NUNO
FERREIRA

Ameaças da próxima geração: Negação de sustentabilidade

Next-Generation Threats: Denial of Sustainability

Universidade de Aveiro
2025

NUNO
FERREIRA

Ameaças da próxima geração: Negação de sustentabilidade

Next-Generation Threats: Denial of Sustainability

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à conclusão da unidade curricular Dissertação, condição necessária para obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Vitor Cunha, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, do Doutor Daniel Corujo, Professor associado com agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho à minha família e amigos.

o júri / the jury

presidente / president

Professor Doutor Paulo Jorge Salvador Serra Ferreira

Professor Associado, Universidade de Aveiro - Departamento de Eletrónica, Telecomunicações e Informática

vogais / examiners committee

Professora Doutora Karima Daniela Velasquez Castro

Professora Auxiliar, Universidade de Coimbra - Faculdade de Ciências e Tecnologia - Departamento de Engenharia Informática

Professor Doutor Vítor António Gonçalves Ribeiro da Cunha

Professor Auxiliar em Regime Laboral, Universidade de Aveiro - Departamento de Eletrónica, Telecomunicações e Informática

agradecimentos / acknowledgements

Agradeço a todos os meus colegas amigos e companheiros pelo apoio, incentivo e partilha de conhecimentos ao longo deste percurso académico. Em especial, gostaria de expressar o meu agradecimento à minha família pelo apoio constante e pelas condições que sempre me proporcionaram, fundamentais para a concretização desta etapa.

Agradeço igualmente ao professor Vítor Cunha e ao professor Daniel Corujo pela orientação e pelo acompanhamento prestado ao longo do desenvolvimento desta dissertação. Expresso ainda o meu reconhecimento ao Instituto de Telecomunicações, particularmente pela disponibilização da infraestrutura laboratorial, essencial para o desenvolvimento deste trabalho.

palavras-chave

Negação de Sustentabilidade Económica, Computação em nuvem, Computação serverless, Segurança na nuvem, Cibersegurança

resumo

A computação em nuvem revolucionou a forma como as organizações implementam e dimensionam aplicações, permitindo uma alocação flexível de recursos em regime de pagamento por utilização. Contudo, este modelo também introduz uma nova classe de ameaças furtivas, como os ataques de Negação Económica de Sustentabilidade, que exploram a elasticidade da nuvem para aumentar os custos operacionais sem interromper o serviço. Esta tese investiga os ataques EDoS em ambientes serverless, focando-se nos mecanismos de autoescalamento em plataformas como o Knative. Foi desenvolvido um Proof of Concept utilizando uma variação do ataque YoYo, que gera cargas de trabalho flutuantes para desencadear eventos de escalamento desnecessários. Para combater este tipo de ataques, propõem-se e avaliam-se experimentalmente duas estratégias de mitigação: uma defesa de autoescalamento aleatório e uma abordagem alternativa mais adaptativa. Os resultados demonstram que estas estratégias conseguem reduzir significativamente o impacto financeiro dos ataques EDoS, mantendo a disponibilidade do serviço. Este trabalho contribui para a compreensão das ameaças económicas na computação em nuvem e oferece defesas leves e práticas para arquiteturas serverless.

keywords Economic Denial of Sustainability, Cloud computing, Serverless computing, Cloud security, cybersecurity

abstract Cloud computing has revolutionized how organizations deploy and scale applications, enabling flexible, pay-as-you-go resource allocation. However, this model also introduces a new class of stealthy threats, such as Economic Denial of Sustainability attacks, which exploit cloud elasticity to drive up operational costs without causing service disruption. This thesis investigates EDoS attacks in serverless environments, specifically targeting autoscaling mechanisms in platforms like Knative. A Proof of Concept is developed using a variation of the YoYo attack, which generates fluctuating workloads to trigger unnecessary scaling events. To counter this, two mitigation strategies are proposed and experimentally evaluated: a randomized autoscaling defense and an alternative adaptive approach. Results show that these strategies can significantly reduce the financial impact of EDoS attacks while maintaining service availability. This work contributes to the understanding of economic threats in cloud computing and offers lightweight, practical defenses for serverless architectures.

reconhecimento do uso de ferramentas de AI

Reconhecimento do uso de tecnologias e ferramentas de Inteligência Artificial (IA) generativa, softwares e outras ferramentas de apoio.
I acknowledge the use of Grammarly (Grammarly, https://www.grammarly.com/) and ChatGPT-3.5 (Open AI, https://chat.openai.com/) to improve grammatical structure, punctuation, and vocabulary.

Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface
APAC	Asia–Pacific
AWS	Amazon Web Services
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoS	Denial of Service
EC2	Amazon Elastic Compute Cloud
EMEA	Europe, Middle East, and Africa
EDoS	Economic Denial of Sustainability
EU	European Union
FaaS	Function as a Service
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
IP	Internet Protocol
IPS	Intrusion Prevention Systems
KPA	Knative Pod Autoscaler
NTP	Network Time Protocol
PaaS	Platform as a Service
PoC	Proof-of-Concept
QoS	Quality of Service
RM	Resource Management
SaaS	Software as a Service
SLA	Service Level Agreement
TCP	Transmission Control Protocol
TTL	Time to Live
UI	User Interface
VM	Virtual Machine

CHAPTER 1

Introduction

1.1 Motivation

The rapid evolution of Cloud Computing into a computing continuum has made traditional Denial of Service (DoS) attacks less effective. Cloud providers can now dynamically scale resources, making it difficult for attackers to exhaust them through simple resource overload, however, this has given rise to a more sophisticated threat: Economic Denial of Sustainability (EDoS).

EDoS attacks target the financial resources of an organization, exploiting cloud scalability by driving up operational costs through malicious consumption of services, rather than disrupting service availability, forcing the victim to bear unsustainable expenses, ultimately leading to service termination due to financial strain. This type of attack capitalizes on the pay-per-use pricing models common in cloud environments.

Furthermore, with the European Union's increasing emphasis on green energy and sustainability metrics, there is a growing risk of EDoS attacks being extended to target an organization's environmental performance. This could degrade their reputation and competitive standing in industries that prioritize sustainability.

Knowing these challenges, understanding and mitigating EDoS attacks has become crucial not only from a cybersecurity perspective, but also to ensure a long-term economic and environmental viability of cloud-based services and take the best of this type of service.

1.2 Goals and Contributions

The primary objective of this thesis is to investigate the emerging threat of EDoS attacks, and to develop a proof of concept experiment that illustrates an exploit scenario targeting the economic model of a cloud based service along with effective mitigation strategies. This work

has as its primary objective to deepen the understanding of how EDoS attacks can drive up operational costs and threaten the sustainability of cloud services.

To achieve these goals, the following contributions were made:

- Designed and implemented a Proof-of-Concept (PoC) that successfully demonstrates how a YoYo attack can exploit autoscaling mechanisms by abusing pay-per-use billing models, as seen in serverless platforms like Knative in this case.
- Proposed and developed two mitigation strategies:
  - **Mitigation 1 - Randomized Autoscaling Defense**: a lightweight adaptive technique to counteract the YoYo EDoS attack.
  - **Mitigation 2 - Dynamic Autoscaling favoring Response Time**: an alternative approach that prioritizes faster response times while still mitigating cost impacts.

1.3 Document Structure

This document is organized into six chapters:

- **Chapter 2 - State of the Art:** This chapter explores the foundational concepts that support the work developed throughout this dissertation. It discusses the definition and types of DoS attacks, their evolution, and impact on services. It then dives into EDoS attacks and their distinction from DoS, explores cloud computing principles such as scalability and elasticity, presents mechanisms used in EDoS attacks through real world case studies, and discusses resource management in cloud environments. Finally, it examines current mitigation strategies and their effectiveness.
- **Chapter 3 – Solution Architecture**: This chapter demonstrates how EDoS attacks exploit autoscaling and pay-per-use billing models in cloud environments. It begins by presenting a general architecture, to model a cloud-style serverless environment. It then focuses on serverless function deployment using Knative (as an example), detailing the Knative Pod Autoscaler (KPA) internal flow, configuration parameters, and ingress behavior. Finally, it describes general monitoring and metering tool options and outlines two lightweight mitigation strategies targeting an actual serverless function in a serverless environment, with two different approaches.
- **Chapter 4 – Evaluation Methodology:** This chapter presents the PoC used to evaluate the serverless function under different scenarios. It describes the architecture that was used to realize the experimental setup, and details the physical and virtual infrastructure as well as the Knative function deployment. It outlines three test scenarios, the baseline traffic (legitimate traffic), the YoYo attack, and the YoYo attack with a mitigation. Finally, it covers monitoring and cost tracking, specifying the metrics collected and their reasoning for evaluating the attack impact and both mitigation effectiveness.
- **Chapter 5 - Evaluation and Results:** This chapter presents the experimental evaluation of the proposed approach under different scenarios. It describes the experimental

setup and metrics used, the results are reported for a baseline scenario, a YoYo attacking scenario, and a mitigation scenario, followed by a comparative analysis of the economic and performance impacts.

- **Chapter 6 - Conclusion:** This final chapter presents the conclusions drawn from the research, along with suggested future work.

CHAPTER 2

State of The Art

The cybersecurity landscape is evolving due to the rising frequency of DoS attacks and the expanding recognition of the influence of cloud computing, which has become more evident in recent years. Traditionally, these attacks focused on exhausting the resources of the targeted systems so that they became unavailable for a certain period and, potentially, taking advantage of it. However, the rise of EDoS attacks signifies a crucial transformation in this field. EDoS attacks seek to undermine the stages of sustainability by taking advantage of the vulnerabilities in profit margins, ultimately targeting the financial resources of the company, in a cloud environment.

Thus, it becomes extremely important to study the way in which EDoS attacks operate, how they arise, and the way they address the pragmatic aspects and issues currently facing organizations when it comes to migrating to cloud infrastructure. This study centers on essential findings from previous research on EDoS attack mechanisms and mitigation strategies, with a focus on how these attacks are being addressed based on sustainability requirements and evolving European Union (EU) policies.

To understand the progression from traditional DoS attacks to EDoS threats, it is essential to talk about the technological advancements that enable these emerging threats. The shift to cloud-based solutions has exponentially increased the attack surface, making systems more vulnerable to stealthy and financial attacks. With cloud providers implementing scalable resource management strategies, EDoS attacks exploit these mechanisms by creating a false behavior of legitimate usage, leading to inflated operational costs for victim organizations.

Moreover, EDoS attacks emphasize the intersection of cybersecurity and financial sustainability, challenging traditional perspectives on the cost of security breaches. Unlike conventional DoS attacks, which focus on service unavailability, EDoS targets the financial viability of businesses, particularly those relying on pay-as-you-go cloud models. This underscores the urgent need for adaptive mitigation strategies that align with the dynamic nature of cloud environments.

In the following sections we will provide a structured examination of the existing literature

and methodologies addressing these challenges. By studying the evolving threat, this work highlights the critical role of proactive defenses and informed resource allocation in ensuring the long-term sustainability of cloud services. Through this lens, the research identifies key areas where future innovations can bridge the gap in current defensive measures, promoting a more resilient and economically viable approach to cloud security.

2.1 Denial of Service Attacks

DoS attacks represent a broad range of cyberattack techniques, designed to overwhelm a network, where the system experiences an excessive number of requests, preventing legitimate users from accessing the servers. These attacks either bring a flood of excessive traffic to their target or exploit system vulnerabilities that will cause disruptions in normal operations. Undoubtedly, DoS forms one of the most dangerous threats in cybersecurity. Such attacks have evolved over the years and have become more complex and more difficult to defend against [1].

This section provides an overview of what constitutes a DoS attack, including the distinctions between DoS and Distributed Denial of Service (DDoS), and examines its various types. It will discuss how DDoS attacks differ by using multiple sources to amplify their effect, followed by an exploration of the evolution of these attacks, their impact on service performance and availability, and common mitigation strategies.

2.1.1 Definition and Types

A Denial of Service attack is a form of cyberattack where an attacker seeks to make a computer or network resource unavailable to its legitimate users by disrupting its normal operations. This is typically achieved by flooding the target with excessive traffic or overwhelming it with many requests, preventing it from processing legitimate user requests. The main characteristic key of a DoS attack is that it is launched from a single machine with only one Internet Protocol (IP) address. On the other hand, a DDoS attack, is a more advanced form of DoS where the attack is carried out from multiple sources, or multiple machines (basically anything with an IP address), that can often utilize (e.g., a network of compromised devices), such as in the case of botnet-driven DDoS attacks [2].

DDoS attacks can be broadly categorized into three types: (i) application-layer attacks; (ii) resource exhaustion attacks; and (iii) volumetric attacks [3].

Application layer attacks exploit weaknesses in applications or services to cause instability, preventing legitimate users from accessing them. These attacks often mimic legitimate traffic patterns, making detection difficult for conventional mechanisms. A notable example is the Slowloris attack [3], which involves sending incomplete HTTP requests at regular intervals to keep connections open for as long as possible. By doing so, the server's connection limit is quickly exhausted, rendering it unavailable to legitimate users. This attack requires minimal bandwidth, allowing it to evade detection systems that monitor for traffic anomalies.

Resource exhaustion attacks aim to deplete hardware resources such as memory, Central Processing Unit (CPU), and storage by exploiting vulnerabilities in network-layer protocols.

These attacks combine specific message patterns to drain the server's capacity. A common example is the TCP SYN Flood [3], which abuses the TCP handshake process. By sending numerous SYN requests with spoofed IP addresses, the attacker forces the target server to allocate resources for incomplete connections. This ultimately exhausts the server's backlog queue, preventing new legitimate connections from being established.

