will have to customize it to fit their specific needs. It represents a great starting point and is a great addition to InnovLab Enterprise Architecture Repository.

This application architecture diagram provide a structural representation of the services that make up an application.



Source: <https://aws.amazon.com/fr/getting-started/projects/build-modern-app-fargate-lambda-dynamodb-python/>

# ARTIFACTS

1. **sa-exercise-cloudformation-template.txt**: CloudFormer template that exports the AWS Web Application final solution.
2. **Create-stack.sh**: Shell script use to recreate the AWS environment (Infrastructure as Code). Takes sa-exercise-cloudformation-template.txt as argument
3. **aws-exercise-workload\_wellarchitected.pdf**: AWS Well-Architected Tool aws-exercise workload- AWS Well-Architected Framework Report

1. REFERENCES & [↑](#footnote-ref-1)
2. Source AWS: https://aws.amazon.com/autoscaling/ [↑](#footnote-ref-2)
3. AWS Application Load Balancer : Source: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html [↑](#footnote-ref-3)
4. Source AWS : https://aws.amazon.com/architecture/well-architected/ [↑](#footnote-ref-4)
5. Cf Artifacts section [↑](#footnote-ref-5)
6. https://aws-quickstart.s3.amazonaws.com/quickstart-bitnami-wordpress/doc/wordpress-high-availability-by-bitnami-on-the-aws-cloud.pdf [↑](#footnote-ref-6)
7. https://blogs.itemis.com/en/serverless-services-on-aws [↑](#footnote-ref-7)
8. https://aws.amazon.com/fr/modern-apps/ [↑](#footnote-ref-8)
9. https://aws.amazon.com/getting-started/projects/build-modern-app-fargate-lambda-dynamodb-python/ [↑](#footnote-ref-9)