POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Cybersecurity

Master's Degree Thesis

Analysis and development of a monitoring system for WAFs using AWS and ELK Stack

Supervisor
Prof. Cataldo Basile

Candidate
Davide Cosola

Academic Year 2023-2024
Turin

Abstract

In the last two decades, websites and web applications have played an essential role in modern society since they allow information sharing, provide a service for business purposes, and connect with multiple people globally.

Unfortunately, legitimate users are not the only source of traffic. Also, attackers and malicious bots can contribute a relevant share. For this reason, a web application needs to rely on a scalable, resilient, and fast infrastructure that provides security from cyberattacks.

Companies could design their solution to resolve this security issue, but due to the complexity, cost, and effort that these architectures' implementation requires, often it is not the best choice. Instead, it is possible to rely on services offered by another company that takes the responsibility to provide the security needed by their websites and web applications.

A web application firewall (WAF) is a cybersecurity solution that aims to analyze HTTP traffic to detect and filter malicious requests received by web applications. Since the WAF acts as a reverse proxy, users must pass through it to reach the web applications behind it. In this case, a WAF's malfunction can propagate to the protected services and cause downtime.

The thesis aims to analyze and develop a monitoring system that could verify the accessibility of the customers' web applications, their configuration, and the overall health status of a WAF infrastructure that relies on services offered by Amazon Web Services (AWS) and Elastic Cloud.

The solution would include periodic reports and alerting functionalities that can notify in case of malfunctions of varying severity. The goal is to improve the WAF's reliability through effective monitoring and timely response to issues to provide a step forward in the quality of service and the protection provided to the customer.

Acknowledgements

I would like to express my gratitude to the supervisors from aizoOn Technology Consulting, Simone Janin, and Giuseppe Vavala', and to all the other colleagues for their guidance and support that created a stimulating work environment. Furthermore, I am thankful to my academic supervisor, Prof. Cataldo Basile, for the opportunity to start this journey and the constructive feedback.

To my mother and father for their sacrifices and the values they instilled in me. I am thankful for their endless love and guidance.

To my brother, aunt, and grandparents for their continuous support and encouragement.

To my girlfriend for her help, patience, and constant belief in me during these challenging years.

Contents

1 Introduction 5
1.1 Thesis Organization . . . 6

2 Thesis Background 7
2.1 Observability and Monitoring . . . 7
2.2 Monitoring System: Impact . . . 8
2.3 Downtime . . . 9
2.4 Web Application Firewalls . . . 11

3 AWS Fundamentals 13
3.1 Introduction to AWS . . . 13
3.2 VPCs and Subnets . . . 15
3.3 AWS Identity and Access Management . . . 18
3.4 EC2 Instances . . . 20
3.5 ECS and Fargate . . . 21
3.6 AWS LBs and Global Accelerator . . . 22

4 Containerization and Simulation 25
4.1 Docker . . . 25
4.2 Logging in Python . . . 27
4.3 Simulating Issues using Python . . . 28
4.4 ECS Cluster Configuration . . . 30

5 Simple AWS Metrics 32
5.1 CloudWatch . . . 32
5.2 Simple Metrics and Logs . . . 33
5.3 Alarms and SNS . . . 34
5.4 Metric Filter . . . 36
5.5 Event-based Metric . . . 37

6 Serverless and the ELK Stack 40
6.1 AWS Lambda . . . 40
6.2 Importing Libraries in AWS Lambdas . . . 42
6.3 Elasticsearch Cluster . . . 44
6.4 Elasticsearch Client in Python . . . 45
6.5 Asynchronous Requests . . . 48

7 Visual Interface in AWS 49
7.1 AWS Dashboard Widgets . . . 49
7.2 AWS SDK Boto3 and Lambda Metrics . . . 51
7.3 Elasticsearch Nodes Metrics . . . 59
7.4 Widget using Query Syntax . . . 61
7.5 Alarm Widget . . . 64
7.6 AWS Custom Widgets . . . 64

8 Debugging and Automation 67
8.1 Reachability Inside and Outside VPC . . . 67
8.2 Links to Debugging Results . . . 72
8.3 Creating CSV Files and Pushing Them into S3 . . . 72
8.4 Daily and Monthly Reports in Python . . . 74
8.5 Email Automation in Lambdas using SES . . . 77
8.6 Costs Report using Excel . . . 79

9 Dashboard Limitations 81
9.1 Kibana . . . 81
9.2 Dashboard Visualizations . . . 82
9.3 Apdex in Kibana . . . 85

10 Conclusions and Future Works 87

Chapter 1

Introduction

Web applications have become core business components, offering solutions to provide new services to customers and maintain a high reputation of the company and user satisfaction. However, they must face several security challenges to protect from cyberattacks. Organizations deploy Web Application Firewalls as a protection layer to overcome this issue. But, even if it prevents unauthorized access and protects sensitive data, it must have an effective monitoring system to maintain high performance and quality of service.

The thesis, developed at aizoOn Technology Consulting, aims to analyze and develop a monitoring system tailored for Web Application Firewalls that utilize the cloud computing capabilities of Amazon Web Services (AWS) and, for storage and analytics features, the ELK Stack. The combination of AWS and Elasticsearch provides a scalable and secure infrastructure. So, it is also possible to enable near real-time analysis and visualization of WAF logs and statistics of several metrics to face daily issues.

Amazon Web Services is a leading cloud services provider that changed how organizations deploy and manage their infrastructure through constant innovation. This thesis explores several crucial AWS services to develop monitoring systems, such as AWS Lambda and CloudWatch. The ELK Stack, a collection of different applications, played a fundamental role in real-time log analysis, searching, retrieving, and data visualization.

The project leveraged a serverless approach in Python to maintain high efficiency while cutting down data gathering costs. The monitoring system consists of two dashboards. The first one provides a general view of the WAF and Elasticsearch Cluster status through default and custom metrics. It also allows debugging customers' configurations directly from buttons added in the AWS dashboard.

To timely respond to issues, AWS Alarms delivers notifications when metrics value hits specific thresholds. However, to overcome flexibility limitations over metrics, another dashboard, built using Kibana, provides more granular and customizable filters, operating directly on documents. Eventually, AWS Lambdas generates daily and monthly reports that aggregate data of a specific time interval for analytics purposes. Then, these reports will be delivered via mail or pushed as CSV/Excel files on an AWS S3 Bucket.

1.1 Thesis Organization

The document structure could be summarized in the following paragraphs:

- Chapter 2 contains an introduction about monitoring, the impact that it could have on a service, and a brief description of web application firewalls and how they work. This chapter would be useful to understand the reason behind the thesis project.
- Chapter 3 provides an overview of a portion of AWS services utilized for the thesis development. However, some of them will be described later.
- Chapter 4 points out the main functionalities that containerization offers. Then, it provides an initial setup for a testing environment.
- Chapter 5 focuses on AWS Cloudwatch. Here are the first simple metrics that would be present in the monitoring system. Then, it shows how to create alarms and notifications using AWS.
- Chapter 6 introduces the AWS Lambda service and the Elasticsearch Cluster. These elements represent the core components for the development of custom metrics.
- Chapter 7 contains the paragraphs of the building process of the AWS Dashboard. There is a description of the steps to include the generated metrics and to obtain an overview of the alarm state.
- Chapter 8 contains the code that permits making the AWS Dashboard interactive, adding some debugging functionalities and reports generation.
- Chapter 9 describes the limitations of the previously developed dashboard, and it contains the steps to create a more flexible dashboard using Kibana.
- Chapter 10 concludes the document and addresses some future works for improvement.

Chapter 2

Thesis Background

This chapter defines the difference between observability and monitoring. Then, the impact that could have on the lifetime of web services and how it affects companies. This introduction could be meaningful to understand the motivation behind the thesis research and the problems related to the subject. In the last paragraphs of the chapter, there is a description of the target of the monitoring system, a cloud-based web application firewall. The latter is a security solution that protects web applications from multiple cyber attacks, and, in this specific scenario, its infrastructure relies on services offered by AWS and ELK Stack.

2.1 Observability and Monitoring

Before diving into the subject of the thesis, it could be meaningful to understand the difference between observability and monitoring:

- **Observability** represents the ability to understand the state of complex systems without directly testing them. It is possible to design data observability tools utilizing Machine Learning solutions to detect anomalies and address issues from output data and interactions between different components. Here, data patterns could lead to a result even if not predicted before. Observability allows a proactive approach to determine why and how something happened.
- **Monitoring** is a process based on collecting and analyzing aggregated data of predefined metrics and logs to define the health status of an IT system and assess its integrity. It is possible to define alerts that allow the team to get notifications promptly to resolve issues and analyze data over long periods. It permits a reactive approach to determine when and what something happened.

This thesis aims to deliver a monitoring system since it is a crucial component that can make a difference in understanding if a product behaves as expected. To summarize, it consists of repeated tracking of different meaningful metrics that allow analyzing the status of the service in terms of responsiveness, network traffic, resource utilization (CPU, RAM, disk, etc.), and other relevant statistics and tools that could be meaningful depending on the type of service provided.

2.2 Monitoring System: Impact

The impact of monitoring for IT services can be summed up in the following key points:

- **Service availability**: View of the past and current health status of the architecture.

- **Detecting human errors**: When a service has a lot of settings, it is possible to have misconfigurations generated by a customer or the developers. An automated debugging system could improve the research of flaws in terms of time and efficiency.

- **Service improvement**: It is possible to collect and evaluate the statistics received to point out non-optimal performances and where it's possible to act to provide a better service.

- **Reaction to issues**: In case of problems, it's crucial to react promptly to resolve them. Receiving a message or an email when a metric surpasses a certain threshold can help spot strange or unexpected behaviors. But even better, it's possible to automate some mitigation measures for some attacks (e.g. DDoS)

- **Company reputation**: Being able to detect and resolve problems faster will limit damage to the company image and improve the trust of the customers in the service.

2.3 Downtime

Downtime is, by definition, the timespan of a service or machine that can not provide its functionality. This effect is not limited to the single compromised system but can spread unexpected consequences to other services that depend on it.

These behaviors will escalate in sales losses for the company, in particular, caused by the inability to generate new ones because the system was not available or, even worse, due to a possible bad reputation that will compromise the future of the business. In every aspect of life, gaining trust and credibility is more challenging than losing it. Nonetheless, IT businesses are not an exception.

There is an interesting study that Uptime Institute posted in 2022. The research area was on outages that occurred in the same year. In particular, they focused their studies on the impact and cost of downtime of data centers.

According to Uptime's Data Center Resiliency Survey[1], 80% of data center managers and operators have experienced a minimum of an outage in the past three years. However, not all the outages have generated a financial loss and downtime, but less than half of them had a significant impact.

From the study, it is also possible to gain information about some real numbers of losses. They found that in 2022, less than 40% of the total downtimes had a cost under 100,000$, whereas the ones that surpassed 1 million $ were 15%. The same article also showed that almost one out of three downtimes in 2021 lasted more than 24 hours. In 2017, the percentage was just 8%.

The impact those losses have on a company depends on the revenue a specific business could generate. And also on the margins of tolerance that they have towards failures. Even following the best practices, it is impossible to bring the chance of downtime to zero. However, countermeasures could significantly reduce its impact.

Ponemon Institute[2] conducted a study analyzing the cost of data center outages in January 2016. The research collected data from 63 data centers located in the United States and compared the new data with the older research of 2010 and 2013. According to the results published: "The average cost of a data center outage rose from $690,204 in 2013 to $740,357 in this report, a 7 percent increase. The cost of downtime has increased 38 percent since the first study in 2010."

Considering the maximum downtime cost, its value was $2,409,991 for 2016. This number increased by 32 percent since 2013 and 81 percent since 2010. Nonetheless, the average cost of unplanned outages per minute was 8,851$ in 2016. Its value was higher than the one found in the previous years, respectively 5,617$ in 2010 and 7,908$ in 2013.

From a more recent worldwide survey of 2020, Statista[3] reported that 88% of 1200 respondents had an average hourly downtime cost for their server greater than 300,000$. And 17% faced costs higher than 5 million dollars per hour.

Furthermore, according to Unitrends[4], there are six major causes of downtime:

- **Human Error**: That can be unavoidable, but it is possible to reduce it with proper checklists, training, automating some processes, and limiting the manual settings as possible.
- **Hardware/Software Failure**: It could be caused by obsolete hardware or software and by patching without exhaustive tests.
- **Device misconfigurations**:These could be caused by configuration errors that allow attackers to exploit existing vulnerabilities. Here, automating and testing could limit the probability that issues happen.
- **Bugs**: This can lead to security issues and could be resolved by applying patches on time with proper testing.
- **Cybersecurity Attacks**: The number of threads is increasing every year and can potentially cause relevant damage to a company, but the investment a business is willing to finance in security often is not sufficient to deal with this issue. Also, not having compromised and periodical backups can play a huge difference in case a disaster recovery is needed.
- **Natural Disasters**: Unpredictable events that are not uncommon in specific places but can cause huge losses. To limit damages, it is possible to rely on a physical copy of data and take advantage of the possibility of spreading resources to different locations.

In conclusion, it is possible to assert that the cost of downtime is a crucial aspect that cannot be diminished. In particular, it could cause several losses financially and reputationally that could lead to the end of a company. Even if it is impossible to avoid, proper prevention and an effective monitoring system may reduce outage frequency and duration.

2.4 Web Application Firewalls

Due to the increasing sophistication and changes that affect cyber threats, security represents a significant challenge to web applications. A WAF (Web Application Firewall) is a security solution that protects web apps. It utilizes monitoring, blocking, and filtering of the HTTP traffic to prevent malicious requests from reaching the services behind it.

This solution uses security rules to protect web applications against multiple threats, for example, zero-day attacks, OWASP top 10 vulnerabilities, malware, distributed denial-of-service (DDoS), and bad bots. These policies require updates as soon as possible to prevent new attacks and can be customizable to satisfy specific scenarios.

A WAF operates at Layer 7 of the OSI model (Application Layer). Therefore, it could provide more exhaustive protection than Layer 3 and 4 Firewalls. The latter could apply packet filtering (based on source/destination IP, port, and protocol) and stateful packet inspection (additionally able to keep track of active connections and allow or deny traffic based on sessions). A WAF can also apply a more granular level of inspection than an IPS/IDS.

WAFs can utilize reputation databases and geo-blocking rules to block requests with specific source IPs and locations. Additionally, it provides content, parameters, and header inspections for the HTTP traffic. For example, it could prevent malicious character sequences, signatures, or patterns indicative of an SQL injection attempt like the following:

```
SELECT * FROM Users WHERE username=″ or ‘1′=′1′ AND

password=″ or ‘1′=′1
```

In this case, the inserted query parameters for username and password are: `′ or ′1′ = ′1`

To provide its functionalities, a WAF must act as a reverse proxy. The previous statement implies that the Web Application Firewall position is between the users and the server that needs protection. So, all the HTTP traffic has to pass through the WAF, which can analyze all the inbound requests and the outgoing responses of the web apps behind it. On the other hand, the server must block all the requests generated from different sources, or the additional protection layer could be bypassed.

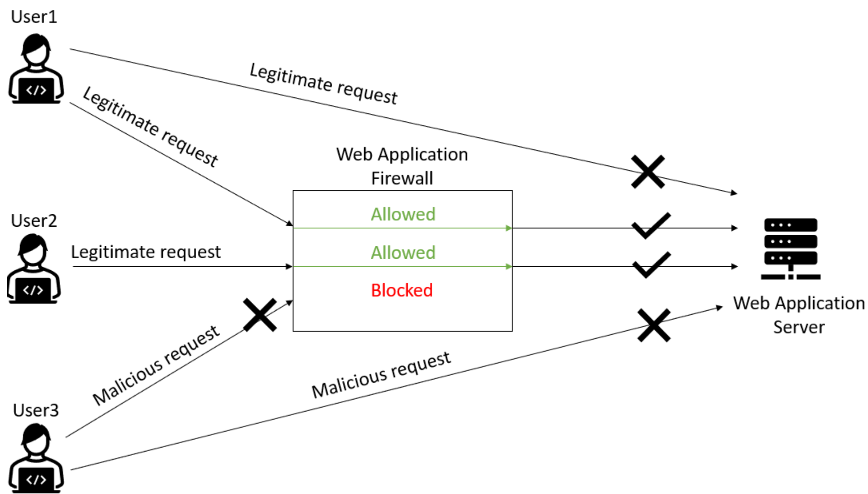


Figure 2.1: Web Application Firewall

In the case of this thesis, the Web Application Firewall that needs the monitoring system is a WAF-as-a-Service. In particular, it relied on services offered by AWS to provide protection to multiple customers. In this scenario, downtime will propagate to all the web applications behind the WAF and make them unreachable.

Chapter 3

AWS Fundamentals

As anticipated before, Amazon Web Services (AWS) is the Infrastructure as a Service (IaaS) that the Web Application Firewall is relying on. This chapter contains an introduction to AWS and some of its services for understanding the structure of the project. In particular, it focuses on networking, managing users' access, VMs, and load balancing.

3.1 Introduction to AWS

Amazon Web Services (AWS), launched in 2006 by Amazon.com, is the most widely-used Cloud Computing platform. The provider offers over 240 cloud services in Q4 2023, and their number is still growing with its different functionalities. It is possible to get some meaningful information from the webpage of the Global Infrastructure of AWS[5]:

AWS has a worldwide infrastructure distributed through different geographic areas called Regions. Every region has a minimum of 3 Availability Zones (AZs). Each contains one or more data centers that are physically independent, isolated, and secure. Between AZ of the same region, there are redundant and ultra-low-latency networks with encrypted traffic.

After the account creation, AWS customers can use more regions isolated from the others by default. It is possible to exploit this configuration for testing new features and services with zero risks of compromising infrastructures hosted elsewhere. On the other hand, it is also feasible to implement an application in multiple Availability Zones. The latter makes it more resilient and provides higher fault tolerance, availability, and scalability.

In November 2023, there were 102 AZs distributed within the 32 regions. The picture below contains their location:

Figure 3.1: AWS Regions

From the image above, it is possible to notice that new regions are coming soon (Canada, Malaysia, New Zealand, and Thailand) and that there is a will to keep improving the quality of the service offered.

Some differences have to be taken into consideration when it comes to choosing which region is better to rely on:

- **Distance**: Choosing a region as near as possible reduces the network latency.
- **Services release**: Some new services will be available first in a limited number of regions.
- **Pricing**: There is a difference in price between regions that cost more than others, so using built-in AWS calculators is essential to provide an exhaustive cost estimation.
- **Regulatory and Compliance**: Due to GDPR or law restrictions, the choice of a specific AWS Region depends on the country.

Since the pool of services is quite large, there are a lot of possible implementations. Those could resolve the same problem in many different ways.

