POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Enhancing Cloud-Based Web Application Firewalls with Machine Learning Models for Bot Detection and HTTP Traffic Classification

Supervisors

Prof. Cataldo BASILE

Dr. Daniele UCCI

Eng. Ivan RUSSO

Candidate

Damiano FERLA

October 2024

Abstract

Recently, cybersecurity attacks has become increasingly complex, with an increase in automated attacks and vulnerabilities exploitations in web applications. Online threats, such as bots or Cross-Site Scripting (XSS) attacks, represent new challenges for data or user protection. Since the birth of the OWASP Top 10 in 2003, XSS attacks have always remained firmly planted in their report. Starting from 2021, XSS attacks have been included in a more general category called "Injection", which includes other threats such as SQL injection. In the 2023 report, the category dedicated to Injection is in third place. In addition, according to the Imperva 2023 report, 49.6% of Internet traffic is composed of bots. Of these, 32% are bad bots, that perform automated tasks with malicious intent, such as extracting data from websites without permission to reuse them and gain a competitive advantage.

Improvements in machine learning, particularly through unsupervised and supervised learning techniques, have opened up new solutions for the detection and prevention of these cyber threats. Past researches have identified machine learning models for detecting bot-generated traffic and for detecting XSS attacks, already demonstrating the potential of these tools. However, implementing these technologies requires a robust and flexible infrastructure, capable of handling large amounts of data and providing adequate computing capacity.

The aim of the following thesis is therefore to implement an architecture on the Amazon Web Services public cloud, to enable the use of machine learning models for the detection of automated bots and XSS attacks. The use of cloud computing offers several advantages, such as scalability, the availability of on-demand resources, and the ability to integrate different services together. This architecture aims to combine the strengths of unsupervised and supervised machine learning techniques with the computational capabilities offered by cloud platforms, providing a scalable solution for web application security.

In this thesis, a cloud architecture will be examined to implement a threat detection system based on machine learning, including the analysis of each architectural component, the integration with other related cloud services, and the integration with a proprietary tool for the defense of web applications. Furthermore, the effectiveness of this architecture will be evaluated on real use cases, in terms of model accuracy but also in terms of execution time.

The result of this research is an architecture developed entirely within the Amazon AWS cloud, consisting of the Amazon MSK, Amazon ECS, Amazon SageMaker and Elastic Cloud services, capable of obtaining predictions for bot detection, which for detection of XSS attack attempts. With the resulting architecture, the average latency time for bot detection is 40.56 seconds, obtained by analyzing about 13,000 sessions, and the average latency time for XSS attack attempt detection is 11.23 seconds, for about 1-2 suspicious requests.

Acknowledgements

I would like to express my most sincere gratitude to all those who accompanied me in this academic journey, concluding in the writing of this thesis.

In particular, I thank *Prof. Basile*, for guiding me through the most important stage.

An heartfelt thanks goes to *Daniele Ucci* and *Ivan Russo*, thesis advisors and colleagues, for their constant support, helpful advice and complicity in the realization of each chapter of the thesis.

I also thank *AizoOn*, the company this thesis was developed with, and *Simone Janin*, as my manager and CTO of the Cyber Security department, for giving me the opportunity to carry out the project providing time and resources to devote to me.

Finally, I thank my family for always being there for me with the highest level of support and understanding.

Thank you all for making this achievement possible.

Damiano

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

AWS

Amazon Web Services

AZ

Availability Zone

CSP

Content Security Policy

DDoS

Distributed Denial of Service

DOM

Document Object Model

ECR

Elastic Container Registry

ECS

Elastic Container Service

ENISA

European Union Agency for Cybersecurity

HTTP

HyperText Transfer Protocol

IaaS	Infrastructure as a Service
IoT	Internet of Things
ML	Machine Learning
MSK	Managed Stream for Kafka
PaaS	Platform as a Service
SaaS	Software as a Service
WAF	Web Application Firewall
WAAP	Web Application & API Protection
XSS	Cross-Site Scripting

Chapter 1

Introduction

In today's world, informatization covers every aspect of society, transforming the way we interact, communicate and do business. Technology has enabled global connectivity, where information and services are accessible anywhere, anytime. According to DataReportal's Global Overview Report 2024 [1], more than 66% of the world's population uses the Internet, and 69.4% own a mobile device.

This increase affects not only the common people but also companies, and the digitization process has a strong impact on their businesses. In fact, according to Flexera 2023 report [2], digital transformation is a high priority for 74% of companies, up from 56% in 2021. In addition, 64% [3] of companies believe they need to implement new digital businesses to stay ahead in the race among competitors.

In this context, web applications have become critical for businesses. They are not just storefronts, but enable them to provide services, manage operations, and interact with customers more quickly and flexibly. For example, many companies implement Customer Relationship Management (CRM) systems to collect, analyze and manage customer data, thereby improving customer loyalty.

However, new security challenges emerge as more digital technologies become available and in use. Cybersecurity has assumed a crucial role in protecting systems, networks and data from a wide range of threats. The ENISA Threat Landscape [4] 2024 report highlights how cyber threats continue to evolve, with increasingly complex attacks. Major threats include ransomware attacks, which continue to affect a wide range of industries, with 18% of commercial service companies and 17% of manufacturing companies, or DDoS attacks, which are the most frequent threat, but also data threats such as data breaches: nearly 20.000 data breach incidents were identified in 2024.

The data from this report offer a clear picture of the growing importance of security solutions such as Web Application Firewalls (WAFs). These firewall components are specifically designed to monitor, filter, and block HTTP requests that pass to and from web applications, protecting the company's IT infrastructure from a wide range of threats.

Web applications are a main target for cyber attacks, as they are publicly accessible and often handle sensitive data and critical business transactions. In particular, companies operating in sectors such as e-commerce, finance and telecommunications, where operational continuity and data protection are essential, rely on WAFs to ensure that their applications are defended from common attacks like Cross-Site Scripting (XSS) and other forms of injection attacks. However, a growing challenge that WAFs must face is detecting and mitigating malicious bots. Bots, often used to automate fraudulent operations, can cause significant damage to web applications. This automated software can be employed for activities such as web scraping, illegally extracting information from websites, or click fraud, which manipulates traffic metrics by generating false clicks on advertisements.

The WAF acts as a barrier that separates web applications from potential external threats by analyzing each request and response to detect known malicious patterns or suspicious behaviour. Thanks to custom rules and advanced algorithms, the WAF can identify and block attack attempts before they compromise the integrity or availability of the applications.

This solution represents an essential component for many companies seeking to maintain high-security standards, protecting not only their data but also the brand's reputation. Indeed, a security breach can have devastating consequences, not only in terms of direct financial losses but also in terms of user trust and damage to the company's image. Implementing a WAF offers a first line of defense and allows companies to maintain strict control over who can access their applications and which requests are permitted.

In recent years, traditional threat detection methods based on static rules and signatures have proven increasingly ineffective. Attackers, using bots and advanced techniques, manage to modify their behaviour and vary their attack patterns to bypass these static defenses. Predefined rules are designed to detect known threats but are ineffective in recognizing new attack variants or anomalous behaviors that do not follow predefined models.

For this reason, adopting machine learning (ML) techniques has become a fundamental part of the development of modern WAFs. ML technologies allow security systems to learn from data continuously, improving their ability to detect zero-day attacks and suspicious behaviors that do not match predefined signatures. Unlike traditional methods, machine learning models can analyze vast amounts of data in real time to identify hidden patterns and correlations between seemingly unrelated activities, which allows emerging threats to be detected proactively.

Thanks to its dynamic nature, machine learning allows WAFs to continually adapt to new threats, improving over time through the analysis of constantly updated data. This flexible approach reduces false positives, a common problem in rule-based methods, increasing operational efficiency and the precision of threat detection.

However, integrating machine learning into WAFs also presents challenges, such as the need for large amounts of data to ensure accurate decision-making and manage the computational power required for processing. Data quality also becomes crucial: models trained on incomplete or non-representative data can lead to inaccurate results. For this reason, it is essential to integrate ML systems with constant validation and updating mechanisms.

Cloud computing has become an essential solution to address these challenges, The cloud offers the infrastructure necessary to handle growing volumes of data and to leverage on-demand computing power without the limitations imposed by local hardware resources. Thanks to the elastic scalability of the cloud, machine learning models can be trained on very large datasets and updated frequently, adapting to new threats in real time.

Furthermore, access to globally distributed data via cloud platforms enables WAFs to leverage information from various sources, improving the accuracy of the models. Cloud service providers, such as *Amazon Web Services (AWS)* or *Microsoft Azure*, offer specific tools for data management and processing, integrated with machine learning algorithms already optimized for their platforms. This simplifies the implementation of advanced models and reduces development time.

The cloud also facilitates the automation of updates: machine learning models can be trained and retrained automatically based on new data, ensuring that threat detection rules remain constantly updated. This continuous updating capability is critical in a context where cyberattacks are constantly evolving.

Another advantage is the global distribution of resources: cloud-based WAFs can distribute the workload evenly, ensuring optimal performance even during traffic spikes. This reduces the risk of slowdowns or service interruptions, thus increasing system reliability.

In this context, my thesis aims to contribute to developing a scalable cloud architecture that enables the integration and makes machine learning models for bot detection and the prevention of XSS attacks production-ready. In particular, the architecture uses Amazon MSK, ECS and SageMaker to provide an end-to-end solution capable of detecting anomalous behaviour and potential attacks promptly and accurately.

This architecture will be integrated into *Mithril* [5], a web application security service developed by *AizoOn Technology Consulting*, a global technology consulting company structured in different market areas and divisions, including Cybersecurity. Mithril is a web application security service designed to protect websites and web applications from various threats and attacks. It offers a comprehensive Web Application and API Protection (WAAP) service that extends beyond traditional

Web Application Firewalls (WAF) capabilities. Mithril integrates several cloud-based solutions, including WAF deployment, bot mitigation, DDoS protection, and API security, ensuring robust website security. The designed system leverages cloud processing platforms to manage real-time data flows and ensures that ML models are continuously updated and adaptable to new threats.

My work differentiates itself from existing contributions by focusing on the flexibility and scalability of the system, integrating machine learning capabilities into a cloud infrastructure that can be easily adapted to different business needs. Integrating a proprietary enterprise tool also improves the protection of web applications from evolving and sophisticated threats, ensuring proactive and continuous protection.

1.1 Problem Statement

The proposed solution aims to develop a scalable and flexible architecture based on AWS cloud, designed to make machine learning models production-ready, specifically developed for detecting requests generated by bots and attempts at XSS attacks. As mentioned before, these models will be integrated within the *WAAP Mithril* project.

The solution requires the following key features and requirements:

- **Batch traffic processing**: Batch processing is a way to analyze large volumes of data as a group by performing operations that accept a set of data as input, and produce a set of data as output. The system must be able to receive and manage traffic in batch mode, which is essential for a certain type of preprocessing.

- **Stream traffic processing**: Stream processing is a method of processing continuous streams of data and obtaining the results within a short period of time. The system must also be able to analyze traffic in stream mode.

- **Input data transformation**: The collected data must be processed and transformed into the formats required by the preprocessing steps.

- **Data storage for retraining**: Processed data must be stored so that it can be reused for future model retraining, ensuring a continuous improvement in detection capabilities.

- **Automated inference**: The system must automatically perform inference on the collected data, providing real-time or near-real-time predictions from machine learning models used in bot detection and XSS detection.

- **Results storage**: Predictions must be stored in a readable and consultable database for analysts, accompanied by useful information to interpret the results and make informed decisions.
- **Low-latency performance**: The architecture must guarantee low execution times while maintaining high accuracy levels.
- **Easy results readability for SOC analysts**: The results must be easily accessible and consultable by SOC (Security Operation Center) analysts, who will be able to examine the predictions and react promptly to detected threats.

1.2 Thesis Structure & Objectives

This thesis is structured to guide the reader through the various phases of the research, from problem analysis to the implemented solution. Each chapter addresses a key aspect of the project, offering a clear and progressive view of the technologies used, the machine learning models developed, and the proposed cloud architecture.

Chapter 2, dedicated to the **Background**, explores the theoretical foundations of machine learning applied to the two models developed. It also provides an overview of the relevant security techniques and the cloud context, along with a description of the proprietary tools in which the solution will be integrated.

Chapter 3 focuses on **Related Work**, analyzing similar solutions proposed by other competitors and reporting contributions from the scientific literature that have influenced the development of the solution presented in this thesis. A general overview of the components used in the architecture will be shown.

Chapters 4 and 5 describe in detail the cloud architecture to achieve the thesis objective, delving into improvements implemented in the preprocessing phases of the machine learning models. Additionally, the process of integrating the architecture within the WAAP Mithril project is illustrated, with particular attention to the interaction between the detection models and the proprietary security system.

Chapter 6 is dedicated to results, execution times and model performance are presented, accompanied by a detailed cost analysis of the AWS cloud infrastructure used for the proposed solution.

Finally, Chapter 7 is dedicated to **Conclusions** and **Future Works**; possible future improvements to the architecture are identified, suggesting avenues for optimizing the solution and for further technological developments.

This research developed an architecture entirely based on the AWS cloud, using several services, including Amazon MSK as a message broker, Amazon ECS for hosting the containers needed for preprocessing, and Amazon SageMaker for obtaining inferences. An average latency time of 40.56 seconds was obtained for bot detection, analyzing about 14,000 sessions per batch. For XSS detection, the average latency time was around 11.23 seconds, analyzing only the requests tagged as malicious by the WAF.

Chapter 2

Background

2.1 Internet Robots

In the modern digital landscape, **bots** - automated software programs - play a significant role in web traffic. Used for a wide range of activities, bots can be both beneficial and harmful. Good bots, like those employed by search engines to index websites or compare prices in e-commerce, are essential for the efficient functioning of the web. However, there are also malicious bots, responsible for harmful activities such as **web scraping**, **click fraud**, and **DDoS attacks**.

Malicious bots, which are becoming increasingly sophisticated, represent a growing threat to the security of web applications. The figure 2.1 shows the difference between good bots and bad bots. These automated software programs can disguise themselves as legitimate users [6], perfectly imitating human behaviors and bypassing traditional security measures. Using advanced techniques, such as headless browsers that execute JavaScript and simulate page rendering, bots can bypass detection systems like **CAPTCHA** and **IP blacklists**. This evolution makes traditional detection methods, which rely on static rules, increasingly ineffective.

In addition to their technical sophistication, malicious bots can compromise website performance by consuming bandwidth and resources, slowing response times, and increasing operational costs. These bots can also violate data security by carrying out **credential stuffing** attacks, where they automatically attempt to access user accounts using combinations of usernames and passwords. This seriously threatens the privacy and security of personal or financial information.

From a commercial standpoint, malicious bots can distort marketing metrics by generating fake clicks on online ads, causing **click fraud**. This practice skews traffic analysis and results in significant financial losses for companies that use **pay-per-click** advertising models.

A critical aspect of malicious bots is their ability to hide their origin by using

dynamic IP addresses or **proxy networks** [7], making detection even more challenging. This camouflage makes distinguishing bots from legitimate users difficult, as IP-based detection methods alone are no longer sufficient. Furthermore, advanced bots can replicate complex behaviors, such as mouse movements or seemingly random navigation paths, to avoid detection by traditional technologies.

Several advanced detection techniques have been developed to address the growing threat posed by bots. One common approach is **behavioral analysis** [8], which monitors user behavior (e.g., mouse movements, interactions, and response times) to detect suspicious patterns that differ from typical human behaviour. However, while useful, this technique is not always sufficient to identify the most sophisticated bots.

Another frequently used technique is **rate limiting** [9], which restricts the number of requests made by a single IP address or user within a given time frame. This measure is particularly useful for mitigating **DDoS attacks** and reducing the volume of bot-generated traffic.

The most recent evolution in bot detection is the application of **machine learning techniques**. These techniques allow for analysing large volumes of data and identifying hidden patterns that traditional methods fail to detect. Thanks to their ability to continuously learn, machine learning models improve over time, adapting to new bot behaviors and offering a proactive and scalable solution to the problem. Additionally, the use of **honeypots** - digital traps design to lure malicious bots - helps identify and study their behavior, further enhancing defense capabilities.

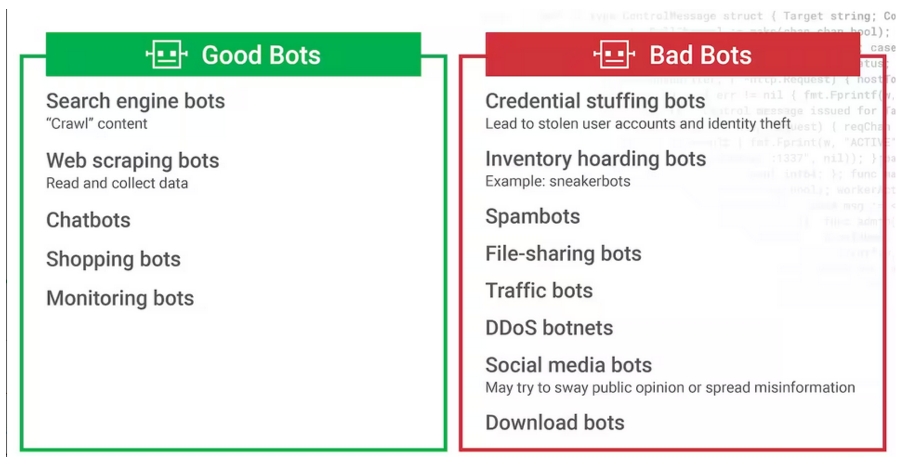


Figure 2.1: Good bots vs bad bots [10]

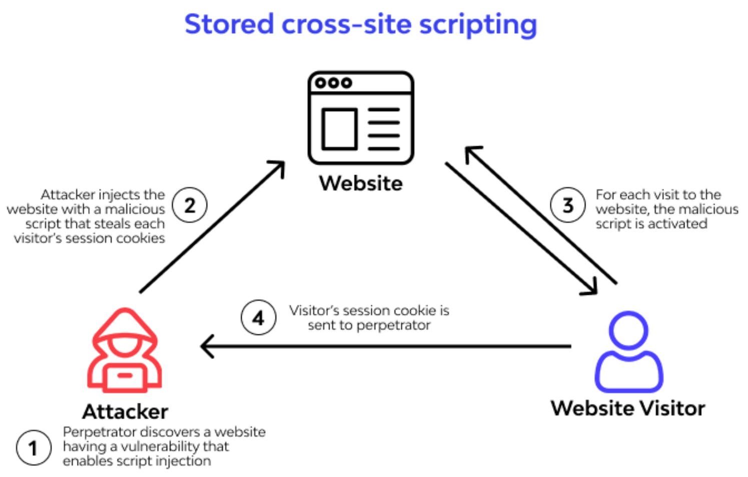

Stored cross-site scripting
Website
2 Attacker injects the website with a malicious script that steals each visitor's session cookies
3 For each visit to the website, the malicious script is activated
4 Visitor's session cookie is sent to perpetrator
Attacker
1 Perpetrator discovers a website having a vulnerability that enables script injection
Website Visitor

Figure 2.2: Stored XSS attacks example [12]

2.2 Cross-Site Scripting

Cross-Site Scripting (XSS) is one of modern web applications' most widespread and insidious vulnerabilities. It occurs when an attacker manages to inject malicious code into a web page, code that is then executed by an unsuspecting user's browser. This type of attack exploits the trust a user places in a legitimate website by manipulating how the content is presented or processed.

XSS attacks occur because of a lack of validation or sanitization of user-supplied input. When a web application accepts data from the user and embeds it into pages without adequate security measures, it creates an opportunity for attackers to insert arbitrary code. This code is executed within the security context of the Web site, giving the attacker access to sensitive information or the ability to manipulate the page behavior.

There are mainly three types of XSS attacks [11]: stored, reflected and DOM-based.

1. In the case of **stored XSS**, shown in Figure 2.2,the malicious script is permanently stored on the application server, such as in a database, comment field, or user profile. When other users access the page containing the malicious code, their browser executes the script, allowing the attacker to steal data or perform other malicious actions. A typical example is when an attacker posts a blog comment containing malicious JavaScript code; any user viewing that comment will unintentionally execute the script in their browser.