Volumetric attacks focus on saturating the victim's communication channels by generating excessive traffic. These attacks typically generate significantly higher volumes of data compared to application-layer or resource exhaustion attacks. A prominent example is the amplification attack, which exploits services like Network Time Protocol (NTP) and Domain Name System (DNS) to increase the size of response packets. By spoofing the victim's IP address, attackers cause the servers to send amplified responses, thereby consuming the target's bandwidth and rendering the system unavailable [4] [3].

In conclusion, DoS and DDoS attacks represent a significant and evolving challenge in cybersecurity, with a profound impact on service availability and performance. Understanding the distinctions between these attacks and their various types is crucial to design effective defense mechanisms, because each type exploits different vulnerabilities and employs unique strategies. As the sophistication of these attacks continues to grow, detection and mitigation strategies are essential to safeguard critical systems and networks.

2.1.2 Evolution of DDoS Attacks

This subsection will explore the evolution of DDoS attacks, focusing on how and why they have adjusted to new network structures leading up to modern cloud infrastructures.

DDoS attacks have come a long way from the early days of simple high-volume traffic floods. Today's attacks are far more sophisticated, often combining multiple techniques to hit different layers of a network at the same time, exploring protocols like Hypertext Transfer Protocol (HTTP), DNS, and Transmission Control Protocol (TCP) to create layered disruptions. Attackers may vary packet sizes, exploit protocol weaknesses, and utilize amplification methods to strain not only the network bandwidth but also server resources directly. While early attacks mainly focused on clogging up network bandwidth with overwhelming amounts of traffic, modern DDoS strategies have evolved to include more targeted approaches like application-layer attacks and amplification methods.

Application layer attacks, for instance, aim to drain server resources by sending fewer but more precise requests, they proceed by sending legitimate HTTP packets where is virtually no difference between an attack and a normal request. The only main difference resides in the intent and not in the content of the packets [5]. This makes most network level packet filters (and even some application layer firewalls) ineffective in detecting these attacks: much harder than the older and more obvious volumetric attacks. Meanwhile, amplification attacks, like DNS amplification, take advantage of certain network protocols to send back a much larger response than the original request, drowning the target in amplified traffic.

To support all that, the rise of botnets has only made things worse, as these networks of hijacked devices, often consisting of poorly secured Internet of Things (IoT) gadgets, allow

attackers to launch massive attacks from a distance. With botnets getting bigger and more powerful, the scale and impact of DDoS attacks have grown significantly, making them more frequent and much harder to defend against. As a result, defending against these kinds of attacks now requires increasingly advanced and adaptive strategies, and new tools [6].

Based on data published by Google from 2010 to 2022 (Figure 2.1) and supplemented by Cloudflare's reports for 2023 and 2024 (Figure 2.2), the growth in DDoS attack sizes over the last decade has been exponential. In the early 2010s, the largest recorded attacks were measured in Gigabits per second (Gbps). However, by the 2020s, the scale has transitioned to Terabits per second (Tbps), demonstrating a dramatic escalation in attack magnitude and intensity [7].

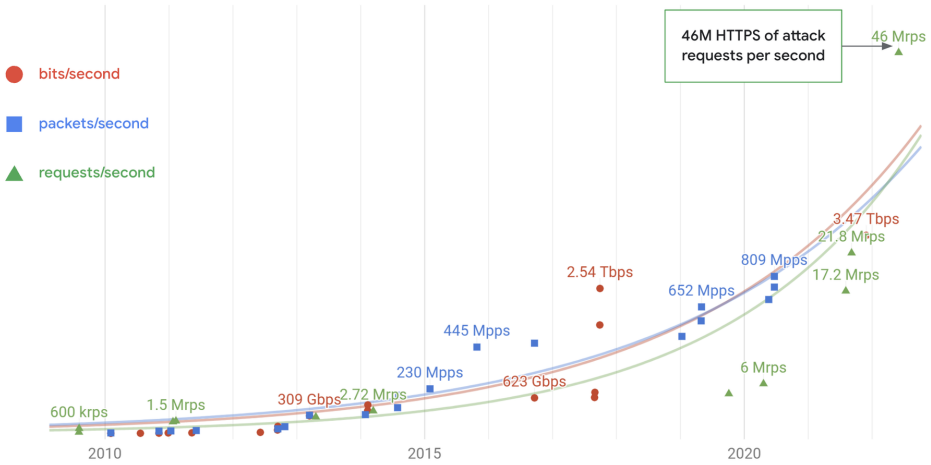

Figure 2.1: Largest known DDoS attacks, 2010 - 2022. (Source: Google) [7].

The evolution of DDoS attack sizes over the last decade demonstrates a clear trend of increasing magnitude and sophistication, with each passing year bringing the potential for more disruptive attacks. Upon analyzing metrics associated with large-scale attacks from the last decade, it is possible to observe an exponential growth pattern rather than linear progression, particularly in recent years.

Furthermore, the number of requests per second (rps) and bits per second (bps) has increased significantly, reflecting a growing capacity for disruption. Historical data from Google reveals a consistent upward trend in attack volume, with a notable peak of 6 million requests per second (Mrps) observed in 2020. This exponential growth illustrates the advancing sophistication of DDoS attacks, which continues to challenge the resilience of modern network infrastructures.

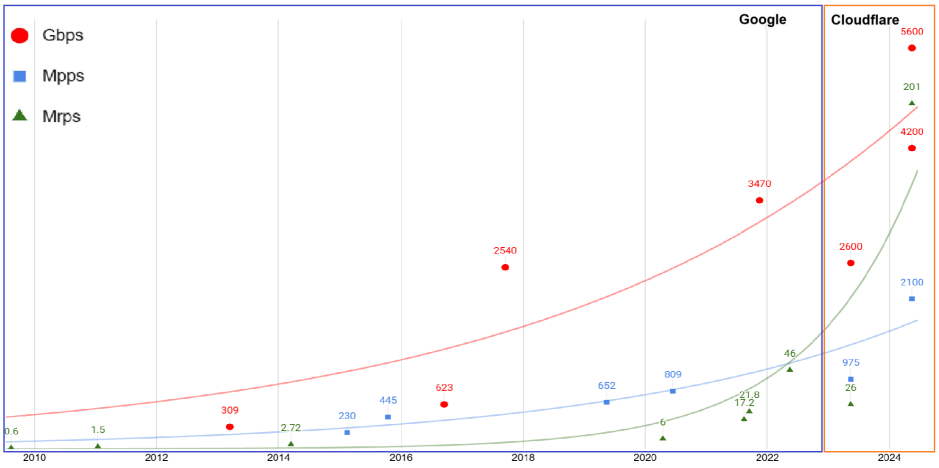

Figure 2.2: Data from last figure extended with large attacks observed by Cloudflare in 2023 and 2024 [7].

According to an annual report by Cisco 2024 Global Threat Analysis Report Executive Summary [8], the frequency and sophistication of DDoS attacks have significantly evolved over recent years. In 2023, there was a 94% increase in the number of DDoS attacks per customer compared to 2022, continuing a steady upward trend observed since 2021, where customers faced an average of 106 attacks per month.

Geographically, America has experienced a 196% increase in DDoS attacks between 2022 and 2023, while the Europe, Middle East, and Africa (EMEA) region saw a 43% increase, and the Asia–Pacific (APAC) region recorded an unprecedented growth of 260%. These statistics underscore the growing prevalence and evolving nature of DDoS attacks, highlighting their substantial impact on global cyber security.

The evolution of DDoS attacks reflects a shift from simple volumetric traffic floods to sophisticated, multi-vector strategies that exploit modern network protocols and architectures. The rise of application-layer and amplification attacks, combined with the increase of massive botnets, has significantly increased both the scale and impact of these threats. The exponential growth in attack size, as evidenced by metrics from Google, Cloudflare, and Cisco, underscores the escalating challenge of mitigating DDoS attacks in modern infrastructures. As these attacks continue to evolve, they demand equally advanced, adaptive, and proactive defense strategies to safeguard critical systems and ensure service continuity.

2.1.3 Impact on Services

DDoS attacks represent a significant threat to online services, not only by completely halting their operation, but also by introducing subtle and often overlooked forms of service degradation. The primary objective of such attacks is to overwhelm system resources, such as memory and disk space, connection limits, network resources, processing power and even critical node capacity, exploring the DNS and even Intrusion Prevention Systems (IPS), making the overall system incapable of responding to legitimate user requests [9]. However, even when full service outages are avoided, these attacks can cause substantial degradation in

performance, including increased latency, packet loss, and a decline in the overall Quality of Service (QoS).

The impact extends beyond technical disruptions, affecting the user experience, financial operations, and organizational reputation, with customers potentially facing long delays or unreliable service access, which can erode trust in the affected provider. In addition, organizations often incur significant costs in mitigating these effects, including the deployment of additional infrastructure or compensating customers due to Service Level Agreement (SLA) violations.

This section explores the diverse impacts of DoS attacks on service availability, performance, and economic factors, illustrating how even partial disruptions can have significant consequences.

E-commerce platforms are particularly vulnerable to DDoS attacks due to their dependence on consistent availability and uninterrupted user experiences. Such attacks exploit system vulnerabilities to disrupt operations, leading to severe consequences for businesses and customers. A DDoS attack can result in an e-commerce platform inaccessible, disrupting transactions and frustrating users. For example, during the infamous DDoS attack on E-bay in March 2014 [1], over 120 million users were affected, resulting in lost sales and a significant breach of personal data. These attacks often peak during high-demand periods, such as holiday sales, amplifying the financial impact. The inability to process orders or even allow customers to browse leads to immediate revenue loss, especially for platforms that rely heavily on real-time transactions [10].

One secondary indirect impact of this is the degradation of the QoS. Even partial disruptions caused by DDoS attacks can degrade the user experience. Customers may experience slow-loading pages, failed transactions, or repeated disconnections. These types of performance issues often result in abandoned shopping carts and discourage users from returning to the same platform. Over time, this decline in service quality erodes customer trust, leading users to competitors who offer more reliable services [10]. Furthermore, during DDoS attacks, critical QoS metrics such as throughput and response time are severely affected, as the authors demonstrate [11] in their graphs, showing a significant decline in throughput "($Figure 3a$)" and an increase in response time by up to five times during attack scenarios "($Figure 4$)". With that, legitimate traffic struggles to reach the server due to the overwhelming flood of malicious packets, leading to increased latency and reduced link utilization. This deterioration of service availability significantly compromises the user experience and further compounds the impact on customer satisfaction.

In addition, the impact can even affect the company's reputation and financial stability. The direct economic impact of these attacks, including lost revenue from interrupted transactions, which can amount to millions of dollars depending on the duration and timing of the attack. Beyond immediate financial losses, secondary effects such as customer confidence, negative media feedback (which nowadays is becoming more popular), and the risk of legal actions

[1] https://www.forbes.com/sites/jaymcgregor/2014/07/28/the-top-5-most-brutal-cyber-attacks-of-2014-so-far/

amplify the damage. The incapacity to maintain uninterrupted service not only frustrates customers, but also drives them towards competitors that offer more reliable alternatives. Over time, these disruptions erode brand loyalty and decrease a company's market share, leaving a lasting impact on its reputation and economic viability [11].

Overall, DDoS attacks can degrade service performance and availability, causing revenue loss and reputational damage even without a complete outage.

2.2 Economic Denial of Sustainability (EDoS)

2.2.1 Definition and Distinction from DoS

Economic Denial of Sustainability is an emerging threat in cloud computing environments that exploits the utility-based pricing and auto scaling capabilities of cloud infrastructure. Unlike the traditional DDoS attacks, which aim to disrupt the availability of services by overwhelming resources with illegitimate requests, EDoS attacks target the economic model of cloud services by generating continuous, seemingly legitimate traffic that gradually triggers auto scaling mechanisms, forcing cloud consumers to pay for additional resources and incur higher costs without proportional business benefit. Consequently, EDoS attacks can financially strain or even bankrupt organizations reliant on cloud platforms, while also causing service degradation for legitimate users [12]. While both DDoS and EDoS attacks may leverage similar techniques to generate traffic, the primary goal of EDoS is financial harm rather than direct service disruption, which differentiates it from conventional DDoS threats. Unlike DDoS, which visibly overwhelms a target's resources, EDoS aims to be as stealthy as possible, gradually increasing costs without immediately alerting the victim to the attack's presence. While EDoS is a form of DoS attack, its purpose differs significantly. Unlike traditional DoS attacks, which aim to disrupt a victim's service immediately, EDoS attacks allow the service to continue but imitate regular client behavior to gradually drain computing resources over time, which is the distinctive characteristic of a EDoS attack [13].

2.2.2 Cloud Computing and its Scalability

As cloud computing continues to gain traction due to its flexibility, elasticity, and cost-effectiveness, the pay-as-you-go model has become fundamental to Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) and particularly Function as a Service (FaaS) or serverless computing. In this model, users are charged based on actual execution time and resources utilized, optimizing cost effectiveness under normal conditions. However, this model also opens opportunities for financial exploitation [14] [15]. For instance, in IaaS and PaaS, automatic scaling adjusts resources based on demand, providing a responsive solution to fluctuating workloads. Similarly, serverless computing platforms, such as Amazon Web Services (AWS) Lambda [16] and Google Cloud Functions [17], automatically allocate resources, enabling developers to focus on code without needing to worrying about infrastructure management.

Two of the most relevant concepts in cloud computing for EDoS, which will be explained in greater detail later, are scalability and elasticity. These concepts are critical yet distinct,

enabling systems to effectively manage varying workloads. Scalability refers to a system's capacity to increase resource limits over time to support long-term growth in demand, this typically involves a planned, gradual allocation of additional resources, such as adding more virtual machines or storage as demand increases. A scalable system maintains performance as demand grows, and it is often measured by its ability to keep this performance stable when resources are doubled or tripled [18].