3.2 VPCs and Subnets

A **Virtual Private Cloud (VPC)** is a network in the cloud. As previously mentioned, AZs are data centers in large clusters that belong to a Region. It is possible to launch, for example, EC2 or RDS instances in Availability Zones. AWS allows to split VMs and services across different AZs, so if one goes down, another would be up.

From the AWS Documentation [6], a VPC is an **isolated network** in a Region, spanning all AZs of that Region. Isolation means that multiple networks in an AWS account are independent (e.i. from a compliance perspective). Instead, subnets are subnetworks inside a VPC and can be assigned to a single AZ. While VPC can span multiple AZs.

In VPCs, there are private IP ranges, but it is possible to assign public IPs also. Each subnet takes a subset of a VPC's IP range. An instance can have a private IP that provides access from inside the VPC and a public IP that allows external reachability. Another essential component is the Internet Gateway, which is crucial for all traffic leaving the VPC and manages connections between a private cloud network and the outside world.

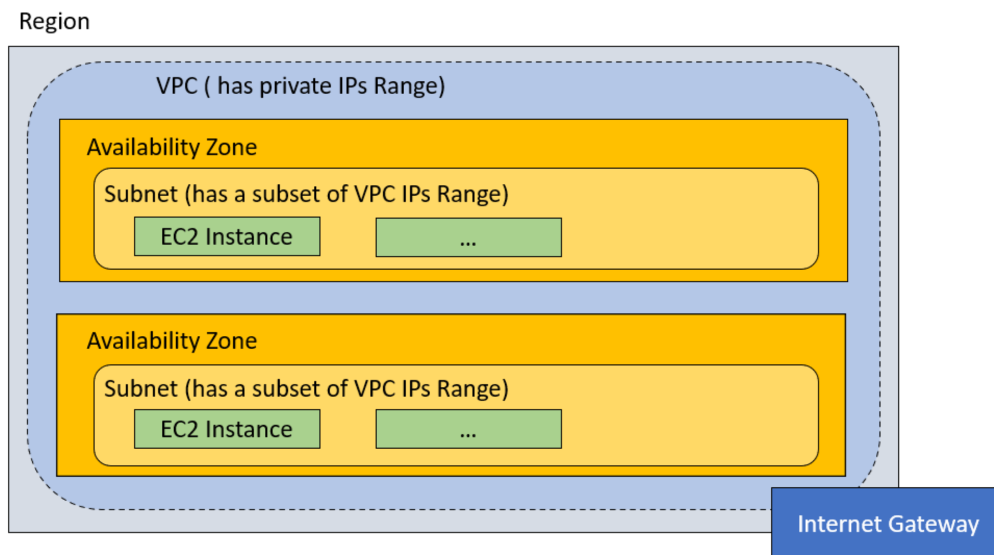


Figure 3.2: VPC and Subnets in a Region

Security Groups[7] applied on a per-instance level can be used to control the inbound and outbound access to AWS resources using Security Groups, applied on a per-instance level. A Security Group acts like a network firewall. AWS customers can assign SGs to more instances or have a different one for each. Instead, a **Network Access Control List** (NACL) works at the subnet level and defines which traffic could access the subnet. A combination of SGs and NACL is supported.

Another meaningful feature is the ability to route requests and, as a result, change the traffic direction. So, even if a component has problems, administrators could manage changes directly by modifying the Route Tables. Routing is set up specifically on subnets, enabling one to choose if the data of specific subnets could go out and where.

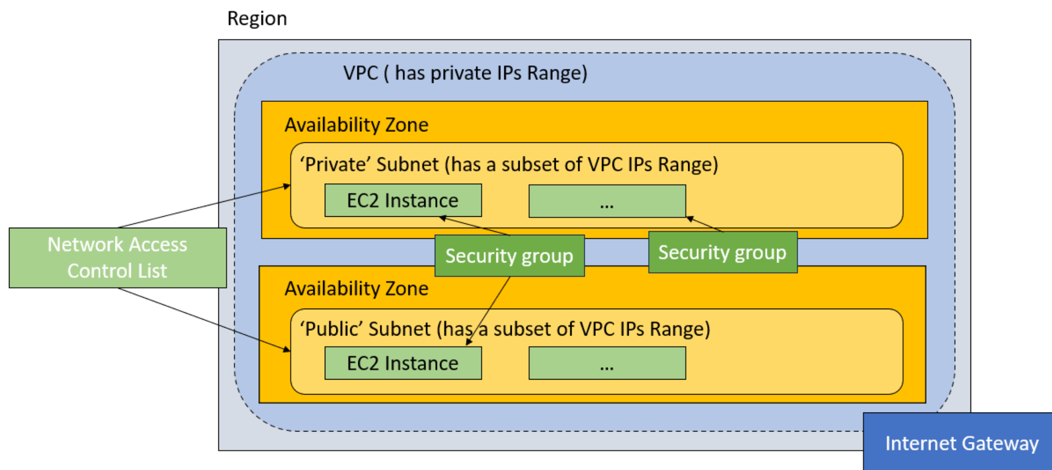


Figure 3.3: Security Groups

An EC2 instance has two ways of getting public IPs. The first is enabling the auto-assign public IP. So, when the VM restarts, it receives a new public address. Nonetheless, it is possible to associate an Elastic IP to the EC2 instance that remains the same after reboots. If an EC2 instance doesn't have a public IP, it could still be reachable from the AWS services inside the VPC through the private IP automatically assigned to the VM.

Blocking all outgoing traffic to the internet through the Route Table and not assigning any public IP in a subnet is the configuration for a **private subnet**. The latter can contain, for example, a database. While in a **public subnet**, external traffic is allowed through an **Internet Gateway**, and internal services have public IP assigned. Here, a server that provides a public service could be hosted.

Since the public and private subnets are in the same VPC, they can still mutually communicate with each other but with different access configurations. A Private Subnet could utilize a **NAT Gateway** (located in a Public Subnet) and redirect all the outgoing traffic to it through the Private Subnet's route table. So, the instances without a public IP can still access the internet while being unreachable from outside the VPC[8].

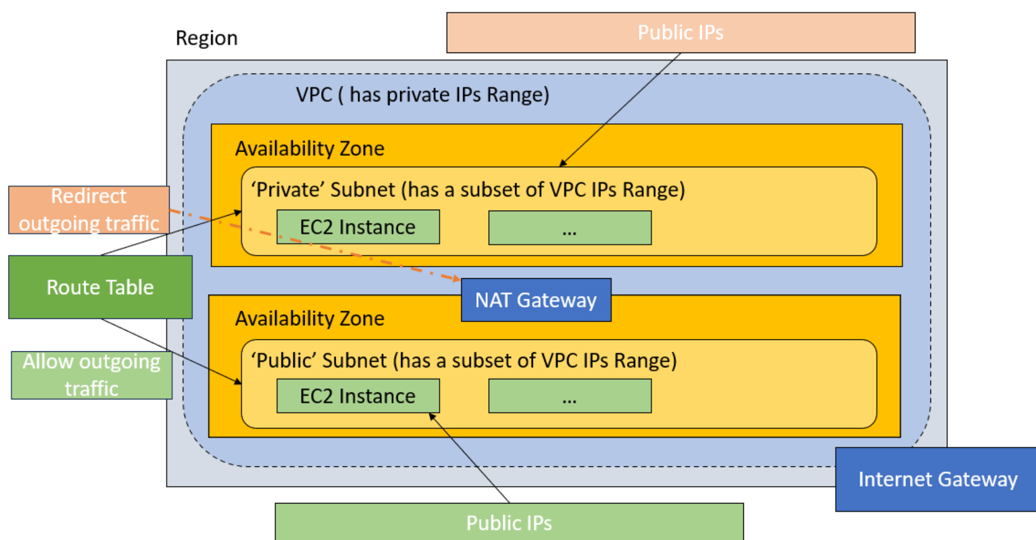


Figure 3.4: Public and Private Subnets

3.3 AWS Identity and Access Management

Identity and Access Management (IAM) is an AWS Service that allows customers to keep their accounts secure. This service is not supposed to provide security to the applications, but it does give the proper permissions to different people and services in the account[9].

There are four main functionality that can be managed in the IAM service:

- **Users**
- **Groups**
- **Roles**
- **Policies**

Every AWS Account can be used by different people, called Users, even if there is one account owner or root user. It's a good practice to create a user for each physical person and limit the use of the Root User.

A new user has no permissions by default at its creation. But it's possible to configure which AWS services he can access by attaching Policies and being granular with their selection. IAM service allows customers to put multiple Users into Groups and assign rights to those Groups (e.g. Admin Group, Read-Only Group . . .).

Policies are a set of rules that contain different authorizations called actions. Some policies already exist. However, it is possible to add new ones. Every policy is customizable in JSON and holds a statement to define the rules. Here, it is feasible to specify the Effect (Allow or Deny), the Action (What to do, e.g. "ec2:*" gives full permissions to EC2 Instances), and the Resource(On what. e.g. a specific EC2 instance or all).

For example, if a new user needs to access just an EC2 instance, it is feasible to give him full permission over that particular machine specifying its ARN (Amazon Resource Names). The latter is an unambiguous identifier for all the AWS resources.

In this case, a possible policy could be:

```
{
    "Version": "2012-10-17",
    "Statement": [
     {
       "Sid": "VisualEditor1",
       "Effect": "Allow",
       "Action": "ec2:*",
       "Resource": "arn:aws:ec2:<REGION>:<ACCOUNT_ID>:instance/<
   INSTANCE_ID>"
     }
    ]
}
```

Listing 3.1: Full access policy for EC2

Here, it is feasible to select a region (e.g. eu-west-3) and the 12-digit account number (ID).

By default, every AWS Service could not access other AWS Services. If an EC2 Instance wants to access the cloud storage service of AWS (S3), it needs the rights to do it. As a solution, a Role with specific Policies has to be attached to the AWS Resource.

It is crucial to be as strict and granular as possible when applying those policies and choose carefully to provide fewer rights that still permit one to achieve a desired task. AWS suggests activating Multi-Factor Authentication (MFA) for each user to increase security. Nonetheless, a Password Policy could be added to set the minimum requirements for the user's password and the expiration period and prevent password reuse.

3.4 EC2 Instances

EC2 stands for **Elastic Compute Cloud**[10]. It is an AWS Service that allows customers to “rent” a small fraction of machines, also known as virtual servers or EC2 Instances. The latter are isolated from the others that share the same physical machine for security reasons. It is an on-demand and scalable service.

EC2 instances could be launched by the console selecting an AMI (Amazon Machine Image). The latter permits the choice of the Operating System with some pre-installed features and which resources need to be allocated to the instance (vCPU, RAM, Disk. . .).

Every Instance type has a tier. Its price grows based on the AMI and the resources utilized. AWS uses a pay-as-you-go approach that allows customers to pay just for the utilized services.

It is fundamental to set the VPC, the Subnet, and the Security Group and possibly assign a public IP to the instance if needed. AWS allows the generation of Key Pairs to connect to the virtual server through SSH.

Instances (1/1) Info | Connect | Instance state | Actions | Launch instances

Find Instance by attribute or tag (case-sensitive)

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
Test	i-026ddff204b4d931d	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	ec2-35-170-52-

Figure 3.5: EC2 Instance running

After the instance gets created, it will reach a "running" state, and it is possible to change the instance state using the drop-down menu:

- **Start**: It runs a stopped instance.
- **Stop**: It switches off a running instance.
- **Reboot**: It restarts a running instance.
- **Hibernate**: It freezes all the running processes, saves the RAM content in a persistent volume (EBS), and then performs a shutdown.
- **Terminate**: It deletes the instance permanently.

Another gripping feature system is the Status Check. The Status Check metrics are the System and the Instance status checks. They monitor software, network, and hardware problems. The first check is applied on the physical machine, while the second is on the virtual server.

3.5 ECS and Fargate

ECS is the **Elastic Container Service** of AWS[11], a managed service that runs Containers (typically Docker). It is possible to choose a Serverless option with FARGATE or a Managed option using EC2. With the EC2 option, the AWS customer is responsible for patches, software upgrades, and security issues.

However, using FARGATE, there is less maintenance involved. From an availability perspective, ECS supports autoscaling to handle huge workloads. It's a good solution for ad-hoc services and for ones that have to scale as needed. ECS is also a cost-effective solution.

There are three main components in ECS:

- **Task**
- **Service**
- **Cluster**

The first one is the concept of Task. The latter is an abstraction on top of containers. But first, a Task Definition Family must be created. It is a template where it is possible to set the Task size in terms of vCPU and Memory. Then, it is required to assign an IAM role and select the Image URI for each container needed in the Task. Below is the syntax and an example of an Image URI:

```
<ACCOUNT-ID>.dkr.ecr.<REGION>.amazonaws.com/ <ECR-REPOSITORY>:<TAG>
```

```
123412341234.dkr.ecr.eu-west-3.amazonaws.com/my-image:latest
```

The Image URI must be present in ECR Service, a Cloud Storage Service for Docker Images. Then, it is possible to set the port mapping and other optional configurations if necessary. On the other hand, an ECS Service contains a group of tasks that has to run in the long term. It allows administrators to specify the desired number that runs simultaneously in an ECS Cluster. An ECS Service uses a scheduler for relaunching each component that failed or crashed and tries to reach the desired number of healthy ECS Tasks.

AWS allows to configure the number of maximum tasks for a service. This feature limits the autoscaling, for example, to reduce costs in case of unexpected behaviors. Eventually, an ECS Cluster is a group of Services or Tasks that contains the Task Definition and the Service configuration. The latter is not mandatory since it is possible to run the tasks directly.

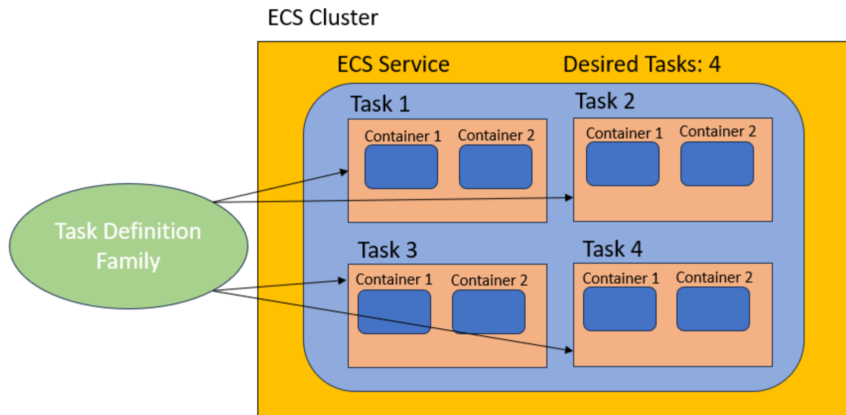


Figure 3.6: ECS Cluster

3.6 AWS LBs and Global Accelerator

The primary functionality of a **Load Balancer** is to equally distribute a share of traffic between more instances, even in different Availability Zones considering AWS. It can check the health status of the target instances and not forward traffic to a compromised one. This configuration makes the LB fault-tolerant, flexible, and scalable [12].

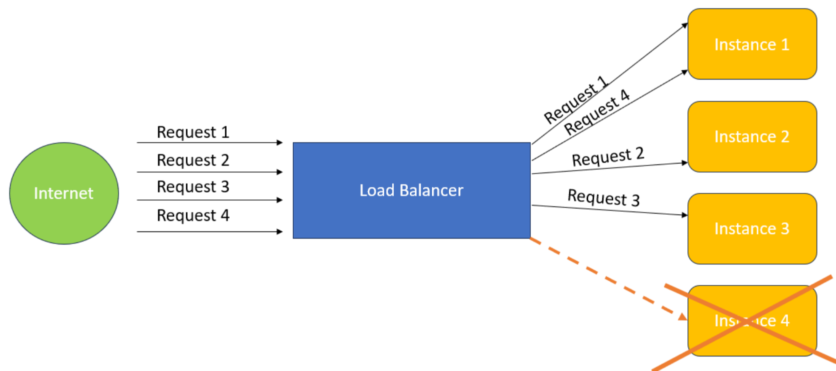


Figure 3.7: Load Balancer

There are three types of Load Balancer in AWS:

- **Application Load Balancer (ALB)**
- **Network Load Balancer (NLB)**
- **Gateway Load Balancer (GLB)**

According to the AWS Documentation[13], the first difference between these Load Balancers is the OSI Layer on which they rely. The Application Load Balancer operates at Layer 7. It can inspect Application-level content and route traffic based on HTTP and HTTPS protocols, while the NLB works at Layer 4 and routes TCP, UDP and TLS traffic.

The Gateway Load Balancer is an optimal solution on the network gateway level, managing traffic across multiple regions or between cloud and on-premises environments. Eventually, it supports IP-based routing for high scalability and availability.

Due to its features, the ALB is more suited to interact with a Web Application Firewall. Since it provides functionalities like SSL termination, session persistence, and content-based routing, it is an ideal solution for managing microservices, containerized environments, and web applications. **Listeners** are components that allow connections to the LB.

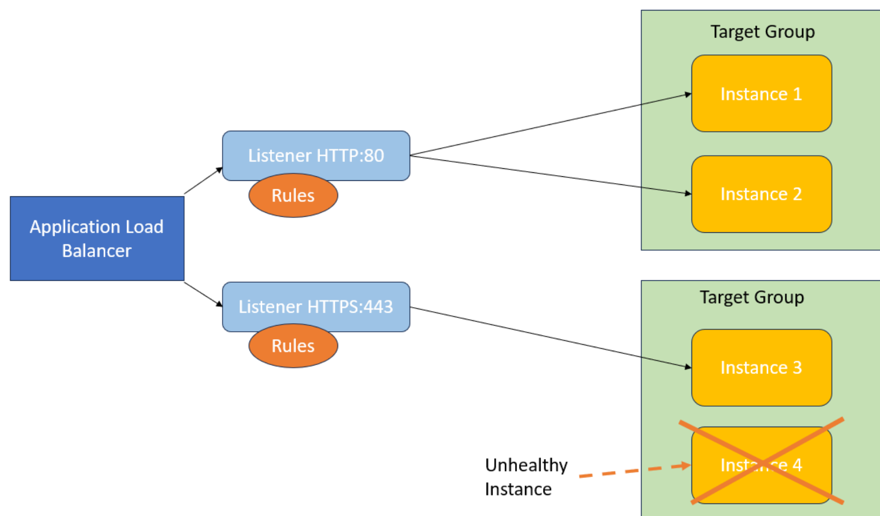


Figure 3.8: Application Load Balancer Structure

In the listener configuration, it is possible to specify the protocol and port.Each ALB Listener uses a set of rules for routing purposes. Every request looks for a match in the rules' condition and takes action. The latter could forward or redirect to a Target Group or return a fixed response. It is mandatory to define a default rule per listener. Target groups define the traffic destinations. They could also determine if a specific target is healthy and stop the connection if the periodical check fails.