2. **Reflected XSS** occurs when the malicious script is reflected off the web

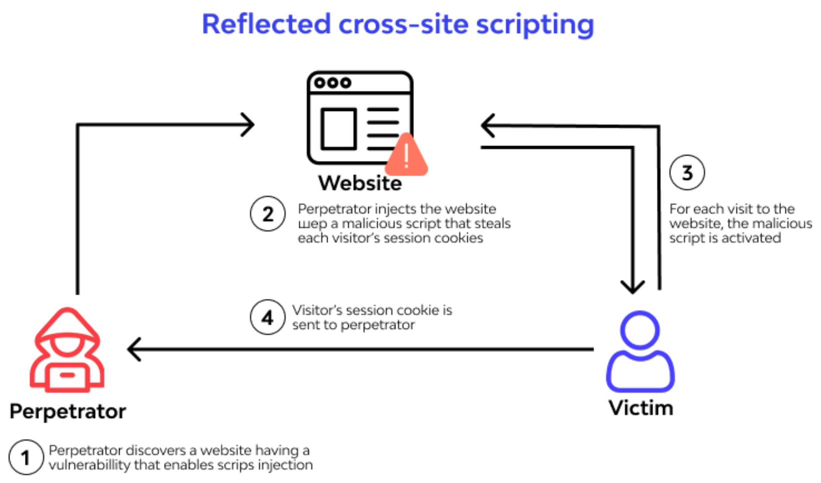

Reflected cross-site scripting
Website
2 Perpetrator injects the website шep a malicious script that steals each visitor's session cookies
3 For each visit to the website, the malicious script is activated
4 Visitor's session cookie is sent to perpetrator
Perpetrator
Victim
1 Perpetrator discovers a website having a vulnerabillity that enables scrips injection

Figure 2.3: Reflected XSS attacks example [12]

server, such as in a search page or, error messages, or any other response that includes some of the input sent to the server as part of the request. An example of this attack is shown on Figure 2.3. These attacks are delivered to victims via another route, such as in an e-mail message. When a user clicks on a malicious link, the injected code travels to the vulnerable website, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. A common example is a phishing link that leads to a legitimate site but with parameters manipulated to execute malicious code.

3. **DOM-based XSS** occurs entirely on the client side when the application's JavaScript code manipulates the Document Object Model (DOM) in an insecure manner using untrusted data, see Figure 2.4. This type of attack does not involve the server in the script injection phase, making it more difficult to detect with traditional server-side security measures. For example, a web application could use `document.write()` or directly modify the HTML of the page with data from the URL without sanitizing it, allowing arbitrary code to be executed.

XSS attacks can have significant consequences. Attackers can steal authentication cookies, allowing them to impersonate victim users and gain access to their accounts. They can dynamically manipulate the content displayed to the user, inserting false or misleading information. In addition, users can be redirected to phishing sites or pages that distribute malware. Attackers can read data displayed on the page, such as personal information or payment details. They can perform

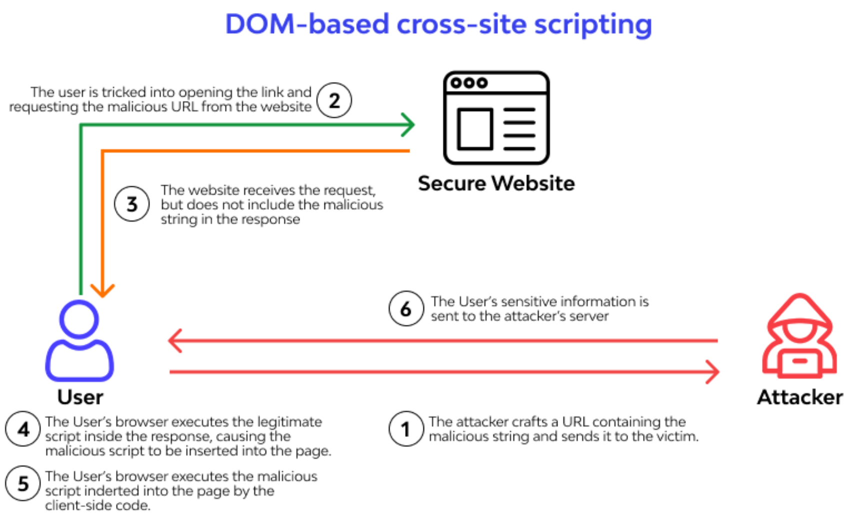

DOM-based cross-site scripting
2 The user is tricked into opening the link and requesting the malicious URL from the website
Secure Website
3 The website receives the request, but does not include the malicious string in the response
6 The User's sensitive information is sent to the attacker's server
User
Attacker
4 The User's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
1 The attacker crafts a URL containing the malicious string and sends it to the victim.
5 The User's browser executes the malicious script inderted into the page by the client-side code.

Figure 2.4: DOM-based XSS attacks example [12]

unauthorized operations on behalf of the users, such as sending messages or making transactions.

Over the years, numerous real-world cases of XSS attacks have had a significant impact. In 2010, an Australian teenager discovered an XSS vulnerability inside Twitter [13]. By posting special tweets, the hacker could get other Twitter users to execute arbitrary code when they moved over the messages. Even if the attack was harmless, this opened the door to exploiting the vulnerability widely. In 2021, a DOM-based XSS was hidden inside PayPal's code [14], which allowed attackers to execute arbitrary code in order to steal data or take control of the device.

Preventing XSS attacks presents several challenges. Input validation is complex; with the increasing complexity of Web applications, ensuring that all input is properly validated and sanitized is an ongoing and challenging task. Using third-party frameworks and libraries can introduce vulnerabilities that are not immediately obvious, as one does not always have complete control over the external code. In addition, attackers are continually developing new methods to bypass existing security measures, evolving their techniques and exploiting new vulnerabilities.

To mitigate XSS attacks [15], it is critical to apply strict validation measures on all data provided by users. Using whitelists to define which characters or strings are acceptable can help prevent the insertion of malicious code. Before inserting dynamic data into web pages, appropriate encoding should be applied based on the context to neutralize potential malicious code. Implementing a Content Security Policy (CSP) can limit the sources from which the browser can load resources and execute scripts, preventing the execution of unauthorized code. It is advisable to choose frameworks and libraries that offer built-in protections against XSS, such as automatic data escaping.

With the increasing adoption of single-page applications (SPAs) and the intensive use of client-side JavaScript, the attack surface for XSS is expanding. Emerging technologies such as WebAssembly [16] and integrating advanced APIs require special attention to security. In addition, the expansion of the Internet of Things (IoT) and the development of mobile applications introduce new contexts in which XSS can occur, necessitating the need for continuous evolution of defense strategies.

2.3 Machine Learning models

2.3.1 Machine Learning fundamentals

The use of machine learning techniques in the context of cybersecurity offers solutions that are dynamic and adaptable to the ever-changing scenarios in which it operates.

Machine Learning is a branch of Artificial Intelligence that aims to learn from the data it receives as input and make decisions without being explicitly programmed. Machine Learning algorithms use mathematical models to detect patterns in the data.

Based on the task these algorithms have to perform, they are divided into two categories: supervised and unsupervised machine learning. Supervised machine learning algorithms learn from a labelled dataset, that is, a dataset in which data are associated with correct answers. In this way, the algorithm will later be able to identify other data, with the labels specified in the training phase. A common use of supervised learning is where past data are analyzed to predict future events based on the probability that they are similar to previous ones. Unsupervised learning involves unlabeled data, thus leaving the algorithm to search for patterns that relate to the input data. These types of algorithms are commonly used for transactional data and to identify anomalies. Hybrid approaches also exist: this is the case with semi-supervised algorithms. Small amounts of labelled data are combined, forming the base learner. Unlabeled data, which are easier and less expensive to obtain, are added to these. Finally, there are reinforcement learning models, based on the concept of rewards and penalties: with each action performed by the agent, the environment provides a reward or penalty. The agent's goal is to maximize the sum of the rewards.

Machine Learning has found fertile ground in many different areas, and its ability to analyze large amounts of data to make predictions has made it a very versatile tool. In the field of healthcare, for example, ML is used to improve disease diagnosis by analyzing medical images to detect anomalies. Still, it is used in the field of finance to analyze millions of transactions in real-time with the goal of detecting potential fraud.

Using Machine Learning in cybersecurity is increasingly relevant to protect computer systems from sophisticated attacks: they can analyze large amounts of data to identify suspicious behavior, such as lateral movements within a network or attempted DDoS attacks. They are particularly effective at detecting bot-generated traffic, distinguishing legitimate and malicious traffic through analysis of user behavior. It can also analyze files and programs for typical characteristics of malware, allowing it to detect variants that escape classic signature checks. In this, Deep Learning is best suited for its ability to analyze execution patterns or binary signatures, without the need for predefined signatures. Finally, machine learning can identify anomalous patterns, such as exploit attempts or the insertion of malicious code within a Web page through the analysis of Web requests.

2.3.2 Bot detection model [17]

In the context of analyzing web client interactions with a website, it is essential to be able to distinguish between human and bot behaviour. Client interactions with a website are represented as sessions, defined as a sequence of HTTP requests from a client during a single visit. Since HTTP is stateless, session information is not stored in access logs, making heuristics necessary to identify sessions.

The commonly adopted methodology for identifying sessions involves grouping HTTP requests based on the same IP address and user agent, using a timeout-based approach to split the clickstream into multiple sessions. However, this technique has clear limitations, particularly in determining the appropriate time threshold for separate sessions. Typically, a threshold of 30 minutes is adopted, but this is not sufficient for longer sessions or continuous browsing behavior. For this reason, a dynamic threshold has been introduced: if the number of requests is below a predefined limit, the 30-minute threshold is maintained. Once this limit is exceeded, the threshold is extended to 60 minutes, allowing for a more accurate handling of longer sessions.

Before applying machine learning models, data must be preprocessed to be ready for analysis. Preprocessing involves several steps:

Data Normalization: Since models like K-Means are sensitive to feature scale, all numerical features are standardized using scikit-learn's `StandardScaler`. This process makes the data comparable, reducing the risk of features with very different values (such as total number of requests against the average time between requests), that influences the model.

Text Processing (User Agent): The User Agent, a string describing the device and browser used to make the requests, is transformed into a numeric representation using the Bag of Words (BoW) technique through `CountVectorizer` class. This allows this textual feature to be included in the model, representing it as a numeric vector.

The features extracted from HTTP requests represent different aspects of web sessions and are essential for detecting suspicious behavior. The main features used are listed below:

- **User Agent**: The User Agent is represented in numerical form using the Bag Of Words, allowing us to analyze information about the device and browser used during the session.
- **Total number of requests**: Indicates how many requests were made during a session. A high number of requests can be indicative of bot activity, such as data scraping or DDoS attacks,
- **Total volume of data transmitted**: Measures the amount of data sent from the server to the client. Malicious bots can generate a disproportionate volume of data compared to legitimate users.
- **Average time between requests**: Indicate the average interval of time between two consecutive requests. Bots tend to make requests much faster than human users.
- **Standard deviation of time between requests**: Measures the variability in the time between requests. Low variability suggests that requests were made automatically at regular intervals, which is typical for bots.
- **Nightly request rate**: Calculates the percentage of requests made between 2:00 AM and 6:00 AM. A high percentage of nightly traffic may reveal automated activity.
- **HTTP error rate (codes >= 400)**: HTTP errors can be generated in greater quantities by bots, which often send invalid requests or fail to complete operations correctly.
- **GET, POST, and HEAD request rate**: Analyzes the use of different HTTP methods during a session. Bots tend to generate abnormal percentages of GET (to extract information) and POST (to send data) requests.
- **Session width and depth**: Width measures how many distinct pages were requested during a session, while depth indicates how deeply a user explores site sections. Bots tend to explore in a more predictable and less varied manner than human users.
- **Null Referrer Request Rate**: Bots tend to make direct requests to servers without going through other pages, resulting in a high percentage of null referrer requests.

Cluster algorithms were used to analyze web sessions and identify anomalous behaviors that could indicate the presence of bots. Specifically, **DBSCAN** and **K-Means** algorithms were implemented and compared. After several experiments, the K-Means algorithm proved to perform significantly better than DBSCAN and was therefore selected.

The Elbow Method and silhouette score analysis were used to determine the optimal number of clusters. The Elbow Method identified the point at which adding additional clusters no longer significantly improved the partition quality, while the silhouette score measured the cohesion and separation between clusters. After applying both techniques, an optimal number of 12 clusters was chosen.

2.3.3 XSS detection model [18]

XSS attacks often exploit the way web applications handle user input in HTTP requests. Malicious payloads are inserted into the request parameters, and then processed by the application. A model has been developed that can identify the presence of payloads with XSS injected into any request body. Obviously, the data cannot be processed in a raw way but must be preprocessed to allow the extraction of features.

DECODING. The initial payload is subjected to a decoding process managed by a `Decoder` block that supports different decoding methods. This block detects the presence of one of the supported decoding methods. When an encoding type is detected, the block decodes the payload and reprocesses the decoded text. The recursion ends when the text remains unchanged, i.e., no other encodings are present.

GENERALIZATION. Next, the payload goes through a `Generalization` block, which takes the taker of removing excess whitespaces, converting all text to lowercase, removing HTML comments and deleting punctuation and repeated characters.

TOKENIZATION. After the payload has passed under the `Generalization` block, it proceeds to the `Tokenization` block where further refinements take place. Here, non-visible characters are removed, and the payload is divided into a list of tokens. This list is fundamental because the analytical techniques and machine learning models depend on it to extract significant passages and make predictions with increased accuracy. The Figure 2.5 shows the model workflow.

Finally, we proceed with the extraction of the features. In the case of XSS detection, two models were developed: one specific to analyze the JavaScript code,

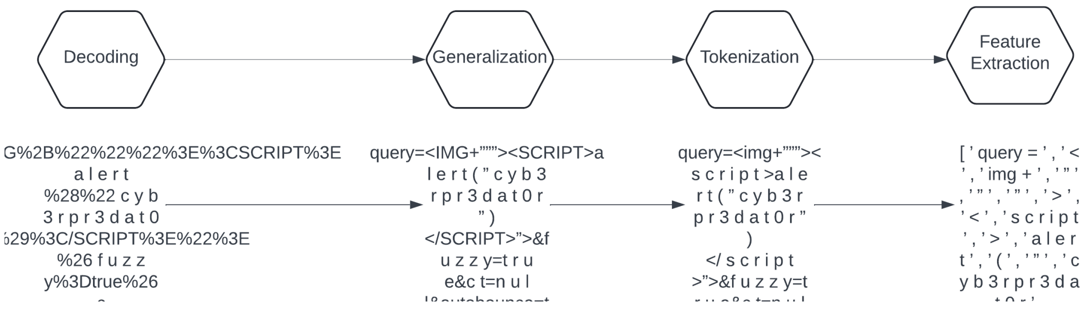


Figure 2.5: XSS detection model workflow

and the other optimized to analyze the HTML code. For this reason, there are two sets of features.

The list of features for the **JavaScript** model is as follows:

- **PayloadLength**: Length of the payload, in logarithmic scale.
- **Document**: Represents the call to the document object representing the web page.
- **jsObject**: Recognizes JavaScript object literals, for example `key: value`.
- **jsAttr**: Recognizes JavaScript attribute references, for example `.attribute`.
- **jsMethods**: Identifies calls to JavaScript methods, for example `.method(`.
- **cookie**: Identifies calls to cookie.
- **structural**: Number of punctuation characters used.
- **structComb**: Counts combinations of punctuations used.
- **sensitiveWords**: Contains a list of words commonly found in malicious payloads.
- **closeB**: Counts the number of closing parentheses.
- **openB**: Counts the number of opening parentheses.

Regarding the features of the **HTML** model, there are some slight differences:

- **PayloadLength**: Length of the payload, in logarithmic scale.
- **HtmlTag**: Recognizes HTML tags.
- **HtmlAttr**: Recognizes HTML attributes.

- **ExtLink**: Search for external links called by `src` or `href` attributes.
- **ExtJs**: Search for scripts inside the payload.
- **sensitiveWords**: Contains a list of words commonly found in malicious payloads.
- **closeB**: Counts the number of closing parentheses.
- **openB**: Counts the number of opening parentheses.

Finally, the features are normalized, so that each feature contributes equally to the analysis and modeling.

For this task, the chosen model is an **OCSVM (One-Class Support Vector Machine)**. This model is particularly suitable for anomaly detection scenarios like the current one, where the goal is to distinguish regular behavior from anomalous one (attack attempts). OCSVM is particularly useful when you have a dataset where the classes are strongly unbalanced (the number of entries labelled LEGIT is much more than those labelled anomalous).

The main goal of OCSVM is to find a hyperplane that maximizes the margin. The margin is the distance between the decision hyperplane and the closest positive class data point. Once the hyperplane is found, the model can be used to classify new data points. The distance between a data point and the decision hyperplane is calculated, which is compared to the margin. Data points within the margin are considered anomalous, while those outside the margin are considered normal (legit).

2.4 Web Application Firewall

A Web Application Firewall is a sub-category of firewalls that filters, monitors, and blocks HTTP traffic to and from a website. By inspecting the traffic, WAFs can protect business-critical applications and web servers from threats such as zero-day attacks, distributed denial-of-service (DDoS), SQL injection, and cross-site scripting. Protection occurs by applying a set of rules to an HTTP conversation. These rules cover the most common attacks.

A WAF can be either a virtual or physical device, but also in the form of a plugin. It is implemented in front of web applications and analyzes bidirectional HTTP traffic, detecting and blocking any malicious elements.

One of the main challenges of traditional web application firewall technology is that security teams must constantly analyze and optimize a set of rules to reflect changes in applications, emerging threats, and updates to WAF solutions.

When a firewall is not configured correctly, it can issue an increasing number of warnings and alerts. Suffering from alarm fatigue, security teams may have difficulty by distinguishing false positives from real attacks. Fearing that their inability to effectively configure rules might disrupt business operations, security teams often remove protections and accept a weakened security posture.

There are 3 main types of web application firewalls: WAF appliances, host-based WAFs, and cloud WAFs.

- **WAF appliances**, typically hardware-based, can be installed locally using dedicated resources, and can be placed as close as possible to the target application to reduce latency. Most hardware-based WAFs allow for copying rules and settings between devices to support large-scale deployments in enterprise networks. The downside is that they require a large initial investment in addition to ongoing maintenance costs.

- **Host-based** WAFs can be fully integrated within the code of the target web application. The advantages of this deployment model include much lower costs and better customization. However, they are more complex to implement, requiring the installation of specific libraries on the application server and relying on the web server's resources to function effectively.

- **Cloud-based** WAFs are a cost-effective option that provides a turnkey WAF solution, with no upfront investment and quick deployment. They are typically subscription solutions and only require a DNS configuration to be applied to start working. They have access to constant threat intelligence and can also offer managed services to help you define security rules and respond to attacks as they happen.

WAFs can operate according to whitelists, i.e. allowing only recognized traffic, or according to blacklists, i.e. blocking traffic that matches any attack patterns or security rules.

The problem with rule-based WAFs is that they require very high maintenance. organizations must meticulously define rules to match their application patterns, which may change over time. This also makes it more difficult to deal with evolving threats: new threats may require new rules.

The figure 2.6 shows a general schema of a WAF Workflow.

2.5 Cloud Computing & Amazon Web Services

2.5.1 Cloud Computing

NIST [20] provides an official definition of cloud computing, highlighting its main characteristics and service models. It defines five key features: **on-demand self-service**, which allows users to manage resources themselves without interaction

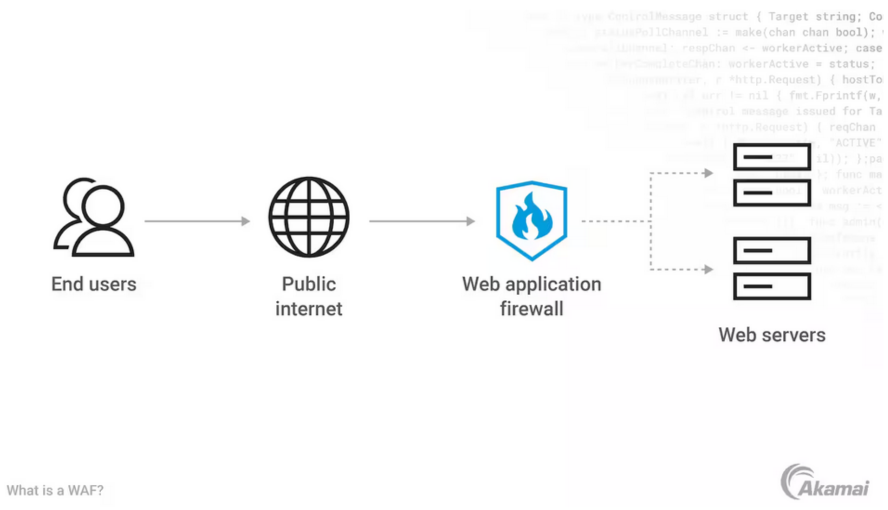


Figure 2.6: How the WAF works [19]

with the provider; **large-scale network access**, with resources accessible via Internet through standard clients; **shared resource pooling**, which involves dynamic sharing of physical resources among multiple through a multi-tenant model; **rapid elasticity**, which allows resources to be expanded or reduced flexibly as needed; and **measurable service**, with automatic monitoring of resources to ensure a payment model based on actual usage.