Meanwhile, elasticity in cloud computing refers to the system's ability to quickly adjust resources up or down to meet sudden spikes or drops in demand, often in real time. This flexibility allows cloud environments to allocate resources dynamically, ensuring they are available precisely when needed. Such elasticity is crucial for handling unpredictable workloads and is a defining feature of cloud services. For example, elastic services like Amazon Elastic Compute Cloud (EC2) [19] automatically scale resources based on demand, providing compute capacity that adapts efficiently to changing workloads, ensuring both optimal resource utilization and stable performance. Together, elasticity and scalability create a responsive, cost-effective environment where scalability supports long-term growth, and elasticity addresses immediate, short-term fluctuations [18].

Auto scaling in Virtual Machine (VM) instances on cloud platforms exemplifies elasticity in action. In this mechanism, the number of VM instances adjusts automatically based on user activity levels. Auto scaling monitors parameters such as CPU utilization, memory usage, response time, and network bandwidth. When CPU utilization exceeds a predefined threshold, the scale-up mechanism is triggered, adding more VM instances to handle heavy traffic and bring CPU usage back to normal. Similarly, when CPU utilization drops below the threshold, the scale-down mechanism is activated, turning off extra VM instances to save resources and costs [20].

In summary, cloud computing's core principles of scalability and elasticity are foundational to its general adoption and utility. These features enable cloud platforms to dynamically respond to both long-term growth and short-term fluctuations in workload demands, ensuring cost-effectiveness and performance stability. However, while these capabilities offer significant advantages, they also introduce vulnerabilities that can be exploited, such as the potential for financial abuse through attacks like EDoS. By understanding the subtle interplay between scalability and elasticity, organizations can better appreciate the strengths and limitations of cloud environments, laying the groundwork for discussions on how these same features may be influenced or targeted in advanced attack scenarios.

2.2.3 Mechanisms of EDoS Attacks: Case Studies and Examples

The emergence of EDoS attacks exploits this model by generating seemingly legitimate traffic designed to gradually trigger automatic scaling of resources. This stealthy approach can impose significant economic burdens, particularly in serverless environments, where scalability is a key advantage and can quickly turn into a financial liability if not carefully monitored. Studies have shown that organizations unaware of these vulnerabilities may experience unforeseen financial strain, emphasizing the need for awareness and mitigation

strategies in the face of evolving cyber threats [21].

This subsection, will explore real world examples and experimental demonstrations of EDoS attacks. By examining practical cases, this work highlights the diverse methods attackers use to exploit the scalability and "pay-as-you-go" model of cloud computing, ultimately draining the financial resources of their targets. These examples provide valuable insight into how EDoS attacks are executed and their potential consequences, laying the foundation for understanding both their impact and possible mitigation strategies.

For example, the Application Programming Interface (API) Abuse exploits legitimate platforms to indirectly target victim servers (2016) [22]. Here, attackers exploit services like Facebook's URL fetcher or Pinterest's image fetching API to generate massive traffic directed at the victim's resources. A significant advantage of this approach is its ability to obscure the attacker's identity, since the requests originate from the infrastructure of trusted third-party providers, so the victim perceives the traffic as legitimate, complicating detection and mitigation. This stealth characteristic is further amplified by the distributed nature of these services, as they often use load balancing across multiple IPs, making IP-based blocking almost impossible. The authors [22] show that, for instance, attackers may use Pinterest's image retrieval API to fetch large images from a victim's server. Despite being a legitimate service, the API can be abused to cause significant financial damage. Research has shown that it can result in a bandwidth amplification factor exceeding 135,000 MB. In one example, the API fetched over 140 MB of data in just seven seconds, using only a single computer, causing substantial costs for the victim while requiring minimal effort from the attacker. While the technique is undeniably effective in evading detection and causing substantial financial strain on the victim, it is not without limitations, beacuse its dependence on third-party API's makes it susceptible to service-specific rate-limiting mechanisms or cache-based defenses. Additionally, the amplification potential, while high, varies significantly among service providers, with some enforcing stricter limits on content size or request frequency. These limitations highlight the evolving nature of this attack vector and the importance of adaptive defenses.

Another example is the recently introduced YoYo attack, which targets the auto-scaling mechanisms of cloud platforms. The YoYo attack, as described by Ko et al. (2020) [20], manipulates the cloud's auto-scaling behavior by alternating between high and low traffic bursts. During the on-attack phase, attackers send short bursts of traffic that exceed scaling thresholds, prompting the platform to allocate additional VM's. In the subsequent off-attack phase, the attacker stops activity, leaving the additional resources idle and causing economic inefficiency. This cycle is repeated, creating substantial economic damage while minimizing the attacker's own costs again. The YoYo attack is particularly dangerous because of its ability to bypass detection by mechanisms like EDoS-ADS (will be later disused on), a mitigation strategy designed to differentiate legitimate and malicious traffic. By carefully timing traffic bursts to avoid triggering "attack mode", the YoYo attack can force unnecessary scaling while evading classification as an attack. As the researchers noted, this technique is not only efficient but also challenging to detect, given the predictability of scale-up and scale-down processes. They also highlight the need for further studies to enhance defense mechanisms against this

evolving threat, showing that the YoYo attack is a real threat to the cloud environment. However, its success relies on the predictability of auto-scaling policies and a lack of adaptive defenses, that the attacks must know or study before using that technique, which reinforces the importance of developing more dynamic and resilient strategies to counteract such attacks.

2.2.4 Computing Continuum

The field of cloud computing is constantly evolving, and a new paradigm is emerging: the computing continuum. This model extends beyond traditional cloud infrastructure to include the network edge, creating a seamless integration of different computing layers such as cloud, edge, and fog computing. By unifying these layers into a reliable ecosystem, the computing continuum offers several advantages, including lower latency, improved scalability, and better resource management. However, it also brings new challenges, especially when it comes to ensuring security and maintaining sustainability in such a complex environment.

The cloud computing, edge computing, and fog computing represent distinct, and interconnected layers of modern distributed computing, each addressing specific challenges of data processing and resource management. Cloud computing relies on centralized data centers that provide vast computational power, scalable storage, and high-level analytics. This paradigm excels in processing extensive datasets and hosting complex applications, but as data traffic surges and latency-sensitive use cases like IoT applications multiply, the limitations of centralized cloud models become apparent. Issues such as high latency, increased bandwidth consumption, and potential service bottlenecks hinder its effectiveness in scenarios requiring real-time responsiveness [23].

To mitigate these challenges, edge computing brings processing power closer to data sources, such as IoT devices and local nodes. By performing computations directly at or near the edge of the network, it minimizes delays and improves responsiveness for applications like smart grids, autonomous vehicles, and industrial IoT. Edge computing supports device mobility and reduces the load on cloud data centers, enabling more efficient use of resources while maintaining real-time operational capabilities [23].

Complementing these paradigms, fog computing acts as an intermediary layer between cloud and edge. Unlike edge computing, which operates directly on or near IoT devices, fog computing utilizes infrastructure like gateways, routers, or local servers to process data within the network itself. This approach addresses challenges such as network congestion, data privacy, and scalability by distributing computational tasks more effectively across the network. It bridges the gap between cloud and edge, offering a hierarchical processing model that supports applications requiring both local autonomy and centralized coordination [23]. Figure 2.3 illustrates how these work individually.

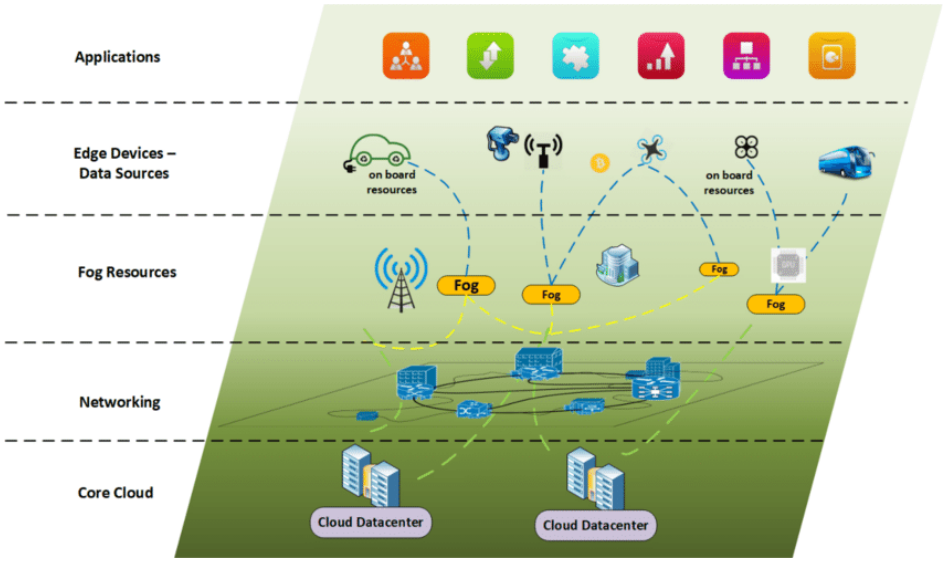

Figure 2.3: Cloud-Computing Continuum [24].

Together, they create a unified framework within the computing continuum, ensuring that data is processed at the most appropriate level based on latency, resource availability, and privacy requirements among others. This integrated approach improves scalability, optimizes resource utilization, and supports a diverse array of applications, from high performance real-time applications to extensive data processing tasks.

In the context of EDoS attacks, the computing continuum plays a crucial role. With resources distributed across multiple layers, attackers can exploit the varying characteristics of each layer, namely cloud, edge, and fog, to launch sophisticated attacks that are difficult to detect and mitigate. For example, while edge computing environments provide low-latency access to data and services, they may lack the robust security measures present in centralized cloud infrastructures, making them attractive targets for exploitation. Similarly, the inter connectivity of these layers introduces additional complexities, where vulnerabilities in one layer can cascade through the continuum, amplifying the potential for financial harm.

Moreover, as organizations increasingly adopt hybrid models where workloads are distributed across cloud and edge platforms, the risk of economic denial of sustainability escalates. Attackers may target the more decentralized edge layers, causing localized disruptions that trigger unnecessary scaling in cloud platforms, thereby increasing costs without directly impacting service availability. This dynamic highlights the need for adaptive defense strategies that can operate across the entire computing continuum, ensuring both the security and sustainability of cloud-based services.

In the next sections, this work will explore the implications of this computing continuum on the sustainability of cloud services, specifically focusing on the economic impact of EDoS attacks and the challenges of defending against such evolving threats.

2.2.5 Resource Management in Cloud Services

Resource Management (RM) plays a central role in the effective functioning of cloud computing environments. As we saw before, cloud computing operates on the principles of virtualization and distributed systems, offering services such as IaaS, PaaS, SaaS, which were already discussed. RM techniques are essential to ensure efficient allocation, monitoring, and utilization of these virtualized resources. By dynamically managing the use and release of resources, RM addresses performance, cost efficiency, and sustainability concerns, which are critical to both service providers and users [25].

The RM process in cloud environments involves meeting the terms of SLA, which outline the expected quality of service, penalties for violations, and payment details. A key feature of these environments is the use of virtualization to create flexible, on-demand provisioning of resources. This flexibility enables scaling up or down based on workload demands, while releasing unused resources to optimize overall system efficiency [25].

Cloud service providers use market-oriented RM techniques to cater to the dynamic nature of resource provisioning. These techniques prioritize economic incentives and scalability, enabling resource sharing among users on a "pay-as-you-go" basis. However, challenges such as SLA violations, energy efficiency, and load balancing often arise, requiring RM strategies to balance conflicting objectives effectively. For example, combining workloads to save energy can increase network loads or SLA violations, complicating optimization efforts [25].

Effective RM must also address evolving demands in hybrid, public, and mobile cloud environments, adapting to varying requirements while ensuring optimal utilization of available resources. A critical component of this involves understanding the various payment models employed by cloud providers, such as Amazon Web Services (AWS) and Google Cloud Platform (GCP).

AWS adopts a dynamic pricing approach, allowing users to pay only for the services they consume through its Pay-as-You-Go model, which charges users based on actual usage without any long-term commitments [26]. In case of predictable workloads, AWS offers Savings Plans and Reserved Instances, which provide significant discounts in exchange for a commitment to specific usage levels over one or three year periods [27], while Reserved Instances are particularly suitable for steady state applications, offering guaranteed capacity and cost savings [28]. For workloads that can tolerate interruptions, AWS provides Spot Instances, leveraging unused EC2 capacity at reduced rates [29].

Similarly, GCP offers its Pay-as-You-Go pricing structure, which allows users to pay only for the resources consumed similary to aws, without upfront fees or termination charges [30]. For users with steady usage patterns, GCP provides Committed Use Discounts, which offer substantial discounts for one or three-year commitments [31]. Additionally, GCP automatically applies Sustained Use Discounts for workloads running consistently during the billing period, ensuring cost efficiency without the need for explicit commitments [32]. Finally, GCP offers Preemptible VMs, which are short-lived compute instances available at a fraction of the cost, ideal for non-critical and interruption-tolerant applications, such as batch processing or machine learning model training [33].

Knowing that cloud providers like AWS and GCP offer flexible pricing strategies tailored to different user needs, attackers can easily study and exploit these systems, the public availability of such pricing information makes it easier to develop attacks designed to exploit the costs.

These pricing strategies exemplify how cloud providers balance flexibility and cost optimization. However, the same mechanisms that enable scalability and affordability also present vulnerabilities. For instance, attackers could exploit the public availability of Spot Instances or Preemptible VMs, orchestrating resource-intensive operations to inflate cloud costs, thereby executing an EDoS attack. Understanding these vulnerabilities is crucial to developing robust mitigation strategies, as will be discussed in the following section.

2.3 Mitigations and Defenses

The increase in DDoS and EDoS attacks over the years has given an incentive to significant research and development efforts to counter these threats. This chapter is focused on the most notable mitigation and defense strategies that may, present the better results or contributed the most to future mitigations. Those mitigations will be in chronological order if possible, highlighting their effectiveness, advantages, and drawbacks. By examining these solutions, this chapter aims to provide a comprehensive understanding of the progress made in combating these attacks and the areas where further improvement is needed.