Another AWS Service that can improve the performance of cloud infrastructure is the AWS Global Accelerator. The latter is a service that utilizes multiple Edge Locations, globally distributed, as on and off ramps to access the AWS Global Network that can drastically reduce network latency. It is possible to add integration of the Global Accelerator to the Load Balancer to enhance the speed offered to customers worldwide since it provides them with an optimal path for the network traffic.

Chapter 4

Containerization and Simulation

Containerization technologies package, distribute, and run applications in isolated environments called containers. Docker is the most widely utilized platform that manages containers. This chapter contains a brief introduction to it. Then, it provides an initial setup for simulating a testing environment that can replicate problems of ECS Tasks, like warnings, errors, and crashes.

4.1 Docker

Docker is one of the most popular open-source platforms nowadays. It is used for building, deploying, and running applications in lightweight and isolated containers. A container is similar to a Virtual Machine (VM). But it contains and packages the needed application, its libraries, runtime, environment variables, and dependencies. This configuration permits a faster, easier, and more efficient deployment[14]. A Virtual Machine simulates virtual hardware components, while a container shares the existing one of the machine. The OS Kernel is the core component that acts as the primary interface between the physical parts of the machine (Hardware) and the applications that run on the Operating System. Docker virtualizes just the Application Layer and utilizes the host Kernel, while a Virtual Machine virtualizes the entire Operating System. This last feature allows VMs to run on any OS host, providing better compatibility than Docker. The latter reduces the needed size for an image and is faster than a VM, but it is impossible to run Linux Images if the host OS is Windows and macOS without using a virtual machine that runs Linux.

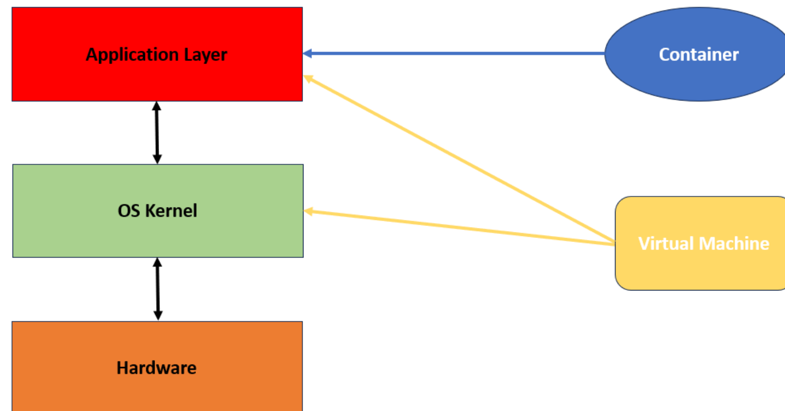


Figure 4.1: Containers vs Virtual Machines

Three elements are fundamentals:

- **Docker Image**: It is a template file/artifact containing the application, the dependencies, and the configurations. Every Image has a version, also known as Tag. The newest version has the tag `latest`.
- **Docker container**: It is a running instance of a Docker Image.
- **Docker registry**: A storage location for Images that can be public (Docker Hub) or private.

A container has all its needed components, so it would be unnecessary for every developer who works on the same project to install all the services the application depends on. It is time-saving, especially in complex projects where lots of services have to interact with each other and provide a replicable environment. It allows us to run different versions of the same applications without conflicts.

To download an Image (e.g. `ubuntu:latest`) from Docker Hub to the local Registry, the following command was used in the Docker command line:

```
docker pull ubuntu:latest
```

Then, it is possible to utilize the `docker images` command in the console to check the local repository:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	ca2b0f26964c	2 weeks ago	77.9MB

Figure 4.2: Local Docker Repository

4.2 Logging in Python

One way to know if something is not working as it should is through logging. Here is Wikipedia's definition[15]:

"In computing, logging is the act of keeping a log of events that occur in a computer system, such as problems, errors, or just information on current operations. These events may occur in the operating system or other software."

In other words, logging is the process of generating messages to understand what happened in a specific scenario and time. It is possible to make statistics or decisions based on them, fix bugs, and improve the system. It's essential to start logging events when an application is more complex than a basic project and then pipe them in a visualization software to get an overview of what is happening.

For this example, Python Programming Language (Python 3.8) was utilized to write a small script that generates random logging errors and warnings. Python has the `logging` module built-in, so it was possible to import it directly.

According to the Python3 documentation[16], logs could belong to different categories or levels. There are six standard logging levels, and these are, in order of value:

- **NOTSET**: In this case, it checks an ancestor logger for level determination.
- **DEBUG**: It gives detailed information in case of diagnosing problems.
- **INFO**: It confirms that something is working as expected.
- **WARNING**: It advertises to us that things are not working as expected.
- **ERROR**: Impossibility to execute a function/task.
- **CRITICAL**: Error that could lead to a crash or disservice in the application.

It is possible to set one of these levels to generate logs of equal or higher impact. The default value is `WARNING`. So, if there are `INFO` or `DEBUG` logs, they would be ignored and not shown in the console. Here there is a simple example of how it works:

```
1 import logging
2
3 logging.basicConfig(level=logging.INFO)
4
5 logging.info("It's all good")
```

Listing 4.1: Example of INFO log

The output is in the picture below:

Figure 4.3: Output of INFO log

Every LogRecord can have different attributes that will determine the format of the customizable logs.

4.3 Simulating Issues using Python

The `time` and `random` Python libraries could be utilized to simulate a real-world scenario where a process can crash with or without logging, for example, if there is an unexpected behavior not managed:

```
import random
import time
import sys
import logging

def crash_func():
    logging.basicConfig(level=logging.INFO, format='%(levelname)s
   :%(name)s:%(message)s')
    time.sleep(random.randint(60, 420))
    err=random.randint(0, 2)
    if err == 2:
        logging.error('An error message')
        sys.exit(2)
    elif err == 1:
        logging.warning('A warning message')
        sys.exit(1)
    else:
        sys.exit(0)

if __name__ == '__main__':
    crash_func()
```

Listing 4.2: Python script that simulate issues

The code above waits for a random time range (1 to 7 minutes) before an action occurs.

It was necessary to create a DockerFile to build a custom Docker Image to package and execute the script using a Docker Container. A DockerFile always starts with a `FROM x` line, where x is another existing Docker Image used as a foundation. Docker allows developers to specify the version of the initial image. The latest Ubuntu image (Ubuntu 22.04) was used for this example.

The command `RUN` executes any command inside the container but not in the host environment. `python3` must be installed in the container to run the script created. It is also possible to use a Python image directly from Docker Hub and avoid the previous step, but it has a bigger size and unnecessary features for this purpose. `WORKDIR` changes the actual directory in the container. The `COPY` command copies a folder's files in another directory inside the container. The last DockerFile command is `CMD`, which executes an entry point command. In this case, it uses `python3 crash.py` to start the Python script in the Docker Container. The code below shows the DockerFile:

```
1 FROM ubuntu:latest
2
3 WORKDIR /home
4
5 COPY ./script .
6
7 RUN apt-get update && apt-get install -y python3
8
9 CMD [ "python3", "crash.py"]
```

Listing 4.3: Dockerfile commands

The script was not a complex application, and the container didn't need a lot of additional packages. However, a good practice is to do all the installations before and then add the script. Since Docker caches the layers created at every step of the Dockerfile build, creating a new Image will be faster due to Docker that can recycle the upper layers in case of changes in the code. The working directory must hold the DockerFile and create another folder that contains the Python script:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        23/03/2023     08:07                script
-a----        23/11/2023     16:37            138 Dockerfile
```

Figure 4.4: Folder Organization

Here, it is possible to use the following command to create the Docker Image and be ready to put it on AWS:

```
docker build -t py-crash:latest .
```

4.4 ECS Cluster Configuration

Considering the Docker introduction, a Registry is the storage location of the Docker images. AWS provides a service called ECR (Elastic Container Registry). The latter manages all the Docker Images securely and reliably in the cloud. Here, a different AWS Region from the one where the Web Application Firewall relied has been chosen to isolate the testing region from the production one. This decision also maintained a clean production environment and avoided unnecessary risks.

A private repository was created in the ECR console to push a custom Docker Image on AWS. It is requested to install and set up the AWS CLI. There are a lot of possible ways to configure the AWS CLI. Since this example is used just for testing, an Access Key for an IAM User could be generated in the IAM console and utilized the following commands in the local environment to use the credentials:

```
$ aws configure
AWS Access Key ID [None]:  **********EXAMPLEAK
AWS Secret Access Key [None]:  ************EXAMPLESAK
Default region name [None]:  example-region-2
Default output format [None]:  json
```

Then, these commands will push a Docker Image in AWS ECR:

```
$aws ecr get-login-password -region eu-west-3 |
docker login -username AWS -password-stdin
111111111111.dkr.ecr.eu-west-3.amazonaws.com

$docker tag py-crash:latest 111111111111.dkr.ecr.eu-
west-3.amazonaws.com/py-crash:latest

$docker push 111111111111.dkr.ecr.eu-
west-3.amazonaws.com/py-crash:latest
```

After the upload, the Docker Image will be inside the private ECR repository. So, it is now available to be utilized in the AWS Environment:

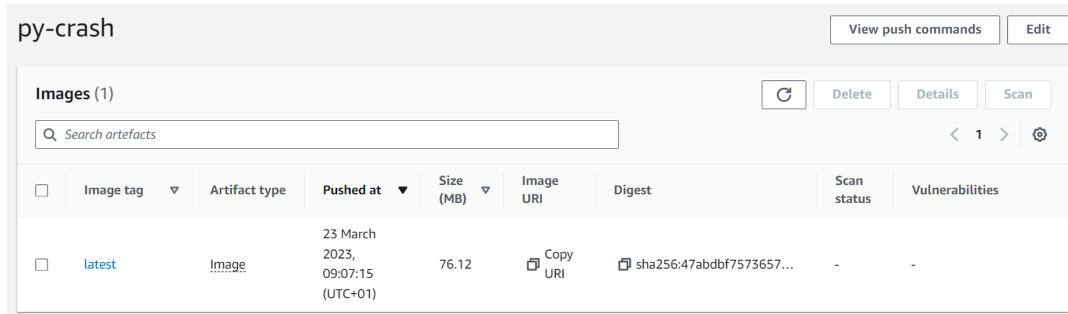


Figure 4.5: Docker Image inside the ECR Repository

In the last step, it was necessary to:

- Create a Task Definition Family in the ECS console and select the custom Image URI as Docker Container from ECR. Then, choose as few resources (vCPU/memory/disk) as possible since it reduces the costs.
- Create a Cluster and a Service with a limited number of desired tasks (1 in this case), select FARGATE, and choose as Family the one previously created. Then, check the "Monitoring" option to use Container Insight. The latter is essential for collecting more metrics of the cluster.

A gripping feature that ECS offers is the quick deployment of new versions of Docker containers. Since ECR allows AWS customers to push more versions of an image in the same repository, new Revisions could be created. Furthermore, it is possible to select the new Docker Image that was pushed with a different tag on ECR using the Task Definition Family console. Then, the ECS service would be updated after choosing a new Family Revision.

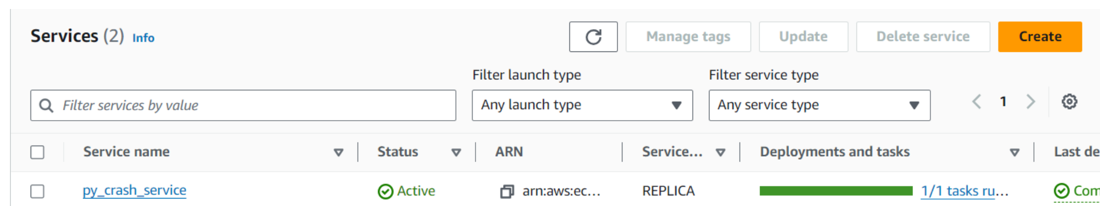


Figure 4.6: Service deployed in ECS

Now that a simple service hosted on AWS has been configured to crash and generate warnings or error logs, it is possible to start to develop the monitoring system.

Chapter 5

Simple AWS Metrics

Firstly, it is required to comprehend which AWS service could provide data visualization functionalities to create the new metrics for the logs and crashes of the compromised Docker container. This chapter will briefly describe the Amazon CloudWatch service and then explain how to detect these simulated issues in ECS Tasks.

5.1 CloudWatch

One of the services that have been crucial for the structure of the thesis is Amazon CloudWatch[17]. It can use multiple metrics for different services and allows customers to monitor all their AWS resources. However, it also supports integrations with architectures outside the AWS cloud environment.

AWS provides features like batching, auto-scaling, and resource scheduling, but data are essential for choosing when to scale or upgrade an architecture. Cloudwatch collects that information, monitors the states over a certain period, and permits it to analyze them and act accordingly.

Once the relevant parameters for infrastructure have been identified, it is possible to build on Cloudwatch automated dashboards that provide a general view of all the systems for each area of the cloud environment.

5.2 Simple Metrics and Logs

A monitoring system relies on values that receive continuous updates over small periods, also known as metrics. The smaller the timespan, the more those metrics are in real-time. AWS automatically creates default metrics for its cloud resources. It also supports generating other custom ones.

Every AWS metric [18] needs a timestamp, a value, a unit, and StatisticValues (e.g. min, max, sum). It is possible to specify a custom retention time for each metric. The latter belongs to one specific Region since it is isolated from the others. Like the Registry for Docker images on ECR Service, metrics also have their storage locations, called Namespaces. During the metric creation process on Cloudwatch, it is possible to select one or more namespaces that contain the metric. The namespace naming convention is "AWS/service". So, it enables a clear separation in macro areas.

Metrics allows AWS customers to specify their Dimensions. The latter identifies the resources linked with a metric and describes other relevant characteristics meaningful for the identification. It is supported by combining them to group different metrics inside a Namespace. A practicable combination of dimensions for a metric of an ECS cluster could be ClusterName and ServiceName in case more than one service is present in the same cluster.

AWS generates two metrics by default for a service in an ECS Cluster:

- Cpu Utilization
- Memory Utilization

ClusterName 6/6	ServiceName	Metric name	Alarms
study_cluster	crash_service	CPUUtilization	No alarms
study_cluster	crash_service	MemoryUtilization	No alarms

Figure 5.1: Default metrics in Cloudwatch

Selecting the `Monitoring` option in the cluster creation will deploy additional metrics (Task/Service number, transferred and received Byte, storage, etc.)

Cloudwatch saves log records with the same source in different Log Streams. The latter will compose a Log Group based on source, monitor, and retention policy. Metrics and logs are sent to Cloudwatch through a CloudWatch Agent installed in the source instance.

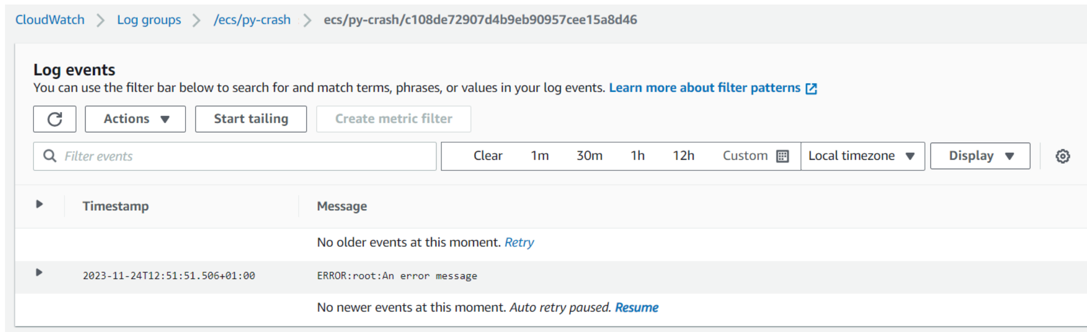


Figure 5.2: Error logs from the Docker Container

The picture above shows a log generated from the compromised Docker Container that is now inside a running ECS Task.

5.3 Alarms and SNS

An effective monitoring system must permit administrators to react fast when something is not working as expected. Having an alarm system that sends notifications is essential if the provided service needs to be always up for several reasons.

CloudWatch supports the creation of alarms based on the value of existing metrics. The latter could be aggregated to obtain specific statistics (AVG, MAX, MIN..) over a precise period of a data point. For example, the maximum value of the CPU percentage over 5 minutes becomes a data point. If the period is lower than 60 seconds, AWS charges customers more.

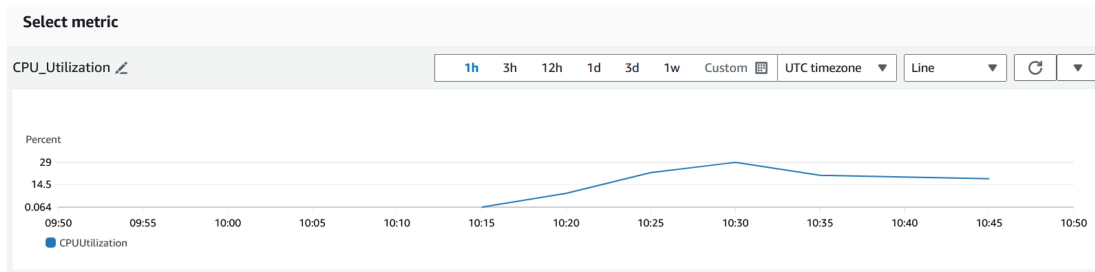


Figure 5.3: Cloudwatch metric graph

For every metric, it was required to set the alarm threshold to define an acceptable range of values and how many data points over the threshold would trigger the alarm. This last feature could reduce the number of false positives. For example, a CPU's spike over the threshold for one data point out of five may not represent the critical situation, but it could be necessary to investigate. Whereas, if there are three positive data points out of five, it would be better to activate the alarm that triggers an action.

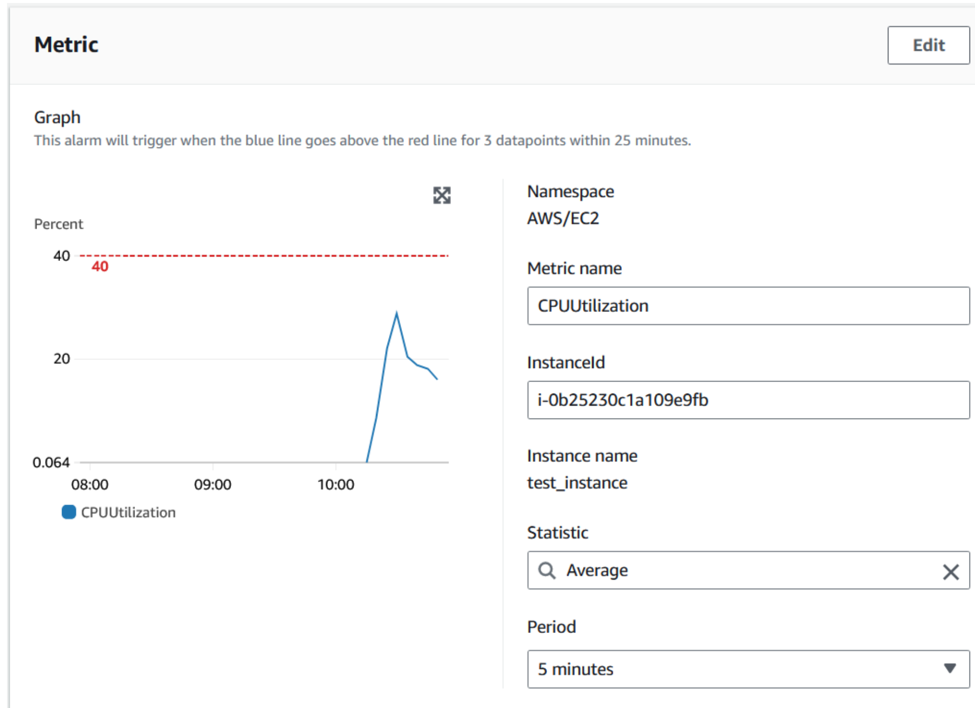


Figure 5.4: Cloudwatch alarm

It is possible to use the SNS (Simple Notification Service) to send a notification when an alarm is triggered. SNS requires a topic that specifies the name of the message and the email endpoints that receive the notice. Furthermore, other actions exist. For example, if an ECS service has a single task, the auto-scaling can be used to deploy more of them to share the workload. Or if the resource is an EC2 Instance, it is possible to change its state (e.g. from running to stopped) and perform other actions utilizing custom serverless functions.

5.4 Metric Filter

Logs could be utilized to deploy new Cloudwatch Metrics and Alarms. The monitoring system needed an alarm that could activate if the ECS Service made more than a certain number of Warning/Error Logs in a period. Since this was not one of the default metrics, it was possible to use a Metric Filter.

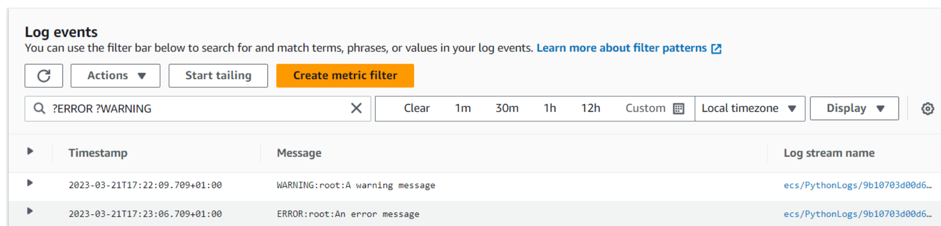


Figure 5.5: Log events

A metric filter[19] checks every log record in one or more Log Streams and creates a new metric based on the ones that match a specific pattern. It requires selecting the Log Group, then choosing `Search All Log Streams`, and providing a filter pattern. Then, it also needs a Namespace for the metric, a value for every match (in this case, the value is 1 since it is a count), and two names, one for the filter and one for the CloudWatch metric. Eventually, it was possible to select the metric filter from the Log Group console on CloudWatch and deploy a new Alarm as done before but utilizing the new custom metric with a different statistic.

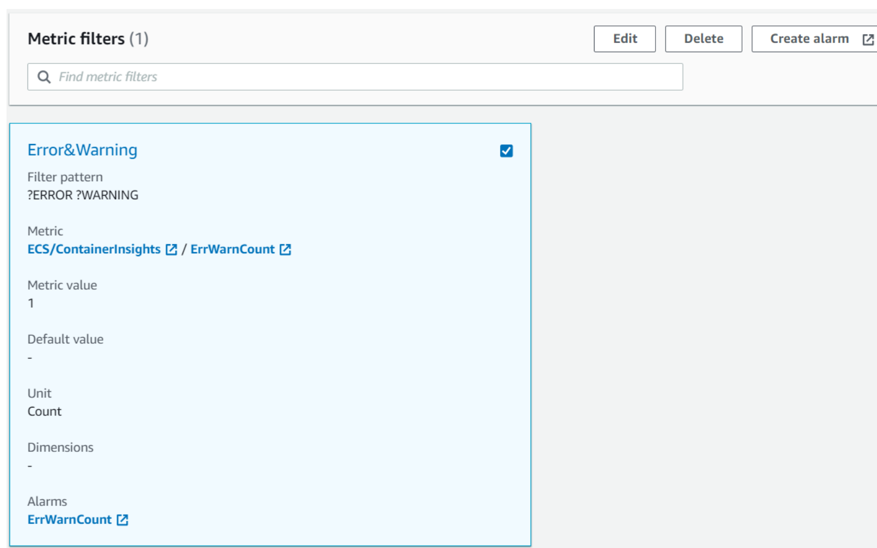
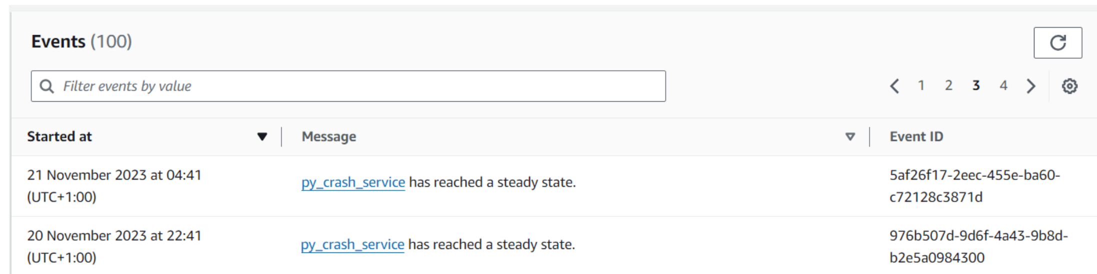


Figure 5.6: Metric filter

5.5 Event-based Metric

It is almost impossible to consider every possible source of error in coding. Especially if the project is large and more people are working on it. ECS FARGATE assures that there are several active tasks equal to the desired number. So, if one goes down, FARGATE automatically launches another one to substitute it.

Every six hours, AWS generates an event if everything is fine in an ECS Cluster (All the tasks are healthy. The autoscaling and new deployments have not happened):



Events (100)

Started at	Message	Event ID
21 November 2023 at 04:41 (UTC+1:00)	py_crash_service has reached a steady state.	5af26f17-2eec-455e-ba60-c72128c3871d
20 November 2023 at 22:41 (UTC+1:00)	py_crash_service has reached a steady state.	976b507d-9d6f-4a43-9b8d-b2e5a0984300

Figure 5.7: Events of ECS Task without issues

But in this case, a compromised Docker image that crashes quite often was deployed for testing. So, AWS will generate additional events because the Task changed its status. The picture below shows the events of the deployed service:

24 November 2023 at 19:19 (UTC+1:00)	service py_crash_service has reached a steady state.	98f17409-c111-4f96-8580-8348a7c1500e
24 November 2023 at 19:16 (UTC+1:00)	service py_crash_service has started 1 tasks: task c7b9505a722c439d9d9c9b07c88f4574.	f88fcd36-fc78-4bb0-bd23-a33c1d2b783a
24 November 2023 at 19:14 (UTC+1:00)	service py_crash_service has reached a steady state.	7bb12879-a97a-4d1e-a70f-4f031412457c
24 November 2023 at 19:11 (UTC+1:00)	service py_crash_service has started 1 tasks: task 2ee28e2d43264fd3a2d32e21eed7a22c.	dc9deb12-3841-41eb-bc03-e17407ed158c
24 November 2023 at 19:06 (UTC+1:00)	service py_crash_service has reached a steady state.	f9d813cf-7a7b-4bd3-b523-3f6b3094163c

Figure 5.8: Events of ECS Task with issues

The task was designed to act like this, but a container may crash in an unpredictable interval in a real-world scenario. Then, FARGATE could launch a new Task to substitute the missing one. So, if the events are not checked daily, it is unlikely to discover that there is a problem immediately. To resolve this issue, AWS Eventbridge could be used.

According to the AWS Documentation[20]:

"EventBridge provides simple and consistent ways to ingest, filter, transform, and deliver events so you can build applications quickly."

It is possible to generate rules that send an Event to a CloudWatch Log Group for every match that occurs using a specific pattern. But first, a new Log Group must be created on CloudWatch to keep the delivered events.

During the rule creation, the Event `source` of the AWS service of interest was chosen (ECS) and inserted in the event pattern form in JSON. Then, `ECS Task State Change` was selected as `detail-type`, since it represents the primary variable enabling crash detection. Afterwards, it was specified the Cluster Arn. Since the objective was to retrieve only the events of the compromised AWS Service in ECS. Below there is the final result of the Event Pattern created:

```
{
    "source": ["aws.ecs"],
    "detail-type": ["ECS Task State Change"],
    "detail": {
        "clusterArn": ["arn:aws:ecs:eu-west-3:111111111111:cluster/study_cluster"]
    }
}
```

Listing 5.1: Event pattern in AWS Eventbridge

In the last step, the destination of the filtered events must be selected. Here, the Log Group previously created was chosen. This configuration allowed the generation of an event in the Log Streams every time the state of the container changed. Since the alarm must detect only when errors and failures occur or the container exits a new Metric Filter has been deployed:

StopCodeFilter

Filter pattern

{ $.detail.stopCode = "*Error" || $.detail.stopCode = "*Failed*" || $.detail.stopCode = "*Exited" }

Metric

ECS/ContainerInsights / StopCode

Metric value

1

Figure 5.9: Metric Filter for crash events

AWS Events contains lots of information, so it is possible to filter granularly and get what is needed. The `$.detail.stopCode` variable has as its value the reason why the task changed its state. A **regex pattern** and **dimensions** of interest were specified. The `$.detail.group` value was used as a **ServiceName** variable, and the `$.detail.containers.name` value was utilized as a **ContainerName** variable in case different containers share the same task.

Lastly, the new alarm could be created utilizing as a threshold the count value of 1. As an activation trigger, it was considered a data point out of 1. So, in case crashes occur, a notification will always be sent.

Chapter 6

Serverless and the ELK Stack

By now, the monitoring system utilizes only Amazon Web Services to set up new alarms and metrics to react in case of issues. However, the WAF's Infrastructure relies also on the ELK Stack. So, the following chapters will describe how to integrate the functionalities of these two platforms to publish new custom metrics and add them to a Cloudwatch Dashboard.

6.1 AWS Lambda

AWS Lambda[21] is an on-demand service that lets customers run functions without worrying about creating an instance and managing its setup and maintenance. It is possible to create a serverless function, insert the code to run, and eventually, the AWS Lambda executes on the cloud. To execute the code, managed VMs could run several containers to start functions in different Programming Languages. The latter supported are Java, Go, PowerShell, Node.js, C#, Python, and Ruby. Thus, it is possible to use other languages through Runtime APIs.

As mentioned above, the setup is quick, but this is not the only reason to use it. In AWS, almost all the services use a pay-as-you-go approach for pricing, with some exceptions. For example, assigning an Elastic IP that is public to an EC2 instance is charged even when the VM is not running. The reason behind this choice is the reservation of that specific IP address, not allowing other customers to utilize it. An elastic IP must be released to limit costs, making the address available again for AWS customers.

Since the monitoring system needs custom metrics sent periodically over a time range, AWS Lambda would be a cost-effective solution. Furthermore, it can rely on other AWS services for data visualization (AWS Cloudwatch), reducing to the minimum the resources needed. Other implementations are possible using:

- **A dedicated machine**: A single customer reserves the physical server. It is the worst solution since it would be underutilized and doesn't allow vertical scaling.

- **A VM**: Multiple customers share the physical server. In this case, the underused resources are less.

- **A container**: A VM runs multiple containers. It is a better solution than the previous two. However, it still requires being up more than needed.

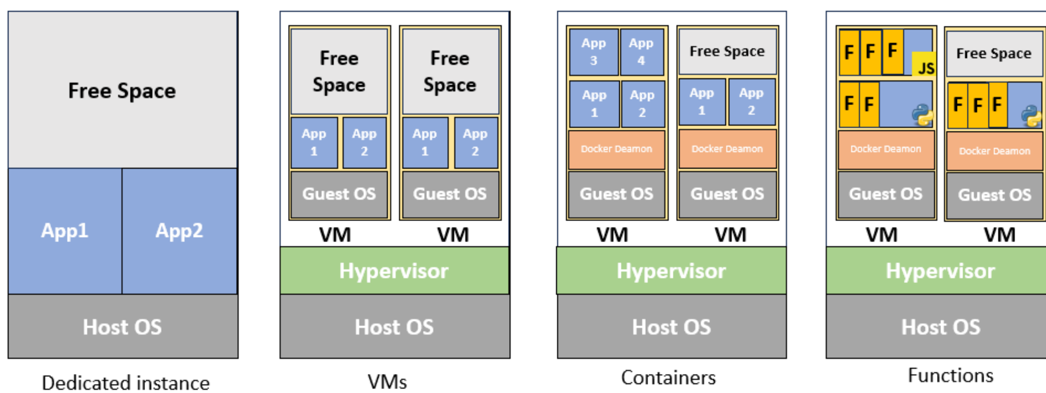


Figure 6.1: Alternative solutions

6.2 Importing Libraries in AWS Lambdas

During the creation of an AWS Lambda function, it is possible to select, as a starting point, a completely new template called `author from scratch`. Then, some additional features need to be configured:

- The architecture: for example x86_64.
- The programming language utilized to write the code (Python 3.8).
- The assignment of an IAM Role to the AWS Lambda.

AWS Lambdas has a lot of other possible configurations. For example, it is possible to assign the function to a VPC in a region and the subnets of interest. It is essential to set a proper timeout for the function. It should not be too low, or the AWS Lambda risks being interrupted while running code. But either too high or, in case of misbehaviors (e.g. endless loops), it will keep going with an increasing cost.

The IAM Role assigned to the Lambda function determines which AWS resources it could access. It is a good security practice to give the IAM Role as few permissions as possible to perform only its task. Additionally, it is possible to add environment variables, which are encrypted at rest, to avoid hard-coded credentials or other confidential data being exposed.

AWS Lambda automatically provides the built-in libraries of the selected programming language of the function. It is also possible to use different modules. However, it is required to create a Lambda Layer that packages together libraries to maintain the code deployment fast.

According to the AWS documentation definition[22]:

“A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.”

Since AWS Lambda runs on Amazon Linux, a good practice is to generate the layer from a cloud-based IDE that lets AWS customers use a Linux server to avoid possible conflicts in different Operating Systems (e.g. Windows and macOS). The service that offers this feature is Cloud9.

To create an environment in Cloud9 service, a small EC2 instance with an Amazon Linux Image could be launched, for example, a t2.micro. A lot of resources are not necessary for this task. Then, a specific role, with permission to publish lambda layers, must be assigned to the new instance:

```
{
    "Version": "2012-10-17",
    "Statement": [
     {
       "Sid": "VisualEditor0",
       "Effect": "Allow",
       "Action": "lambda:PublishLayerVersion",
       "Resource": "*"
     }
    ]
}
```

Listing 6.1: Lambda layer policy

Then, the same version of Python selected for the Lambda must be installed:

```
$ sudo amazon-linux-extras install python3.8
$ curl -O https://bootstrap.pypa.io/get-pip.py
$ python3.8 get-pip.py --user
```

A folder that contains the library (e.g. elasticsearch[async] that will be used later) must be created and installed inside the instance:

```
$ mkdir python
$ python3.8 -m pip install elasticsearch[async] -t
python/
```

The last step is to compress (.zip) the folder and publish it to AWS:

```
$ zip -r layer.zip python
$ aws lambda publish-layer-version --layer-name
elastic-async-layer --zip-file fileb://layer.zip
--compatible-runtimes python3.8 --region eu-west-3
```

Eventually, it is possible to select and add the new custom layer to the AWS Lambda. More layers could be assigned to the same function, and a singular layer could contain multiple libraries.

6.3 Elasticsearch Cluster

Elasticsearch is a search engine and part of the Elastic Stack[23]. It allows users to take data from any source, store it, search, and make an analysis. It's used mainly for:

- **Logging**: store and use log data.
- **Metrics**: gathering descriptive statistics.
- **Security and Business Analytics**: securing communications through channels and improving business capabilities by analyzing data and making predictions based on them.

It is possible to explore, interact with, and visualize data using Kibana in the Elastic stack. Multiple integrations are supported to connect different data sources, ingest them, and generate alarms.

A **node** is an instance of Elasticsearch with a unique name, ID, and Cluster. An Elastic Cluster can contain more nodes distributed over different machines. Elasticsearch stores JSON objects with unique IDs called **documents**. **Indexes** group documents that are related to each other. So, it is possible to divide them and make it easier to find specific information. Disks do not store documents inside indexes but in **shards** across nodes. An index is a virtual concept that keeps track of the location of the shards and manages them. It is possible to create an index with multiple shards, and it would split them into different nodes to improve performance. A **shard** is primary if it contains the original copy of the documents. But to increase reliability and fault tolerance, replicas of shards could be distributed on different nodes. So, if a node goes down or gets corrupted, the documents inside it are not lost.

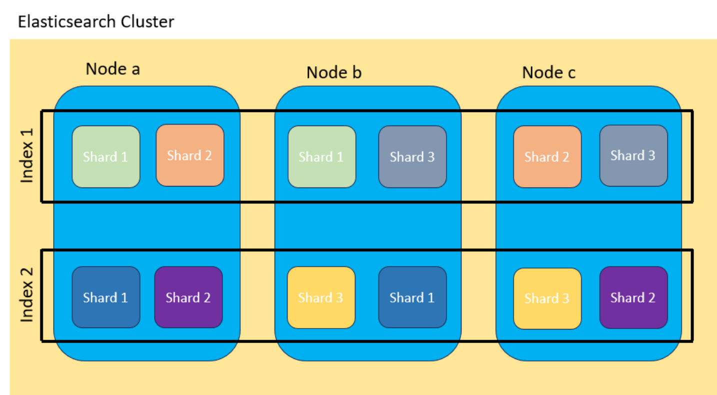


Figure 6.2: Disposition of the Elasticsearch nodes inside a cluster

6.4 Elasticsearch Client in Python

The previous paragraphs briefly described what Elasticsearch is and how it works. Considering a cluster already configured and with documents flowing to the shards, it would be useful to retrieve data from the Elasticsearch documents and build a CloudWatch dashboard that integrates statistics from the Elastic Cluster.

Elasticsearch provides an official low-level Python library that lets developers connect and interact with their Elasticsearch Cluster[24]. But before using it, connections from specific IP addresses must be allowed from the Elastic Cloud configuration. For security reasons, a default traffic filter denies every request from a source IP different from the specified ones (Whitelist).

It is possible to connect to the Elasticsearch Cluster through its endpoint, creating a Client. However, users must provide authentication using an API Key ID and its secret or a username and password.