NIST outlines three main service models. **IaaS** (Infrastructure as a Service) provides basic resources such as virtual servers, storage and networking. The **PaaS** (Platform as a Service) provides a platform for application development and deployment while fully managing the underlying infrastructure. Finally, **SaaS** (Software as a Service) offers ready-to-use applications accessible via the Internet, with infrastructure management delegated to the vendor. See figure 2.7 for a recap schema of the service models.

Four deployment models are also defined: the **public cloud**, where infrastructure is commercially provided to multiple users; the **private cloud**, intended exclusively for a single organization; the **hybrid cloud**, which combines public and private cloud resources; and the **community cloud**, which shares infrastructure among organizations with similar goals.

2.5.2 Amazon Web Services

Amazon Web Services (AWS) was launched by Amazon in 2006 as an on-demand cloud services platform for businesses and individuals. The idea behind AWS was to offer on-demand IT infrastructure, eliminating the need for companies to invest in expensive hardware and in-house server management. Initially, AWS

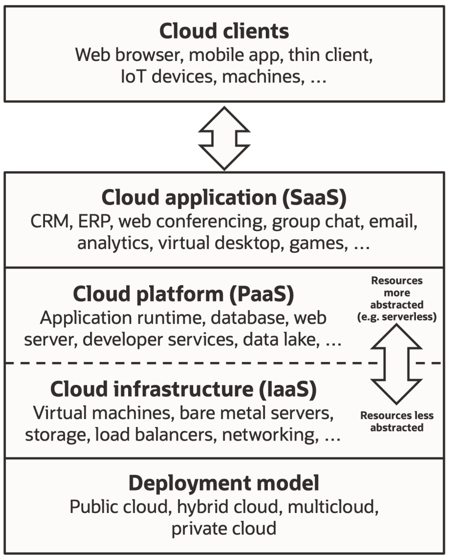

Cloud clients
Web browser, mobile app, thin client, IoT devices, machines, ...
Cloud application (SaaS)
CRM, ERP, web conferencing, group chat, email, analytics, virtual desktop, games, ...
Cloud platform (PaaS)
Application runtime, database, web server, developer services, data lake, ...
Resources more abstracted (e.g. serverless)
Cloud infrastructure (IaaS)
Virtual machines, bare metal servers, storage, load balancers, networking, ...
Resources less abstracted
Deployment model
Public cloud, hybrid cloud, multicloud, private cloud

Figure 2.7: Cloud computing service models [21]

offered core services such as **Amazon S3** for object storage and **Amazon EC2** for scalable computing power. These services allowed companies to quickly launch applications and manage variable workloads without large upfront investments. Over the year, AWS has greatly expanded its portfolio, introducing a wide range of services, including managed databases (such as **Amazon RDS**), data analytics tools, artificial intelligence, Internet of Things, Advanced Security, and Developer solutions.

AWS has consolidated its position as a leader in the cloud computing industry through its comprehensive offerings and continuous innovations. In 2023, AWS was recognized by Gartner as one of the strategic cloud platforms of choice, confirming its important to business and institutions globally. The platform offers scalability, flexibility and a wide range of services. Gartner's latest Magic Quadrant [22], shown in figure 2.8, highlights AWS's leadership due to its ability to execute and strategic vision completeness. AWS has maintained leadership for over a decade because of its ability to adapt to customer needs and constantly innovate with new services. In addition, AWS continues to demonstrate the importance of security and compliance in the cloud, offering robust solutions that enable organizations to comply with global security and privacy regulations. This focus on severity has made AWS a primary choice for regulated industries such as finance and government.

AWS's global architecture [23] consists of three main elements: regions, availability zones, and edge locations, design to provide high availability, resiliency, and low latency. **Regions** are geographic areas that contain two or more **Availability**

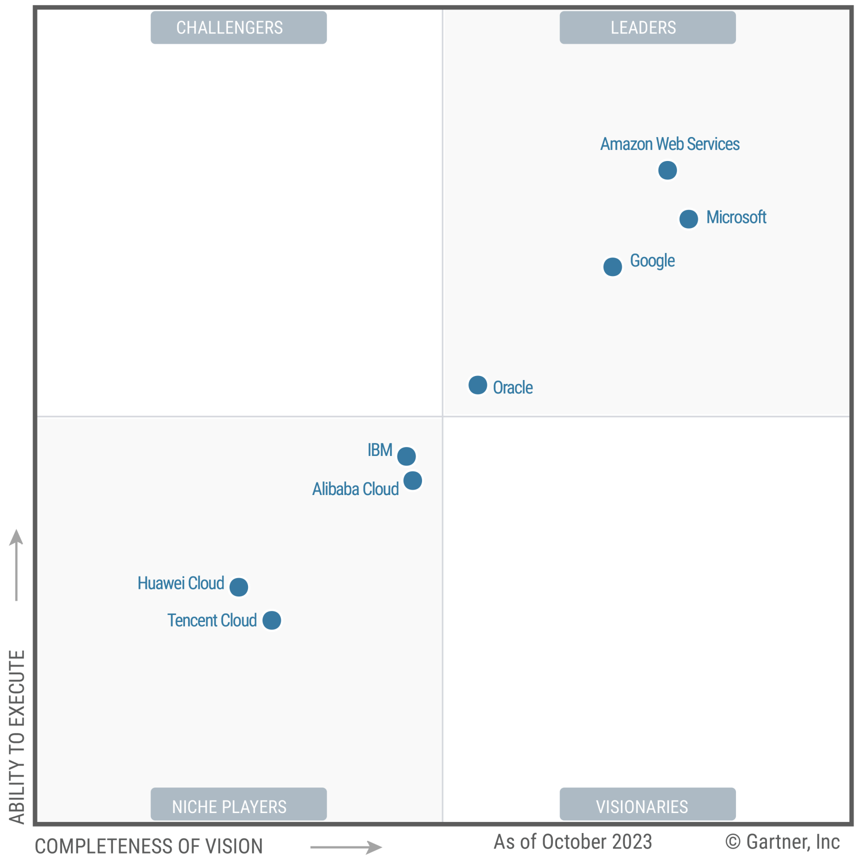


Figure 2.8: 2023 Gartner Magic Quadrant [22]

Zones, each operating as an isolated data center. Each Availability Zone (AZ) is connected via low-latency networks but is physically separated to provide redundancy in case of failure. This isolation of availability zones allows highly reliable and fault-tolerant applications to be built. Currently, AWS offers more than 30 regions worldwide, with more than 100 availability zones distributed among them. **Edge locations** are points of presence located near end users and play a crucial role in improving the performance of services such as **Amazon CloudFront**, a Content Delivery Network (CDN) that reduces latency, enabling faster access to data. AWS uses more than 400 edge locations globally.

Geographic resilience is critical to ensure that applications remain operational even in the event of failure or natural disaster in a specific region. Distribution across multiple availability zones and regions provides critical redundancy that protects applications from unforeseen events. AWS enables real-time disaster recovery, allowing companies to replicate data and operations across multiple regions, thereby reducing the risk of downtime. Latency and performance are equally important characteristics, especially for applications that require fast response times. AWS's global network, with its regions and edge locations, minimizes the physical distance between end users and servers, ensuring ultra-low response times. This geographic distribution reduces data transmission bottlenecks and ensures that applications remain fast and responsive, regardless of user location. The figure 2.9 shows how spread is the AWS Architecture.

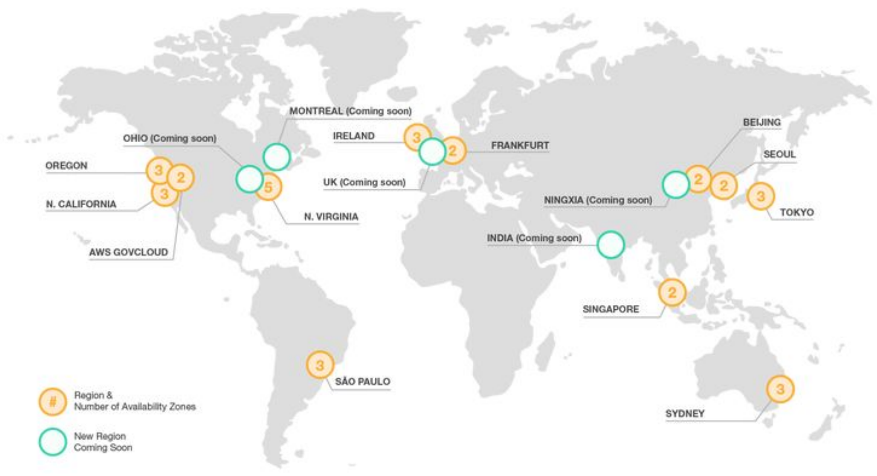


Figure 2.9: AWS Global Architecture [24]

2.6 Architecture Overview

In this section, we will discuss about each component of the developed architecture.

2.6.1 Apache Kafka

Apache Kafka [25] is a distributed platform for acquiring, storing and processing real-time data streams. It is designed to handle large volumes of data generated continuously from thousands of sources, with the capabilities to process that data sequentially and incrementally. Data streams, also called streams, are produced by devices or applications that simultaneously send data records, and Kafka is optimized to handle these continuous influxes.

Kafka provides three key features:

1. Publishing and subscribing to streams of events (records) in real-time.

2. Durable and reliable storage of these events in the order in which they are generated
3. Processing of these streams as they occur or retrospectively

These capabilities make it ideal for building stream-matching data pipelines and creating real-time streaming applications. The combination of messaging, storage and processing allows the management of historical data and real-time streams with a single scalable solution.

Kafka is a distributed system consisting of servers and client communicating through a high-performance network protocol (TCP). It can run on various environments, such as bare-metal, virtual machines and containers, in both on-premise and cloud environments.

- **Servers (Brokers)**: Kafka servers form a cluster that can span multiple data centers or cloud regions. Some of these servers act as brokers, storing and replicating data, while others run Kafka Connect for continuous integration of data to or from other systems, such as relational databases or other Kafka clusters. This distributed architecture ensures high availability and fault tolerance, because if one of the servers fails, the others take over the workload.
- **Clients**: Kafka's clients enable the creation of distributed applications that read, write and process event streams in a parallel, fault-tolerant manner. Kafka provides clients with various programming languages, including Java, Scala, Python, Go and C/C++, as well as a REST API. These clients enable applications to process real-time data efficiently, even during network failures or hardware problems.

In Kafka, data is treated as events, which record the fact that "something has happened". Each event has a key, value and timestamp, as well as any metadata. Producers are applications that publish events to Kafka, while consumers are those that subscribe to and read those events.

Events are organized into topics, which can be thought of as folders in which data are stored. A topic in Kafka can have multiple producers and consumers, and events are not deleted immediately after being read, unlike many traditional queues. Data in a topic can be read as often as necessary, and Kafka allows you to configure how long events should be kept.

Topics are divided into partitions, which allow data to be distributed across different Kafka brokers, improving scalability. Events with the same key are written to the same partition, ensuring that consumers read events in the same order as they were written.

To ensure reliability, Kafka supports replication of data between multiple brokers and even between different geographic regions or data centers. Each partition can have multiple replicas, which means that even in the event of a failure, the data remains accessible. For example, in a production environment, a typical replica has a factor of 3, which means there are 3 copies of the data distributed across different brokers, ensuring continuity of service.

Kafka combines two messaging models:

1. Queuing, which allows traffic to be distributed among multiple consumers, improving scalability;
2. Publish-Subscribe, which allows multiple consumers to read the same message.

This hybrid model, based on a partitioned log, allows Kafka to combine the benefits of both systems while providing scalability and reproducibility.

Kafka proves particularly useful in contexts where large volumes of data need to be managed in real-time. It is ideal for creating data pipelines that need to move and process data reliably and scalably. Some typical scenarios include:

- Integration of heterogeneous systems: Kafka acts as an intermediary, facilitating the exchange of data between different applications;
- Real-time processing: In applications such as network monitoring, Kafka enables analysis and processing of data as it is generated;
- Streaming pipelines: Kafka is used to build streaming pipelines that collect, process and distribute data to different systems, making it ideal for microservice-based architectures.

Kafka thus offers a complete solution for managing data in motion, combining publishing, archiving and event processing in a highly scalable and distributed platform. Its flexible architecture can handle a wide range of workloads, ensuring business continuity even in high traffic or server failure scenarios.

2.6.2 Amazon MSK

Amazon Managed Steaming for Apache Kafka (Amazon MSK) [26] is a fully managed service that enables enterprises to build and launch Apache Kafka-based applications for real-time data stream processing. Amazon MSK simplifies the entire Kafka cluster lifecycle, managing both control-plane operations (such as cluster creation, update and delete) and data-plane operations (such as streaming data production and consumption).

Amazon MSK runs open-source versions of Apache Kafka, ensuring compatibility with existing applications, tools, and plugins without the need for application code changes. This offers broad flexibility and allows companies to leverage the Kafka ecosystem without having to directly manage the underlying infrastructure.

The Amazon MSK architecture is based on a number of key components that ensure scalability, resilience, and service reliability:

1. **Broker nodes**: When an MSK cluster is created, you can specify how many broker nodes you want for each availability zone. Brokers are responsible for managing streaming data traffic and event persistence. Each broker in Amazon MSK can be configured to operate in high availability, with at least one broker per availability zone.

2. **Zookeeper nodes**: Amazon MSK also creates and manages Zookeeper nodes, which act as distributed coordinators. Apache Zookeeper synchronises brokers and maintains data consistency within the cluster, ensuring reliable and redundant coordination.

3. **Producers and consumers**: Amazon MSK users can leverage Apache Kafka data-plane operations to create topics, publish data, and consume events in real-time. Amazon MSK enables interaction with the cluster via Kafka's native API, providing a seamless experience for producers and consumers.

4. **Cluster operations**: Amazon MSK allows Kafka clusters to be managed through the AWS Management Console, AWS Command Line Interface (CLI), or SDK APIs. These tools provide a full range of operations for cluster control and configuration, allowing you to easily scale and monitor your Kafka infrastructure.

One of the main advantages of Amazon MSK is its ability to automatically handle failures and ensure system reliability. Amazon MSK is designed to recognize and handle common failure scenarios, automatically initiating recovery procedures to maintain cluster uptime. When an error occurs at the broker level, Amazon MSK can mitigate the impact of the error by replacing the existing one, while reusing the previous broker's storage to reduce the amount of data to replicate.

These automated fault management minimizes the impact on system availability by limiting it to the time required to detect and correct the problem. After recovery, producer and consumer applications continue to communicate using the same IP addresses as previous brokers, ensuring continuity of operations without requiring significant manual intervention.

2.6.3 Amazon ECS

Amazon Elastic Container Service (Amazon ECS) [27] is a fully managed container orchestration service that enables easy deployment, management and scaling of containerized applications.

As a fully managed service, Amazon ECS incorporates AWS operational and configuration best practices. It is integrated with both AWS and third-party tools such as Amazon Elastic Container Registry and Docker. This integration allows to focus more on building applications rather than managing the environment. Is it possible to run and scale container workloads across AWS regions, both in the cloud and on-premises, without the complexity of managing the control plane.

Amazon ECS consists of three main levels:

1. **Capacity**: the infrastructure on which containers run;
2. **Controller**: deploys and manages applications running in containers;
3. **Provisioning**: the tools used to interface with the scheduler in order to deploy and manage applications and containers.

The Amazon ECS *Capacity* represents the infrastructure where containers run, offering several options:

- **Amazon EC2 instances in the AWS cloud**: is it possible to select the type and the number of instances, managing sizing directly.
- **Serverless with AWS Fargate in the AWS cloud**: AWS Fargate is a serverless computation engine that eliminates the need to manage servers, sizing or container workloads.
- **On-premises virtual machines or physical servers**: Amazon ECS supports registration of external istances, such as on-premises servers or virtual machines, within the ECS cluster.

The Amazon ECS scheduler is the software responsible for managing applications.

The figure 2.10 illustrates the application life cycle and interaction with Amazon ECS components: It is critical to design applications so that they can run in containers. A container is a standardized unit of software development that includes everything needed to run the applications, such as code, runtime, system tools, and libraries. Containers are created from a read-only template called *image*. Images are generally built using a Dockerfile, a simple text file that contains instructions for creating the container. Once built, images are saved in a registry, such as Amazon ECR [28], from which they can be downloaded.

After creating and storing the image, a *Task definition* is established. The Task definition consists of a schema of the applications, represented by a file in JSON format that describes the parameters and one or more containers that make up the application. For example, it may specify the image to be used, operating systems parameters, which containers to implement, which ports to open for the application, and which data volumes to associate with the containers.

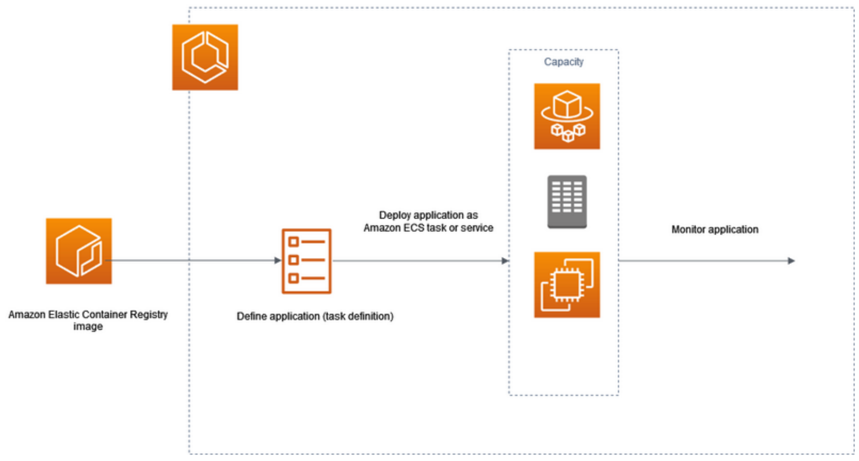

Capacity
Deploy application as Amazon ECS task or service
Monitor application
Amazon Elastic Container Registry image
Define application (task definition)

Figure 2.10: Amazon ECS lifecycle [27]

Once the task is defined, it is deployed as a service or task on the cluster. A cluster is a logical grouping of tasks or services running on the capacity infrastructure registered in the cluster.

A task is an instance of a task definition within the cluster. It is possible to run a task independently or as part of a service. Using an Amazon ECS service, it is possible to execute and maintain the desired number of tasks simultaneously in an Amazon ECS cluster. Suppose one of the tasks fails or stops for any reason. In that case, the Amazon ECS service scheduler starts another instance based on the task definition, ensuring that the desired number of tasks are maintained in the service.

The container agent runs on each instance within the cluster. This agent sends information about running activities and container resource utilization to Amazon ECS. It also starts and stops activities whenever it receives a request from Amazon ECS.

2.6.4 Amazon SageMaker

Amazon SageMaker [29] is a fully managed machine learning service. It provides a user interface for running machine learning-related workflows, making its tools available in multiple integrated development environments (IDEs). Amazon SageMakers offers a number of tools and services that enable one to follow the entire lifecycle of a machine learning model.

The figure 2.11 shows the lifecycle of a machine learning model.

For what concerns the first phase, that is, the generation of training data, there

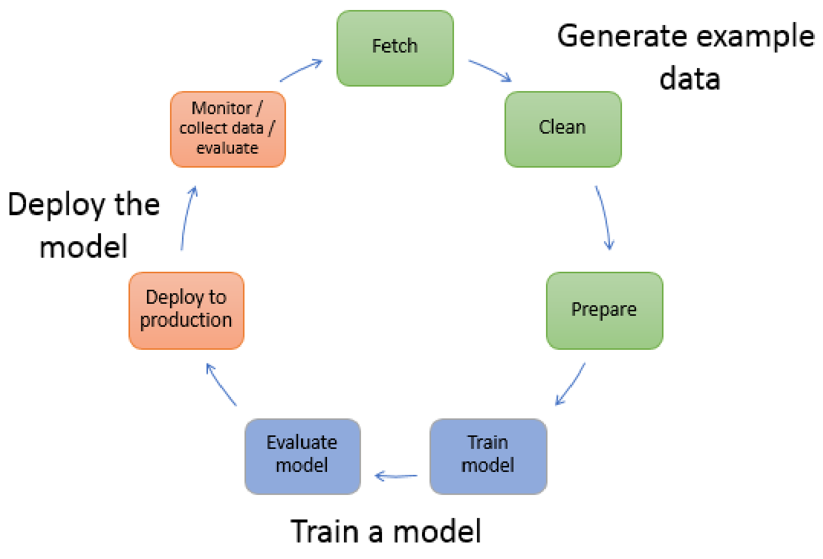


Figure 2.11: Machine learning model lifecycle [29]

can be several usage scenarios:

1. For those who prefer a graphical interface, SageMaker offers the **Data Wrangler** service within **Amazon SageMaker Canvas**. SageMaker Canvas is a code-free development environment design that allows non-expert users to create, use and train machine learning models. Data Wrangler, on the other hand, is responsible for preparing the data and manipulating the datasets before using them in Canvas.