2.3.1 Current Mitigation Strategies and Their Effectiveness

The dangerous evolution of DoS, DDoS, and EDoS attacks has driven researchers to develop diverse mitigation strategies over the years. These strategies range from reactive solutions like Secure Overlay Services to more advanced and proactive frameworks such as Enhanced DDoS-MS and EDoS-ADS. Each approach has aimed to address specific vulnerabilities, balancing security, performance, and user accessibility. This subsection will provide a chronological review of these mitigation techniques, analyzing their evolution, strengths, and limitations in the context of modern cloud environments. This analysis will highlight the ongoing challenges in defending against stealthy and economically driven attacks, emphasizing the need for adaptive and intelligent defense mechanisms.

Starting on the DDoS, several techniques have been proposed to mitigate those attacks, among these, Secure Overlay Services (SOS) (2003) [34] is recognized as a reactive approach. According to [35], SOS was the first solution to utilize overlay techniques, which redirect incoming packets and conceal the location of protected web servers to defend against DoS attacks. However, SOS has limitations: it restricts access for legitimate and anonymous users and makes reaching the protected server challenging. Despite this, it effectively safeguards web servers by requiring attackers to expend significant resources to launch a successful attack, which may discourage attackers. A major drawback is that SOS is vulnerable to threats from spoofed IP addresses, which can launch DDoS attacks within internal firewalls, and even latency considerations are largely ignored in this technique.

Building on SOS, a proactive approach called WebSOS was proposed by [36] (2003) . While maintaining the same structure as SOS, WebSOS now integrates a Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) to strengthen the authentication process. However, once more, it does not take in consideration latency issues.

Another reactive technique, Kill-Bots [37] (2005), functions as a kernel extension designed to protect web servers from application layer DDoS attacks. Similar to WebSOS, it employs CAPTCHA for client verification, while also modifies the TCP three-way handshake, delaying socket creation until the handshake is completed, thus mitigating flooding attacks. Despite its benefits, Kill-Bots introduces complexity by allowing multiple CAPTCHA attempts and incorporating mechanisms like bloom filters and admission control. Moreover, applying its protective measures directly on the server, rather than at the network edge (firewall), increases risks. The latency impact of these factors is also significant.

Then we have the DaaS framework [38] (2011) that proposes a metered resource pool that surpasses the capacity of botnets, simplifying the management of idle resources. However, DaaS suffers from elevated latency compared to earlier mechanisms. Additionally, relying solely on puzzles as a countermeasure proves insufficient due to their inherent limitations.

Lastly, there is The Cloud-Based Attack Defense System (CLAD) (2012) that is designed to protect web servers using a robust security framework that functions as a network service, working on the network level. Operated within a cloud infrastructure acting as a supercomputer, it defends against flooding attacks by mitigating network-layer threats targeting CLAD nodes, which act as web proxies running on applications or virtual machines [39]. The system consists of a network of CLAD nodes and a DNS server, with each node implementing mechanisms like admission control, congestion management, network-layer filtering, authentication, and pre-emption. The protected server remains concealed from the public, allowing only traffic from CLAD nodes to access it. The DNS server responds to client requests with the IP address of a CLAD node, while only the nodes themselves are aware of the protected server's actual IP address. To maintain system reliability, CLAD nodes exchange health status information, which is managed by an authoritative DNS server that dynamically allocates healthy nodes in real-time. This ensures users are directed to operational nodes seamlessly. Session management relies on HTTP session keys stored in session tables, with their size determined by the number of concurrent users, where admission control is enforced by limiting the creation of new session keys, which are required for access and are either stored as cookies or generated through hashing the user's IP address and an expiration time using a private hash function. When a client sends a request, the DNS server directs it to a healthy CLAD node. The node verifies the client through a graphical Turing test, issuing a session key upon successful validation. Verified requests are then forwarded to the protected server. While effective in filtering out illegitimate traffic, CLAD introduces latency since all client packets must pass through the overlay system. The infrastructure ensures only valid HTTP requests reach the target server, discarding other traffic types. However, CLAD's design makes it more suited for small enterprises due to scalability limitations and the added latency from its layered architecture.

From this period on, this section will address the mitigations already implemented for EDoS, in the order in which they were implemented and their relevance, trying to describe the most important ones.

Self Verifying Proof of Work (sPoW) is an early-stage approach designed to mitigate EDoS attacks by integrating smoothly with existing cloud technologies. It was proposed by Khor and Nakao [40] in 2009 where the sPoW functions as a DNS-like mechanism, that instead of returning an IP address, like normally would, it provides a cryptographic puzzle instead, alongside an encryption key. In this way, the clients interact with servers through plugins embedded in static web pages, the server plugin generates puzzles by creating temporary encryption channels and sends these to the clients as requests to solve. Upon solving the puzzle, a secure communication channel is established between the client and the server for information exchange, but if a puzzle solving attempt fails, the server escalates the difficulty of the crypto-puzzle, aiming to prioritize legitimate connections. While innovative, the sPoW model has not been evaluated with specific metrics to validate its effectiveness, the authors even acknowledge certain limitations, including the risk of "puzzle accumulation attacks," where attackers bypass solving crypto-puzzles, and a higher false positive rate that may deny legitimate users access due to puzzle complexity. Additionally, the computational overhead of solving puzzles could increase response time, introducing delays. Despite these challenges, the adaptability of sPoW to cloud environments has gathered attention as a promising direction in mitigating EDoS attacks.

Idziorek and Tannian (2011) [41] focused on the vulnerabilities within the current cloud utility pricing models, highlighting how EDoS attacks, through fraudulent resource consumption, can gradually impact cost structures. Their study utilized the pricing model of Amazon EC2 to assess the financial implications of aggregate EDoS attacks over a defined time frame. To address these challenges, the authors proposed two detection methodologies: (i) Zipf's Law, which identifies anomalous traffic patterns, and (ii) entropy-based detection, which examines the nature of individual user traffic by analyzing the entropy of session lengths over a specific period. Both approaches were evaluated in terms of their false positive and false negative rates independently. However, this approach has some limitations, including the presence of false positives, the lack of a defined response time, minimal consideration of CPU utilization, and associated implementation costs.

One notable solution to mitigate EDoS attacks is the EDoS-shield (2011) [42], which introduced the Verifier nodes (V-nodes) and Virtual Firewalls (VFs) as core components for attack detection and mitigation. The V-nodes are evaluating each client request by verifying its legitimacy, doing this using CAPTCHA responses. Based on the outcome, requests are categorized into blacklists or whitelists, allowing the Virtual Firewalls to either block or forward them to the cloud. We can see the proposed architecture in Figure 2.4

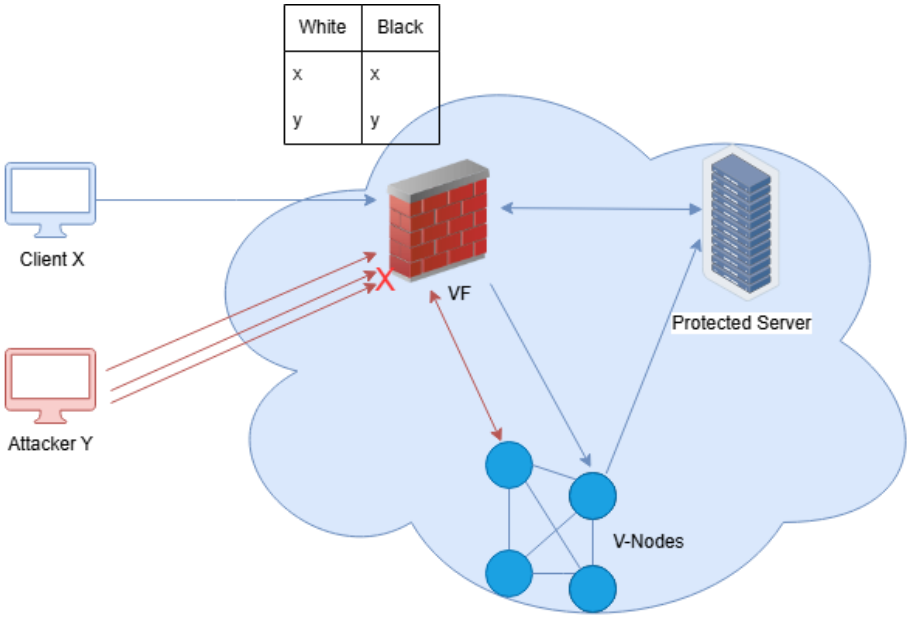

Figure 2.4: Proposed EDoS-Shield Architecture.

The Figure 2.4, illustrates the system's components and their interactions in mitigating EDoS attacks.

In this architecture, a Client X sends requests towards a Protected Server. These requests first pass through a Virtual Firewall (VF), which acts as a gatekeeper. An Attacker Y is also shown, attempting to send malicious requests to the VF.

A crucial part of the system is the V-Nodes, depicted as a cluster of interconnected nodes within the cloud. These V-Nodes evaluate client requests, by verifying their legitimacy using CAPTCHA and based on this evaluation, requests are categorized as indicated by the small table above the VF with White and Black lists.

The VF, informed by the V-Nodes, then either blocks incoming requests or forwards legitimate ones to the Protected Server.

An improved version of this is the Enhanced EDoS-shield (2012) [43], expanded on this approach by incorporating additional data, such as the Time to Live (TTL) values of IP packets, alongside IP addresses. This enhancement allowed for more precise updates to the blacklists and whitelists and informed more effective request handling.

Similarly, in the methodology proposed in [44] the same authors built on their own foundation of [42] and [43], but with two significant additions: an IPS to detect malicious files within IP packets and a Reverse Proxy to conceal the cloud server and load balancer from external users. Although it does not evaluate the false rate, the graphical Turing tests and frequent IP list updates makes this approach resource intensive, leading to increased CPU utilization and adds some latency to it. Consequently, while it enhances verification mechanisms, its suitability for real-time scenarios or accessibility for disabled individuals is significantly limited.

Kumar et al. (2012) in [45] suggests another framework in their study, referred to as the In-Cloud eDDoS Mitigation Web Service (Scrubber Service). This service pretty much relies on the creation and validation of cryptographic puzzles with two distinct levels of difficulty, customized to verify clients based on the nature of the attack targeting the protected system. This framework operates in two modes: the suspected mode and the normal mode, applying different levels of examination depending on the perceived threat level.

While innovative, this approach is heavily reliant on puzzles, which present their own inherent limitations. For example, puzzles may increase computational overhead on both the server and client sides, potentially impacting the user experience for legitimate users. Moreover, since the framework requires the verification of every packet, it introduces significant delays, particularly in high-traffic scenarios. This response time issue remains a critical drawback, as the latency introduced by continuous packet verification may compromise the usability and efficiency of the service, particularly in real-time applications. While the Scrubber Service demonstrates a unique approach to client verification and EDoS mitigation, its focus on puzzles as the primary mechanism limits its scalability and suitability for environments where low latency and high performance are essential. A more comprehensive strategy incorporating additional verification techniques could enhance its effectiveness without sacrificing performance.

EDoS Armor (2013) introduces a dual-layered solution that integrates admission control and congestion control to mitigate EDoS attacks. This approach consist of three core components: (i) a Challenge Server, (ii) Admission Control, and (iii) a Congestion Control.

The Challenge Server provides either image-based or cryptographic challenges to clients initiating connections, and only those clients that successfully solve the challenge are allowed to proceed to the admission control stage. In this second stage, Admission Control rate-limits the connections from authenticated clients, assigns random access keys, and hides the server's port number, further improving security. Finally, Congestion Control allocates server resources among the permitted clients by maintaining a client priority table that tracks the priority levels of each client based on their activity and the type of resources they access [46].

A notable strength of this strategy is its ability to define and manage the number of clients allowed to send requests and to prioritize access based on the nature of the client's activities. This approach can be more relevant for internal organizational networks, where rate-limiting connections from specific networks can prevent resource abuse.

However, this technique has again some limitations. While the use of challenge-based authentication and client prioritization adds layers of protection, it also introduces significant overheads. The port translation process, generation of challenges, and verification of trusted nodes increase computational complexity with CPU utilization, and resource usage. Additionally, relying heavily on a priority-based system might lead to unintended biases, where legitimate low-priority clients are underserved, potentially impacting service quality. This approach will heavily depend on what the company prioritizes.

The Enhanced DDoS-MS framework [47] integrates multiple layers of protection to safeguard VM's from DDoS attacks. This approach incorporates again a firewall, a verifier node,

a puzzle server, an IPS, and a reverse proxy, all strategically placed in front of the protected zone of VM's.

The framework relies heavily on the firewall, which acts as the central control point for the protection process. The verifier node performs graphical Turing tests to validate client requests and reports the results back to the firewall. The IPS inspects incoming packets for malicious activity, while the reverse proxy serves multiple roles: concealing the target server, managing the load balancer, and monitoring traffic rates. To further enhance security, a client puzzle server is deployed to limit the actions of users flagged as suspicious by the reverse proxy.

While it demonstrates a good multi-layered defense strategy, it has some drawbacks. The complex design introduces significant complexity, making it challenging to implement effectively in dynamic and scalable cloud environments. In addition, the reliance on multiple authentication processes, including graphical Turing tests and puzzle-solving, substantially increases latency. This latency can degrade the quality of service for legitimate users, especially in time-sensitive applications, and raise accessibility concerns for users with disabilities. So, while Enhanced DDoS-MS offers a complex and comprehensive solution for mitigating DDoS attacks, its practical adoption in real-world cloud environments is set back by its high design complexity and latency overheads.