```
from elasticsearch import Elasticsearch

client = Elasticsearch(['https://my-elasticsearch-endpoint:port'],
                       basic_auth=('username', 'password'),
    request_timeout=30)
```

Listing 6.2: Elasticsearch client

Once the authentication phase is successful, proper permission must also be configured. For example, it supports different functionalities, such as creating new indexes, adding, modifying, retrieving, deleting documents, and more. However, the primary functionality needed is to search and get the documents from an index and perform data aggregations for the monitoring system. The documents' variables are called **fields** in Elasticsearch (e.g. Timestamp).

It would not be efficient and cheap to retrieve all the documents in an index and perform some transformations on the data in a local machine. If a cluster has N nodes, it will be more effective to let its instances do all the transformations on the documents and then retrieve only the final result.

Elastic Cloud charges customers more for the quantity of data that they get from the cluster and for internode transfers surpassing 100GB a month. Distributing all the workload between the nodes relieves a local machine from unnecessary stress and time spent.

Here is an example of how to retrieve data in Python from the nodes:

```
response = client.search(index="my-logs", query={
    "bool":{
        "filter": [{
            "range": {
                "@timestamp": {
                    "gte": "now-5m",
                    "lte": "now",
                }
            }
        }]
    }}, size=0, aggs={
    "hostnames":{
        "terms": {
            "field": "transaction.hostname.keyword",
            }
        }
    })
```

Listing 6.3: Example of Elasticsearch library usage

The index containing the required documents must be specified, but only after a connection is established to the Elasticsearch endpoint. Then, a simple bool query could filter out all the documents with a timestamp value older than 5 minutes. It is possible to use the `range` parameter to provide a boolean result to the filter.

Elasticsearch queries provide `Date Math` parameters that simplify dealing with dates. In particular, the actual time could be retrieved using `now` and then adding or subtracting different time units. For example, if today is the first day of November at 8 am, writing `now-1M` becomes the 1st of October at the same hour. These are the possible time units that are supported:

- **y**: Years
- **M**: Months
- **w**: Weeks
- **d**: Days
- **h** or **H**: Hours
- **m**: Minutes
- **s**: Seconds

There are other types of bool queries. For example, it is possible to combine more **filters** to extract the documents granularly. Considering the timestamp example, more time ranges could be used to retrieve more documents with a single query. Elasticsearch also supports these parameters:

- **must**: it considers a document valid if all the rules in the array are satisfied.

- **must_not**: it filters out every document that matches at least one condition in the list.

- **should**: a bool query type that assigns a score based on the number of rules matched.

After the documents of interest were retrieved through filtering, aggregations were applied to the results to reduce the volume of data extracted from the Elasticsearch nodes.

Two principal types of **aggregation**[25] were utilized to reach the thesis objective. The first one is metric aggregation, which permits obtaining numeric results based on the values that specific fields would have. Some examples are the average, the min, and the max over different documents. The second type of aggregation is the bucket aggregation. The latter is similar to a `group by`. It divides the documents into small subgroups (buckets) that have some similarities in the field values.

In the last part of the code above, there is the `aggs` parameter that performs the aggregation after the query. Then, it requires inserting the name assigned to the result and, eventually, the aggregation type. The `terms` aggregation creates a bucket for each hostname field value and performs a count of all the documents. The `.keyword` parameter added after the field name allows nodes to compare text format values efficiently. Eventually, it is possible to set the `size` parameter to 0 to specify the interest of retrieving only the aggregation result of all the documents.

6.5 Asynchronous Requests

As mentioned before, the cost of AWS Lambda functions depends on how much time the code is running, but not only on that. It depends also on the architecture, the number of requests, and the allocated memory and storage. It is necessary to optimize the code to reduce the amount of money required for a repeated launch of the AWS Lambda. Since the function needs to send web requests to Elasticsearch to retrieve metrics, it is possible to take advantage of concurrency in Python.

Concurrency means that the code makes progress in more than one task. If an application has more tasks to complete, the processing unit can switch between them even before one has completed its execution. So, they are not running in parallel. Applications may involve waiting (e.g. web request waiting for a response). In this case, it would be more efficient to let the processing unit do something else.

The asyncio Python Standard Library allows developers to manage concurrency in the application. In particular, it is possible to utilize `async` before a method or a function, allowing them to execute code concurrently. The `await` statement specifies that, after a concurrent operation, the code has to wait for its result before starting its execution.

Chapter 7

Visual Interface in AWS

Dashboards allow users to observe relevant information and metrics through graphs, tables, and other visual tools. It is possible to utilize these visual interfaces to get an insight into how a system works and if some unexpected behaviors require further investigation.

7.1 AWS Dashboard Widgets

AWS CloudWatch supports the development of custom dashboards that could contain multiple **widgets**. The latter lets customers insert in the dashboard not only graphs but also text in Markdown, log tables, alarms, and even full custom widgets integrated with Lambda functions.

AWS CloudWatch provides a small panel to set a custom period for all the dashboard's widgets. It is required to provide a source to create widgets. There are two possibilities:

- **Metrics**
- **Logs**

With the deployment of the ECS Cluster, some standard metrics have already been generated.

In particular, it is possible to add to the dashboard the following default metrics:

- CPU usage percentage
- RAM usage percentage

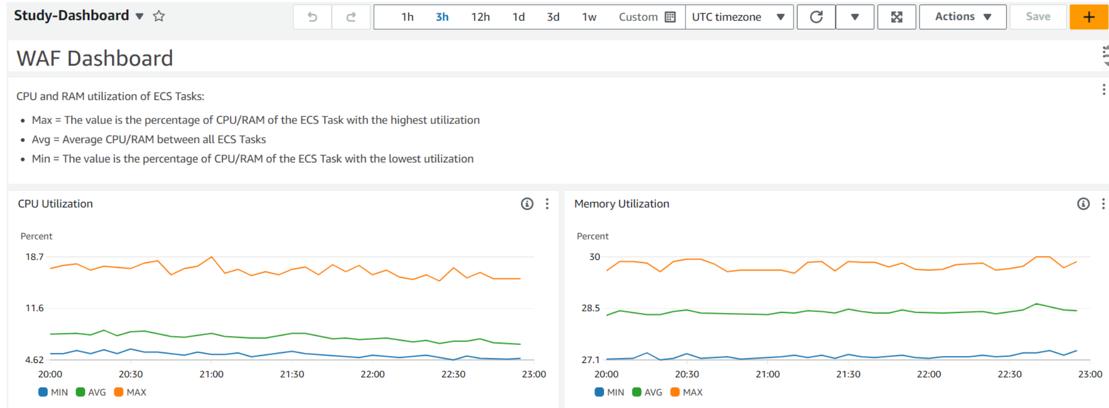


Figure 7.1: CPU and RAM Utilization

The number of active tasks in an ECS service is a metric that could be obtained by setting the monitoring option during the cluster's creation:

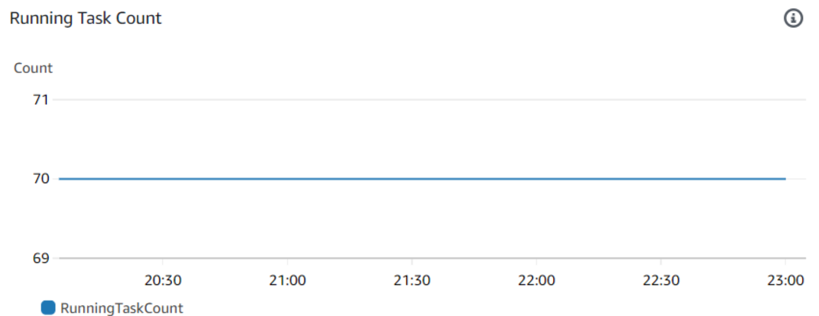


Figure 7.2: Number of active ECS Tasks

The following paragraph will describe how to publish custom metrics into CloudWatch using AWS Lambdas.

7.2 AWS SDK Boto3 and Lambda Metrics

AWS provides tools and libraries that allow developers to interact with multiple AWS services using different programming languages. These tools are AWS SDK, which stands for Software Development Kit. They are built-in libraries into the AWS Lambdas function. `Boto3`[26] is the AWS SDK for Python, but other programming languages are supported (Java, Go, JS, Kotlin, etc.). It is possible to directly import this library in an AWS Lambda function and start generating the metrics that will be used in the dashboard widgets.

Some meaningful metrics that can be retrieved from Elasticsearch to monitor how the WAF, in particular, is behaving are:

- The average latency in milliseconds that the WAF adds to the requests with and without timeouts. This metric will be referred to as WAF Latency.
- The average WAF latency of the top 5% slowest requests with and without timeouts (e.g. WAF Latency $>$ 240 seconds).
- The percentage of requests that are allowed, blocked, or alerted.
- The variation of the WAF latency in different periods.

The deployed AWS Lambda function contains two Python files:

- `lambda_function.py`: it manages all the requests.
- `query.py`: it contains the queries that retrieve the data from the Elasticsearch cluster and push the results on CloudWatch. The previous file will import all of its code.

It is possible to use the `asyncio.run()` function to start the top-level entry-point function for the async requests:

```
import json, asyncio, os, query
from boto3 import client
from botocore.exceptions import ClientError
from elasticsearch import AsyncElasticsearch
from datetime import datetime, timedelta

def lambda_handler(event, context):
    asyncio.run(dashboard())
    return {
        'statusCode': 200, 'body': json.dumps('Success!')
    }
```

Listing 7.1: Lambda handler function

The Elasticsearch client could be created using the environment variables saved in AWS Lambda's configuration. But this time, the `AsyncElasticsearch` library was utilized to support asynchronous requests instead of its synchronous version.

Both the AWS CloudWatch and Logs clients must be created using Boto3. Eventually, it is possible to launch all the asynchronous code contained in `query.py` through the `asyncio.gather()` function:

```python
async def dashboard() -> None:
    #Elasticsearch client
    esclient = AsyncElasticsearch([os.environ['HOST']], basic_auth=(os.environ['USERNAME'], os.environ['PASSWORD']), request_timeout=30)

    #Cloudwatch client
    cw=client('cloudwatch', region_name=os.environ['REGION'])

    #Cloudwatch logs client
    cw_logs=client("logs", region_name=os.environ['REGION'])

    data=datetime.utcnow()
    ed=data.strftime('%Y-%m-%dT%H:%M:00.000') + 'Z'
    sd=(data-timedelta(minutes=5)).strftime('%Y-%m-%dT%H:%M:00.000') + 'Z'

    #Launch async queries to Elasticsearch and create new metrics and logs in AWS Cloudwatch
    await asyncio.gather(
        query.avg_waf_latency(esclient, cw, sd, ed),
        query.avg_waf_latency_no_to(esclient, cw, sd, ed),
        query.elastic_stats(esclient, cw, sd),
        query.avg_waf_latency_95th(esclient, cw, sd, ed),
        query.avg_waf_latency_95th_no_to(esclient, cw, sd, ed),
        query.wafstatus_count(esclient, cw, sd, ed),
        query.waf_latency_diff(esclient, cw, sd, ed),
        query.avg_waf_latency_host(esclient, cw_logs, sd, ed))
    await esclient.close()
```

Listing 7.2: Dashboard function

In the `query.py file`, an asynchronous function was added for every metric that appears in CloudWatch to retrieve the data from the documents in Elasticsearch. The following code will retrieve the Average WAF Latency metric:

```
async def avg_waf_latency(esclient, cw, sd, ed) -> None:
    try:
        response = await esclient.search(
            index=os.environ['LOG_INDEX'],
            query={
                "bool": {
                    "filter": [{
                        "range": {
                            "@timestamp": {
                                "gte": sd,
                                "lt": ed,
                            }
                        }
                    }]
                }},
            size=0,
            aggs={
                "avg-waf-latency":{
                    "avg": {
                        "field": "transaction.response.waf-latency
"
                    }
                }
            }
        )

        avg_time =round(
            response["aggregations"]['avg-waf-latency']['value'],
            1)

        startd=datetime.fromisoformat(sd[:-5])
     except Exception as e:
        return {
                    'statusCode': 500,
                    'body': json.dumps(f"Waf-Latency Exception: {e
}")
                }
```

Listing 7.3: Average WAF Latency metric

Then, the result of the aggregation was pushed to CloudWatch using Boto3. The latter requires to specify the Namespace that collects the new metric and then all the MetricData that contains:

- **Name**
- **Dimensions**
- **Timestamp**
- **Value**
- **Unit**

```
try:
    response = cw.put_metric_data(
    Namespace='ECS/ContainerInsights',
    MetricData=[
        {
            'MetricName': 'WafLatency',
            'Dimensions': [
                {
                    'Name': 'ClusterName',
                    'Value': 'WafCluster'
                },
                {
                    'Name': 'ServiceName',
                    'Value': 'WafService'
                },
                {
                    'Name': 'TaskDefinitionFamily',
                    'Value': 'WafTaskDef'
                },
            ],
            'Timestamp': startd,
            'StatisticValues': {
                'SampleCount': 1.0,
                'Sum': avg_time,
                'Minimum': avg_time,
                'Maximum': avg_time
            },
            'Unit':'Milliseconds'
        },
    ])
# Display error
except ClientError as e:
    print(e.response['Error']['Message'])
```

Listing 7.4: Pushing a custom metric into Cloudwatch

Printing the output into the Lambda's console can be used for finding errors since it will automatically send the result to the Log Group of the Lambda function. Also, it is possible to print a limited number of characters in the console. The same steps could be repeated to create the custom metrics for the AWS dashboard. The new metrics published on AWS Cloudwatch will take minutes to be available for the first time.

Since Lambda functions are not instances always running, it is necessary to configure triggers that start the code execution. There is an extensive list of possible trigger implementations. The latter can include APIs, other Lambdas, batch data processing, and even services outside of the AWS environment.

An **EventBridge cron-based schedule** would make the code of the AWS Lambda function a periodic execution. There is no ideal rate, but there could be compromises considering multiple factors that are correlated:

- How fresh the data must be
- The number of requests per day/month
- The workload on the Elasticsearch Cluster
- The Cost

A valid rate could be to trigger the Lambda function every 5 minutes, taking as query intervals the same duration so it can reduce the workload on the Elastic nodes and maintain relatively fresh data. Below there is the cron expression added to EventBridge:

```
(0/5 * * * ?  *)
```

On average, the AWS Lambda deployed takes 3 seconds to complete its execution and repeats 288 times a day (24 hours / 5 minutes) and 8640 times a month. It is possible to use the AWS Pricing Calculator and select the minimum resources for memory (128 MB) and ephemeral storage (512 MB):

Pricing calculations
8,640 requests x 3,000 ms x 0.001 ms to sec conversion factor = 25,920.00 total compute (seconds)
0.125 GB x 25,920.00 seconds = 3,240.00 total compute (GB-s)
3,240.00 GB-s x 0.0000166667 USD = 0.05 USD (monthly compute charges)
8,640 requests x 0.0000002 USD = 0.00 USD (monthly request charges)
0.50 GB - 0.5 GB (no additional charge) = 0.00 GB billable ephemeral storage per function
Lambda costs - Without Free Tier (monthly): 0.05 USD

Figure 7.3: Estimated cost for the AWS Lambda of the Dashboard

To the costs of the Lambdas, there is an additional 5$ a month for the dashboard and 0.30$ for each custom metric created (Price of November 2023).

These are the metrics obtained considering a data point every 5 minutes:

1. The average Latency added to all the requests and just the slower top 5% by the WAF with and without timeouts (Latency > 240 seconds). It is possible to add the following condition to the filter's list to remove timeouts:

```
{
    "range": {
        "transaction.response.waf-latency": {
            "lt": 240000
            }
    }}
```

Listing 7.5: Timeouts filtered out

Furthermore, two queries were utilized to retrieve only the top 5%. The first one gets the 95th percentile value by applying an aggregation, while the second filters out all the values below that number in milliseconds and calculates the average value.

```
aggs={
    "top_5_perc": {
        "percentiles": {
            "field": "transaction.response.waf-latency",
            "percents": [95]
        }}
    }
```

Listing 7.6: 95th percentile value

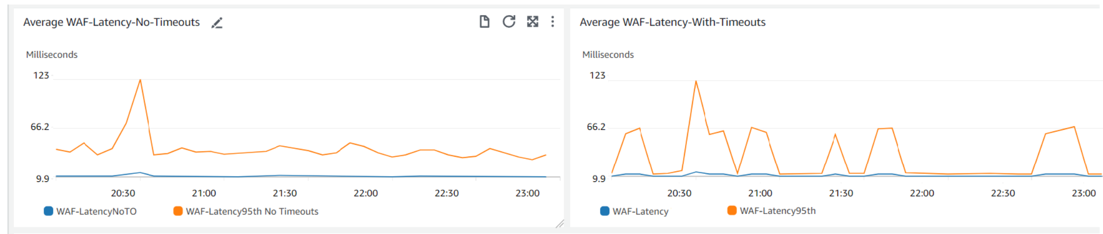


Figure 7.4: Average WAF Latencies

The picture above contains two graphs that show the four metrics created. On the left, the average WAF Latency excludes timeouts. On the right, there is the graph that includes all the documents.

2. The percentage of requests that are allowed, blocked, or alerted. Here, it was possible to utilize the `terms` aggregation. The latter counts the document's occurrences of a specific field value. Then, the three values retrieved were pushed separately to create a custom metric for each value count.

```
aggs={
        "wafstatus":{
            "terms": {
                "field": "status.keyword",
                "include": [ "allowed", "blocked", "alerted"
    ],
                "min_doc_count": 0,
                "size":3
            }
        }}
```

Listing 7.7: Waf Status Aggregation

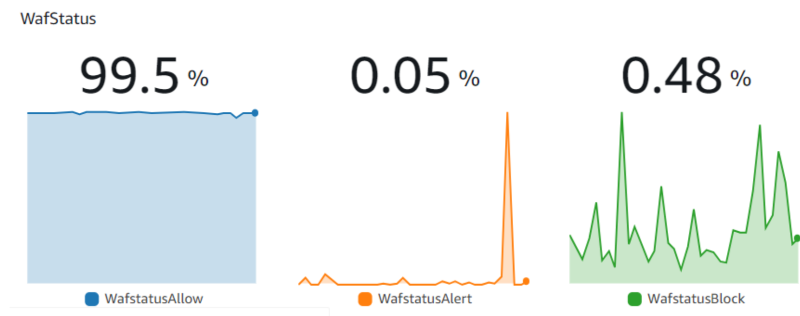


Figure 7.5: WAF requests's status percentage

3. Latency added by the WAF considering different time slots (Now, 1H ago, 6H ago, 12H ago, 1D ago, and 3D ago). The `date_range` bucket aggregation could be utilized to group documents sharing the same timestamp interval. Eventually, it was possible to calculate the average WAF Latency value and push the results separately into AWS Cloudwatch.