2. Using **Amazon SageMaker Studio**, it is possible to connect to an SQL database such as **Amazon RDS** or a data warehouse such as **Amazon RedShift** to run SQL queries and have the results directly in a Jupyter Notebook, so the data can be manipulated using Python and Pandas.

3. For those who need to work with map-reduce programming paradigms or use frameworks such as Apache Spark, Amazon SageMaker Studio is integrated with **Amazon EMR** service, a fully managed cluster that enables distributed workloads.

Regarding the training phase, Amazon SageMaker offers several pre-trained algorithms and models that are ready to use, making available both supervised and unsupervised algorithms as well as algorithms for textual analysis of documents or image processing.

Finally, the last step in the life cycle concerns the deployment of the model. The microservice that takes care of this is called **AWS SageMaker Inference**. Even in the case of simple inference, we fall into three use cases:

1. **Model deployment in a low-code environment**: In this case, it is possible to use the **Amazon SageMaker JumpStart** service through the SageMaker Studio interface, without the need for complex configuration.

2. **Using code to deploy machine learning models with greater flexibility**: The `ModelBuilder` class of the SageMaker Python SDK can be used to do this, allowing granular control over various settings, such as instance types, network isolation and resource allocation.

3. **Machine learning models on a large scale**: To do so, use the AWS SDK for Python (Boto3 [30]) and **AWS CloudFormation**, an infrastructure as Code (IaC) tool for automating resource management.

The present research falls into a hybrid use-case because the Python Boto3 SDK will be used to deploy the model without the usage of the ModelBuilder class as it involves the use of pre-trained SageMaker models, but an ad-hoc *Model* type object will be created for both bot detection and XSS detection.

SageMaker offers several inference options:

1. **Real-time inference**: Ideal for interactive workloads, with low latency requirements.

2. **Serverless inference**: Useful for those who do not have to manage the infrastructure underlying the model. This option is ideal for those intermittent workloads where there are idle periods and can tolerate cold starts.

3. **Asynchronous inference**: This option puts incoming requests into a queue and processed asynchronously. This option is ideal for those who need to handle large payloads.

The schema in figure 2.12 shows the different deployment options.

Finally, AWS SageMaker offers the ability to monitor and optimize the costs of the tools used:

- With **SageMaker Neo**, is it possible to optimize model code so that it performs better, minimizing computation costs.

- Using **Autoscaling**, it is possible to adjust the resources allocated to endpoints in a coherent way respect the amount of incoming traffic, optimizing costs.

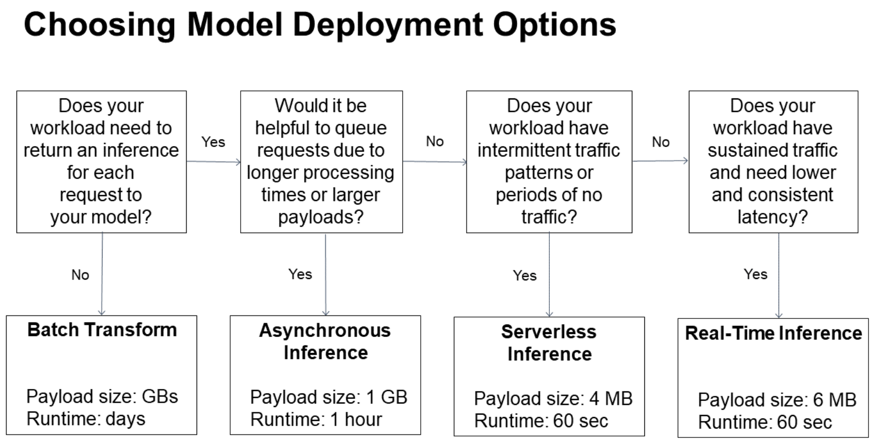

Choosing Model Deployment Options
Does your workload need to return an inference for each request to your model?
Yes
Would it be helpful to queue requests due to longer processing times or larger payloads?
No
Does your workload have intermittent traffic patterns or periods of no traffic?
No
Does your workload have sustained traffic and need lower and consistent latency?
No
Yes
Yes
Yes
Batch Transform
Payload size: GBs
Runtime: days
Asynchronous Inference
Payload size: 1 GB
Runtime: 1 hour
Serverless Inference
Payload size: 4 MB
Runtime: 60 sec
Real-Time Inference
Payload size: 6 MB
Runtime: 60 sec

Figure 2.12: Model Deployment Options [29]

2.6.5 Amazon ElastiCache

Amazon ElasticCache [31] is a service that simplifies the management of an in-memory data store in a cloud environment. It offers a high-performance caching solution, eliminating the complexity associated with implementation. It is possible to implement a serverless caching mechanism or create your own cluster, more flexibly.

ElastiCache Serverless allows you to create a cache in high reliability in less than a minute, eliminating the need to size instances or configure nodes and clusters. In addition, ElastiCache Serverless removes the need to manage memory sizing, as the service itself monitors memory in use, network bandwidth and CPU utilization, automatically scaling when necessary. It then offers a simple endpoint to make contact, making the entire infrastructure underneath transparent.

If granular control over the cluster is needed instead, it can be designed from scratch. ElastiCache allows you to choose the node type, number, and geographic location among the different AWS Availability Zones. However, it continues to be a fully managed service, so hardware provisioning, monitoring, replacing nodes when they fail, and patching continue to be transparent.

Chapter 3

Related Works

Web application security has recently gained significant importance, mostly because of the increase in complex assaults like DDoS, SQL injection, and Cross-Site Scripting. Web application firewalls have rapidly evolved in response, particularly with the incorporation of cutting-edge technology like machine learning. To detect new threats and zero-day assaults, a number of open-source and commercial firewalls (WAFs) have started utilizing deep learning-based engines. This reduces the dependence on static rules and increases the accuracy of identifying aberrant behavior.

3.1 Commercial WAFs

The cloud services provider that facilitates and secures web services, **Akamai Technologies** [32], announced in March 2024 that they had added features to their WAF, such as *Browser Impersonation Detection*, which uses machine learning to deepen browser behaviors, reducing false positives and improving the detection of malicious bots. A new feature named *AkaNAT* was also published in March 2024; it deals with identifying shared public IPs, or IP addresses linked to several users or devices. They used an XGBoost classifier and a new machine-learning technique to accomplish this. The dataset consisted of a list of IP addresses with binary labels (1 for shared IP and 0 for unshared IP).

Another web security competitor, **Imperva** [33], also employs machine learning in a number of ways in their security solutions. For example, their *CounterBreach* function allows them to monitor user behavior using behavioral patterns and identify departures from those patterns. They can also identify APBs (Advanced Persistent Bots) in this way.

For several of its WAF features, like evaluating abnormalities in traffic, detecting DoS attacks at the application layer (layer 7 of the ISO/OSI stack), and detecting anomalies through behavioral monitoring of legitimate user-generated traffic, **F5** also employs machine learning models. Lastly, it employs a machine learning technique in its *API endpoint discovery* function to identify abnormalities in API behavior and prevent unauthorized usage of JWT tokens.

Lastly, leading companies like Palo Alto and Fortinet also use machine learning in their web application security solutions.

Through its *Fortiweb Cloud* [34], **Fortinet** primarily use machine learning to safeguard APIs, but it also leverages Support Vector Machine algorithms to examine traffic profiles and identify sophisticated bots by contrasting unusual activity with typical user behavior.

Palo Alto Networks uses machine learning and threat intelligence with its *Prisma Cloud* [35] to detect threats and identify different *MITRE ATT&CK* strategies. The same basic idea underlies both phases of use: Prisma Cloud learns each customer's cloud environment's typical network behavior in the first phase, then efficiently detects network irregularities and zero-day assaults while reducing false positives. Furthermore, it allows users to explicitly select the trade-off between false positives and false negatives based on their security requirements.

Transparency in technical specifics regarding the implementation of proprietary WAFs like FortiWeb, F5, and Imperva is sometimes limited, despite the potential and effectiveness of machine learning in these systems. Companies typically just supply generic information; it rarely includes detailed insights into algorithms, performance benchmarks, or open comparisons that may be used to confirm their offerings' real efficacy or efficiency. This occurs because the technology in question are proprietary, and the corporations are not inclined to provide technical details that would provide rivals in the cybersecurity space with a competitive edge.

3.2 Open Source WAFs

Several open-source alternatives provide greater transparency into the setting of the machine learning models employed as well as how they operate.

One of the most widely used open-source web application firewalls is **ModSecurity** [36], which is renowned for its capacity to defend against various web application threats, including those that rank in the top 10 in OWASP. The project was initiated by Trustwave in 2002, but OWASP acquired it in February 2024. The core of ModSecurity is the Core Rule Set (CRS), which is a collection of known threat signatures gathered by the OWASP foundation. The majority of open-source WAFs follow this set of rules. The guidelines are meant to recognize particular kinds of attacks. Every rule has a unique number that corresponds to the particular assault class. There are severity and paranoia levels linked to rules. The rules that are enabled are chosen based on the level of paranoia. The CRS has four paranoia levels, and each rule has a corresponding paranoia level. Lastly, a severity level—a positive integer value that denotes the degree of hazard associated with an HTTP request—is also assigned to each rule.

A previous attempt [37] was made to develop a machine learning model in a different container than the one hosting the ModSecurity engine, in an attempt to merge the Core Rule Set and machine learning techniques. Due to communication overhead, there is an additional latency; however, the model is loaded and created only at server startup, rather than for each request. An Isolation Forest algorithm was employed in that endeavor. The integration was carried out as follows:

1. A LUA script takes care of getting the useful information from the engine and creates a POST request with that information in the body, sending it to the ML server.

2. The ML server would receive the information in the body, create features and pass them to the pre-trained machine learning model. The model's response was returned to the Lua script using two HTTP status code: 200 for normal requests, 401 for attacks.

However, because it was unable to raise the CRS's performance, this endeavor did not yield the expected outcomes.

OpenAPPsec [38] is an open-source security project developed by Checkpoint specifically for cloud environments. It leverages machine learning to detect and stop attacks, including those from the OWASP Top 10. According to its white paper, the project's machine learning-based engine, which operates in two stages, is the brains behind it. Phase 1 of the learning process involves the learning engine looking for attack indicators in the HTTP request. Every indicator comprises a brief sequence that suggests a possible resemblance to HTTP requests that are exploited to compromise security. Based on millions of benign and malicious requests, a supervised machine learning model is used to evaluate HTTP requests, identifying indicators, and assigning them a statistical probability score that indicates their likelihood of being a component of an attack. Phase 2 involves further analysis of requests deemed suspicious by phase 1 in a second engine. The objective is to eliminate the likelihood of false positives and increase the certainty that the request is an attack. This is accomplished by taking into account the environment in which the WAF operates, including the application's layout and how users interact with it. An unsupervised machine learning model that is continuously updated in real time depending on incoming traffic is then used to conduct the evaluation. The flow of the architecture is shown in figure 3.1. Following the subsequent phase, the second engine generates additional scores by thoroughly examining the payload, parameters, URL, and user reputation, as shown in the table 3.1.

In conclusion, proprietary WAFs have adopted machine learning techniques; nonetheless, a drawback is the absence of performance benchmarks and transparency.

1-10 Higher is less suspicious	**User Reputation score** Did user already show suspicious behavior like sending requests to non-typical URLs? Did user already send suspicious requests before?
1-10 Lower is less suspicious	**Payload score** Indicators appearing in the request represent the likelihood of an attack.
1-10 Lower is less suspicious	**URL score** The system learns if the URL is prone to attacks or false positives and also learns to offset the overall score accordingly.
1-10 Lower is less suspicious	**Parameter score** The system learns if the URL is prone to attacks or false positives and also learns to offset the overall score accordingly.
1-10 Lower is less suspicious	**Combined total score**

Table 3.1: OpenAPPsec - Second engine score computation

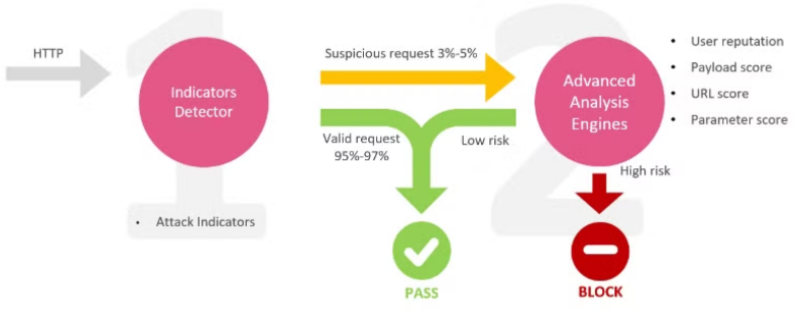


Figure 3.1: OpenAPPsec architecture flow [39]

Open source solutions, on the other hand, are more flexible, but they still have potential for development, particularly when it comes to large-scale integration with cloud technologies and latency reduction, which is the focus of this study.

3.3 Real-time ML pipeline framework

The management of real-time data streams has grown in significance in recent years, particularly with the emergence of concepts like smart cities and the Internet of Things (IoT). Implementing scalable solutions now faces a critical challenge: integrating continuous data streams with artificial intelligence and machine learning models.

Within this framework, Martín et al. [40] present Kafka-ML, an open-source framework that makes it possible to manage ML/AI pipelines by taking advantage of real-time data streams through Apache Kafka, which functions as a distributed messaging system for managing real-time inference as well as model training. Although static file systems are not needed, the data is temporarily kept in Kafka's distributed logs to provide durability and the ability to continue the operation in case of a mistake. This eliminates the requirement for persistent data lakes by enabling the scalable and realistic training and deployment of ML models using data streams. The use of containerization technologies like Docker and Kubernetes, which offer high availability, fault tolerance, and simplicity of deployment in a production setting, is one of the main contributions of Kafka-ML. Additionally, the platform provides an easy-to-use Web interface enabling users to oversee an ML model's lifespan, from training to inference.

Data are transmitted to a Kafka topic, a channel where messages come from heterogeneous sources, such IoT devices or corporate applications. Multiple consumers can be used by Kafka for the same topic, enabling different ML models to receive the data stream.

Docker containers are used to handle the machine learning models and other

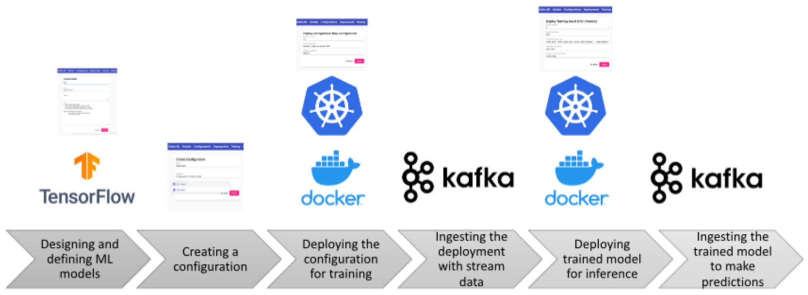

TensorFlow
docker
kafka
docker
kafka
Designing and defining ML models
Creating a configuration
Deploying the configuration for training
Ingesting the deployment with stream data
Deploying trained model for inference
Ingesting the trained model to make predictions

Figure 3.2: Kafka-ML pipeline [40]

Kafka-ML components. This containerized methodology provides application isolation, portability, and on-premises or cloud deployment options. The models are made robust to failures and scalable to the amount of the data stream by using Kubernetes to coordinate the execution of the containers.

The Kafka-ML framework, shown in figure 3.2 allows for real-time model training utilizing data obtained from streams of data that are transmitted to Kafka. Parts of the data stream are separated for training and, if needed, evaluation. Moreover, the framework supports the usage of flow control through control messages that instruct the model on which parts of the flow to utilize for validation and training. Models may be updated continually as fresh data come in this way.

A model may be used for real-time inference in a production setting once it has been trained. Inference occurs when the model processes fresh data received through an input Kafka topic, outputs its predictions in a different output Kafka topic, and repeats the process. Applications that require the forecasts' outcomes can subscribe to this output topic and get the conclusions instantly. Because Kafka-ML can handle many replications of the inference process and divide the load across numerous copies of the model, this technique is very scalable.

Furthermore, Kafka-ML is modularly constructed, enabling users to add other AI and machine learning tools to the framework. Even though it supports well-known frameworks like TensorFlow at the moment, it is made to support more technologies in the future, making it appropriate for various uses in fields like industrial data analysis and the Internet of Things.

Therefore, Kafka-ML represents an innovative approach that addresses the current challenges of real-time data management in the ML/AI context, offering itself as a viable alternative or supplement to existing solutions.

Chapter 4

Methodology

This chapter describes the architecture developed to enable the implementation of Machine learning models on the cloud, with a focus on integration with real-time data streams. The architecture is designed to meet the scalability, flexibility and automation requirements of modern distributed machine learning systems, with a focus on two use cases: bot detection and XSS detection.

The adopted framework involves using Amazon Managed Streaming for Apache Kafka (MSK) as a distributed messaging system, Amazon Elastic Container Service (ECS) for container orchestration and execution, and Amazon Sagemaker for ML model inference. These components work synergistically to ensure efficient real-time data management, improving the system's scalability.

The following section details each architecture component, highlighting their role and how they interact to meet design requirements.

4.1 Architecture implementation

This chapter will be structured as follows: initially, the individual components will be analyzed in detail, and then they will be integrated to obtain an overview of the architecture. The components that will be analyzed are Amazon Managed Streaming for Kafka, Amazon Elastic Container Service, Amazon SageMaker and Elastic Cloud, with brief references to Amazon ECR, Amazon S3 and Amazon ElastiCache.

4.1.1 Amazon MSK

The MSK implementation was designed as the central point for managing data flows between the various services. Specifically, MSK acts as a message broker to ensure efficient and asynchronous communications between the Amazon ECS containers, which host the model preprocessing services, and the WAF engine, which sends HTTP requests passing through it to the broker.

Amazon offers two modes for cluster creation: Serverless and Provisioned. This project's provisioned option was chosen to provide more flexibility and control over resources. This approach allows fine-tuning of configurations and resource provisioning compared to the Serverless model, where AWS automatically manages capacity.

Regarding the version of Kafka, version 3.5.1 was selected and released on September 6, 2023. The community currently supports this version and guarantees regular updates for at least 12 months, with the possibility of future deployments via the AWS console. Regarding broker configuration, these are run on EC2 instances within Amazon MSK. It was chosen to use instances of type *kafka.t3.small*, which offer 2 vCPUs, 2 GB of RAM and up to 5 Gbit network bandwidth. Brokers were distributed across two availability zones to provide greater resilience and redundancy, with one broker per zone.

Each broker was configured with 25 GB of space on the storage side, based on an estimated traffic volume of around 50 GB per day in the production environment. Finally, some additional properties were set to optimize broker behaviour. Specifically, the *log.retention.bytes* property was enabled, with a value set to 16GB, to manage log retention and ensure that the expected storage capacity is not exceeded.

4.1.2 Amazon ECS

As specified in the previous architecture overview related to Amazon ECS, to proceed with the implementation, it is necessary to:

- Prepare a Docker container that contains business logic;
- Prepare a Dockerfile
- Build a Task Definition needed by Amazon ECS
- Provision resources and proceed with deployment

Docker container preparation will be divided into two separate subsections, each dedicated to a different machine learning model. The first part will focus on the integration and the working-logic of preprocessing for bot detection model, while the second part will cover preprocessing for XSS attack detection model.

Docker container for bot-detection

Regarding preparing the Docker container for preprocessing for the bot detection model, it is necessary to group HTTP requests from the WAF engine to ensure proper operation. For this purpose, the Kafka broker was used as a temporary buffer. The preprocessing code is executed every 5 minutes, allowing a significant volume of HTTP requests to accumulate before reading them from Kafka.

To connect to Kafka, a consumer was configured via the `KafkaConsumer` object, defined into `kafka-python` library. The main parameters used are shown below:

- *group_id*: identifies the consumer group to which the client belongs, for managing the distribution of messages among members of the same group. In this case, the value was set to *1*.
- *auto_offset_reset*: set to *earliest*, ensures that if there is no previous offset, the consumer reads data from the first available message in the topic.
- *enable_auto_commit*: the flag was enabled to allow the consumer to automatically update the offset of messages already enabled.
- *max_poll_records*, *fetch_min_bytes*, *fetch_max_bytes* and *max_partition _fetch_bytes* have been configured so that at each fetch from the broker, as many messages as possible are read.

According to this configuration, it is possible to retrieve up to 40.000 requests per iteration of the code.