Building on the proactive measures of previous approaches, the authors in [48] (2019) implemented three distinct virtual firewalls positioned at various stages along the path to the cloud, aiming to stop attacks as early as possible. To address stealth attacks capable of bypassing these defenses, they incorporated a performance monitoring system, further improving the overall effectiveness of their strategy.

Lastly, Ko et al. (2020) [20] was proposed as an enhanced strategy, integrates several sophisticated methods to distinguish legitimate from malicious requests. It employs network level metrics, URL redirection, and CAPTCHA tests to validate user interactions. The mechanism operates across four modes: Normal, Suspicion, Flash Overcrowd, and Attack, these are based on CPU utilization thresholds and user behavior patterns, aiming to dynamically classify and manage traffic.

Central to its functionality, EDoS-ADS uses metrics like Concurrent Requests Per Second (CRPS) and a Trust Factor (TF), derived from CAPTCHA success, to determine client legitimacy, the way its designed, ensures that legitimate users are always prioritized, and suspicious activities are closely monitored to mitigate resource overuse.

While EDoS-ADS outperforms previous solutions like Auto-Scaling Only and EDoS-Shield in identifying and dropping malicious requests, it still misses some points. The framework heavily relies on predictable CPU thresholds and fixed duration timers, which makes it vulnerable to strategic attacks such as the YoYo attack. This type of attack manipulates the periodicity of malicious requests to avoid detection while forcing unnecessary scale-ups, leading to financial losses.

Furthermore, the dependency on CAPTCHA's for user validation introduces latency and potential accessibility barriers, particularly for users with disabilities or limited computational

resources. Although the Trust Factor adds a sophisticated layer of defense, its reliance on simplistic metrics like response times and request frequency makes it susceptible to circumvention by more sophisticated bots, especially those that employ machine learning models trained to mimic human behavior, which is now more common. It can be considered that, EDoS-ADS represents a significant advancement in EDoS mitigation and is one of the most recent and effective strategies developed to date. However, its practical deployment is limited by predictable scaling behaviors and inherent trade-offs between user accessibility and security. The authors end the paper saying that, they emphasize the need for further improvements to address challenges such as the YoYo attack, saying that they are going to develop more ideas to enhance the security against the YoYo attack.

CHAPTER 3

Solution Architecture

This section describes a general serverless style architecture. It modeles a cloud-native environment hosted on top of a virtualized host, in which a monitoring and metering subsystem is integrated to collect both performance and cost metrics. By presenting this foundational architecture, including a hypervisor layer, a guest VM, and a serverless platform (e.g., Knative or any managed FaaS service), it becomes possible to clearly illustrate how autoscaling behavior is triggered and how attackers exploit it. In the following subsections, the roles of each component are detailed, and the points of measurement for observing CPU, latency, and billing data are highlighted, establishing the foundation for the YoYo attack demonstration and subsequent defense mechanisms. After that, two possible mitigations will be discussed that are easy to implement and lightweight as well.

3.1 Architecture

This overall architecture, illustrated in Figure 3.1, models a cloud style serverless environment running on top of a virtualized host. It is designed to be as general as possible with integrated metering and monitoring subsystems to observe both performance and cost metrics.

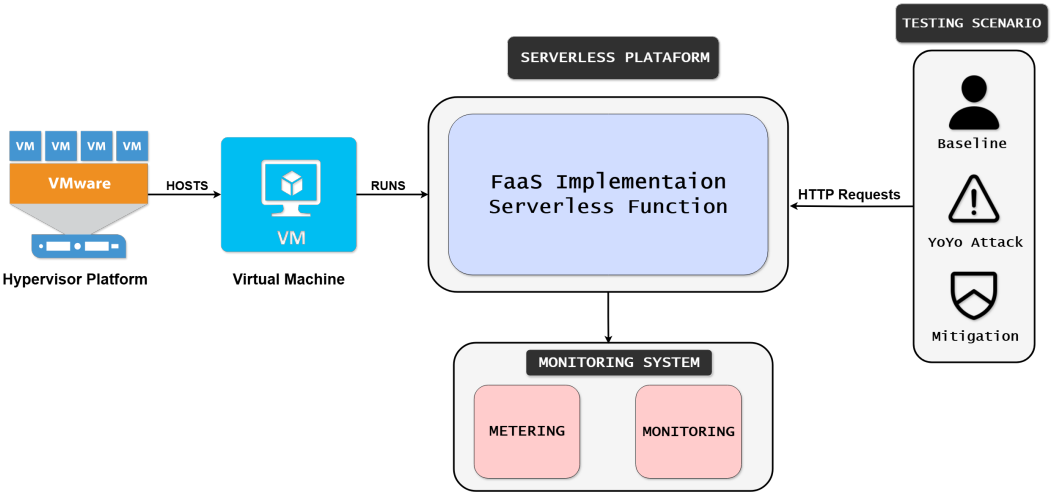

Figure 3.1: General Serverless Architecture.

The architecture includes the following components:

- **Virtualization Layer:** A hypervisor platform that hosts one virtual machine. This layer abstracts physical hardware and enables the deployment of isolated VM instances.
- **Virtual Machine:** A guest operating system instance running on the hypervisor. This VM provides the compute node on which the serverless platform is installed and running.
- **Serverless Platform:** The platform layer within the VM that enables serverless function execution. It receives incoming events or HTTP requests and orchestrates the execution of serverless functions. This could be any kind, like managed offerings: AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Code Engine or self-hosted frameworks like Knative or OpenFaaS.
- **FaaS Implementation:** The core component of the serverless platform responsible for the management, scaling, and routing of the function life cycle, the serverless function itself.
- **Monitoring System:** A dedicated stack to collect operational data:
  - Metering: Gathers resource consumption metrics (e.g., CPU time, memory usage) for cost attribution and billing analysis.
  - Monitoring: Captures performance metrics (e.g., latency, error rates, throughput) and cost data as well.
- **Testing Scenario:** This is where the scripts with the normal user behavior, the attack, and the mitigation would interact with the Serverless platform itself.

3.2 Serverless Function Deployment

The way serverless function are deployed, such as automatic scaling, event driven execution, and simplified application life cycle management, are consistent across various platforms, whether using managed cloud services like AWS Lambda or Azure Functions, or self-hosted solutions like Knative or OpenFaaS. While the implementation details may differ, the core

architectural patterns remain similar. This section focuses on the configuration of Knative, a Kubernetes based serverless platform, to illustrate these concepts in a self managed environment as an example of the ones listed above.

3.2.1 Knative Pod Autoscaler Architecture and Flow

To explore this architecture we need to better understand how the serverless autoscaling behavior is triggered and managed, and deeply understand how the Knative Serving architecture actually works, and analyze the internal flow of the KPA. All of this is shown in Figure 3.2, where the fundamental components of the Knative Serving layer and their interactions are depicted in subsection 3.2.2.

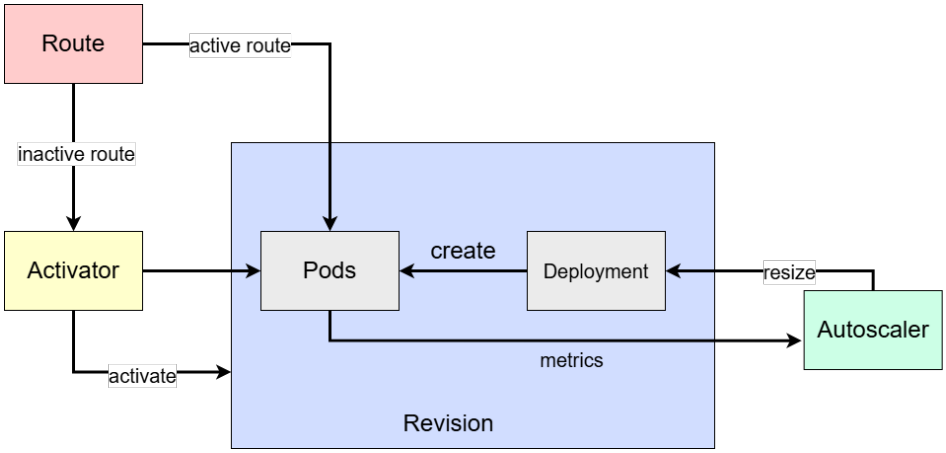

Figure 3.2: Knative Serving Architecture.

The Knative Serving architecture operates in two primary modes, depending if there are any active pods or not. In normal conditions, when the Knative service is scaled to zero (meaning no active pods are running), any incoming request is first routed through the Route component. Upon detecting that the service is inactive, the Route forwards the request to the Activator, then the Activator acts as an intermediary, buffering the incoming requests while simultaneously signaling the need for scaling to the Autoscaler KPA. The KPA then instructs the Deployment controller to scale up by creating new pods based on the latest Revision. Once the new pods are ready, the Activator releases the buffered requests to them, and subsequent requests bypass the Activator entirely, going through the active route, flowing directly from the Route to the newly created pods, to ensure minimal latency.

The Deployment continuously manages the life cycle of these pods, ensuring they match the desired state defined by the latest Revision.

The Autoscaler, which in this case is the KPA, dynamically adjusts the number of pods based on real time metrics. The KPA continuously collects metrics from the pods, like concurrent request counts and target-utilization-percentage (Listing 1), and applies the autoscaling formula (explained in the next subsection) to decide whether to scale the number of pods up or down. When a scaling decision is made, it resizes the Deployment accordingly, depending on the number of pods needed at that moment.

Additionally, Kourier was chosen as the ingress controller due to its simplicity and efficiency, and its a more simple and lightweight solution, making it easier to deploy and manage. It also provides the essential routing functionality required by the serverless function that was used [49] [50]. Plus, Istio proved to be more complex, with higher overhead and had more resource consumption, adding complexity to the system.

With the Serving flow now clear, the following subsection details the exact configuration used for the Knative function, including the autoscaling formula and its parameters, container image, and related settings.

3.2.2 Knative Function Configuration

Knative is a Kubernetes based platform that allows serverless workloads to run in containers, with features such as automatic scaling and traffic-based deployments. Knative Serving is used to extend Kubernetes by introducing custom resources that manage the life cycle of a serverless function, and will be explored in more detail in the next chapter.

A Knative service handles deployment, configuration, and revision control, automatically creating new snapshots of the application when new updates occur. These revisions are exposed through routes, which manage incoming traffic and can split it between versions as we have seen before. This architecture allows for flexible traffic control and simplified management of serverless applications [51].

The autoscaling of Knative is handled by the Knative Pod Autoscaler or KPA, which adjusts the number of running pods based on real-time traffic. If no requests are received, the system can scale down to zero and then back up as demand increases [52] [53]. To illustrate, listing 1 shows the actual deployed Knative Service manifest:

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: knative-fn4
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: '10'
        autoscaling.knative.dev/min-scale: '0'
        autoscaling.knative.dev/target: '50'
        autoscaling.knative.dev/target-utilization-percentage: '100'
    spec:
      containers:
      - image: boostedlee/knative-fn4:latest
        ports:
        - containerPort: 8080
```

Listing 1: Knative Service YAML manifest deployed.

Knative was configured having auto scaling with some default values to make sure it works for a general scenario (Listing 1). Only the "target-utilization-percentage" was changed to

100, for simple math and easily understanding when a pod would auto scale automatically. KPA uses two key parameters to decide when and how many pods to scale:

1. Pod maximum concurrency: The default maximum concurrency of pods, which was changed to 50.
2. Target utilization: The target-utilization-percentage annotation (which was set to 100) defines the fraction of the pod concurrency that the autoscaler aims to maintain during operation.

The KPA determines the desired number of replicas based on:

$$\text{desired pods} = \frac{\text{current concurrent requests}}{\text{maxConcurrency} \times \text{targetUtilization}} \tag{3.1}$$

So, in this work's example, since 'maxConcurrency=50' and 'targetUtilization=100% ', then a burst of 101 concurrent requests would scale to:

$$\text{desired pods} = \left\lceil \frac{101}{50 \times 1.0} \right\rceil = 2.02 \approx 3 \tag{3.2}$$

Since Knative always rounds up fractional pod counts, a burst of 101 concurrent requests would scale the service to exactly 3 pods. This is an important aspect for an attacker to know, since these are the thresholds that trigger the scaling of pods, consequently increasing the service's cost.

The annotations here (Listing1), "autoscaling.knative.dev/target: 50" and "autoscaling.knative.dev/target-utilization-percentage: 100" make that the autoscaler aims at approximately 50 concurrent requests per pod and should not scale up until this limit is reached [52] [53]. In practice, this means that a single pod will handle up to 50 concurrent requests before the KPA adds another pod. Once the additional pods are scaled up, Knative will use its ingress layer, Kourier (in the case of this work), to manage the routing of requests between the pods [51]. Then those requests are distributed using a 'power-of-two random choices' algorithm, which offers near round-robin balancing while improving scalability and reducing load imbalance compared to simple random or round-robin methods [54] [55].

That said, with only a single pod running, all 51 concurrent requests go to it, as soon as that pod's concurrency hits 50, the autoscaler triggers the creation of a second pod. The 51st request still goes to the first pod until the second pod is ready. So basically, the first pod will process the first 50 requests, and only the overflow will go to the new created pod.

Then once two pods are active, after the second pod becomes ready, new incoming requests are distributed between the two existing pods. With 51 total concurrent requests and two pods, each will get about 25 to 26 concurrent connections. Knative's ingress controller randomly balances connections, so requests are evenly shared across the available pods. Thus, neither pod should remain fully saturated unless there is a sudden new spike, this is very important to understand the results later on.

While this behavior is efficient for managing resources, it also creates opportunities for exploitation, particularly by attacks designed to trigger unnecessary pods or nodes. The next chapter will explore how this scaling logic can be abused through the most recent attack pattern described in the literature on EDoS: the YoYo attack (previously discussed in chapter 2). This attack uses designed traffic patterns to repeatedly trigger scaling actions, leading to excessive resource consumption and inflated operational costs.

After understanding this internal flow and the architecture behind Knative Serving, it can now be possible to explore how the autoscaler could be manipulated through the YoYo attack, by examining the way Knative reacts to bursts and idle periods. This process and its results will be discussed in detail in the next chapter.

3.3 Monitoring and Metering Tools

For a good and effective evaluation of a serverless experiment, two separate tools are required: one for real time monitoring of system performance, and another for detailed measurement of resource usage and cost attribution. The monitoring tools capture operational metrics such as request rates, function latency, error rates, and pod scaling events, enabling visibility into the platform behavior when different load patterns are running. Metering tools, on the other hand, collect precise data on resource consumption like CPU, memory and even network, and then translate these metrics into monetary cost models, allowing precise analysis of the economic impact. In practice, a combination of a monitoring system like Prometheus or Grafana and a cost metering solution such as OpenCost or Kubecost, provides the comprehensive data needed to correlate performance events with billing metrics and assess both technical and financial dimensions of the EDoS attack. These tools combined, are key to extract the results necessary to evaluate both the effectiveness of the attack, and the proposed mitigation solution.

3.4 Mitigation Strategy

This section proposes and evaluates two mitigation techniques designed to reduce the economic damage of EDoS attacks against the Knative serverless function. Both mitigations adaptively modify the Knative service file to disrupt an attacker's ability to predict and trigger pod autoscaling. The first mitigation implements a lightweight, randomized target adjustment script, while the second one offers an alternative mitigation focused more on service response time, while still reducing the cost of the attack.

3.4.1 Overview

The main idea behind both mitigation strategies is to dynamically vary some configurations in the autoscaling thresholds on the Knative service. By randomizing or adapting this value in response to unusual autoscaling events, the aim is to:

- Increase the uncertainty for an attacker trying to trigger pods at fixed concurrency levels.

- Limit excessive scale-ups by increasing the concurrency threshold when bursts are detected.
- Maintain acceptable response times and cold start behavior for legitimate users.

In chapter 5 each mitigation will be evaluated based on three key metrics: a) the baseline response time delay, which measures the additional latency experienced by legitimate users due to the mitigation logic, particularly in terms of cold starts or limited scale-ups, b) the total cost after the 12 hour scenario, which is the main focus, c) the implementation complexity, based on the amount of configurations required, external dependencies, extra tools needed, and the level of effort required to make it work with the existing Knative deployment.

3.4.2 First Mitigation: Randomized Autoscaling Defense

Mitigation Mechanism

The first mitigation technique aims to detect sudden spikes in the number of running pods by periodically requesting the Kubernetes API. By using the command 'kubectl get pods' and 'wc -l', it counts the number of running pods associated with the Knative service, maintaining a short sliding history window with the number of the recent pod counts (HISTORY_WINDOW = 6). If there is an increase of at least 3 pods compared to the minimum value in that HISTORY_WINDOW, the mitigation activates and starts the defending process. First, it dynamically randomizes the autoscaling target by randomly selecting a new concurrency target between 70 and 90, as represented in Algorithm 1. After that, it adjusts the scale-to-zero grace period to 10 seconds, reducing it by a third of the default value. Finally, it sets the pod retention period to 0 seconds, aiming to make scaling behavior faster and less predictable to the attacker. The final Knative service configurations YAML file, after the mitigation was triggered, should look similar to Listing 2. The values in black, lines 12, 13, 14, were the ones applied in comparison to the default configurations in Listing1.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: knative-fn4
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: '10'
        autoscaling.knative.dev/min-scale: '0'
        autoscaling.knative.dev/scale-to-zero-grace-period: 10s
        autoscaling.knative.dev/scale-to-zero-pod-retention-period: 0s
        autoscaling.knative.dev/stable-window: 30s
        autoscaling.knative.dev/target: '82'
        autoscaling.knative.dev/target-utilization-percentage: '100'
    spec:
      containers:
      - image: boostedlee/knative-fn4:latest
        ports:
        - containerPort: 8080
```

Listing 2: Knative Service YAML manifest deployed.

Once the Knative service YAML file is updated, it is immediately reapplied to the cluster via the command 'Kubectl apply -f file.yaml', and after that change, the script takes a pause to allow the new configuration to be successfully applied, and it resets the pod history to prevent repeated triggers from the same burst.

This rapid and lightweight adjustment forces the attacker to constantly adapt to varying concurrency thresholds, reducing the effectiveness of fixed rate or scripted YoYo attacks, since they are known to have a fixed patterns, while maintaining reasonable service responsiveness for legitimate users, because it does not require one to actually shut down pods, or canceling new upcoming requests.

Algorithm 1 POC 1: Randomized Autoscaling Defense

Global variables:
SERVICE_NAME = `"knative-fn4"` ▷ Name of the Knative service
NAMESPACE = `"default"` ▷ Kubernetes namespace
YAML_FILE = `"knative-service4.yaml"` ▷ Path to Knative service manifest
CHECK_INTERVAL = 50 ▷ Seconds between each pod-count check
CHANGE_THRESHOLD = 3 ▷ Pod-count jump threshold for triggering update
HISTORY_WINDOW = 6 ▷ Number of previous counts to track
SLEEP_AFTER_UPDATE = 5 ▷ Pause after annotation change
TARGET_MIN = 70, TARGET_MAX = 90 ▷ Range for random autoscale target

function GetPodCount ▷ Query running pods for the service
    cmd ← FormatCommand(SERVICE_NAME, NAMESPACE)
    output ← ShellRun(cmd)
    count ← ParseInt(output, default=0)
    return count
end function

function UpdateAutoscaling(new_target) ▷ Load YAML, update annotations, apply
    config ← YamlLoad(YAML_FILE)
    annotations ← Descend(config, {`"spec"`,`"template"`,`"metadata"`,`"annotations"`})
    annotations[`"autoscaling.knative.dev/target"`] ← ToString(new_target)
    annotations[`"autoscaling.knative.dev/scale-to-zero-grace-period"`] ← `"10s"`
    annotations[`"autoscaling.knative.dev/scale-to-zero-pod-retention-period"`] ← `"0s"`
    YamlDump(config, YAML_FILE)
    ShellRun(`"kubectl apply -f "` ‖ YAML_FILE)
end function

function DetectSpike(history) ▷ Return `True` if latest jump '≥' threshold
    if $|history| < 2$ **then**
        return `False`
    end if
    min_count ← Min(history)
    current ← history[-1]
    return $(current - min\_count) \geq$ CHANGE_THRESHOLD
end function

function Main
    history ← []
    current_target ← `None`
    while `True` **do**
        current_count ← GetPodCount
        history.**append**(current_count)
        if $|history| >$ HISTORY_WINDOW **then**
            history.**pop**(0)
        end if
        if DetectSpike(history) **then**
            new_t ← RandomInt(TARGET_MIN, TARGET_MAX)
            while new_t = current_target **do**
                new_t ← RandomInt(TARGET_MIN, TARGET_MAX)
            end while
            UpdateAutoscaling(new_t)
            current_target ← new_t
            history (← [current_count] ▷ Reset history
            Sleep(SLEEP_AFTER_UPDATE)
        end if
        Sleep(CHECK_INTERVAL)
    end while
end function

3.4.3 Second Mitigation: Dynamic Autoscaling favoring Response Time

Mitigation Strategy

As an alternative to the previous mitigation, this one prioritizes maintaining low response times, even under attack, by dynamically adjusting the autoscaling threshold higher in response to sustained spikes. Rather than picking a new target from a fixed range, this method incrementally increases the current concurrency target whenever a YoYo pattern is detected. As a result, the service may experience a slight increase in cost, but never to the extent caused by the attack itself, resulting in a trade-off between performance and cost for the user to consider.

To mitigate the YoYo attack, this mitigation keeps the same detection mechanism logic as before, identifying sudden spikes in pod counts by periodically querying the Kubernetes API. Utilizing the same 'kubectl get pods' and 'wc -l' approach, to monitor running pods for the Knative service, maintaining the HISTORY_WINDOW of 6 recent counts and triggering if an increase of at least 3 pods (CHANGE_THRESHOLD) is observed. However, its defense activation process now introduces significant operational changes.

Upon detecting the attack, this strategy does not randomize the autoscaling target to a fixed range. Instead, it incrementally increases the current autoscaling target, slowly growing the Knative function configurations with the traffic keeping a more stable number of pods while getting attacked. A randomly selected integer between 10 and 15 (INCREMENT_MIN and INCREMENT_MAX) is added to the autoscaling target value that was active immediately prior to the latest attack trigger. Once those mitigation measures are applied, a COOLDOWN_PERIOD of two hours starts, and if no further YoYo attack patterns are detected in two hours, it applies back to the default Knative configurations values shown in Listing 1.

However, if another attack is identified during those two hours cooldown, the mitigation triggers again and the autoscaling target is once again incremented by a random value of 10 to 15 from its already adjusted level, so if the target was at 50, now it will add 50 + random of 10-15, and then for example 66 + random of 10-15, and the two hour cooldown timer resets. This iterative and stateful approach aims to progressively adapt the autoscaler's sensitivity during sustained attack periods while ensuring a better response time compared to the previous mitigation.

CHAPTER 4

Evaluation Methodology

This chapter aims to demonstrate how EDoS attacks can be launched in cloud environments, by taking advantage of autoscaling and pay per use billing models. This chapter will show how a simple and effective EDoS attack can be launched against a serverless function in the cloud, by using the YoYo attack as a variable approach.

This chapter will detail the practical methodology used to evaluate the autoscaling of Knative's function under different load patterns. First, it will describe the metrics collected and the tools employed for visualization, then outline the three test scenarios implemented in the Kubernetes cluster: (1) a baseline run with only legitimate traffic, (2) a YoYo attack layered on top of the baseline load, and (3) the YoYo attack again on top of the baseline, with the mitigation script running. For each scenario, it will explain the workload generation scripts, autoscaling configuration, and data collection procedures. The actual results and analysis are presented in the following chapter.

4.1 Architecture

The proposed architecture of the deployed PoC is illustrated in Figure 4.1. It is designed to emulate a cloud based serverless environment within a private Kubernetes cluster, enabling a controlled demonstration of the EDoS attack and the mitigation as well.

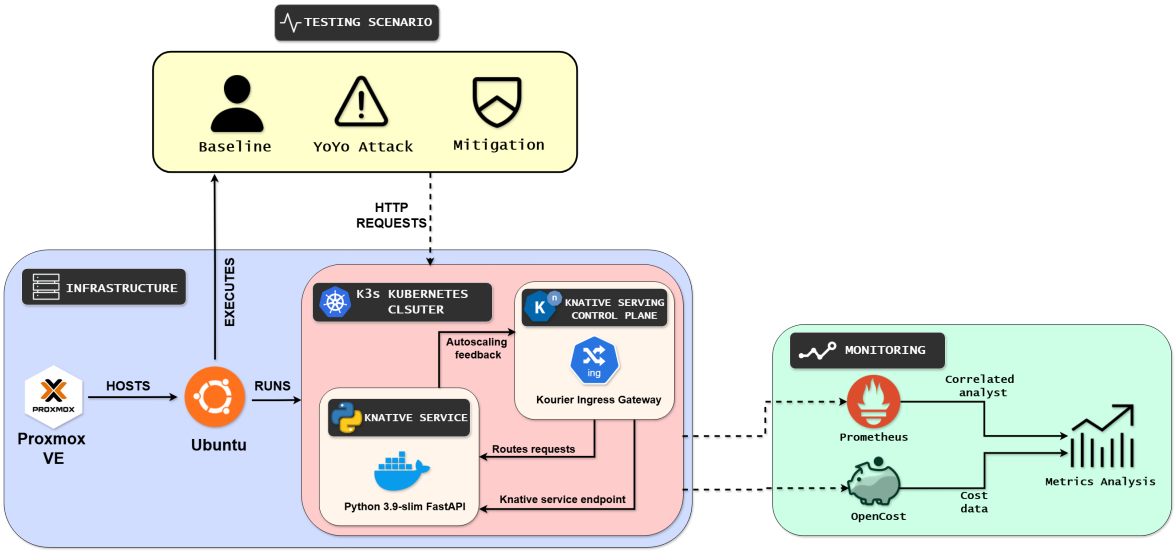

Figure 4.1: POC Architecture.

The architecture consists of these main components:

- **Proxmox VE Host (blue):** A physical server running Proxmox Virtual Environment, which provides virtualization capabilities for hosting the used VM.
- **Ubuntu VM:** A virtual machine deployed on Proxmox, serving as the base OS for the Kubernetes cluster using K3s.
- **K3s Kubernetes Cluster (red):** A lightweight Kubernetes distribution running on the Ubuntu VM. This cluster, orchestrates all containerized workloads and provides core Kubernetes services (API server, scheduler, kubelet).
- **Knative Serving Control Plane:** This is deployed on top of K3s, where Knative serving extends Kubernetes with serverless capabilities. It manages the life cycle of the developed function, traffic routing, and autoscaling via the KPA.
- **Kourier Ingress Gateway:** This is the gateway component for Knative Serving, which is responsible for receiving external HTTP requests and routing them to the Knative function endpoint /fib (since there is only one), and reporting autoscaling metrics back to the control plane.
- **Knative Service (Serverless function):** This is where the serverless function is hosted, in a containerized Python 3.9-slim FastAPI application providing the Fibonacci endpoint (/fib).
- **Monitoring Stack (green):** This is where all the important metrics are hosted. Prometheus and OpenCost are used to collect application and cluster metrics, including pod level cost data. With everything gathered here, the metrics can easily be acceded through Python scripts and extract the results needed.
- **Test Scenarios (yellow):** In this block, is where all Python scripts are ran from the baseline scenario to the YoYo attack and the mitigation scripts.