```
aggs={
    "time_per_date": {
        "date_range": {
            "field": "@timestamp",
            "format": "strict_date_optional_time",
            "ranges": [
                    {
                    "key": "Now",
                    "from": startdatenow,
                    "to" : enddatenow
                    },
                    {
                    "key": "1H",
                    "from": startdate1h,
                    "to" : enddate1h
                    }, {
                    "key": "6H",
                    "from": startdate6h,
                    "to" : enddate6h
                    }, {
                    "key": "12H",
                    "from": startdate12h,
                    "to" : enddate12h
                    },{
                    "key": "1D",
                    "from": startdate1d,
                    "to" : enddate1d
                    },{
                    "key": "3D",
                    "from": startdate3d,
                    "to" : enddate3d
                    }
                ]
        }, "aggs": {
            "avg_time": {
                "avg": {
                    "field": "transaction.resp.waf-latency"
                }}
    }}}
```

Listing 7.8: Average Waf Latency in different intervals

The following graphs show the precise value of the average WAF Latency for a 5-minute interval, and they also contain a preview of what happened before:

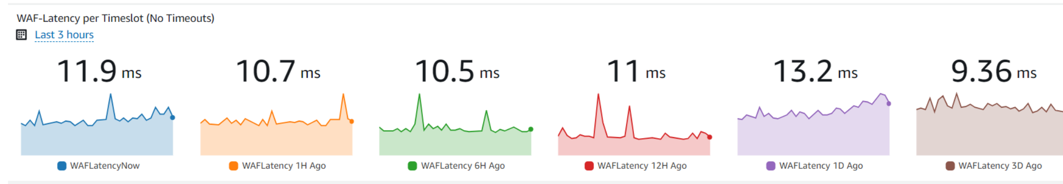


Figure 7.6: Average WAF Latencies of 5-minute intervals

7.3 Elasticsearch Nodes Metrics

The previous paragraph showed some examples of possible metrics that were obtainable by analyzing the content of the documents in Elasticsearch. However, there is no information about the Elasticsearch Cluster. The Elasticsearch Python library also provides access to APIs that enable to get additional statistics about the nodes.

In particular, an Elasticsearch client could be utilized to call the `nodes.stats()` [27] method in Python and retrieve these metrics:

- **fs**: it contains information about the file system and disk storage.
- **process**: it has statistics about CPU usage.
- **JVM**: it keeps statistics about heap memory.

```
res = await esclient.nodes.stats(metric=['fs', 'process', 'jvm'],
            filter_path=["nodes.*.name",
                         "nodes.*.fs.total.total_in_bytes",
                         "nodes.*.fs.total.free_in_bytes",
                         "nodes.*.jvm.mem.heap_used_percent",
                         "nodes.*.process.cpu.percent"])
```

Listing 7.9: Elasticsearch nodes statistics

The `filter_path` parameter enables retrieving only the specified values. The `nodes.stats()` method provides numerous statistics, so applying a filter was necessary. The code above will get the name, the free disk storage, the total disk storage, and the percentage of JVM heap and CPU used for every node. Then, it is possible to add these new metrics in a new CloudWatch namespace (Elastic/stats). Eventually, the names of the cluster and the nodes as Dimensions were required.

Three metrics would be obtained for each node of the Elasticsearch cluster and added to the AWS dashboard:

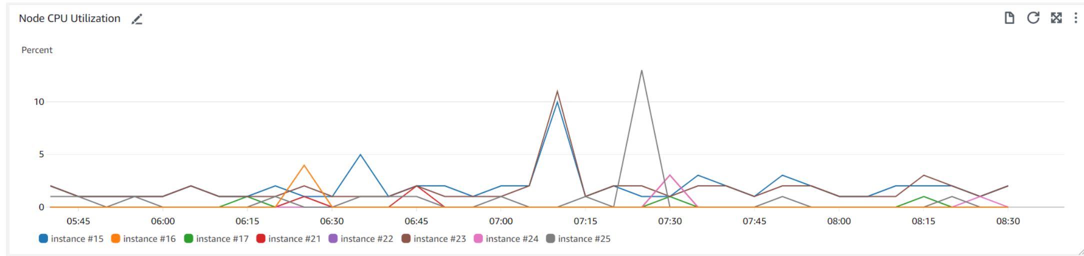


Figure 7.7: CPU Utilization of Elasticsearch Nodes

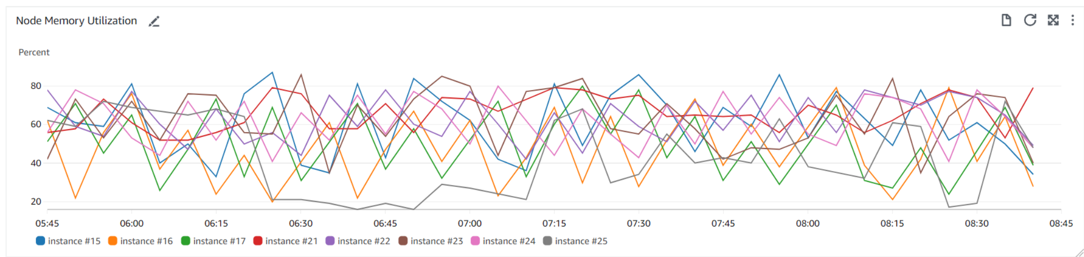


Figure 7.8: Memory Utilization of Elasticsearch Nodes

A **Gauge chart** could be used for the free disk percentage since the nodes' storage percentage changes slower than the CPU and Memory values:

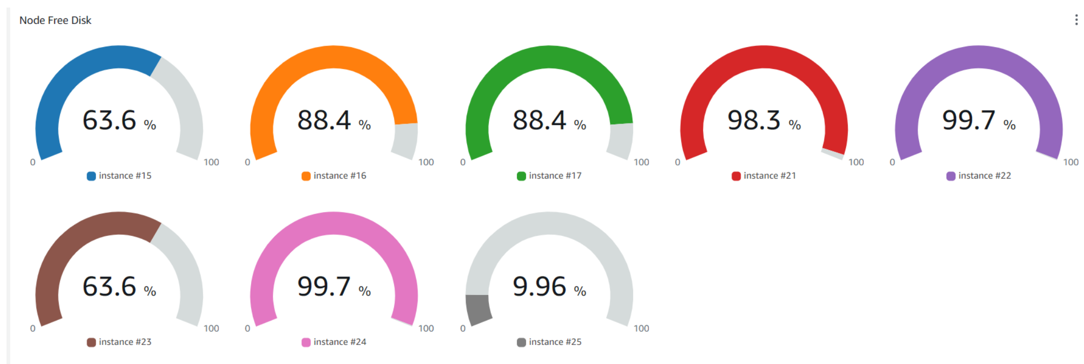


Figure 7.9: Free disk percentage of Elasticsearch Nodes

7.4 Widget using Query Syntax

A WAF is supposed to offer its protection to more than one web application. Kibana would provide a complete view of all the statistics for each of them. But to integrate a preview of how much latency the WAF is adding for single applications in the AWS Dashboard, the steps used before would generate a metric for each website.

Since every custom metric has a fixed cost per month in CloudWatch, this approach is not scalable. Even when creating metrics only for the top 15 slower web applications, they must have the same combination of dimensions. And one of them must be the hostname to recognize the web apps.

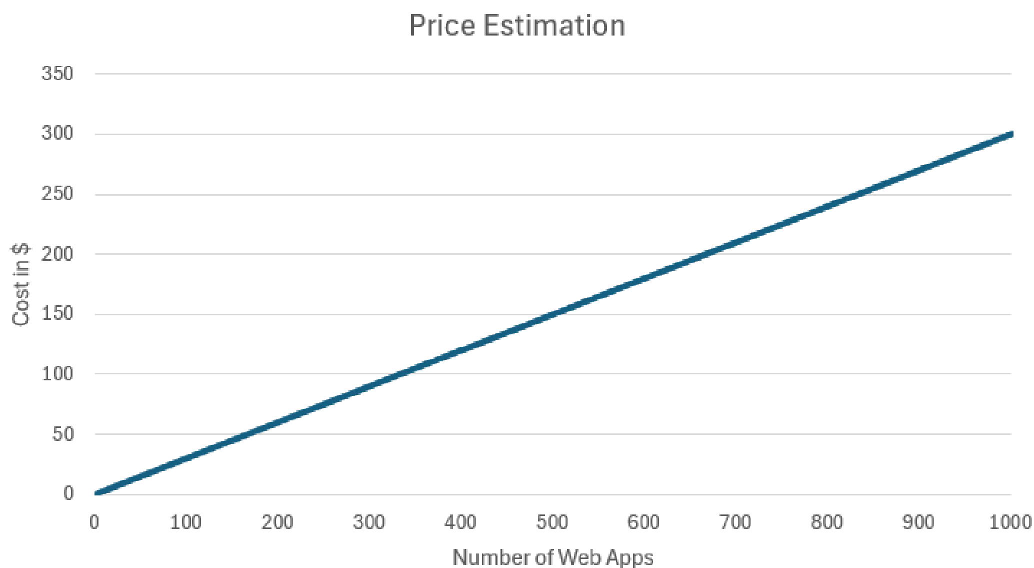


Figure 7.10: Price estimation using the previous approach

For example, if there are 1000 web applications, each would have a different hostname. Some services may be slower than others, but the ranking of latency value might change every 5 minutes. Maybe not all of them, but it is highly possible that in the long run, at least half of the web applications appear at least once in the top 15, considering the small time interval.

All the documents were grouped by hostname in 5-minute intervals using an Elasticsearch query to solve this issue. Then, it was possible to aggregate each bucket by computing the average over the WAF latency value.

Eventually, the results of each bucket were sorted using the descending order, and just the top 15 were kept:

```
aggs={
    "host": {
        "terms": {
            "field": "transaction.host.keyword",
            "size": 500
        }, "aggs": {
            "avg_time": {
                "avg": {
                    "field": "transaction.response.waf-latency"
                }},
            "time_bucket_sort": {
                "bucket_sort": {
                    "sort": [
                        {"avg_time": {"order": "desc"}}
                    ],
                    "size": 15
                }}
    }}}
```

Listing 7.10: Top 15 web apps with high Average WAF Latency

Afterwards, it was possible to create a list of logs, change the timestamp in a compatible format for AWS, and push them into a Log Group in CloudWatch:

```
list_host=[]
timestamp = int(round(datetime.fromisoformat(ed[:-5]).timestamp()
    *1000))
for r in response_host["aggregations"]['host']['buckets']:
    list_host.append({"timestamp": timestamp,
                      "message" : f"host: {r['key']},
                      average: {round(r['avg_time']['value'], 2)},
                      doc_count: {r['doc_count']}"})
```

Listing 7.11: Logs format for Cloudwatch

```
try:
    res= cw_logs.put_log_events(
    logGroupName='dashboard/host-waf-latency',
    logStreamName='host-pl-avg',
    logEvents=list_host
    )
# Display error
except ClientError as e:
    print(e.response['Error']['Message'])
```

Listing 7.12: Sending Logs to CloudWatch

The AWS Lambda function adds 15 logs to CloudWatch every 5 minutes. A new Widget[28] has been deployed in the dashboard that performs a query from the selected Log Group and displays just the newest ones ordered by latency value:

```
parse '*: *,*: *, *: *' as i, host, c, average, d, doc_count
|stats earliest(average) as avg_latency by host, @timestamp
|sort @timestamp desc, avg_latency desc
|limit 15
```

Run query | Cancel | Save | History

Figure 7.11: Logs query for the AWS Widget

This will be the result in the AWS Dashboard:

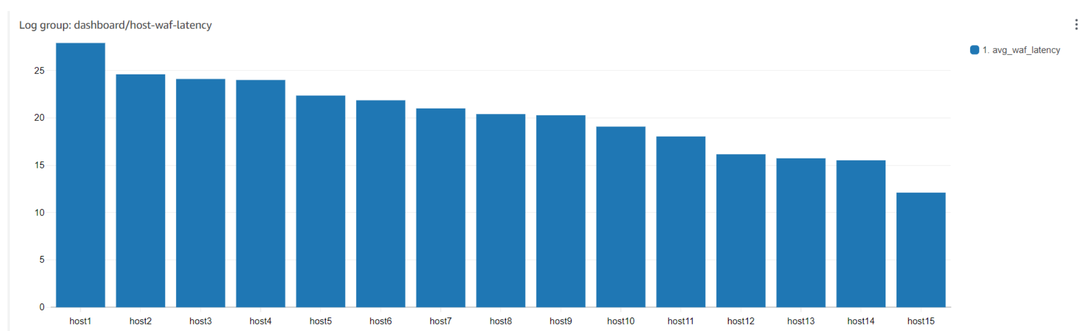


Figure 7.12: Top 15 hostnames with the highest Average WAF Latency in a 5 minute interval

7.5 Alarm Widget

The AWS dashboard has a built-in widget that allows customers to monitor all the alarms they created in their Region, and it gives a quick preview in case something is not working as it should.

The deployed alarms are:

- Health status of EC2 Instances.
- Warning and Error Logs count for ECS.
- CPU and RAM usage for multiple AWS services.
- Active tasks number in ECS.
- Status change for ECS, EC2, and other resources.
- Received Bytes of Network Traffic in ECS.

The following figure shows the alarm widget:

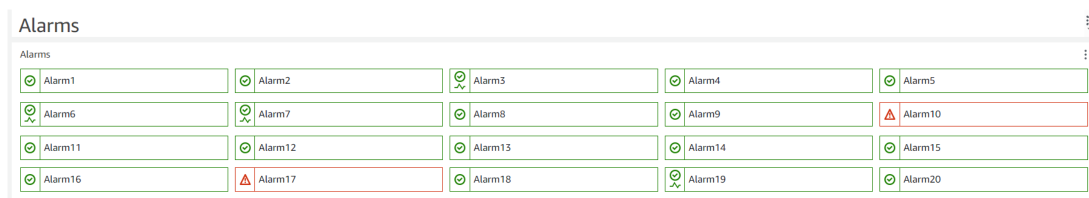


Figure 7.13: Alarm list

7.6 AWS Custom Widgets

The widgets of the dashboard are updated every 5 minutes, but they did not support any user interaction outside of the observed time interval. However, AWS provides an alternative to the standard widgets. It is possible to design custom widgets using AWS Lambda functions[29].

When a custom widget is created, an AWS Lambda can be linked to it, and whatever the Lambda returns in HTML, it will show the result in the AWS Dashboard. CSS and SVG are also supported, but Javascript is not allowed for security reasons.

It is possible to use the <cwdb-action> tag to interact with the elements of the Lambda function from the dashboard and trigger another AWS Lambda call that can execute some code. Then, the result will appear in a new popup or as a substitute for the custom widget.

A list of HTML buttons could be added to the dashboard. So, when someone clicks them, a function performs a specific action. An AWS Lambda could serve more than one button, passing as payload a variable that determines what to do.

In the following code, the customer's list is retrieved automatically from the database. So, every time a new customer arrives, it gets added to the AWS Dashboard.

```
#Create a list of html text that will be used to return the
    buttons in the Study Dashboard that calls other lambdas
string_list=[]

string_list.append('<h3>Check Routes Reachability through WAF
    inside VPC:  ')
for c in customers:
    name=f"HWaf_{c}"
    button=f'''<a class="btn btn-primary">{c}</a>
            <cwdb-action action="call" endpoint="arn:aws:lambda:eu
    -central-1:{os.environ['ACCOUNT_ID']}:function:Check-Services">
    ''' + json.dumps({ "CheckType" : name }) + "</cwdb-action>"

    string_list.append(button)
string_list.append('</h3><br>')
```

Listing 7.13: Customer buttons generation

Eventually, the HTML strings contained in the list variable will be joined to return the final HTML in the custom widget:

```
html2= ''.join(string_list)
return html2
```

Listing 7.14: The code joins and returns the HTML text

In the new Lambda function, it is possible to access the payload sent from the event variable. The latter could be utilized to assign different portions of the code to determine specific tasks for each button:

```
#event contains the check type and the customer selected by the button
checkType = event.get("CheckType", '')

#Example: checktype_customer1
check=checkType.split("_",1)
```

Listing 7.15: Retrieving the check type

The buttons created with the custom widget are shown in the following image:

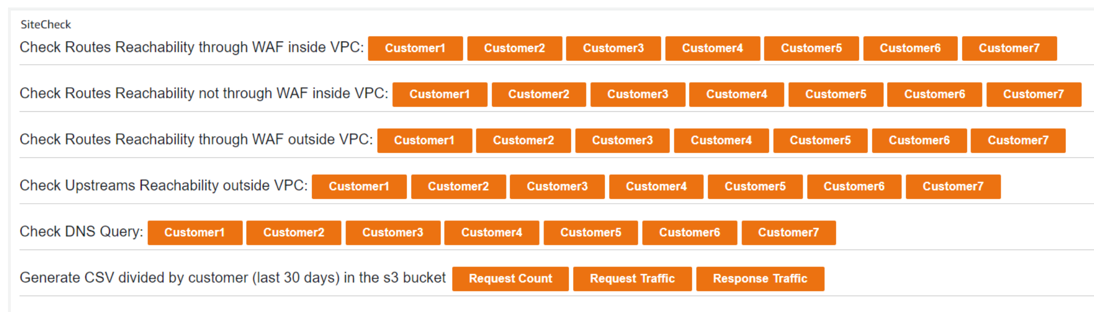


Figure 7.14: Custom widget

Chapter 8

Debugging and Automation

As mentioned before, when a WAF is in front of Web Applications, the latter must block all the traffic not passing through the WAF. Since it is the responsibility of the web app owner to restrict access, there could be misconfigurations that could lead to possible attack vectors bypassing the WAF protection. In this case, an attacker who discovers the public IP of the web application can freely act without one layer of defense. This Chapter describes how to utilize the buttons created before for debugging. Then, it shows how to create reports and send them via mail or save them into a S3 Bucket.

8.1 Reachability Inside and Outside VPC

The buttons, created as a custom widget in the AWS Dashboard, call another AWS Lambda, which performs a specific task. In particular, it sends asynchronous HEAD requests and DNS lookups to all the web applications of a precise customer to check if every configuration is as it should be.

Five checks are possible:

1. Check reachability passing through WAF where the AWS Lambda is inside the same VPC and subnet.
2. Check reachability not passing through the WAF where the AWS Lambda is inside the same VPC and subnet.

3. Check reachability passing through WAF where the AWS Lambda is outside its VPC.
4. Check reachability from outside not passing through the WAF.
5. Check if the DNS lookups return the correct IP addresses.

The code's structure for the first four checks is quite similar but with some minor differences. All of them start with the encapsulation of a connection pool in a session utilizing the `aiohttp` Python library:

```
async def check_host(web_apps):
    async with aiohttp.ClientSession(cookies=None) as session:
        await fetch_all(session, web_apps)
```

Listing 8.1: Client session

Considering customers that could have multiple web applications, an array of tasks was created in Python. So, AWS Lambda could send requests to every hostname of each customer's application. And then, it puts all the results in a Log Group on CloudWatch.