The previous research, which developed the preprocessing and subsequent machine learning model, had the already structured HTTP requests dataset as input. Because of this, it was necessary to adjust the data from how it arrived in Kafka to make it consistent with what was developed before. Specifically, in the dataset of the first research, each individual entry was accompanied by information about the geolocation of IP addresses. To achieve the same result, I obtained this information through the use of the **GeoLite2-city** database from MaxMind [41], an intelligence project specializing in geolocation and fraud detection. It is a binary database, which makes it compact and quick to query.

In this way, I was able to obtain the information needed to reconstruct the Pandas DataFrame coherently. This DataFrame includes the following fields:

- **@timestamp**: timestamp of when the request occurred;
- **customer**: name of the company to which the traffic belongs;
- **region_code**: country of the IP address from which the request started, in ISO 3166-2 format;
- **real_client_ip**: IP address from which the request started;
- **service_id**: identification code of the contacted upstream;

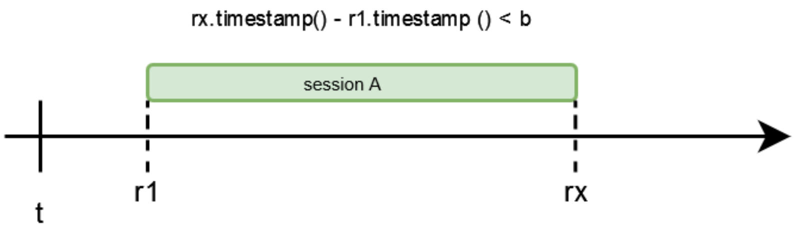


Figure 4.1: Complete session - Example A

- **request_header**: header in JSON format of the HTTP request;
- **request_method**: HTTP method of the request;
- **request_uriPath**: URI path of the HTTP request;
- **response_header**: header in JSON format of the HTTP response;
- **response_http_code**: HTTP code of the response;
- **response_ua**: User-Agent header of the client that made the HTTP request.

After building the DataFrame, is it possible to proceed to preprocessing and creating sessions.

In this regard, an important contribution has been made concerning the creation of sessions: thus, having a batch approach for a continuous data flow, it may happen that at iteration *t+1* sessions have been fetched that belong to the sessions identified and defined at iteration *t*.

For this reason, the concept of **incomplete session** was defined: At the n-th code execution, a session s, made by HTTP requests $r_1, ..., r_x$, is defined **incomplete** if the last session r_x is not older than a first threshold a and if the time between the requests r_1 and r_x is not below than a second threshold b

Here, the mathematical description of this new criteria:

Given a session $s = r_1, r_2, \ldots, r_x$, then s is incomplete if and only if r_x.timestamp() $> a$ & (r_x.timestamp() $- r1$.timestamp()) $< b$

In example A (figure 4.1), session A is complete because the second threshold is met, r_x.timestamp() $> a$ and r_x.timestamp() $-r_1$.timestamp() $> b$

In example B (figure 4.2), session B continues to be complete because, even if the second requirement is not met, still r_x.timestamp() $< a$.

Finally, in example C, shown in figure 4.3, session C is invalid because r_x.timestamp() $< a$ and r_x.timestamp() $-r_1$.timestamp() $< b$, so the session is not large enough, nor old enough, to be considered valid for the processing.

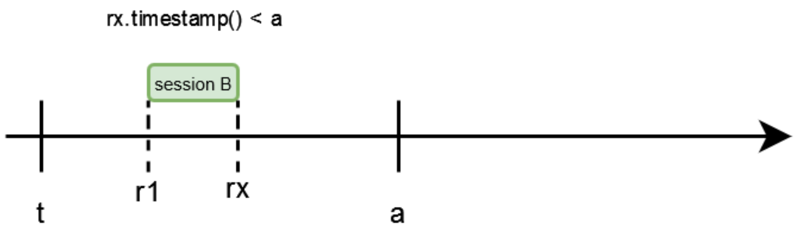


Figure 4.2: Complete session - Example B

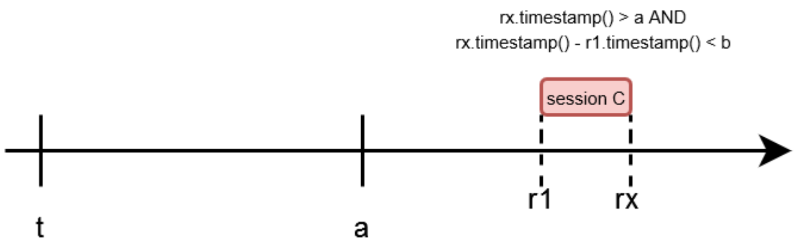


Figure 4.3: Incomplete session - Example C

According to this criterion, at iteration *t1*, the session *s* that is defined as incomplete is sent to a temporary, Redis-based database using the **AWS ElastiCache** service. Redis is an extremely fast in-memory key-value data structure store. The key consists of a string of the type: `BD-<real_client_ip>-<user_agent>-<service_id>`, to avoid duplicates and different sessions with the same key. The value consists of a string representation of the session itself. At the next execution of the code, it is checked whether the session *s2* has the same key as one of those on Redis: if so, the two sessions are merged, and the condition is re-checked. Also, a timeout mechanism was implemented on the entries that exists on Redis, so that upon its expiration the sessions are automatically considered valid.

Before proceeding with inference, we send the newly processed dataset to an S3 bucket, a low-cost object storage. In this way, the data processed at each iteration is not lost, but can be re-used to perform a subsequent retrain of the model.

Having then the dataset available to send to the SageMaker endpoint, we can proceed to inference. When finished, the predictions obtained as a response from the SageMaker endpoint are enriched with the previous computed features and additional information (like the percentage of 200 HTTP code and the percentage of 403 HTTP code) before being sent to the Elasticsearch database, which we will discuss later.

Docker container for XSS detection

Unlike the container responsible for bot detection, the code was scheduled to be launched once every 2 minutes to be more responsive.

The goal of the model responsible for XSS attack detection is to detect any false positives generated by the WAF engine. The dataset the machine learning model needs, consists of a list of payloads. Therefore, after fetching from the Kafka broker (with *group_id* equal to 2, to be discriminated from the other containers), incoming logs were filtered, taking into account only those tagged as malicious by the WAF engine. Next, the payload was extracted from the logs and added to a list, which will then be processed using the steps mentioned before.

Finally, as with bot detection, the processed payloads will be sent to the Amazon SageMaker endpoint, and the results will be written to the Elasticsearch database, enriched by the features extracted for that entry, the malicious payload either in normal form (as it comes from the logs) or in generalized form.

To make the program executable, it is necessary to prepare a Dockerfile, build the docker image, and load it into a Docker registry. For this purpose, the **Amazon Elastic Container Registry (ECR)** service, a fully managed docker image registry, was chosen. With Amazon ECR, the user simply needs to specify the name of the registry to be created and use the *aws cli* to be able to proceed with pushing the image using the command.

```
docker push <ADDRESS_AMAZON_ECR>/<docker_image_name>:<tag>
```

Please refer to Appendix A for the complete example of the Dockerfile used for these containers.

As mentioned earlier, the task definition consists of a JSON object that defines how to execute a task within the Amazon ECS. Please refer to Appendix B for an example of a task definition.

4.1.3 Amazon SageMaker

AWS SageMaker was used only in the third phase of the lifecycle of the machine learning model: for the generation of training data, AWS services were not wide enough to handle a data stream. As for the training phase, it was impossible to use made available by AWS because the task considered in this research is very specific; moreover, previous research already made the models available.

SageMaker was configured to interact with Docker containers to execute inference code. A persistent endpoint was created to get one or more predictions at each invocation, using SageMaker's hosting services.

For model inference, SageMaker launches the container using the command:

```
docker run <image_name> serve
```

According to this syntax, a Python script must be named "serve" inside the docker container, which is responsible for deploying a Web Server end exposing two endpoints.

For model loading, AWS suggests specifying the `ModelDataUrl` or `S3DataSource` parameter when using the `CreateModel` API, which is required for creating the *Model* object. SageMaker copies model artifacts from the specified S3 bucket to the `/opt/ml/model` directory of the container.

For this research, it was decided to load model artifacts directly inside the Docker container, especially for bot detection, as it is also necessary to use a second artifact, comprising the bag-of-words DataFrame. This solution was adopted because concatenating the dataframe with the bag of words and the session dataframe is far more performant than repeated computation of the bag of words, each time the inference endpoint is called.

To obtain inferences, the client sends a POST request to SageMaker endpoints. SageMaker forwards the request to the container, and returns the inference result from the container to the client. To receive inference requests, the container must have a web server listening on port 8080 and accept POST requests to the `/invocations` endpoint and accept GET requests to the `/ping` endpoint.

After the container is launched, SageMaker sends periodic GET requests to the `/ping` endpoint. In the simplest case, the container must respond with HTTP status code *200* and an empty body. This way, SageMaker recognizes that the container is ready to accept inference requests on the `/invocations` endpoint.

According to these criteria, a Docker container structured in the following way was created:

```
\opt\
    program\
        nginx.conf
        predictor.py
        serve
        wsgi.py
    ml\
        bow_df.joblib
        model.joblib
```

The `serve` script does nothing more than launch **nginx** and **gunicorn**. Nginx, a famous HTTP proxy server, simply listens on port 8080 and forwards requests to gunicorn. The latter, whose full name is Green Unicorn, is an HTTP server that implements WSGI (Web Server Gateway Interface), a standard for communication between web servers and Python. It also gives the ability to handle multiple processes to balance the load of requests, improving performance.

The back-end, as anticipated, consists of a Python application written using the **Flask** web framework, a simple and flexible Python module that allows web applications to be built quickly and efficiently. Obviously, the key component of the back-end is the prediction system based on the different machine learning models.

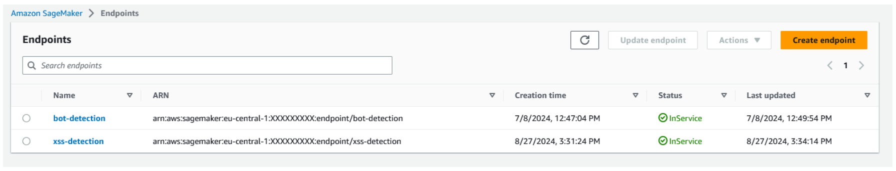
To manage the loading and usage of the model, the **Singleton** pattern was adopted. This creational design patterns ensures that the machine learning model is loaded only once and shared among all subsequent requests. Specifically, the model is loaded into memory only the first time it is invoked, and all subsequent calls reuse the same model instance, thus avoiding unnecessary overhead due to repeated loading. The Singleton pattern is implemented in the `ScoringService` class, which handles the loading and usage of the machine learning model. That class consists of three basic elements:

- Class attribute `model`: attribute shared by all instances of the class and initialized to `None`.
- Method `get_model()`: method responsible for loading the model. It checks whether the `model` class attribute has already been loaded: if it is `None`, then the model is loaded directly from the container file system using the `joblib` library; if it has already been loaded, than it simply returns the existing model.
- Method `predict()`: method that is responsible for doing the prediction and returning the results.

Therefore, the `/ping` route simply calls `get_model()` to load the model artifact returning status *200* if the operation was successful or *404* if the model was not loaded correctly. The difference between the two specific endpoints for bot detection and XSS detection lies precisely in the `/invocations` route, which actually contains the business logic of inference.

As for bot detection, all that is done is to retrieve the payload from the request body and save it as Pandas DataFrame, to be passed to the predict method of the `ScoringService` class. Here, the data frame is prepared consistently with what the model expects to then obtain the clustered partitioning of the data. Next, for each cluster, it assigns the label Bot/Human based on the majority of user agents in that cluster. Finally, it returns the label assigned for each session.

As for XSS attack detection, however, the working principle is slightly different. Actually, each request receives two predictions, one according to the model whose features belong to the HTML domain and the other according to the model whose features belong to the JS domain. The decision in classifying whether the traffic is lawful is made by checking the individual predictions: if at least one of the two predictions classified the request as unlawful, then the request is indeed unlawful. In addition, considering that AWS SageMaker exposes an endpoint, reachable directly via a URL, it is possible to move the preprocessing logic of XSS detection to the WAF engine, so that data processing can then be achieved in streams and no longer in batches.



Name	ARN	Creation time	Status	Last updated
bot-detection	arn:aws:sagemaker:eu-central-1:XXXXXXXXX:endpoint/bot-detection	7/8/2024, 12:47:04 PM	InService	7/8/2024, 12:49:54 PM
xss-detection	arn:aws:sagemaker:eu-central-1:XXXXXXXXX:endpoint/xss-detection	8/27/2024, 3:31:24 PM	InService	8/27/2024, 3:34:14 PM

Figure 4.4: Amazon SageMaker endpoints

To start the build process, it is necessary to prepare a Dockerfile again. The staring images were *Ubuntu 22.04*, to which the above-mentioned tools and Python libraries needed to run the back-end were added. The build images were then uploaded to the docker registry hosted on **Amazon ECR**, named **mithril-ml-bot-detection** and **mithril-ml-xss-detection**. Again, please refer to Appendix C for the Dockerfile definition.

To complete the inference endpoint deployment process, it is necessary to create the *Endpoint Configuration* object, in which the type of endpoint (whether provisioned or serverless) and any production variants are specified. For research purposes, for both models, resources were chose to be statically provisioned, choosing the instance *ml.t2.medium*, with 2 vCPUs and 4GB of RAM. The production variants are the models with which this type of configuration is associated. Thus, the production variants were the templates for each Endpoint Configuration object to be associated with the individual endpoint. Again, the two Endpoint Configuration objects were created via the AWS SDK.

Finally, it was possible to create the endpoint itself, again through the AWS SDK, which was sufficient to specify the name of the endpoint configuration to assemble everything and thus create the two endpoints ready to be invoked.

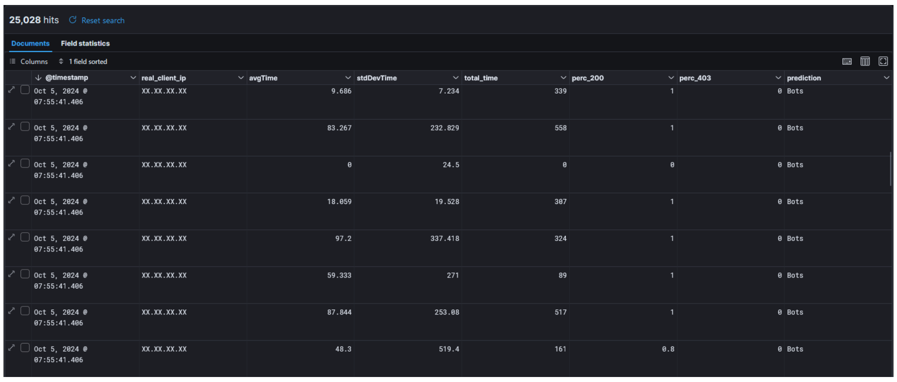
The figure 4.4 shows the two created endpoint.

4.1.4 Elastic Cloud

Elastic Cloud is a fully managed cloud platform that provides the Elastic Stack in SaaS, including tools such as Elasticsearch and Kibana.

Elasticsearch is the core of the Elastic stack. It is a multi-node distributed search engine whose functionality is exposed entirely through a RESTful interface. It is possible to send data in the form of JSON documents using APIs or possible collectors such as Logstash or, within AWS services, using services such as **Amazon Kinesis Data Firehose**.

A key aspect of Elasticsearch is its ability to scale horizontally, thanks to the concept of indexes and shards. An **index** in Elasticsearch is a collection of documents (e.g., logs from the same source), and it can be split into multiple shards. **Shards** allow Elasticsearch to distribute data among multiple nodes in



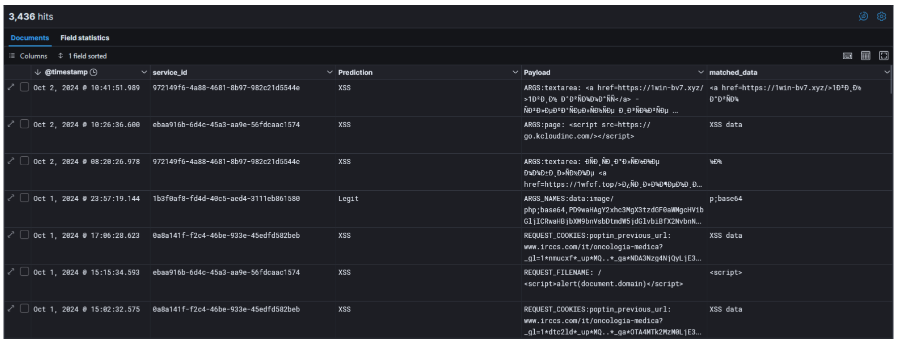
25,028 hits Reset search

Documents | Field statistics

Columns | 1 field sorted

@timestamp	real_client_ip	avgTime	stdDevTime	total_time	perc_200	perc_403	prediction
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	9.686	7.234	339	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	83.267	232.829	558	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	0	24.5	0	0	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	18.059	19.528	307	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	97.2	337.418	324	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	59.333	271	89	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	87.844	253.08	517	1	0	Bots
Oct 5, 2024 @ 07:55:41.406	XX.XX.XX.XX	48.3	519.4	161	0.8	0	Bots

Figure 4.5: Kibana dashboard for bot-detection index



3,436 hits

Documents | Field statistics

Columns | 1 field sorted

@timestamp	service_id	Prediction	Payload	matched_data
Oct 2, 2024 @ 10:41:51.989	972149f6-4a88-4681-8b97-982c21d5544e	XSS	ARGS:textarea: <a href=https://1win-bv7.xyz/ >1Ð²Ð¸Ð½ Ð°Ð²ÑÐ¾Ð¼Ð°ÑÑ</a> - ÑÐ²Ð»ÐµÐºÐ°ÑÐµÐ»ÑÐ½ÑÐµ Ð¸Ð³ÑÐ¾Ð²ÑÐµ …	<a href=https://1win-bv7.xyz/>1Ð²Ð¸Ð½ Ð°Ð²ÑÐ¾
Oct 2, 2024 @ 10:26:36.600	ebaa916b-6d4c-45a3-aa9e-56fdcaac1574	XSS	ARGS:page: <script src=https:// go.kcloudinc.com/></script>	XSS data
Oct 2, 2024 @ 08:20:26.978	972149f6-4a88-4681-8b97-982c21d5544e	XSS	ARGS:textarea: ÐÑÐ¸ÑÐ¸Ð°Ð»ÑÐ½Ð¾Ðµ Ð¼Ð¾Ð±Ð¸Ð»ÑÐ½Ð¾Ðµ <a href=https://1wfcf.top/>Ð¿ÑÐ¸Ð»Ð¾Ð¶ÐµÐ½Ð¸Ð…	¼Ð¾
Oct 1, 2024 @ 23:57:19.144	1b3f0af8-fd4d-40c5-aed4-3111eb861580	Legit	ARGS_NAMES:data:image/ php;base64,PD9waHAgY2xhc3MgX3tzdGF0aWMgcHVib GljICRwaHBjbXM9bnVsbDtmdW5jdGlvbiBfX2NvbnN…	p;base64
Oct 1, 2024 @ 17:06:28.623	0a8a141f-f2c4-46be-933e-45edfd582beb	XSS	REQUEST_COOKIES:poptin_previous_url: www.irccs.com/it/oncologia-medica? _gl=1*nmucxf*_up*MQ..*_ga*NDA3Nzg4NjQyLjE3…	XSS data
Oct 1, 2024 @ 15:15:34.593	ebaa916b-6d4c-45a3-aa9e-56fdcaac1574	XSS	REQUEST_FILENAME: / <script>alert(document.domain)</script>	<script>
Oct 1, 2024 @ 15:02:32.575	0a8a141f-f2c4-46be-933e-45edfd582beb	XSS	REQUEST_COOKIES:poptin_previous_url: www.irccs.com/it/oncologia-medica? _gl=1*dtc2ld*_up*MQ..*_ga*OTA4MTk2MzM0LjE3…	XSS data

Figure 4.6: Kibana dashboard for xss-detection index

the cluster, balancing the load and improving the performance. Each primary shard can have one or more replicas, ensuring high data availability. When sending data to Elasticsearch, data management within the different shards is completely transparent to the user.

Kibana, on the other hand, is Elastic's data visualization and user interface tool. It allows users to explore the data stored in Elasticsearch through dashboards, queries, and graphics. Creating correlation rules highlighting any cause-and-effect relationships from different log sources is also possible.

The two containers responsible for sending the result obtained write to two different indexes in Elasticsearch, **bot-detection** and **xss-detection**.

Two dashboards, shown in figure 4.5 and figure 4.6, have been set up to collect and display data on each entry received. These dashboards allow SOC analysts to monitor not only the predictions generated by the models, but also to obtain additional insights to evaluate the consistency and significance of the information displayed.