4.1.1 Physical and Virtual Infrastructure

The experimental setup was hosted on Proxmox servers provided by "IT - Instituto de Telecomunicações". Within this environment, an Ubuntu virtual machine serves as the host for deploying K3s, a lightweight Kubernetes distribution. This configuration enables agile management of containerized applications while ensuring a realistic emulation of a cloud infrastructure behavior, which is an important part. K3s is deployed as the primary Kubernetes engine, managing a cluster of nodes where each pod is containerized using Docker.

With this Kubernetes-based environment set up, the Knative function service that was implemented will now be described.

4.2 Function Service Implementation

The serverless function is a simple FastAPI service that exposes a single endpoint (/fib), that when called, computes the 10th Fibonacci number using an iterative loop. This function is just a simple example of what could be a Knative function. While a more complex or more lightweight function would change the absolute compute time and cost, it would not affect this experiment, because what matters here is the relative scaling thresholds, not the raw cost or latency of the function's logic. In other words, regardless of whether the function is simpler or more complex, the autoscaler will respond the same way to traffic spikes, and the relative cost effects will be similar.

To package the service for Knative, a minimal Dockerfile based on python:3.9-slim was used, taking advantage of the lightweight nature of the K3s nodes. The dependencies from requirements.txt are installed without caching to reduce image size and the application is served via Uvicorn so that it can receive traffic within the Kubernetes cluster. This container image is then pushed to a personal registry and referenced by the Knative Service manifest (Listing 1).

Once deployed, the Knative service will automatically add or remove pods based on the autoscaling settings we defined before.

The Knative function and the Dockerfile can be found in a open source repository at GitHub [1].

4.3 Monitoring Cost and Metrics Collection

4.3.1 OpenCost and Prometheus

Monitoring is the most important key in evaluating the impact of an EDoS attack. For that, an open-source tool like OpenCost was chosen, which provides the system with every single metric needed to track resource usage and operational costs in real time, without relying on any other service or additional deployments. Metrics related to container usage, network traffic, and auto-scaling events are collected via Prometheus, which feeds data into OpenCost, whose access is allowed using the OpenCost API. This integration allows to quantify the cost

[1] https://github.com/NunooFerreira/Tese-edos

and resource impacts of a simulated attack and provides a detailed view of system behavior under stress.

The collected metrics are visualized using dashboard tools integrated with Prometheus, using the OpenCost User Interface (UI), facilitating both real time analysis and historical data review. To help getting these values accurately, Python scripts were developed to obtain the exact values of whatever Knative function needed, by filtering it by name, pod, window, or day.

These scripts interact directly with the OpenCost API and Prometheus endpoints to retrieve detailed cost data for each container or pod. With this, it is possible to correlate the timestamp of each autoscaling event with the corresponding increase in CPU and memory consumption, translating that directly into cost variations. This comprehensive visibility is crucial in a PoC like this, where the goal is to measure how minor, periodic spikes in traffic can lead to high increases in resource allocation and cost.

Additionally, Prometheus also collects and analyzes custom application level metrics, such as request count per second or response latency, allowing for a deeper understanding of the system's performance during and after the attack. These metrics collected will help to validate the effectiveness of the YoYo attack pattern and will later support the evaluation of the mitigation strategy proposed in the previous one.

4.3.2 Metrics Collected and Rationale

To evaluate the impact of both the attack and our mitigation strategies on the Knative function, the following key metrics were collected over each 12 hour test run.

1. **Average number of active Pods** Tracks how many pods are running.
2. **Average CPU utilization (%)** Measures how much of the requested CPU resources were actually used. Helps assess whether autoscaling settings achieve cost-effective use of CPU.
3. **Cost per minute** This shows how the function's expenses change continuously, highlighting rapid cost increases during traffic spikes.
4. **Total cost** Summed over the 12 hour period, for a direct comparison of overall expense between scenarios.
5. **Response time metrics (min, max, average)** Captures only the baseline response time, to understand how the normal user behavior would be affected by either the attack and the mitigation.

- **Pods and CPU utilization:** Queried from Prometheus via its HTTP API:

- Pod Count:

```
count by (namespace) (
  kube_pod_status_phase{
    phase="Running",
    namespace="default"
  }
)
```

- CPU usage:

```
(
  sum(
    rate(container_cpu_usage_seconds_total{
      namespace="default", pod=~"knative-fn4-.*", container!=""
    }[5m])
  )
  /
  sum(
    kube_pod_container_resource_requests{
      namespace="default", pod=~"knative-fn4-.*", resource="cpu"
    }
  )
) * 100
```

- **Cost metrics:** Retrieved from the OpenCost API, which reports per resource cost rates. We recorded both the per-minute rate and the total.
- **Response times:** Logged by the baseline python script (one request per worker per second), where was then extracted the minimum, maximum, and average response times from the logs.

4.4 Attack Script: YoYo variation

4.4.1 Attack description

The YoYo attack was implemented using an asynchronous Python script built on top of the aiohttp [2] and asyncio [3] libraries to simulate bursts of traffic against the Knative serverless function. This asynchronous model allows to simulate high concurrency while maintaining efficiency and scalability on the client side.

The attacking script alternates between two phases:

1. Attack Phase: During this phase, a high number of concurrent HTTP requests, 265 concurrency requests, are sent to the target function endpoint for a fixed duration of 35 seconds. This duration is based on observed testing to ensure that the Knative Pod Autoscaler has enough time to scale up the number of pods in response to the traffic. This value also takes in consideration the "totalEfficiency" of each pod, the lower the value the better. The number 265 was based on the Knative configuration used, where the autoscaling.knative.dev/target was set to 50. Given this threshold, the traffic is expected to trigger a new pod for approximately every 50 concurrent requests, thus, sending 265 concurrent requests ensures that at least four pods are instantiated, exceeding the 200 request mark to guarantee the desired scale up behavior, as shown in Algorithm 2.

[2] https://docs.aiohttp.org/en/stable/
[3] https://docs.python.org/3/library/asyncio.html

2. Cool Down Phase: After the attack phase, the script enters a low traffic idle period of 900 seconds, during which no requests are sent to make sure the pods are scaled down, and some extra time to keep things quietly, simulating a drop in usage, allowing the Knative to scale down as it normally would in behavior (Algorithm 2).

This cyclical pattern alternating between intense and idle periods mimics the exact behavior of what a YoYo attack is, which aims to force the system to scale up and down repeatedly without alerting too much attention, which is especially effective against scale to zero architecture.

The entire simulation runs for 12 hours to demonstrate the economic impact over half a day. This duration was chosen to simplify estimation by doubling the measured cost, usually obtained at a full day estimate, and by further scaling it up the expense for an entire week or even a month can be predicted. The script even logs the response times in milliseconds along with timestamps, enabling correlation with cost metrics retrieved via OpenCost and Prometheus.

This implementation effectively simulates a real world EDoS scenario, where attackers do not aim to crash the service, but rather to keep it operating inefficiently, maximizing cost while maintaining availability.

This methodology establishes the foundation for the experimental evaluation presented in chapter 5. By systematically collecting pod count, CPU utilization, cost, and response-time metrics under each scenario, it becomes possible to observe how the YoYo traffic patterns manifest in measurable resources and billing variations.

In the next chapter, these same metrics are analyzed to evaluate the effectiveness and reliability of the proposed mitigation strategies, comparing their impact on both operational cost and service performance.

Algorithm 2 YoYo Attack

Global variables:
TARGET_URL = `"http://knative-fn4.default.127.0.0.1.nip.io/fib"` ▷ My serverless function endpoint
NORMAL_CONCURRENCY = 0 ▷ Requests per second during cool-down
ATTACK_CONCURRENCY = 265 ▷ Requests per second during attack burst
ON_ATTACK_DURATION = 35 ▷ Seconds to maintain high traffic
OFF_ATTACK_DURATION = 900 ▷ Seconds to wait for resources to scale down
RUN_DURATION = $12 \times 60 \times 60$ ▷ Total experiment time (12 hours)

function MAIN
    start_time ← NOW(()) ▷ Timer
    while (NOW − start_time) < RUN_DURATION **do**
        RUNPHASE(ATTACK_CONCURRENCY, ON_ATTACK_DURATION, TARGET_URL)
        RUNPHASE(NORMAL_CONCURRENCY, OFF_ATTACK_DURATION, TARGET_URL)
    end while
end function

function RUNPHASE(concurrency, duration, url)
    stop_time ← NOW + duration ▷ Calculate when this phase should end
    connector ← INITCONNECTIONPOOL ▷ Open HTTP connections for all workers
    tasks ← [] ▷ Keep track of our parallel workers
    for $i = 1$ **to** concurrency **do**
        ▷ Create a new worker that will repeatedly send requests
        tasks.**append**(LAUNCHWORKER(connector, stop_time, url))
    end for
    WAITALL(tasks) ▷ Hold here until every worker finishes
end function

function WORKER(connector, stop_time, url)
    while NOW < stop_time **do**
        t0 ← HIGHRESTIMESTAMP(()) ▷ Record time just before sending
        SENDGET(connector, url) ▷ Send one HTTP GET to the target
        t1 ← HIGHRESTIMESTAMP(()) ▷ Record time immediately after response
        RECORDLATENCY(t1 − t0) ▷ Store how long the request took
    end while
end function

CHAPTER 5

Evaluation and Results

5.1 Experimental Setup and Metrics

In this chapter, we will present the experimental results obtained over twelve hour test runs for each scenario: (i) baseline, (ii) YoYo attack, and (iii) mitigation strategies, and use those data to assess both the effectiveness and validity of the mitigation strategies proposed earlier. By comparing pod counts, CPU utilization, cost rates, and response-time metrics across these scenarios, it is demonstrated how well each approach balances operational cost reduction with service performance.

All tests performed against the Knative function were supported by a FastAPI with a Fibonacci endpoint.

In chapter 4, these metrics were collected for each different scenario, to give a good understanding on what happened:

1. Average number of active Pods
2. Average CPU utilization
3. Cost per minute
4. Total cost
5. Response time metrics (min, max, and average)

5.2 Baseline Scenario

The evaluation started by running the baseline simulation, triggering the Knative function for 12 hours with a constant concurrency of 1, to emulate a legitimate user behavior, making periodic requests every second.

5.2.1 Autoscaling and Cost

1. **Number of active Pods:** As expected, Knative only scaled 1 pod consistently, in those 12 hours Figure 5.1.

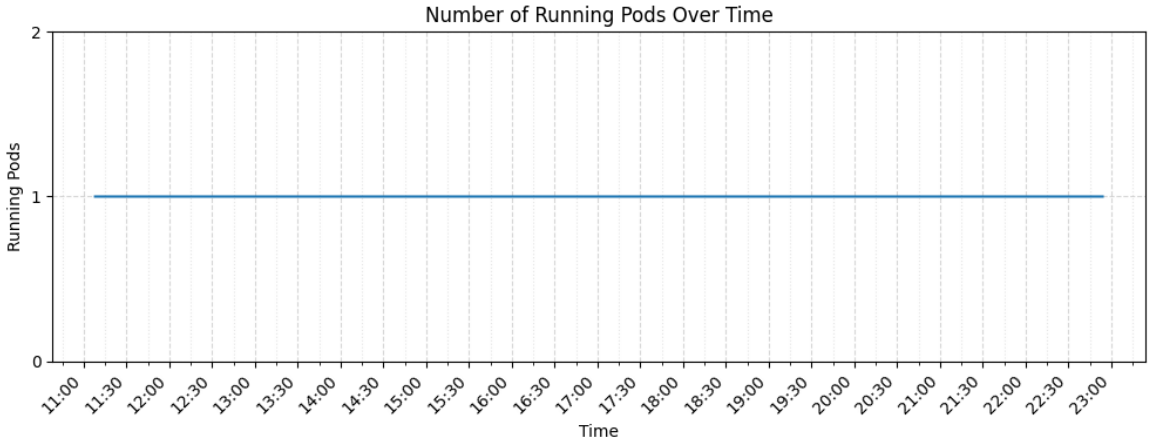

Figure 5.1: Baseline - Number of Active Pods.

2. **CPU usage:** The CPU usage of the one pod remained low and stable with a constant value of approximately 17% corresponding to an average actual usage of 0.00716 vCPU against a 0.025 vCPU request, resulting in a CPU efficiency of 28.65 % seen in Figure 5.2.

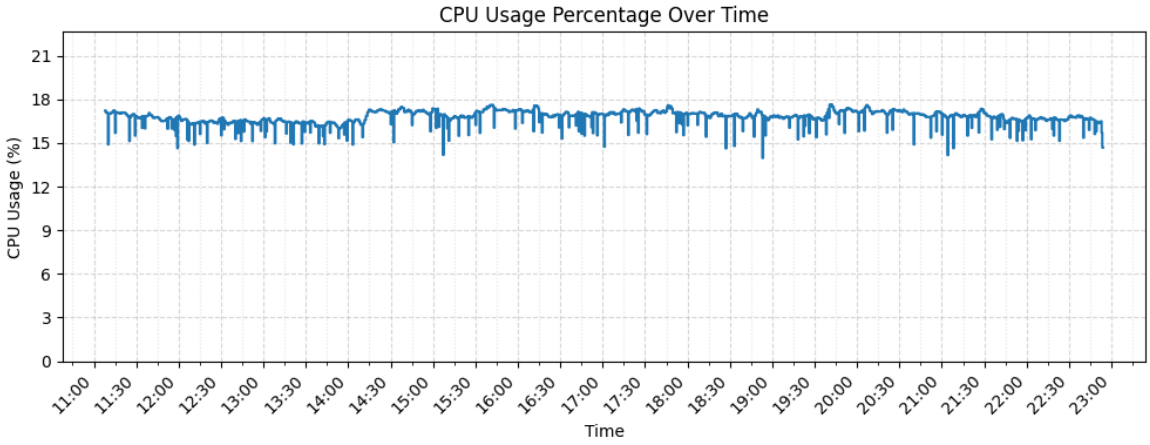

Figure 5.2: Baseline - CPU Usage.