```
async def fetch_all(s, web_apps):
    #sends all the requests to the hostnames of a customer and push the results in a AWS Log Group
    tasks = []
    events= []
    timestamp = int(round(datetime.datetime.utcnow().timestamp()*1000))
    for service in list(web_apps):
        for hostname in service['hostnames']:
            tasks.append(asyncio.create_task(fetch(s, hostname, events, timestamp)))
    await asyncio.gather(*tasks)
    try:
        cw_logs=client("logs", region_name=os.environ['REGION'])
        res= cw_logs.put_log_events(
        logGroupName='dashboard/check_logs',
        logStreamName='hosts-reachability',
        logEvents=events
        )
    except Exception as e:
        print(e)
```

Listing 8.2: Managing the tasks and logging the result

Every task would be slightly different. This is the case 1 and 3, where the requests are checked by the WAF:

```
async def fetch(s, hostname, events, timestamp):
    #send an async request to a hostname and append the result to the "events" list
    url = 'https://' + hostname
    try :
        async with s.head(url, timeout=15, ssl=False) as r:
            events.append({"timestamp": timestamp, "message" : f"Host: {hostname},    Reachable: Yes,    Status_Code: {r.status}"})
    except Exception as e:
        events.append({"timestamp": timestamp, "message" : f"Host: {hostname},    Reachable: No"})
```

Listing 8.3: Case 1 and 3: Task

In case 2, the requests are sent directly to the web application's public IP address. This check is meaningful to exclude the WAF guilt when it is needed to find the solution to an issue.

```
async def fetch_no_waf(s, hostname, ip_address, port, events, timestamp):
    #send an async request to an upstream and append the result to the "events" list
    url = 'https://' + ip_address
    header= hostname + ':' + str(port)
    try :
        async with s.head(url, timeout=15, headers={"Host": header}, ssl=False) as r:
            events.append({"timestamp": timestamp, "message" : f"Host: {hostname},    Upstream: {ip_address},    Reachable: Yes,    Status_Code: {r.status}"})
    except Exception as e:
        events.append({"timestamp": timestamp, "message" : f"Host: {hostname},    Upstream: {ip_address},    Reachable: No"})
```

Listing 8.4: Case 2: Task

In the first two cases, the AWS Lambda was in the same Subnet of the WAF. While in cases 3 and 4, the function was not assigned to any specific VPC. This last configuration simulates a user who wants to send a request to the web application.

Case 4 simulates a user who tries to send the request directly through the public IP of a web application. In this scenario, all the web apps must be unreachable:

```
async def fetch_no_waf(s, host, ip_address, events, timestamp):
    #send an async request to an upstream and append the result to
    the "events" list
    url = 'https://' + ip_address
    try :
        async with s.head(url, timeout=15, ssl=False) as r:
            events.append({"timestamp": timestamp, "message" : f"
   Host: {host},    Upstream: {ip_address},    Reachable: Yes,
   Status_Code: {r.status}"})
    except Exception as e:
        events.append({"timestamp": timestamp, "message" : f"Host:
    {host},    Upstream: {ip_address},    Reachable: No"})
```

Listing 8.5: Case 4: Task

Eventually, the last check would be on the configuration of the web application's DNS. The DNS Lookup provides specific public IPs as a response. In particular, the users must reach the WAF before the offered service.

In the code below, the `socket` Python library was utilized to resolve the DNS lookup:

```
def check_dns(web_apps):

    #Load balancers client
    lbclient = client('elbv2', region_name=os.environ['REGION'])
    response = lbclient.describe_load_balancers()
    list_ip=[]

    #IPs of the LBs
    for l in response['LoadBalancers']:
        addrinfo = socket.getaddrinfo(l['DNSName'], None)
        for info in addrinfo:
            _, _ ,_ ,_, address = info
            list_ip.append(address[0])

    #IPs of Global Accelerator
    addrinfo = socket.getaddrinfo(os.environ['GA_DNS_NAME'], None)
    for info in addrinfo:
            _, _ ,_ ,_, address = info
            list_ip.append(address[0])


```

```
    #Add elastic IPs of the NAT Gateways (FIXED)
    list_ip.append(os.environ['NAT_GATEWAY_IP1'])
    list_ip.append(os.environ['NAT_GATEWAY_IP2'])

    #Check if the IPs returned from DNS lookups are in the list
    events = []
    timestamp = int(round(datetime.datetime.utcnow().timestamp()
*1000))
    for u in web_apps:
        for hostname in u['hostnames']:
            dns_lookup(hostname, u['address'], timestamp, events,
list_ip)
```

Listing 8.6: Retrieving the list of IPs and launching the DNS Lookups

```
def dns_lookup(hostname, upstream_ip, timestamp, events, list_ip):
    try :
        addrinfo = socket.getaddrinfo(hostname,None)
        ip= []
        for info in addrinfo:
            _, _ ,_ ,_, address = info
            ip.append(address[0])
        check="YES"
        for i in ip:
            if i not in list_ip:
                check="NO"
                break
        events.append({"timestamp": timestamp, "message" : f"Host:
 {hostname},    Upstream: {upstream_ip},    IPs found: {list(
set(ip))},    Internal IPs: {check}"})
    except Exception as e:
        events.append({"timestamp": timestamp, "message" : f"Host:
 {hostname},    Upstream: {upstream_ip},    Error resolving DNS
 Lookup"})
```

Listing 8.7: Checking the responses and updating the event list

8.2 Links to Debugging Results

In the previous paragraph, The AWS Lambda pushed all the requests' results in a CloudWatch Log Group. To make access to the generated logs faster, a small textual widget in Markdown was added to the dashboard that would contain links to the location of the logs:

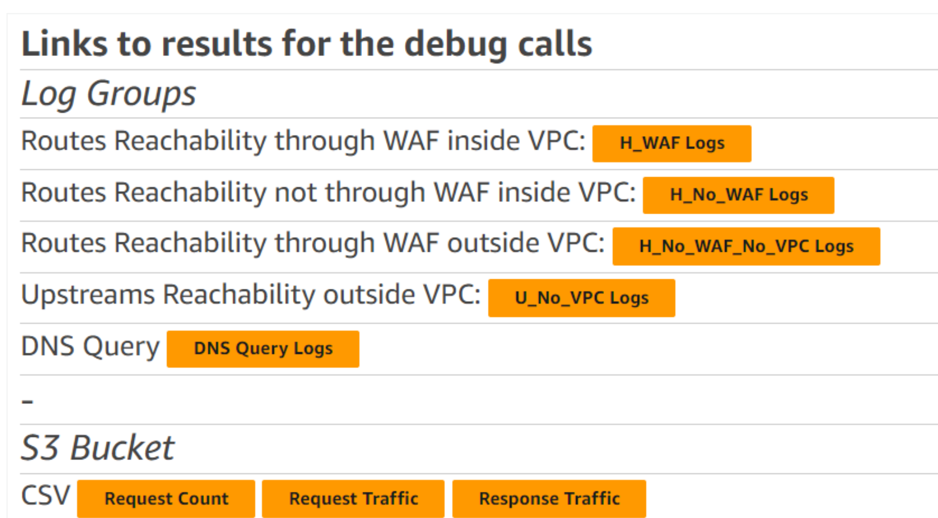


Figure 8.1: Links to results' location

It is possible to obtain the above picture's output by creating a Markdown Widget in the AWS Dashboard that contains the following lines multiple times:

```
# **Links to results for the debug calls**

## *Log Groups*

## Routes Reachability through WAF inside VPC: [button:
primary:H_WAF Logs](link-to-log-group)
```

8.3 Creating CSV Files and Pushing Them into S3

The deployed AWS Dashboard allows administrators to check metrics through graphs and send requests to multiple web applications using one click. Eventually, another feature was added to the dashboard. It collects data from an extensive period using custom widgets. For example, it can count the requests that every hostname received in the last 30 days.

Since analyzing these records could be inconvenient in Log Groups, the results retrieved have been saved into a CSV file from an AWS Lambda using Python. The Python Standard Library already has a built-in module that manages the CSV files. So, it was not necessary to create another Lambda Layer.

Elasticsearch queries retrieved the data of interest (Requests count and the sum of the Content of requests and responses in MB) sent to singular hostnames and divided by customers. In this case, all the documents older than the last 30 days were excluded. Then, it was required to create a bucket of Elasticsearch documents for each customer (`group by`) and apply the same aggregation again for each hostname. Eventually, the sum metric aggregation was applied to the field of interest.

It is possible to create files in AWS Lambda functions, but since they have just ephemeral storage, all the data would be lost when the execution ends. It was needed to temporarily generate a new CSV file in the `/tmp/` folder with the permission to write on that file. Afterwards, a writer object from the CSV library converted the data in the correct format for the CSV file, and its `writerow()` method added the values to the CSV file. The code creates a separate file for each customer through a for loop and orders the lines based on the descending value of the selected field before putting the data in the file.

```
1 for customer, v in result.items():
2     d=dict(sorted(v['host'].items(), key=lambda item: item[1],
    reverse=True))
3     with open(f'/tmp/{date.day}-{date.month}-{date.year}_{customer
    }_req.csv', 'w', newline ='') as f:
4         thewriter = csv.writer(f)
5         thewriter.writerow(['Hostnames','Request Traffic (MB)'])
6         for h, c in d.items():
7             thewriter.writerow([h,f'{int(round(c/(1024*1024),3))
    :,}'])
```

Listing 8.8: CSV File creation

As mentioned before, AWS Lambda can't maintain the files created. But, there is an AWS service that can solve this issue. Amazon S3 (**Simple Storage Service**)[30] is a service that offers storage capabilities and security to save data on the cloud. S3 considers every file as an object. The latter has its unique key and can even support versioning. S3 objects are stored in customizable Buckets with different configurations and policies to control access and organization.

After bucket creation, called `my-s3bucket` in this case, it was possible to utilize the s3 client from the AWS Lambda using the boto3 AWS SDK:

```
s3_client = client('s3',
region_name=os.environ['REGION'])
```

The parameters required for the `upload_file()` method were:

- The source file that is contained in the tmp folder.
- The destination bucket.
- The destination path and file name. It is possible to create a new folder if not found.

```
resp = s3_client.upload_file(f'/tmp/{date.day}-{date.month}-{date.year}_{customer}_req.csv',
        os.environ['S3_BUCKET'],
        f'request_traffic/{date.day}-{date.month}-{date.year}_{customer}_req.csv')
```

Listing 8.9: Pushing into the S3 Bucket

8.4 Daily and Monthly Reports in Python

The developed monitoring system provides a general visual preview and statistics about simple and custom metrics on both the ECS and Elasticsearch Cluster, an alarm system that sends email notifications in case of problems, and a debug feature that allows checking the customers' configurations and creating custom CSV file saved on S3. Having a dashboard is useful when it is required to analyze how an architecture is behaving, but not everyone has the will to learn, set, and manage an AWS User account.

The AWS Dashboard provides the functionality to share the dashboard via mail to existing users with credentials or to make it even public and accessible using links. However, there are some security issues to consider since collecting sensitive information from the dashboard is possible.

For example, all the users that access the AWS Dashboard can read not only metrics and EC2 instances present in the monitoring system. But all the other ones are present in the account. Even if there is no need to share the AWS dashboard outside of the cloud environment, it is possible to exploit the AWS Lambda function for generating daily and monthly reports that provide a summary based on data from the previous day or month.

A daily or monthly report executes the Lambda function's code with a lower frequency than the one used for the metrics (every 5 minutes). But also the quantity of documents it has to work on is higher. In this scenario, it would be better not to use asynchronous queries since the lambda would run once a day/month. In particular, considering several simultaneous queries on documents collected in approximately 30 days, there could be a heavy workload on the nodes that could be avoided by simply executing the queries sequentially.

One way to send reports is using emails. It was possible to get the statistics from Elasticsearch, applying minor changes to the utilized queries to create metrics for the dashboard. The following code saves the result of the Elasticsearch in a list of lists. It is one of the supported inputs to create the table's rows:

```
custres=[]
for r in query_response["aggregations"]['customers']['buckets']:
    cust_count=r['doc_count']
    custres.append([r['key'], f'{int(cust_count):,}'])
```

Listing 8.10: Tabulate input format

Python's `tabulate` library allows developers to create tables and, in general, to format tabular data. The `tabulate()` function, provided by the homonymous library, was utilized to give a more ordered view of the results and add multiple tables to the HTML body:

```
#Email body
BODY_HTML = f'''<html>
<head></head>
<body>
<h1>Daily Report</h1>
<h2>From: {sd}</h2>
<h2>To: {ed}</h2>
<br>
<h3>Customers Count:</h3>
{tabulate(custres, headers=["Customer--------------","Number of
   requests"], tablefmt="html")}
<br>

...

</body>
</html>'''
```

Listing 8.11: Simplified HTML email body

Then, it was required to specify the headers and the format, which would be HTML in this case. A new Lambda Layer must be created and added to the AWS Lambda function to utilize `tabulate`.

Several different tables and data could be inserted in the mail body. Considering a daily report, information that could be useful is the average WAF latency divided by the hour from Elasticsearch. In particular, it could help to investigate when something was not working and research the reason behind it. It was possible to obtain these data using the bucket aggregation `date_histogram` with a specific time interval:

```
aggs={
    "req_per_hour": {
        "date_histogram": {
            "field": "@timestamp",
            "calendar_interval": "hour"
        },
    "aggs": {
        "avg-waf-latency":{
            "avg": {
                "field": "transaction.response.waf-latency"
            }
        }
    }}}
```

Listing 8.12: Calendar interval aggregation

The following code creates three columns in the table (Time interval, average WAF Latency, Request Count) and a row for each hour of the day:

```
hour_table=[]
for r in res_hours["aggregations"]['req_per_hour']['buckets']:
    d=datetime.fromisoformat(r['key_as_string'][:-5])
    request_count=r['doc_count']
    hour_table.append([d, str(round(r['avg-waf-latency']['value'])
    ), "", f'{int(request_count):,}'])
```

Listing 8.13: Tabulate table format

8.5 Email Automation in Lambdas using SES

The previous paragraph showed how to create a simple mail body in HTML with the option to add all the relevant data and needed tables. The Boto3 AWS SDK also supports email delivery. In particular, it can interact with AWS SES (**Simple Email Service**).

Here is a description of how it works from the AWS Documentation[31]: "Amazon Simple Email Service (SES) is an email platform that provides an easy, cost-effective way for you to send and receive email using your own email addresses and domains."

A "Verified identity" must be configured in AWS SES to send emails. It represents proof of ownership of the email address or the domain. However, only the sender needs verification. Then, specific access policies must be added to the Lambda function's role and set a periodical trigger on AWS Eventbridge after the email verification. Eventually, it was possible to create an AWS SES client using Boto3 and send the custom email previously created:

```
try:
    ses_client=client('ses', region_name=os.environ['REGION'])
    #Send email
    response = ses_client.send_email(
    Destination={
        'ToAddresses': [ 'mydestination@example.com' ],
    },
    Message={
        'Body': {
            'Html': {
                'Charset': 'UTF-8',
                'Data': BODY_HTML,},
        },
        'Subject': {
            'Charset': 'UTF-8',
            'Data': f'Daily Report {date.day}-{date.month}-{date.year}',
        },
    },
    Source='verifiedmail@example.com')
# Display error
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    print(f"Email sent! Message ID: {response['MessageId']}")
```

Listing 8.14: Sending emails using AWS SES

The following picture contains a preview of two tables created for the daily report:

Daily Report

From:2024-02-11 00:00:00
To: 2024-02-11 23:59:59

Customers Count:

Customers---------------	Number of requests
Customer1	298,000
Customer2	84,000
Customer3	500,000
Customer4	52,000
Customer5	300,000

Avg WAF Latency by hour:

Hour--------------------	Avg WAF Latency (ms)--------	Number of requests
2024-02-11 00:00:00	18	22,000
2024-02-11 01:00:00	13	20,000
2024-02-11 02:00:00	14	24,000
2024-02-11 03:00:00	13	30,000
2024-02-11 04:00:00	13	40,000
2024-02-11 05:00:00	17	40,000
2024-02-11 06:00:00	45	30,000
2024-02-11 07:00:00	23	44,444
2024-02-11 08:00:00	11	20,000
2024-02-11 09:00:00	22	30,000
2024-02-11 10:00:00	44	10,000
2024-02-11 11:00:00	12	30,000
2024-02-11 12:00:00	12	20,000
2024-02-11 13:00:00	10	40,000
2024-02-11 14:00:00	10	40,000
2024-02-11 15:00:00	10	50,000
2024-02-11 16:00:00	82	70,000
2024-02-11 17:00:00	210	120,000
2024-02-11 18:00:00	150	85,000
2024-02-11 19:00:00	11	20,000
2024-02-11 20:00:00	19	30,000
2024-02-11 21:00:00	10	50,000
2024-02-11 22:00:00	15	30,000
2024-02-11 23:00:00	16	30,000

Figure 8.2: Example of mail report

8.6 Costs Report using Excel

The previous steps were utilized to automate the generation of Excel files that provide a monthly cost analysis. In particular, an AWS Cost Explorer client was created using Boto3 to retrieve the monthly cost of the AWS infrastructure.

Furthermore, Elastic Cloud provides an API that allows customers to obtain the costs for the nodes of the Elastic Cluster. In this case, an API KEY and the Organization ID were required to access:

```
ce_client = client('ce', region_name=os.environ['REGION'])

ce_resp = ce_client.get_cost_and_usage(
TimePeriod={
    'Start': f'{previous.year}-{previous.month}-01',
    'End': f'{current.year}-{current.month}-01'
},
Granularity='MONTHLY',
Metrics=[
    'AmortizedCost',
])

aws_price = round(float(ce_resp['ResultsByTime'][0]['Total']['
    AmortizedCost']['Amount']),2)

es_resp = requests.get(f"https://api.elastic-cloud.com/api/v1/
    billing/costs/{os.environ['ELASTIC_ORG_ID']}?from={previous.
    year}-{previous.month}-01T00:00:00.000Z&to={current.year}-{
    current.month}-01T00:00:00.000Z", headers= {"Authorization": f"
    ApiKey {os.environ['API_KEY']}"})
es_price=json.loads(es_resp.text)['costs']['total']

total_cost=aws_price+es_price
```

Listing 8.15: Retrieving costs from AWS and Elastic Cloud

Once all the metrics of interest from AWS and Elasticsearch were retrieved, the `openpyxl` Python library was utilized to create and add several values and formulas to different Excel Sheets.

Then, the monthly Excel file will be uploaded to an S3 Bucket. The latter also contains an annual file called `Total_{previous.year}.xlsx` that will be downloaded in the AWS Lambda `/tmp/` folder.

Afterwards, a Python function will insert the values of the monthly Excel in the annual one, and eventually, the AWS Lambda pushes the updated version again to the S3 Bucket:

```
#Upload excel to S3
try:
    s3_client = client('s3', region_name=os.environ['REGION'])
    resp = s3_client.upload_file(f'/tmp/Costs_{previous.month}-{previous.year}.xlsx', os.environ['S3-Bucket'], f'Cost_Reports/{previous.year}/Costs_{previous.month}-{previous.year}.xlsx')

    first=0
    if previous.month == 1:
        first=1
    else:
        d_resp = s3_client.download_file(os.environ['S3-Bucket'], f'Cost_Reports/{previous.year}/Total_{previous.year}.xlsx', f'/tmp/Total_{previous.year}.xlsx')

    merge_script.fill_excel(first, previous.year, previous.month)
    resp = s3_client.upload_file(f'/tmp/Total_{previous.year}.xlsx', os.environ['S3-Bucket'], f'Cost_Reports/{previous.year}/Total_{previous.year}.xlsx')

except Exception as e:
    print(f"Error in uploading the Excel to s3: {e}")
```

Listing 8.16: Pushing the monthly Excel and updating the annual one

Chapter 9

Dashboard Limitations

The dashboard previously created on AWS CloudWatch has some limitations. It provides a general view of the statistics. However, it is not flexible about data. It allows users to create a lot of custom metrics and aggregate them based on the data points published on CloudWatch metrics. But, if a specific customer web application must be analyzed considering multiple fields in an Elasticsearch document, it would not be feasible.

It is possible to automate the creation of all metrics based on document variables, but how scalable could that approach be? Likely, checking just one field is not enough to make decisions. A combination of more fields could be required. In particular, in case the reason behind the misbehavior of the architecture is unknown.

9.1 Kibana

As mentioned before, the Elastic Stack is not composed only of Elasticsearch. The latter is responsible mainly for document management and storage, whereas it is possible to utilize **Kibana** for observability and monitoring of the Elasticsearch data.

Kibana is a powerful application that allows users to visualize and create dashboards with high document flexibility and fast data analysis capabilities. It also supports alerting, log interaction, and a lot more.

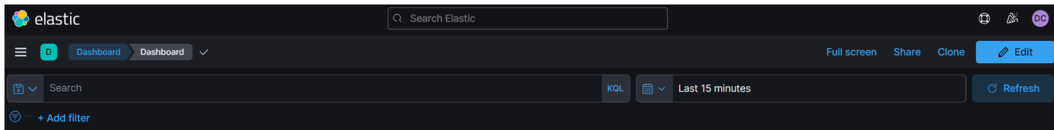


Figure 9.1: Kibana's filter panel

Kibana supports two query languages to filter documents:

- KQL (Kibana Query Language)
- Lucene

KQL allows users to combine multiple text-based queries using `AND` or `OR` operators and supports wildcards for fields or values. For example, it is possible to filter out all the documents with a latency lower than 50 ms for a specific customer with this command:

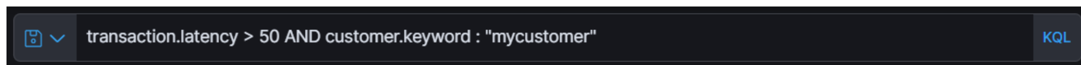


Figure 9.2: Example of KQL

It offers maximum flexibility over the documents required for analysis. It can also be utilized to set a custom filter and time interval for documents that would be applied automatically as the default configuration of Kibana.

9.2 Dashboard Visualizations

In the Kibana Dashboard, the term **visualization** refers to the widgets (called in the AWS Dashboard) that could be utilized to create custom graphs and more. Every visualization has a visualization layer that determines the structure of the widget (e.g. vertical bar, line, pie charts, gauge, etc.)

Kibana dashboards support switching modality from `view` to `edit` to add new visualizations. Then, for every widget, it is required to specify the index that collects the documents and what it will contain as horizontal and vertical axis values.

For example, to get the average latency added by the WAF, it is possible to choose the timestamp value as the horizontal axis and a function as the vertical one. The average metric aggregation is one of the simple functions available by default, and it could be applied to fields present in the documents.

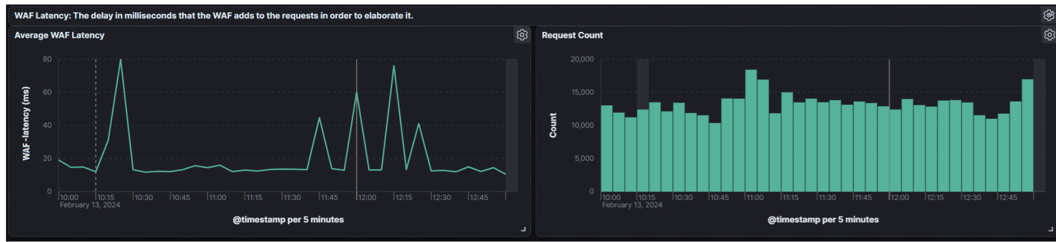


Figure 9.3: Left: Average WAF Latency. Right: Requests count

Similar steps were utilized to add the count of all the requests by choosing another aggregation type. It is possible to generate all the basic graphs needed using Kibana, and its high flexibility, allows users to change the targets of analyses in seconds using filters.

The first graph below compares the impact the WAF applies on the latency of the requests with the one that the customers' application would have without the active protection. The second one counts the number of requests considered legit (allowed), blocked, and alerted. In this case, a standard function is not enough:

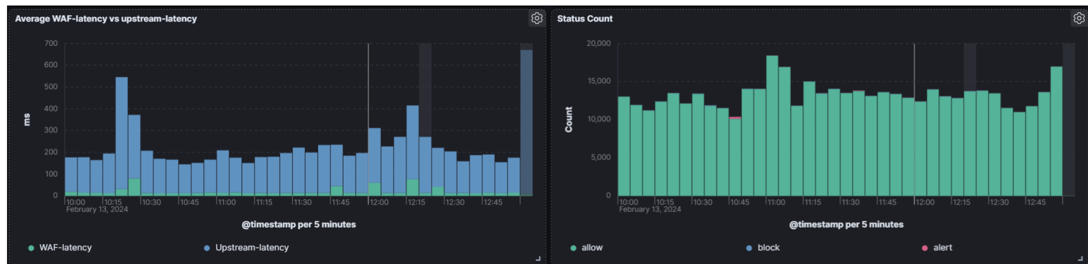


Figure 9.4: Left: Average WAF Latency + Average latency of customer web apps without the WAF. Right: WAF status count

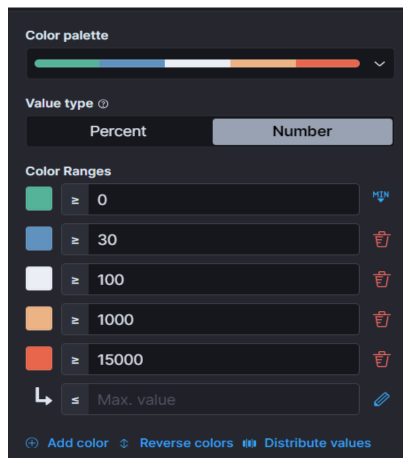
A custom formula must be added on the vertical axis for each metric. In particular, the KQL was utilized to apply a filter before the count aggregation:

1. `count(kql='status.keyword : allowed')`
2. `count(kql='status.keyword : blocked')`
3. `count(kql='status.keyword : alerted')`

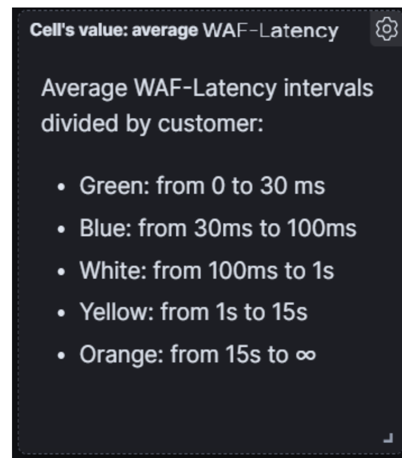
As mentioned before, Kibana supports more types of visualization. Since line and bar charts can become confusing when having more than 3-4 metrics in the vertical axis, a **heatmap** has been used to achieve a more granular visualization. It was possible to check the average latency added by the WAF grouped by customer:

Figure 9.5: WAF Latency Heatmap

Then, every document in the customer bucket has been assigned to different intervals based on custom limits. It is possible to configure different colors to get a clear view of how everything is working. In the visualization's configuration, values could be added in cell value intervals from a panel once the aggregation metric for a field has been done:



(a) Configuration of intervals



(b) Description

Kibana's dashboard provides the functionality to create tables that perform custom aggregations and collect all the results in rows. The following two tables hold information about the count of requests that fall in a specific interval of status code value, divided by customer and hostname in descending order, considered over the sum of the hostname requests.

Then, it could be meaningful to get an ordered list of IPs that send numerous requests to perform investigations and analysis:

Status Code Count

Customers	Hostnames	Status Code	Count of records
Customer1	hostname1	200	88,888
Customer1	hostname1	302	444
Customer1	hostname1	404	2,222
Customer1	hostname2	200	66,666
Customer1	hostname2	302	333
Customer1	hostname2	403	3
Customer1	hostname2	404	1,111

IP Count

Top 25 values of ip	Count of records
11.11.11.11	61,199
22.22.22.22	29,232
33.33.33.33	18,632
44.44.44.44	11,577
55.55.55.55	10,215
66.66.66.66	6,948
77.77.77.77	6,107

Figure 9.7: Left: Status Code count. Left: Request count by IPs.

The two tables below show the total number of requests per customer if a data breach happened and how many have been detected, divided by hostname and customer. In this last case, an AWS Lambda function periodically generates a report with the information to inform the customers.

Request count per customer

Customers	Count of records
Customer1	2222
Customer2	1111
Customer3	12345
Customer4	13456
Customer5	111
Customer6	1713

BeenPwned Count per customer

Customers	Hostnames	Count
Customer1	www.example.com	12
Customer1	www.example2.com	7
Customer2	www.my-site.com	3

Figure 9.8: Left: Request count by customer. Right: havebeenpwned count

9.3 Apdex in Kibana

In the previous paragraph, some simple visualizations were created in Kibana. However, it is possible to add a new metric to provide a different evaluation of how the WAF is performing.

This metric is called **Apdex** and from Wikipedia's definition[32]:

"Apdex (Application Performance Index) is an open standard developed by an alliance of companies for measuring the performance of software applications in computing. Its purpose is to convert measurements into insights about user satisfaction by specifying a uniform way to analyze and report on the degree to which measured performance meets user expectations."

Apdex is a metric that provides a score to the overall performance based on the value of a threshold T. In particular, it requires defining a target value (e.g. 30 ms) for the latency the WAF adds to the requests.

The following formula was utilized to calculate the Apdex score percentage:

$$Apdex = 100 * \frac{Satisfied * 1 + Tollerated * 0.5 + Frustrated * 0}{TotalRequests} \ . \tag{9.1}$$

In (9.1) the values have the following meaning:

1. **Satisfied**: The latency added by the WAF to a request is lower than the target value of 30 ms (T). The request counts as 1.
2. **Tolerated**: The latency added by the WAF to a request has a value between T(30 ms) and 4T(120 ms). The request weighs 0.5.
3. **Frustrated**: The latency added by the WAF to a request is higher than 120 ms (4T). The request is not considered.
4. **TotalRequests**: Sum of all the requests.

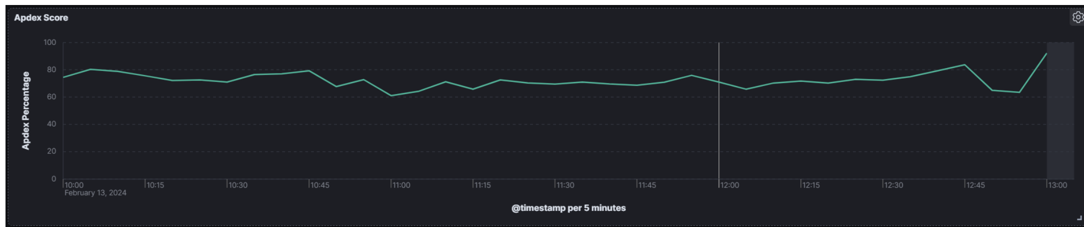


Figure 9.9: Apdex score graph

Eventually, it is possible to use the custom formula panel that Kibana offers in the vertical axis to get the graph of the Apdex score. In this case, only the allowed requests were considered in the following formula that utilizes KQL:

```
100*(count(kql='transaction.waf-latency < 30 and
status.keyword :  "allowed"')*1 + count(kql=
'transaction.waf-latency >= 30 and
transaction.waf-latency <120 and status.keyword :
"allowed"')*0.5)/count(kql='status.keyword :  "allowed"')
```

The combination of all these visualizations provides an extensive view of the web applications statistics that resolved the limitation of the previous Dashboard.

Chapter 10

Conclusions and Future Works

In conclusion, the research results demonstrate how it is possible to develop a monitoring system that can offer additional views for new metrics and statistics of the overall health status of the WAF infrastructure. In particular, the serverless approach was utilized to maintain high efficiency while cutting down data gathering costs and to integrate the functionalities of Amazon Web Services and Elastic Stack. Furthermore, the deployed alerting and debugging system allows the detection of issues in minutes. While the automation of reports on a daily and monthly basis provides supplemental summaries that could be useful for performing analysis.

So, it is possible to assert that all the objectives set at the beginning of the thesis have been achieved, allowing to improve the quality of service provided to the customers.

Considering future works, it could be possible to integrate the debugging system developed utilizing AWS Lambda functions into WAF's dashboard provided to the customers. It would enable them to be aware of misconfigurations that could compromise the security of their web applications and possibly reduce the workload for the service provider company. Additionally, it could be possible to analyze over a prolonged period the alarms' metrics to improve thresholds and reduce even more false positives. Eventually, a machine-learning solution could be deployed to prevent and discover new issues.

Bibliography

[1] Uptime Institute. Uptime institute's 2022 outage analysis finds downtime costs and consequences worsening as industry efforts to curb outage frequency fall short, 2022. URL https://uptimeinstitute.com/about-ui/press-releases/2022-outage-analysis-finds-downtime-costs-and-consequences-worsening.

[2] Ponemon Institute. Cost of data center outages, 2016. URL https://www.vertiv.com/globalassets/documents/reports/2016-cost-of-data-center-outages-11-11_51190_1.pdf.

[3] Statista. Average cost per hour of enterprise server downtime worldwide in 2019, 2020. URL https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/#statisticContainer.

[4] A. Marget. Unitrends. Downtime: Causes, costs and how to minimize it, 2021. URL https://www.unitrends.com/blog/downtime-causes-costs-and-how-to-minimize-it.

[5] Amazon Web Services Inc. Global infrastructure, n.d.. URL https://aws.amazon.com/about-aws/global-infrastructure/?pg=WIAWS-N&tile=learn_more.

[6] Amazon Web Services Inc. What is amazon vpc? - amazon virtual private cloud, n.d.. URL https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.

[7] Amazon Web Services Inc. Infrastructure security in amazon vpc - amazon virtual private cloud, n.d.. URL https://docs.aws.amazon.com/vpc/latest/userguide/infrastructure-security.html#VPC_Security_Comparison.

[8] Amazon Web Services Inc. Subnets for your vpc - amazon virtual private cloud, n.d.. URL https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html.

[9] Amazon Web Services Inc. Aws identity and access management, n.d.. URL https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.

[10] Amazon Web Services Inc. What is amazon ec2? - amazon elastic compute cloud, n.d.. URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[11] Amazon Web Services Inc. What is amazon elastic container service? - amazon elastic container service, n.d.. URL https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[12] Amazon Web Services Inc. What is elastic load balancing? - elastic load balancing, n.d.. URL https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html.

[13] Amazon Web Services Inc. Load balancer types - amazon elastic container service, n.d.. URL https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html#alb-considerations.

[14] Docker Inc. Docker overview, n.d.. URL https://docs.docker.com/get-started/overview/.

[15] Wikipedia Foundation Inc. Logging (computing), 2023. URL https://en.wikipedia.org/wiki/Logging_(computing).

[16] Python Software Foundation. logging — logging facility for python, n.d. URL https://docs.python.org/3/library/logging.html.

[17] Amazon Web Services Inc. What is amazon cloudwatch? - amazon cloudwatch, n.d.. URL https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.

[18] Amazon Web Services Inc. Amazon cloudwatch concepts - amazon cloudwatch, n.d.. URL https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html.

[19] Amazon Web Services Inc. Filter pattern syntax for metric filters - amazon cloudwatch logs, n.d.. URL https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/FilterAndPatternSyntaxForMetricFilters.html.

[20] Amazon Web Services Inc. What is amazon eventbridge? - amazon eventbridge, n.d.. URL https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html.

[21] Amazon Web Services Inc. What is aws lambda? - aws lambda, n.d.. URL https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[22] Amazon Web Services Inc. Working with lambda layers - aws lambda, n.d.. URL https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html.

[23] Elasticsearch B.V. Elastic stack, n.d.. URL https://www.elastic.co/elastic-stack/.

[24] Elasticsearch B.V. Python elasticsearch client — python elasticsearch client 8.11.0 documentation, n.d.. URL https://elasticsearch-py.readthedocs.io/en/v8.11.0/.

[25] Elasticsearch B.V. Aggregations | elasticsearch guide [8.11] |, n.d.. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html.

[26] Amazon Web Services Inc. Quickstart - boto3 1.33.1 documentation, n.d.. URL https://boto3.amazonaws.com/v1/documentation/api/latest/guide/quickstart.html.

[27] Elasticsearch B.V. Nodes stats api | elasticsearch guide [8.11] |, n.d.. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/cluster-nodes-stats.html.

[28] Amazon Web Services Inc. Cloudwatch logs insights query syntax - amazon cloudwatch logs, n.d.. URL https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html.

[29] Amazon Web Services Inc. Add a custom widget to a cloudwatch dashboard - amazon cloudwatch, n.d.. URL https://docs.amazonaws.cn/en_us/AmazonCloudWatch/latest/monitoring/add_custom_widget_dashboard.html.

[30] Amazon Web Services Inc. What is amazon s3? - amazon simple storage service, n.d.. URL https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html.

[31] Amazon Web Services Inc. What is amazon ses? - amazon simple email service, n.d.. URL https://docs.aws.amazon.com/ses/latest/dg/Welcome.html.

[32] Wikipedia Foundation Inc. Apdex, 2023. URL https://en.wikipedia.org/wiki/Apdex.