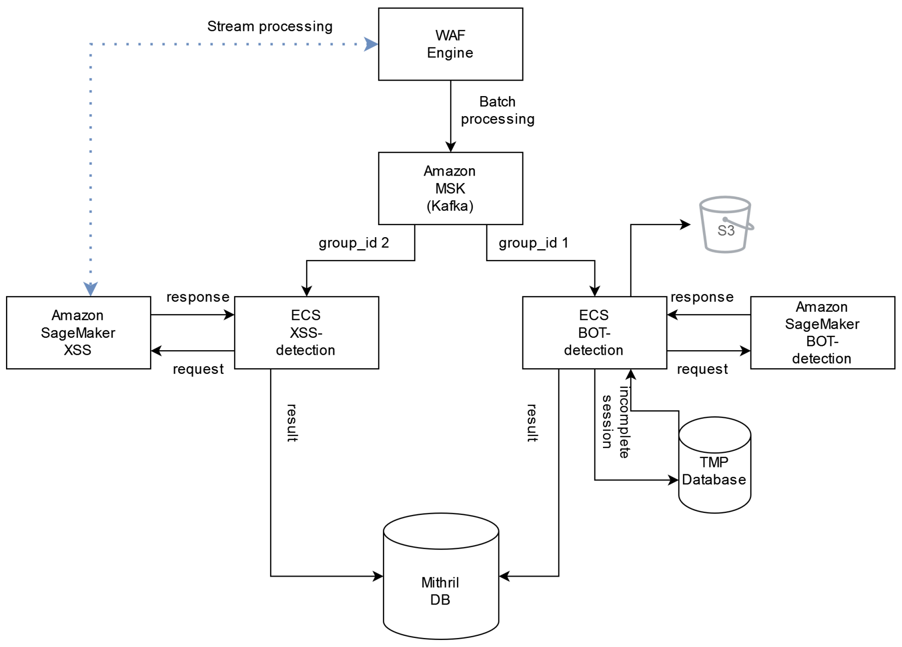


Figure 4.7: Final architecture schema

Having analyzed the individual components, we can now examine the flow of data within the architecture. The figure 4.7 is an illustration of the path the data follows, from the time it is sent to the Broker to the final stage of processing.

As shown in the image, the flow starts from the WAF engine, which not only parses the HTTP requests it receives, but also sends the logs produced to Amazon MSK (Kafka).

ECS containers, using two different group_ids so that they operate independently, read logs from the Kafka broker to initiate preprocessing of each task.

Regarding bot detection, grouping into sessions is a key preprocessing step. When the computation is complete, invalid sessions are temporarily sent to the Amazon ElastiCache service; valid sessions, on the other hand, are sent to the SageMaker endpoint to receive the prediction.

In contrast, for XSS detection, preprocessing is simply done, and then contacting its related SageMaker endpoint to receive the predictions.

Finally, after both ECS containers have received the predictions, the results are sent to the Mithril Elasticsearch database.

Chapter 5

Cost Analysis

When implementing a cloud-based architecture, one of the considerations to be made is the costs associated with each individual service. This is a key factor in determining the feasibility of the solution, especially in public clouds like AWS.

In the following chapter, we will elaborate on each component's costs, get a general overview, and then evaluate possible optimisation strategies in the Future Works section. For this research, the region of use is **eu-central-1 (Europe - Frankfurt)**. Estimated costs vary by region of use.

5.1 Amazon MSK

As for the **Amazon Managed Streaming for Apache Kafka** service, you pay an hourly rate for using the Apache Kafka broker instance, with rates varying based on the size of the broker instance and based on the number of brokers. In addition, storage price is also considered: it is calculated by adding up the GB for which provisioning was done per hour and dividing by the total number of hours per month. No charge is made for data transfer between cluster nodes, but standard AWS charges will be made for data transfer to and from Amazon MSK clusters (inbound and outbound traffic).

Below are the basic points for making calculations:

- The hourly price of instance **kafka.t3.small** is **0.0526 USD**.
- The price per GB per month for primary storage is **0.119 USD**.
- As mentioned earlier, **25GB** of storage is associated for each broker, so that, with retention configured, about 20 hours of logs can be maintained.
- The WAF engine (also hosted in AWS), in 24 hours, produces about 28GB of logs. In 30 days, about **900GB** of logs are produced.

- No transfer costs are charged to and from the AWS cloud since this is intra-region traffic, so the costs are equal to **0**

According to these assumptions, the monthly price for Amazon MSK service is:

- **Broker**: 2 instances $*$ 0.0526 USD $*$ 730 hours in a month = 76.80 USD/month
- **Storage**: 2 broker nodes $*$ 25GB $*$ 0.119 USD = 5.95 USD/month

For a total of **103.23 USD/month**.

5.2 Amazon ECS

There are two different billing models for Amazon ECS, depending on whether the Fargate or EC2 usage model was chosen.

For this research, container deployment via Amazon ECS was chosen using the AWS EC2 model. In this, it was possible to monitor resource utilization and have more flexibility.

Several payment options are available for EC2 instances:

- **On demand**. On-demand instances allow you to pay for processing capacity consumed per hour or per second.
- **Saving Plans**. This is a flexible model, offering lower prices than those derived from on-demand pricing in exchange for a definite commitment to use over a one-year or three-year period. AWS offers three types of Saving Plans:
    1. Compute Saving Plans: automatically apply to EC2 instance usage regardless of instance family or size, and also apply to AWS Fargate or AWS Lambda usage.
    2. EC2 Saving Plans: this type of saving plan provides the lowest prices, guaranteeing savings of up to 72%, in exchange for a commitment to use these instances within a Region.
    3. Amazon SageMaker Saving Plans: flexible pricing model, exclusive to Amazon SageMaker.
- **Reserved Instances**. It is often used when the workload and computational demand are constant and predictable. This billing model saves up to 75%, and is based on 3 key points:
    1. The instance attributes, i.e., the type of instance, the Availability Zone it belongs to, and the type of platform in use on that EC2 instance.

2. The service commitment time: you choose how long to commit with payment; it can be for 1 year or 3 years
3. Type of payment chosen: again, it is possible to choose between full upfront, in which the full cost of the expense is advanced, partial upfront, in which a short down payment is made and then a constant payment over the life of the commitment, no upfront i.e., no down payment.

- **Spot Instances**. These types of instances leverage unused resources from the AWS cloud at a discounted price (up to 90% off), but with the risk that AWS will terminate the instance if the resources are requested by other on-demand instances.

For this research, it was chosen to create EC2 instances with **on demand** payment model, since this is an experimental phase and it might have happened that the type of instance would have to be changed, choosing one with more resources.

To estimate the price of an EC2 instance according to the on-demand payment method, we need to take into consideration:

- Hourly price of the chosen instance: for both machines, a **t4g.medium** instance was chosen, with 2 vCPUs and 4GB RAM, whose hourly cost is **0.0384** USD.
- Storage associated with the instance: for both machines, and for the purposes in use, minimum storage was chosen, equal to 10GB (without any snapshot/backup), with a price of **0.95 USD/month**.

Therefore, the total price of each estimated instance is about 28.03 USD/month for the instance, and about 0.95 USD/month for storage, for a total of **28.98 USD/month**.

Considering both instances, one for bot-detection, the other for XSS detection, the total price is about **57.96 USD/month**.

5.3 Amazon ElastiCache

Again, is it possible to use Amazon ElastiCache with the serverless option, or by creating a cluster, flexibly managing each node individually. For this research, the **serverless** model was chosen. Regarding Amazon ElastiCache serverless billing, two key factors should be considered:

1. Archived data: archived data is billed in gigabyte-hours, with a price of **0.151 USD/hour**.

2. ElastiCache Processing Units (ECPUs): serverless ElastiCache requests are paid in ECPUs, a unit of measure that includes both vCPU time and data transfer. Reads and writes require 1 ECPU per kilobyte of data transferred, with a price of **0.0041 USD/million ECPU**

In this research, at steady state, there are about 10,000 entries on Amazon ElastiCache, with a total weight of about 9.50 MB. In addition, about 1,500 valid sessions are reclaimed from ElastiCache with each code launch, and as many are pushed, a total of about 3,000 requests are made to ElastiCache, with each batch processing the bot-detection.

In terms of price, this corresponds only to **110.45 USD/month**, corresponding to an available cache of 1 GB (minimum amount offered by the service), covering about 36,000 requests to and from ElastiCache.

5.4 Amazon SageMaker

Again, there are two pricing models for Amazon SageMaker: on-demand or with a saving plan, mentioned earlier, in which the user guarantees fixed resource usage for one to three years in exchange for savings of up to 64%.

For the same reasons discussed in Amazon ECS, this research needed to have flexibility on resources, so the on-demand usage (and payment) plan was chosen.

Amazon SageMaker's on-demand billing is based on 3 points:

- Type of instance chosen: as mentioned in the previous chapter, an instance **ml.t2.medium** was chosen for both endpoints, with an hourly price of **0.064 USD**

- Associated storage quantity: The storage price of the SageMaker endpoints is **0.167 USD/month**. The two endpoints do not have large storage needs, so only 4GB per endpoint was chosen.

- Data processing: we consider a price of **0.016 USD** per GB of traffic, both incoming and outgoing. Given such a low unit price, the total price for data processing can be considered negligible.

According to these considerations then, each SageMaker endpoint will cost about 46.08 USD/month for the instance and 0.68 USD/month for storage, for a total price of **46.75 USD/month**.

The price should be considered doubled, since there are two SageMaker endpoints (one for bot detection, the other for XSS detection), for a total price of **93.5 USD/month**.

5.5 Elastic Cloud

For this research, it was decided to use an ad-hoc instance of Elastic Cloud to facilitate later the integration of the dashboards shown above within Mithril.

Elastic Cloud billing is based on the hardware profile that is intended to be used. A hardware profile is a mixture of virtual storage, RAM and vCPU. The profile consists of several resources, each associated with each individual component of the Elastic stack. Needing only to display the data in the appropriate format, the only tools in the Elastic stack used were **Elasticsearch** and **Kibana**, sized minimally to reduce costs.

- **Elasticsearch**: only one hot node was chosen, with a single availability zone, possessing 80 GB storage, 1 GB RAM and up to 2 vCPUs (instance type: **aws.es.datahot.i3en**).
- **Kibana**: again the associated resources are minimal, with 1 GB RAM and up to 6.4 vCPUs, on a single availability zone (instance type: **aws.kibana.c5n**)

Therefore, considering this configuration, the hourly price is **0.0346 USD**, for a total of **25.25 USD** per month.

5.6 Total costs

The only component that remained excluded from this analysis is the **bucket S3** in which the subsequent training data for the bot-detection model is saved. This is because the bucket usage is extraordinarily low, and the billing price (0.0245 USD/GB) can be considered negligible. In conclusion, the studied and implemented architecture, with the assumptions and estimates listed above, has a monthly cost of approximately **390.39 USD/month**

Chapter 6

Results

In this chapter, the results obtained from this research will be analyzed. Hardware resource utilizations will be shown for each component of the architecture. Not having the systems overloaded is important for both batch and stream data processing; therefore, resource sizing must be accurate.

Finally, the results in terms of latency of the data flow from input by the WAF engine to writing to the Elasticsearch database will be shown.

6.1 Network Usage

Monitoring network traffic is crucial as it helps to understand how the system uses network resources, especially in distributed architecture contexts such as in this research. If network usage saturates the available bandwidth, performance degradation may occur. During the testing and monitoring phase, the implemented architecture received a variable amount of traffic, characterized by very high initial peaks at the beginning, followed by a decrease, indicating that the network traffic is quite irregular due to variable workloads. As we can see from figure 6.1, the number of received packets starts to grow more gradually and steadily, due to the increase in load during the working day. Finally, at the end of the day, the number of received packets starts to decrease.

6.2 Amazon MSK

As for MSK, CPU, RAM, storage, and network utilization will be analyzed in detail.

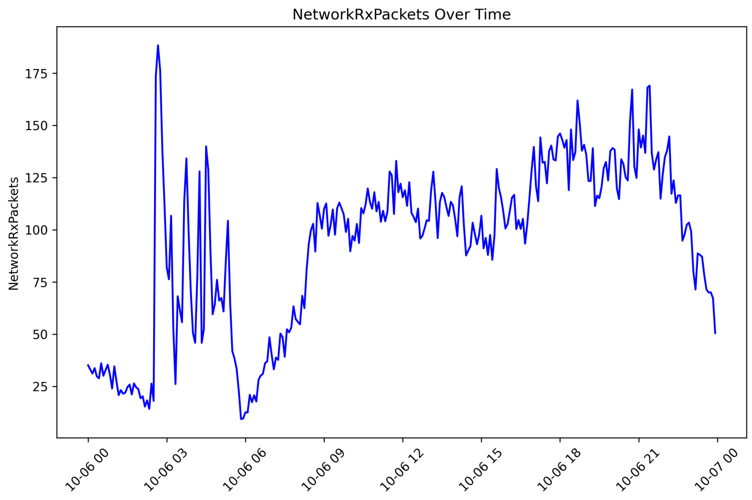


Figure 6.1: Daily usage of Network (Received Packets)

CPU

CPU utilization refers to the percentage of processing capacity utilization to run processes. In Amazon MSK, the CPU is used to handle data flow and message processing. High CPU utilization indicates that the system is processing a high volume of data. It is a key metric for assessing the efficiency of the Kafka broker: if it is constantly under stress (high utilization rate), then bottlenecks or slowdowns may occur. As can be seen from the graph in the figure 6.2, CPU utilization fluctuates regularly over time, with small spikes approaching 10%, correlated with duty cycles from the Kafka broker. A cyclic pattern of CPU utilization is present due to constant daily workflows. CPU utilization is around 5-6%, suggesting that the system still has sufficient capacity to handle additional workloads, a positive factor for performance and scalability.

RAM

RAM usage is critical to the proper functioning of MSK, as Kafka loads temporary data and operations involving topics and messages in transit into memory. Insufficient RAM can cause slowdowns, as it would mean that data must be written to disk. Optimal RAM management allows Kafka to process messages more efficiently, preventing it from swapping frequently, dramatically slowing down the execution flow. The graph in figure 6.3 shows very stable RAM utilization, standing at about 1 GB throughout the monitoring period. This shows that the architecture is well balanced in terms of memory and that MSK does not undergo significant changes

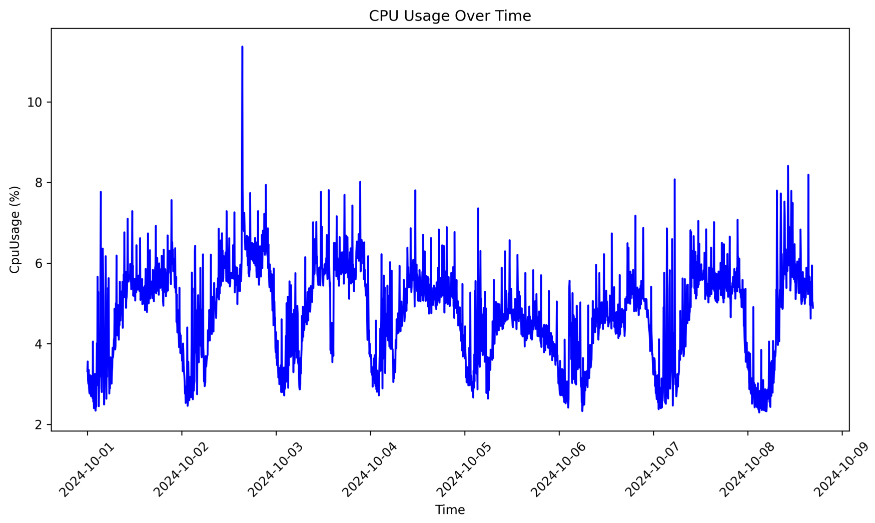


Figure 6.2: Amazon MSK: Weekly usage of CPU(%)

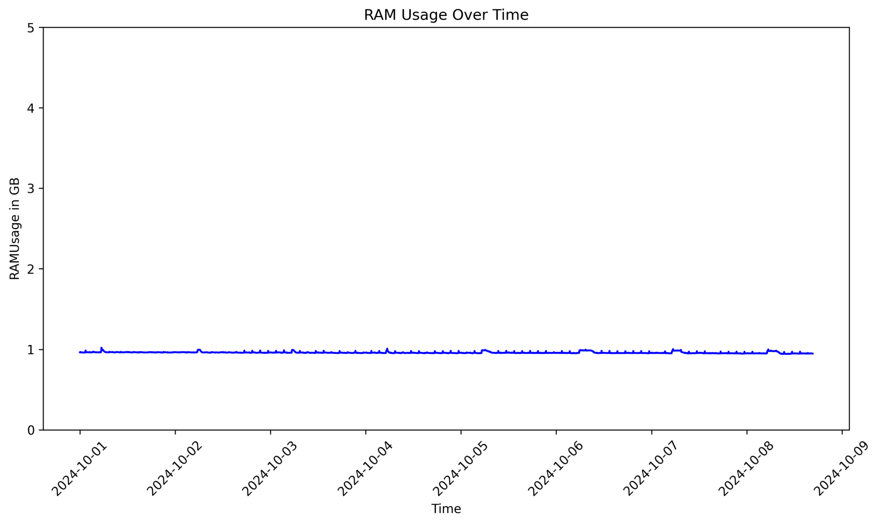


Figure 6.3: Amazon MSK: Weekly usage of RAM (GB)

in workload such that it needs to be used more. No noticeable spikes in RAM utilization are observed, an indication that the allocated memory is sufficient to handle the workload. Considering that RAM utilization remains constant and does not exceed 1GB, memory utilization is efficient, with no waste.

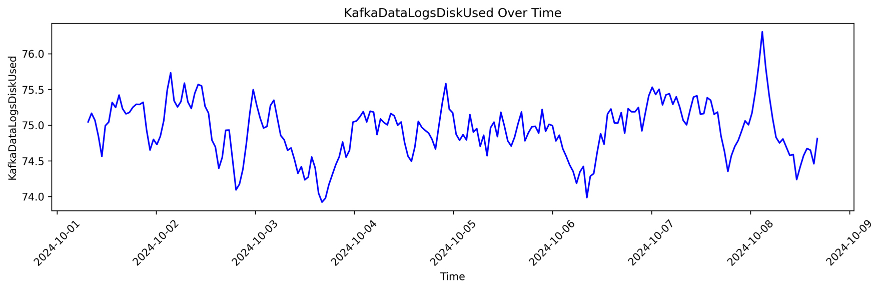


Figure 6.4: Amazon MSK: Weekly usage of Storage (%)

Storage

Finally, storage on MSK represents the physical space on which data, including Kafka message logs, are stored. As a system based on continuous logging of messages, storage is critical to ensure that all messages are retained for as long as necessary to be processed by consumers later. It is also important for the temporary persistence of messages: Kafka allows messages to be reprocessed in the event of failure, and efficient storage allows for greater resilience and security in handling data in transit. The graph in figure 6.4 shows regular oscillations in disk utilization, with values ranging between 74% and 76%. These oscillations are due to Kafka writing log operations, and subsequent deletion due to configured retention. This behavior reflects the natural operation of Kafka, which handles logs cyclically, deleting messages only when they have been compacted into chunks. Despite the fluctuations, the overall storage utilization remains within a moderate range, indicating that Kafka is managing disk space efficiently, without excessive accumulation of data that could lead to disk saturation.

6.3 Amazon ECS

In this section, an analysis of the use of Amazon ECS container resources specific to bot detection and XSS detection will be presented. Again, the key components to be monitored, to assess the efficiency of the architecture, are the CPU and RAM.

CPU

CPU utilization represents one of the key metrics for monitoring the performance of containers managed by Amazon ECS. It is needed to monitor the computational resources used by containers to execute the workload. In this context, it is essential to keep track of this metric to ensure system scalability and efficiency. Regarding bot detection, the figure 6.5 shows peak CPU utilization reaching 60%, alternated

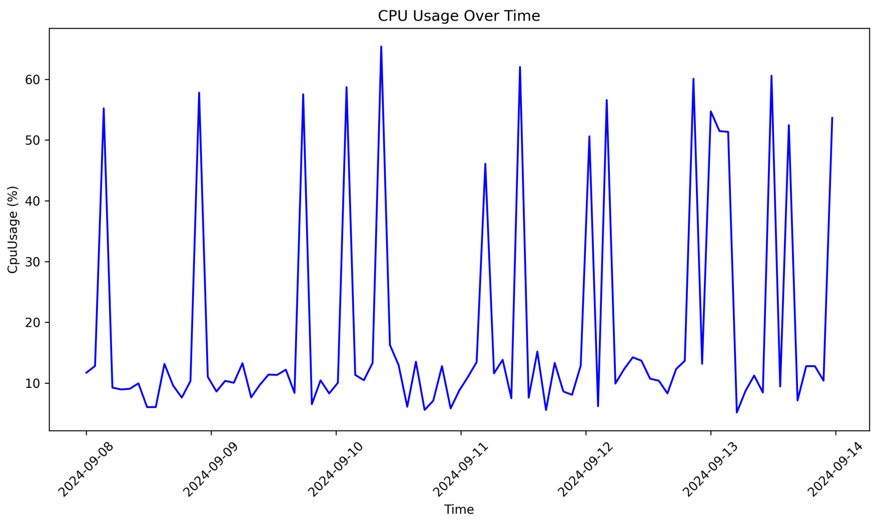


Figure 6.5: Bot detection: Weekly usage of CPU (%)

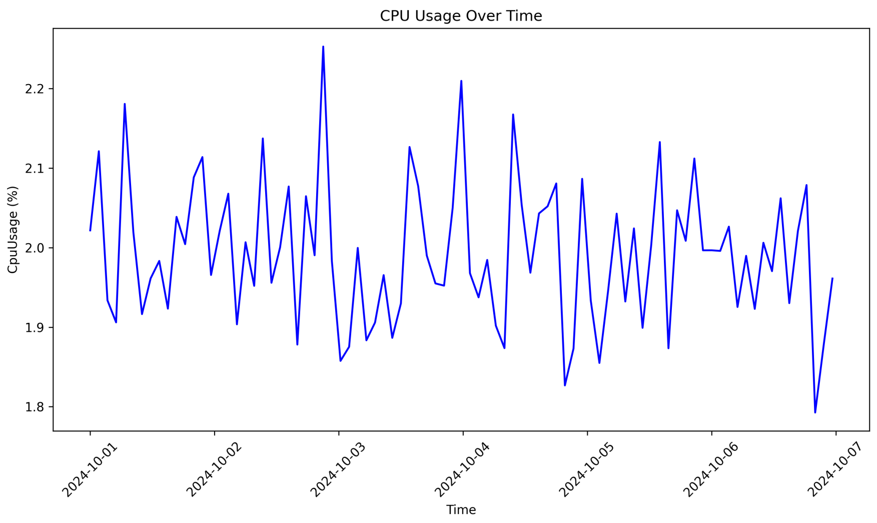


Figure 6.6: XSS detection: Weekly usage of CPU (%)

with periods when it drops dramatically to very low values. This is because, as has been known, the workload is intermittent. Regular spikes indicate that the system is entering high load phases related to preprocessing and clustering activity in sessions. Despite regular spikes, CPU utilization does not become saturated, indicating that the system has room to handle more intense workloads, and there are no obvious signs of CPU overload. As for the container responsible for XSS detection, however, as we can see from figure 6.6, CPU utilization is maintained

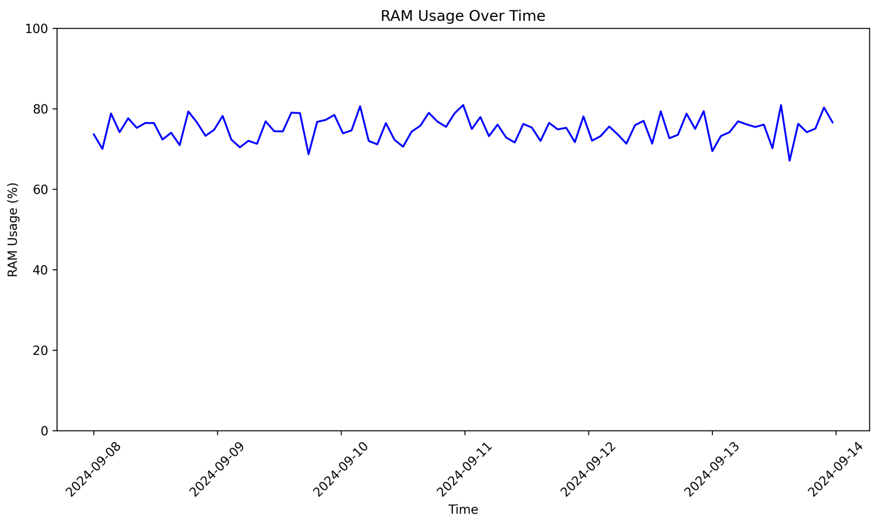


Figure 6.7: Bot detection: Weekly usage of RAM (%)

between 1.8% and 2.2%, demonstrating the model's lightness. No significant spikes are observed, indicating that the system is not subjected to sudden intensive processing demands, so the container is not facing bottlenecks or situations where there is a shortage of resources. Therefore, it can be said that the container hosted on ECS is correctly sized.

RAM

RAM utilization is the other key metric for monitoring the performance of containers managed by Amazon ECS. Proper monitoring of this resource helps ensure that containers have enough memory to perform their task without slowdowns or errors due to its deficiency. In the bot detection container (figure 6.7), RAM utilization remains constant, hovering around 80%. This indicates that the system uses a significant part of the available memory, but without reaching critical levels. Although RAM usage is high, it does not approach saturation, but code optimization techniques can still be considered to reduce its usage. In contrast, the graph in figure 6.8 represents the RAM utilization over time for the container responsible for XSS detection preprocessing. RAM utilization remains constant between 10% and 12%. The system handles a light workload. There are no significant spikes or dips in RAM utilization, which means that the container handles a very stable workload over time.

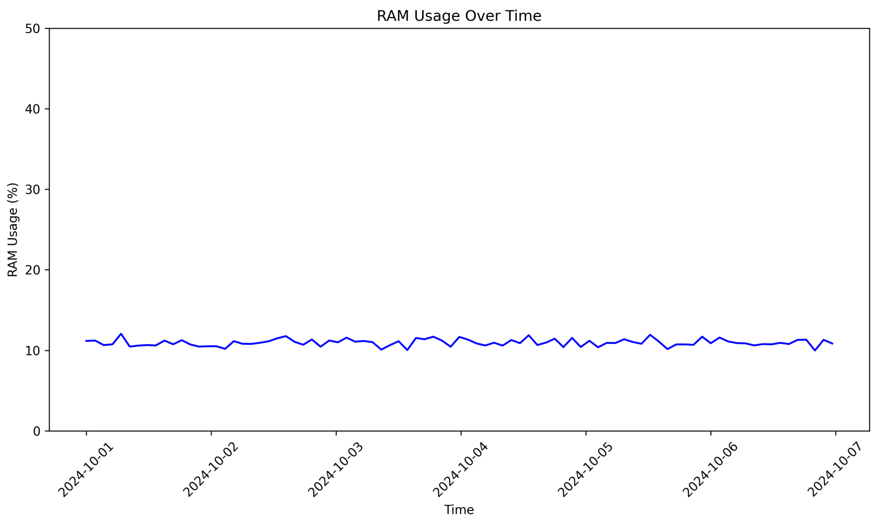


Figure 6.8: XSS detection: Weekly usage of RAM (%)

6.4 Amazon SageMaker

Regarding SageMaker, the CPU utilization, RAM, latency of each model will be analyzed in detail. Next, the number of times the endpoint is contacted, and the size of the dataset at each iteration will be shown. Finally, an overview of the execution time of the entire data stream.

CPU

The CPU is the core of the computational process. Whenever the machine learning model is invoked, the CPU handles the operations necessary to perform inference. In Amazon SageMaker, CPU utilization is closely related to the speed at which a model can process data. If the CPU is overloaded, the inference time increases, slowing down the performance of the model. As can be seen from the bot detection graph in figure 6.9, the CPU utilization trend is intermittent, with frequent periods when CPU utilization near 0%. This is the expected behavior, as the model is invoked non-constantly, for the batch processing mentioned earlier. The graph shows some peaks in utilization, with the highest reaching about 9%. This is caused by times when the model had to process large volumes of data (when the incomplete session timeout was triggered, for example). In each case, the low CPU utilization is under-loaded relative to the available CPU resources, thus indicating that the resources are oversized relative to the actual load. As for XSS detection on figure 6.10, on the other hand, CPU utilization remains practically zero, with values less than 0.25%. This happens because of the filters that are applied on

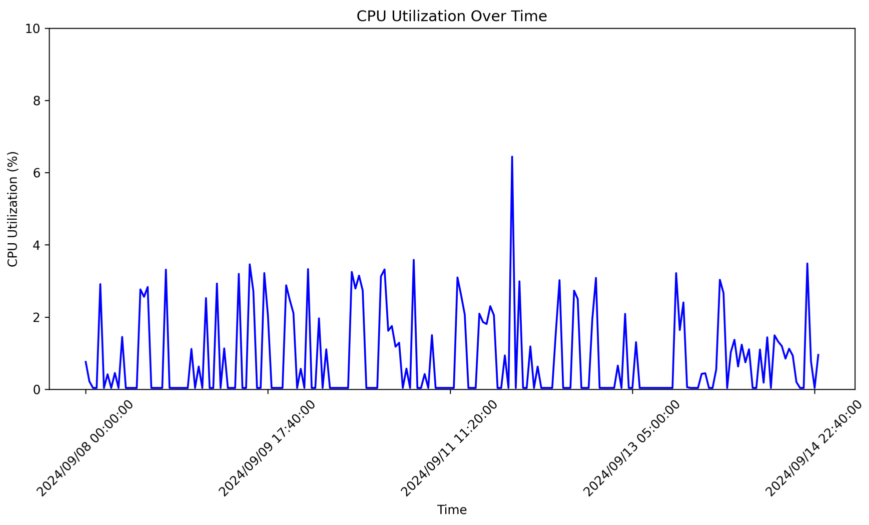


Figure 6.9: Bot detection: Weekly usage of CPU (%)

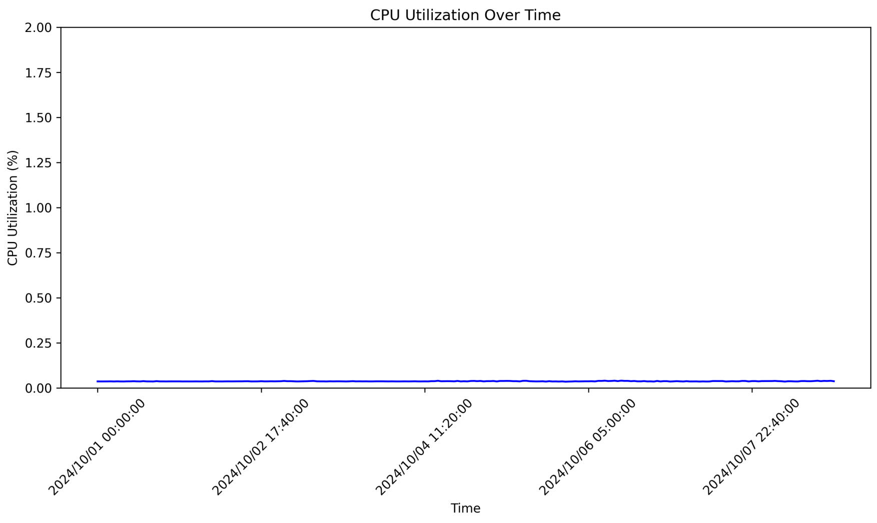


Figure 6.10: XSS detection: Weekly usage of CPU (%)

upstream requests, sending to the endpoint only those that have been tagged as XSS by the WAF engine.

RAM

RAM is essential for loading and handling temporary data. When a machine learning model is invoked, data must be loaded into memory before being processed.

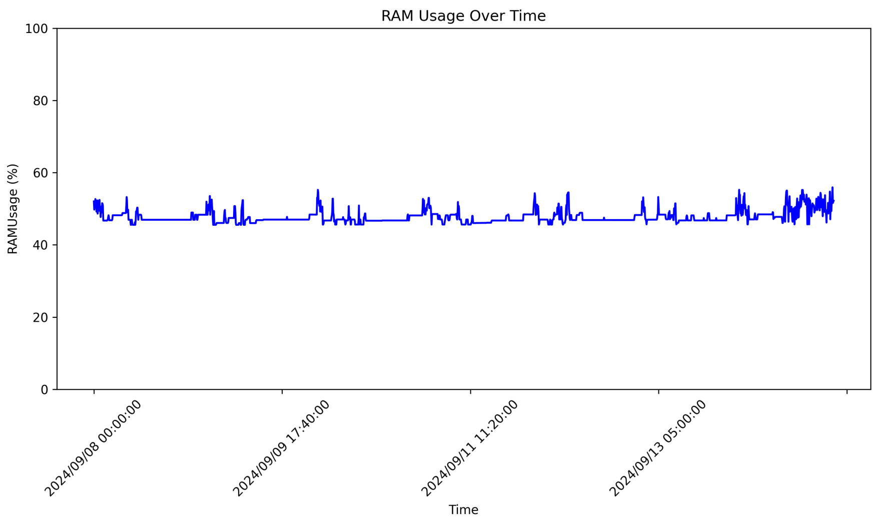


Figure 6.11: Bot detection: Weekly usage of RAM (%)

The amount of RAM required varies with the dataset's size, the model's complexity, and the batches processed. The graph in figure 6.11 indicates that RAM utilization is stable for bot detection, hovering around 50-60% No significant spikes above 60% are seen, implying that the model has not reached the maximum available RAM limit. This is good, as it suggests that the system does not risk running out of available memory during inference operations. The model can be run without the risk of performance degradation. RAM utilization around 50-60% indicates that the resources are appropriately sized for the current workload. If the model were loaded with more data, there would still be sufficient space to handle it. For XSS detection on figure 6.12, on the other hand, similar to CPU utilization, RAM utilization is also minimal, for the same reasons mentioned above. RAM utilization is consistent with the amount of data the model receives when invoked.

Latency

Latency measures the time between the moment the model is invoked and the moment it provides a response (inference). It is a key metric for quality of service. Latency is critical for measuring model efficiency in real-time. A model with low latency responds quickly and can be used in critical contexts, just like bot management. For bot detection, model latency varies significantly (figure 6.13), ranging from very low values to peaks exceeding 5 seconds. This variability depends on the workload, which is often erratic because of the way sessions are created. There is no clear trend of latency stabilization over time, indicating the need for better model optimization to ensure more stable performance. Much better,

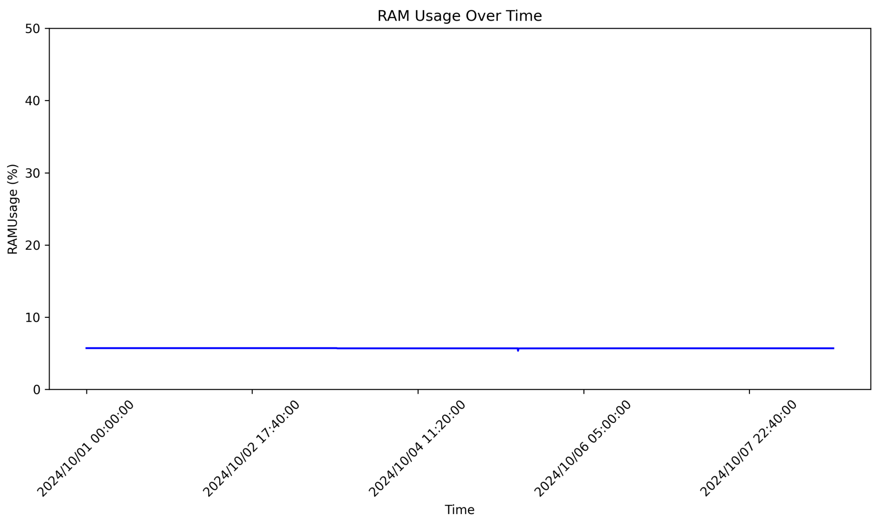


Figure 6.12: XSS detection: Weekly usage of RAM (%)

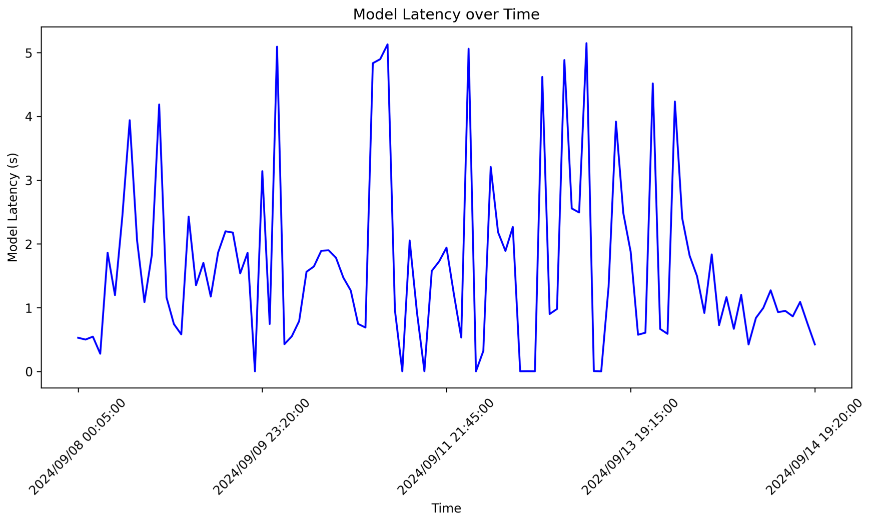


Figure 6.13: Bot detection: Weekly latency (s)

however, is the latency in the model related to XSS detection, in figure 6.14. It fluctuates between 2.5 ms and 4.5 ms, thus keeping within a narrow range. This indicates that the model responds fairly stably. A latency of this range indicates good model efficiency, especially if the goal is to ensure fast responses in security scenarios, such as XSS attack detection.

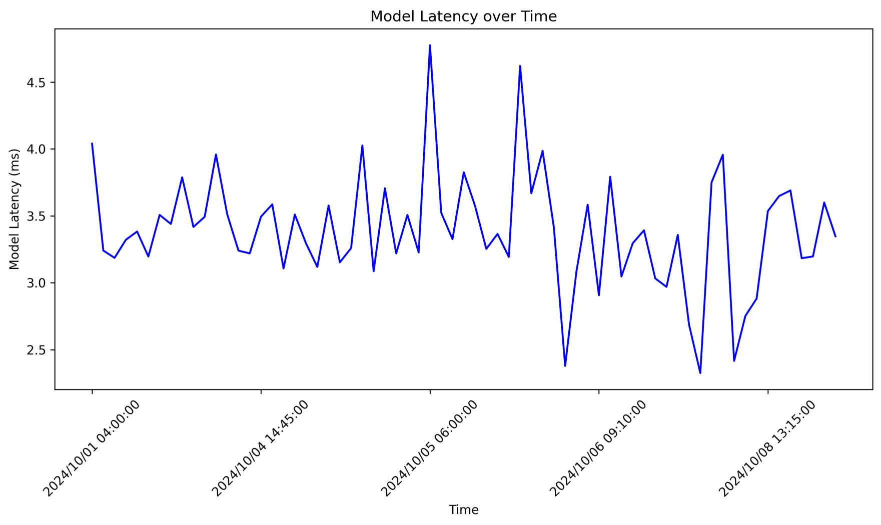


Figure 6.14: XSS detection: Weekly latency (ms)

Number of messages per SageMaker Invocations

The number of messages per invocation refers to the amount of data in the dataset that the model must process each time it is invoked. It thus represents the amount of work the model has to deal with per request. The number of entries directly affects the model's resources. Optimizing the number of entries per invocation balances resources and performance, allowing the model to handle variable loads efficiently. The graph in figure 6.15 shows variability in the number of messages received over a 24-hour period. This variability is due to the way sessions are grouped, which is often irregular. In any case, predicting web traffic trends is a challenge. There is a period of particularly heavy traffic between the lowest point and the highest peak on the graph. This maximum traffic phase is simply related to a time slot of heavier web traffic. The graph reflects the need for the system to be able to handle spikes in activity. We have seen that they do not affect computational resources, so overall, the architecture meets its objectives.

Regarding XSS detection on figure 6.16, on the other hand, the model is not invoked often precisely because of the filters that are applied upstream within the ECS container. This greatly reduces the number of payloads to be analyzed: as shown, 1 to 4 messages are analyzed per invocation.

6.5 Architecture Latency and Model Performance

In this section, the general latency of the whole architecture will be analyzed, specifically for bot detection and XSS detection.

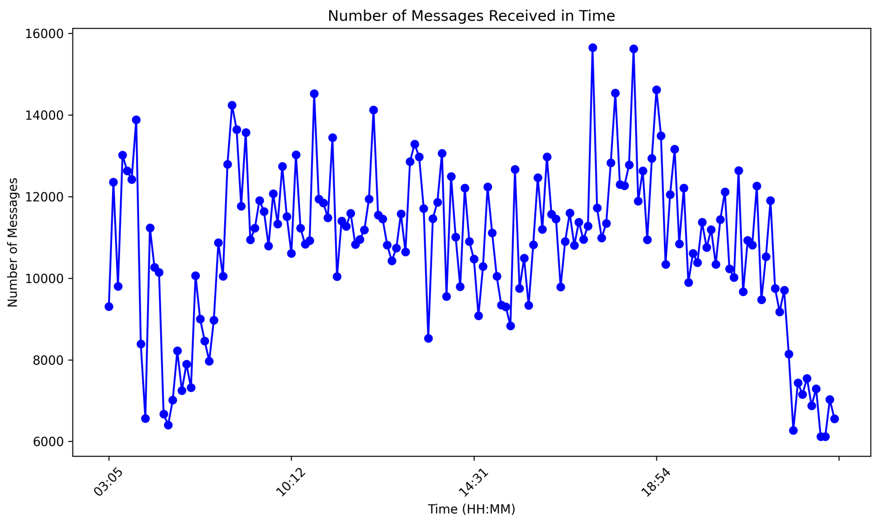


Figure 6.15: Bot detection: Number of sessions sent to SageMaker

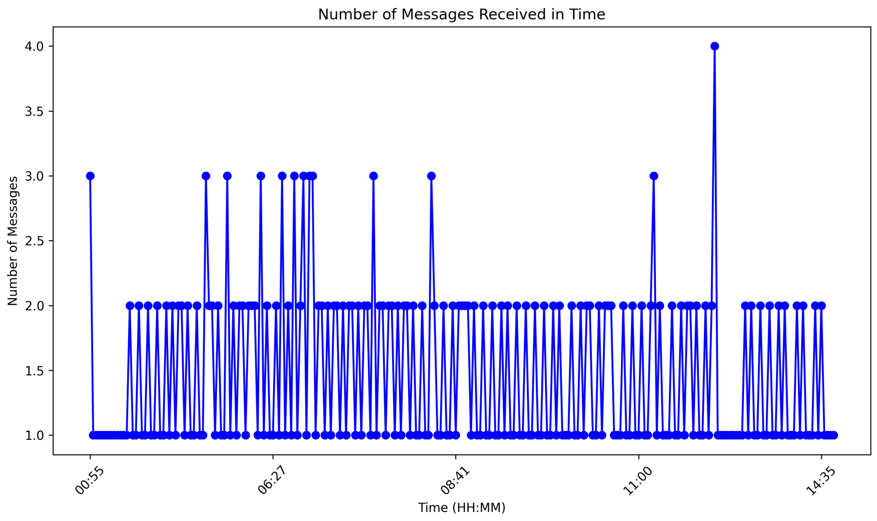


Figure 6.16: XSS detection: Number of requests sent to SageMaker

The graph in figure 6.17 shows the overall latency of the architecture, specifically for bot detection

The graph shows significant variations in latency, with values ranging from a low of 20 seconds to peaks exceeding even 60 seconds. These spikes suggest that there are times when the architecture is under particularly heavy workloads, specifically when batch processing is done. Due to this type of processing, we also see a cyclic pattern in latency, precisely due to batch data handling, where the system is

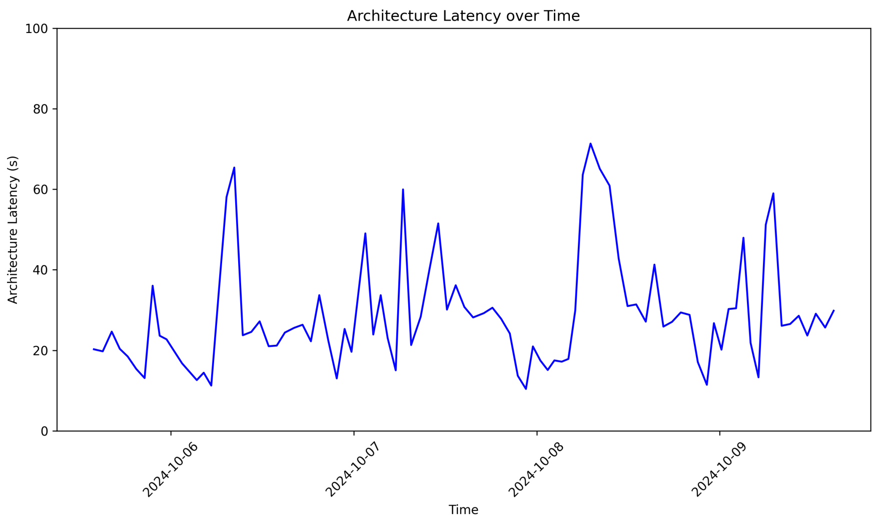


Figure 6.17: Overall latency: bot detection

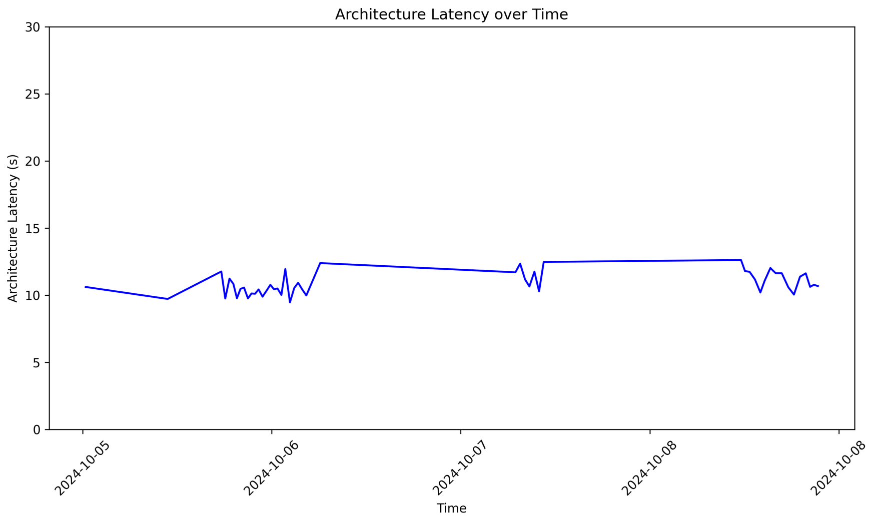


Figure 6.18: Overall latency: XSS detection

subjected to heavier loads at certain times. Although there are moments of high latency, the system does not achieve excessively long response times (100 seconds), which shows that the architecture can still maintain an acceptable performance level.

With respect to XSS detection (figure 6.18), on the other hand, the latency is mostly between 10 and 15 seconds, with slight fluctuations, indicating that the architecture can respond quickly and stably. No significant latency spikes are present, which means the architecture is appropriately sized, and no bottlenecks are present. Moreover, low variability is a positive sign, as it indicates that the architecture can handle the load predictably and reliably.

The results obtained regarding the performance of machine learning models were in accordance with expectations. Regarding bot detection, the k-means cluster algorithm has shown high effectiveness in correctly grouping sessions into clusters. Specifically, out of a sample of 200 sessions analyzed, SOC analysts confirmed that 96.05% of the sessions identified as bots by the algorithm actually corresponded to bots. This result underscores the model's accuracy in accurately distinguishing legitimate from malicious sessions.

Regarding the detection of XSS attack attempts, the main objective was to identify any false positives among requests marked as malicious by the WAF engine. Analyzing a sample of 500 requests tagged as potential XSS attacks by the engine, the machine learning model identified 12 false positives, i.e., requests that, although tagged as malicious, were correctly labelled as lawful. This result highlights the model's effectiveness in refining classification and reducing the risk of errors in threat identification.

Chapter 7

Conclusions and Future Works

The main objective of this thesis was to design and implement a cloud architecture capable of detecting bots and XSS attack attempts using machine learning models. In addition, such an architecture had to be easily scalable, and integrable within the Mithril WAAP context. To this end, an entirely AWS cloud-based architecture was developed, using several services, including Amazon MSK for managing message flows, Amazon ECS for running the containers responsible for preprocessing each model, and Amazon SageMaker for implementing the models themselves.

Regarding bot detection, the k-means clustering algorithm was employed to classify web traffic sessions into the appropriate clusters. On a sample of 200 sessions, SOC analysts confirmed that 96.95% of the sessions labelled as bots were actually bots. The XSS attack detection algorithm played a crucial role in identifying false positives among requests tagged as malicious by the WAF: the model identified 12 false positives, i.e., requests that were mistakenly tagged as malicious but were actually from legitimate users.

The cloud architecture proved efficient in terms of resource utilization. Resource monitoring (CPU, RAM, network, and storage) showed well-balanced utilization that met expectations, with overall architecture latency for both components remaining stable and low. In particular, the average latency for bot detection remained between 20 and 30 seconds, while for XSS attack detection, it was below 15 seconds.

This work has made a significant contribution by demonstrating how machine learning models can be used to improve web application security. The cloud-based implementation made the architecture scalable and flexible, allowing it to handle high traffic volumes effectively. The reduction in false positives, particularly for XSS attacks, demonstrates the practical value of this approach in cybersecurity, with significant spin-offs for companies wishing to improve the protection of their web services.

For future developments, the following evolutionary lines can be considered:

- To greatly lower latency, the session concept used for this research should be revised: it could greatly help if WAF used a session cookie to automatically recognize requests with the same cookie.
- In this regard, it would certainly be helpful to optimize the code that deals with the preprocessing of bot detection, replacing the GeoLite-City2 binary database in a lighter and more flexible one, perhaps in CSV format and with the information strictly necessary for geolocation.
- Making XSS attack detection in real-time: thus avoiding the use of Amazon MSK and Amazon ECS for message fetch and preprocessing, directly tasking the WAF engine and contacting the SageMaker endpoint directly.
- Building a training pipeline: having saved the data on persistent storage such as Amazon S3, you now have easy access to the data. In this sense, appropriately modifying the SageMaker endpoint to also accept requests to re-train the model and use it in subsequent inferences is definitely a viable option.
- Cost optimization: in this regard, it might make sense to use a single endpoint for both models, further exploiting the potential of Amazon SageMaker.
- Finally, an update of the machine learning models, replacing the two individual algorithms in a single neural network capable of performing double classification.

Bibliography

[1] Simon Kemp. Digital 2024: Global Overview Report. URL: https://datareportal.com/reports/digital-2024-global-overview-report (cit. on p. 1).

[2] Flexera. 2023 Tech Spend Pulse. URL: https://info.flexera.com/FLX1-REPORT-State-of-Tech-Spend (cit. on p. 1).

[3] J. Galvin and L.LaBerge with E.Williams. The new digital edge: Rethinking strategy for the postpandemic era. URL: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-new-digital-edge-rethinking-strategy-for-the-postpandemic-era (cit. on p. 1).

[4] European Union Agency for Cybersecurity. ENISA Threat Landscape 2024. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024 (cit. on p. 1).

[5] URL: https://www.mithrilwebsecurity.com (cit. on p. 3).

[6] G. Suchacka, A.Cabri, S.Rovetta, and F.Masulli. «Efficient on-the-fly Web bot detection». In: 223 (July 2021) (cit. on p. 7).

[7] ThreatX. Malicious Bot Detection Through A Complex Proxy Network. URL: https://www.threatx.com/blog/malicious-bot-detection-through-a-complex-proxy-network/ (cit. on p. 8).

[8] A.Acien, A.Morales, J.Fierrez, and R.Vera-Rodriguez. BeCAPTCHA-Mouse: Synthtic Mouse Trajectories and Improved Bot Detection. URL: https://ar5iv.labs.arxiv.org/html/2005.00890 (cit. on p. 8).

[9] Imperva. Rate Limiting. URL: https://www.imperva.com/learn/application-security/rate-limiting/ (cit. on p. 8).

[10] Akamai. What is a Bot? URL: https://www.akamai.com/glossary/what-is-a-bot (cit. on p. 8).

[11] OWASP Foundation. Types of Cross-Site Scripting. n.d. URL: https://owasp.org/www-community/Types_of_Cross-Site_Scripting (cit. on p. 9).

[12] Wallarm. What is XSS. URL: https://www.wallarm.com/what/what-is-xss-cross-site-scripting (cit. on pp. 9–11).

[13] P. Mutton. Twitter users fall victim to new XSS worm. URL: https://www.netcraft.com/blog/twitter-users-fall-victim-to-new-xss-worm/ (cit. on p. 11).

[14] A. Asokan. PayPal Mitigates XSS Vulnerabilty. URL: https://www.bankinfosecurity.com/bounty-hunter-finds-paypal-xss-vulnerability-a-15984 (cit. on p. 11).

[15] OWASP. Cross Site Scripting Prevention Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (cit. on p. 11).

[16] M. Sipek, D. Muharemagic, B. Mihaljevic, and A. Radovan. «Next-generation Web Applications with WebAssembly and TruffleWasm». In: (Sept. 2021), pp. 1695–1700. URL: http://dx.doi.org/10.23919/MIPRO52101.2021.9596883 (cit. on p. 12).

[17] S. Palladino. «Machine learning model for bot and applications detection on Web Application Firewall data». MA thesis. Genova: Università degli Studi di Genova, 2023 (cit. on p. 13).

[18] R. Gnisci. «Machine Learning-Based Detection of Cross-Site Scripting Attacks». MA thesis. Genova: Università degli Studi di Genova, 2023 (cit. on p. 15).

[19] Akamai. What is a Web Application Firewall (WAF)? URL: https://www.akamai.com/glossary/what-is-a-waf (cit. on p. 19).

[20] P. Mell and T. Grance. The NIST Definition of Cloud Computing. 2011. URL: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf (cit. on p. 18).

[21] Wikimedia Commons. File: Cloud computing service models. 2022. URL: https://commons.wikimedia.org/w/index.php?curid=114211203 (cit. on p. 20).

[22] Gartner. Magic Quadrant for Cloud Infrastructure and Platform Services. 2020. URL: https://www.gartner.com/en/documents/4020235 (cit. on pp. 20, 21).

[23] Amazon Web Services. AWS Global Infrastructure. Accessed: 2024-10-04. 2024. URL: https://aws.amazon.com/it/about-aws/global-infrastructure/ (cit. on p. 20).

[24] Pinterest. Pinterest Pin. Accessed: 2024-10-04. 2024. URL: https://ar.pinterest.com/pin/263390278185693134/ (cit. on p. 22).

[25] Apache Software Foundation. Apache Kafka. URL: https://kafka.apache.org/35/documentation.html (cit. on p. 22).

[26] Amazon Web Services. Amazon Managed Streaming for Kafka Documentation. URL: https://docs.aws.amazon.com/msk/ (cit. on p. 24).

[27] Amazon Web Services. Amazon Elastic Container Service Documentation. URL: https://docs.aws.amazon.com/ecs/ (cit. on pp. 25, 27).

[28] Amazon Web Services. Amazon Elastic Container Registry Documentation. URL: https://docs.aws.amazon.com/ecr/ (cit. on p. 26).

[29] Amazon Web Services. Amazon SageMaker Documentation. URL: https://docs.aws.amazon.com/sagemaker/ (cit. on pp. 27, 28, 30).

[30] Amazon Web Services. AWS Boto3 documentation. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html (cit. on p. 29).

[31] Amazon Web Services. Amazon ElastiCache Documentation. URL: https://docs.aws.amazon.com/elasticache/ (cit. on p. 30).

[32] Akamai Technologies. How Akamai Uses Machine Learning to Detect Shared IPs. 2023. URL: https://www.akamai.com/blog/security/how-akamai-uses-machine-learning-to-detect-shared-ips (cit. on p. 31).

[33] Imperva. Advanced Bot Protection Datasheet. Accessed: 2024-10-04. 2024. URL: https://www.imperva.com/resources/datasheets/Datasheet-Advanced-Bot-Protection.pdf (cit. on p. 31).

[34] Fortinet. FortiWeb: Web Application Firewall. 2024. URL: https://www.fortinet.com/it/products/web-application-firewall/fortiweb (cit. on p. 32).

[35] Prisma Cloud. Prisma Cloud Documentation. 2024. URL: https://docs.prismacloud.io/en (cit. on p. 32).

[36] ModSecurity. ModSecurity: Open Source Web Application Firewall. 2024. URL: https://modsecurity.org (cit. on p. 32).

[37] C. Folini. A New Attempt to Combine the CRS with Machine Learning. 2021. URL: https://coreruleset.org/20210519/a-new-attempt-to-combine-the-crs-with-machine-learning/ (cit. on p. 33).

[38] OpenAppSec. OpenAppSec: Open Source Web Application Security Solution. 2024. URL: https://www.openappsec.io/ (cit. on p. 33).

[39] OpenAppSec. OpenAppSec Whitepaper. 2024. URL: https://www.openappsec.io/whitepaper (cit. on p. 35).

[40] C. Martin and P. Langendoerfer et al. «Kafka-ML: Connecting the data stream with ML/AI frameworks». In: 125 (2021), pp. 324–336. DOI: `10.1016/j.future.2021.01.017`. URL: `https://www.sciencedirect.com/science/article/pii/S0167739X21002995` (cit. on pp. 35, 36).

[41] URL: `https://www.maxmind.com/en/company` (cit. on p. 39).

Appendix A

Dockerfile for ECS (bot-detection)

Listing A.1: Dockerfile object for bot-detection ECS

```
FROM ubuntu:22.04

MAINTAINER Damiano Ferla <damiano.ferla@aizoongroup.com>

RUN apt-get update && apt-get install -y \
    python3 \
    vim \
    dos2unix \
    cron \
    python3-pip \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

RUN mkdir -p /app

RUN pip --no-cache-dir install kafka-python-ng pandas geoip2
    pycountry redis sagemaker boto3 elasticsearch

WORKDIR /app

CMD ["sh", "-c", "python3 ./entrypoint.py"]
```

Appendix B

Task Definition for ECS (bot-detection)

Listing B.1: Task Definition Object for bot-detection ECS

```
{
    "taskDefinitionArn": "arn:aws:ecs:eu-central-1:XXXXXXXXX:task-definition/bot-detection-preprocessing:3",
    "containerDefinitions": [
        {
            "name": "bot-detection",
            "image": "mithril-bot-detection:latest",
            "cpu": 0,
            "portMappings": [
                {
                    "name": "bot-detection-container-443-tcp",
                    "containerPort": 443,
                    "hostPort": 443,
                    "protocol": "tcp"
                }
            ],
            "essential": true,
            "environment": [
                {
                    "name": "BROKER-KAFKA-IP",
                    "value": "XX.XX.XX.XX"
                },
                {
                    "name": "MITHRIL-REDIS-IP",
                    "value": "XX.XX.XX.XX"
```

```
             },
             {
                 "name": "MITHRIL-EC-IP",
                 "value": "XX.XX.XX.XX"
             },
             {
                 "name": "MITHRIL-EC-TOKEN",
                 "value": "XXXXXXXXXXXXXXXX"
             },
             {
                 "name": "MITHRIL-SM-IP",
                 "value": "XX.XX.XX.XX"
             },
             {
                 "name": "MITHRIL-SM-TOKEN",
                 "value": "XXXXXXXXXXXXXXXX"
             }
         ],
         "mountPoints": [
             {
                 "sourceVolume": "bot-preprocessing-
volume",
                 "containerPath": "/app",
                 "readOnly": false
             }
         ],
         "volumesFrom": [],
         "logConfiguration": {
             "logDriver": "awslogs",
             "options": {
                 "awslogs-group": "/ecs/mithril-bot-
detection",
                 "awslogs-region": "eu-central-1",
                 "awslogs-stream-prefix": "ecs"
             }
         }
     }
 ],
 "executionRoleArn": "arn:aws:iam::XXXXXXXXX:role/
ecsTaskExecutionRole",
 "networkMode": "awsvpc",
 "revision": 3,
 "volumes": [
     {
         "name": "bot-preprocessing-volume",
```

```json
            "host": {
                "sourcePath": "/ecs/data"
            }
        }
    ],
    "status": "ACTIVE",
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "cpu": "2048",
    "memory": "5120",
    "runtimePlatform": {
        "operatingSystemFamily": "LINUX"
    },
    "registeredAt": "2024-05-19T12:24:07.462Z",
    "registeredBy": "arn:aws:iam::XXXXXXXXX:user/damiano.
  ferla",
    "tags": []
}
```

Appendix C

Dockerfile for SageMaker (bot-detection)

Listing C.1: Dockerfile object for bot-detection SageMaker endpoint

```
FROM ubuntu:22.04

MAINTAINER Damiano Ferla <damiano.ferla@aizoongroup.com>

RUN apt-get -y update && apt-get install -y --no-install-recommends \
              wget \
              python3-pip \
              python3-setuptools \
              nginx \
              ca-certificates \
              vim \
              dos2unix \
         && rm -rf /var/lib/apt/lists/*


RUN pip --no-cache-dir install joblib==1.3.2 pandas==2.2.0 numpy
    ==1.26.3 scikit-learn==1.4.1.post1 user_agents flask gunicorn

ENV PYTHONUNBUFFERED=TRUE
ENV PYTHONDONTWRITEBYTECODE=TRUE
ENV PATH="/opt/program:${PATH}"

COPY ./Container/decision_trees/ /opt/program
COPY ./Container/model/ /opt/ml/
RUN dos2unix /opt/program/*
RUN dos2unix /opt/ml/*
WORKDIR /opt/program
```