3. **Cost rate:** Since the delta cost reflects the variation in cost per minute, and there was consistently only 1 pod running, there was effectively no variation. As a result, the cost per-minute chart in Figure 5.3 remains a perfectly flat line at approximately 1.82×10^{-5} \$/min.

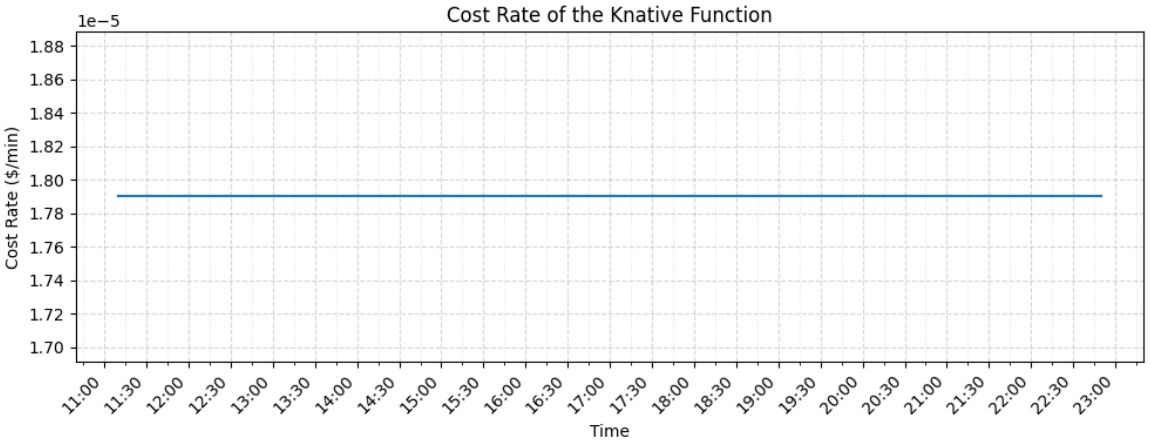

Figure 5.3: Baseline - Cost Rate Baseline.

4. **Total cost:** The total cost of the single pod was 0.013$.

5.2.2 Performance Metrics

This subsection presents legitimate request response times for each scenario. For each one, the maximum, minimum, and average time taken to complete those requests over the 12 hour period were measured. These metrics will provide a good insight into performance variations and will be crucial for evaluating the effectiveness of the mitigation strategies discussed later on this chapter.

All requests succeeded with an HTTP code of 200 so there were no misses. The response time plot depicted in Figure 5.4 shows every request to the Knative function response time over the 12 hours, with cubic spline interpolation [1] highlighting the consistently steady performance.

[1] https://scipy.org/

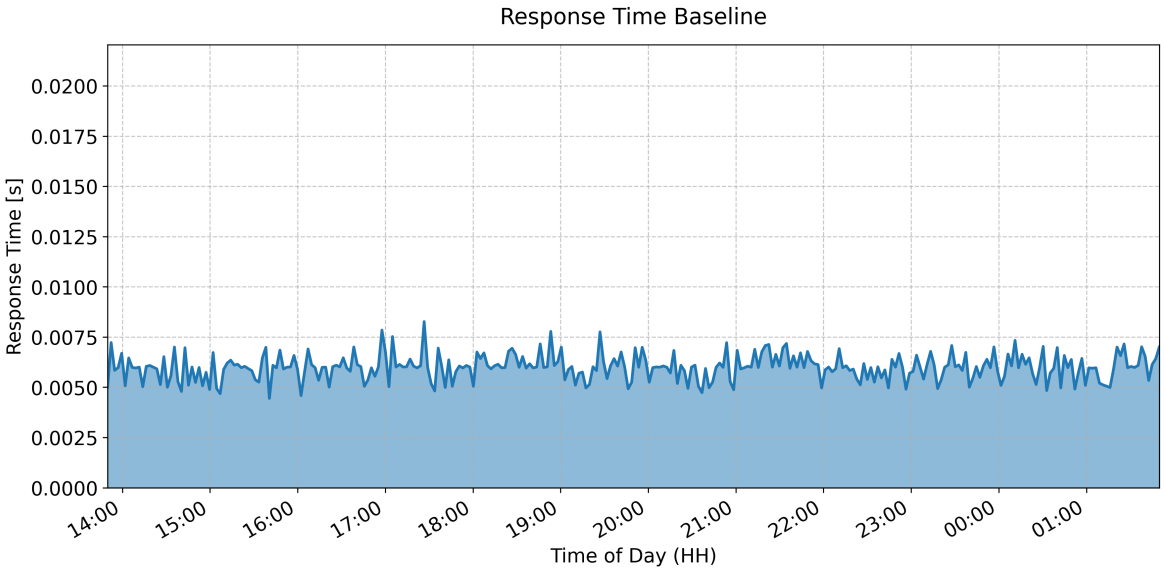

Figure 5.4: Baseline - Response Time.

Table 5.1 shows that the maximum recorded value was 0.021s, which is relatively low, with a minimum of 0.004s and an average of 0.005972s, which will serve as the baseline for future results.

Metric	**Baseline (s)**
Minimum	0.004
Maximum	0.021
Average	0.005972

Table 5.1: Baseline Response Time

The histogram of response times in Figure 5.5 shows a tight distribution between 0.004s and 0.008s, with a median of 0.006s.

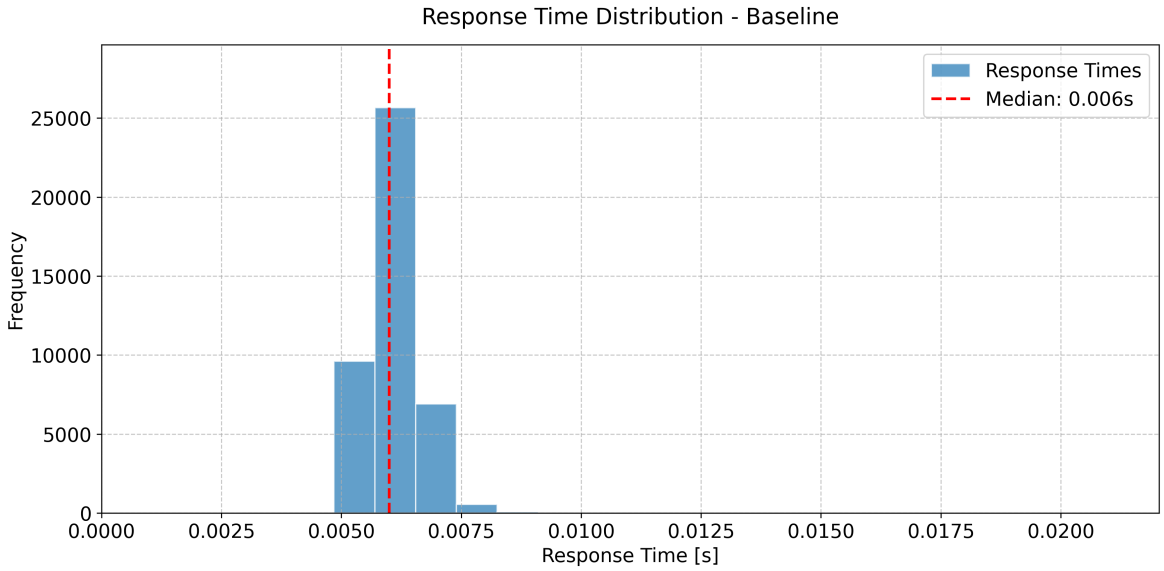

Figure 5.5: Baseline - Response Time Distribution.

So in the baseline scenario, the service shows perfectly stable autoscaling behavior (no unexpected scale-ups), low and consistent CPU utilization, low cost, and fast responses. These values will be used as a baseline for other scenarios.

5.3 YoYo Attacking Scenario

The YoYo attack scenario replicates the baseline setup, but introduces the attack traffic designed to trigger rapid autoscaling events. The same metrics were collected over the 12 hours.

5.3.1 Autoscaling and Cost

Number of Active Pods: Knative consistently scaled up to 5 pods for each attack cycle. We can see 47 spikes in Figure 5.7 because in the 12 hour (43200 s) window, the first burst starts after 2 minutes (120 s), leaving 43080 s of active time. Each attack cycle takes:

$$35\,\text{s} + 900\,\text{s} = 935\,\text{s}, \tag{5.1}$$

so:

$$\frac{43080}{935} \approx 46.07 \text{ full cycles} \tag{5.2}$$

Each spike represents a burst of 265 concurrency requests in a time spam of 35 seconds, leading KPA to cold start 5 more pods leaving them to run for approximately 60 seconds, which is the default value of the scale to zero grace period.

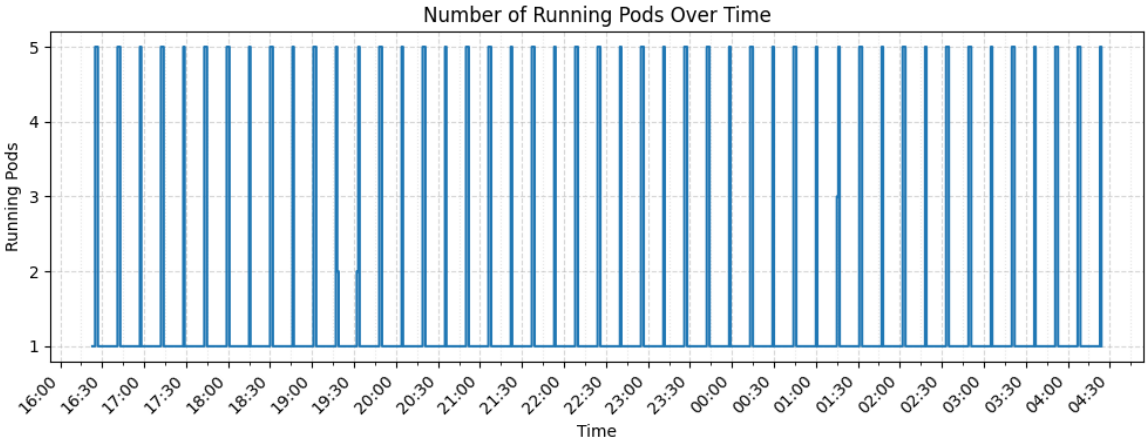

Figure 5.6: YoYo - Number of Active Pods.

CPU usage: In Figure 5.7 we can clearly see the YoYo pattern of the CPU utilization over time. At the start of a burst, the CPU jumps nearly from 17 % to a peak of 480 %, the expected value, as Knative scales up 5 pods (each using about 100 % CPU). Then, just as those pods finish their cold starts, the attack ends and utilization goes back to 17%, the baseline average CPU usage.

In some bursts we only see a total CPU usage between 120-320 % because, after the first pod's concurrency hits 50, the autoscaler spins up a second pod but only observes that new pod once it's actually ready and during that short interval most of the work still sits on the original instance. When more pods keep scaling up, each one can momentarily spike toward 100 % before the next polling interval, sometimes causing aggregate peaks of around 480 %. Another reason is that because the 35s burst of the attack, often ends before all 5 pods can join, some pulses only trigger 1 - 2 new pods (120–240 % CPU), others manage 3 – 5 pods (360 – 480 %). Lastly, the low value of the target utilization threshold affected the autoscaler's behavior, because if it had been better optimized, the autoscaler would have had enough time to scale up all five pods, and we would have seen peaks consistently reaching the expected 480 %.

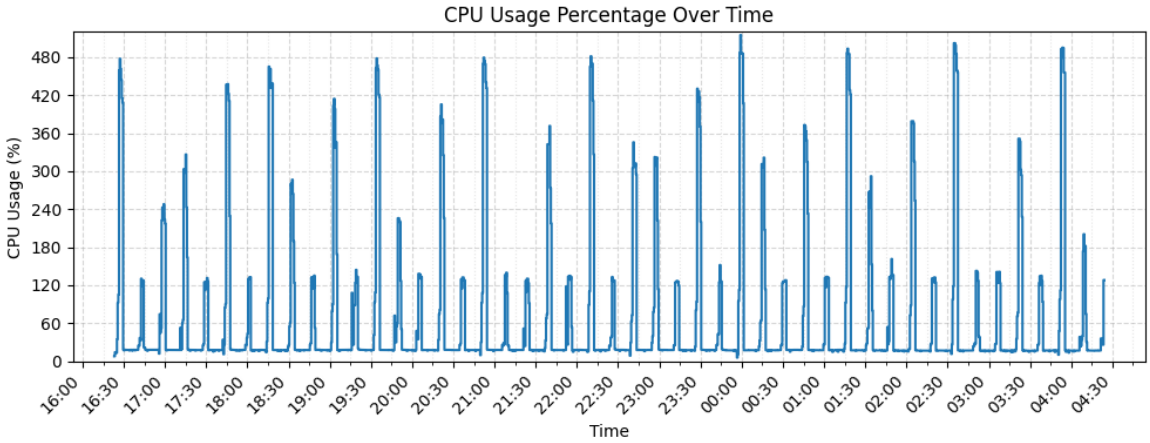

Figure 5.7: YoYo - CPU Usage.

Cost rate: Here in Figure 5.8, we can clearly see the variation in the delta cost showing how effective the attack was on the cost part:

- Negative drops down to –0.00010$ to -0.00015$ per minute correspond to those same pods idling out of the grace period and being torn down so the instantaneous differential cost goes negative.
- Positive spikes of roughly 0.00004$– 0.00006$ per minute occurs each time the YoYo's burst forces Knative to cold start 4 extra pods.
- Between bursts, on the silence phase, the cost rate is nearly zero, just like we saw on the baseline.

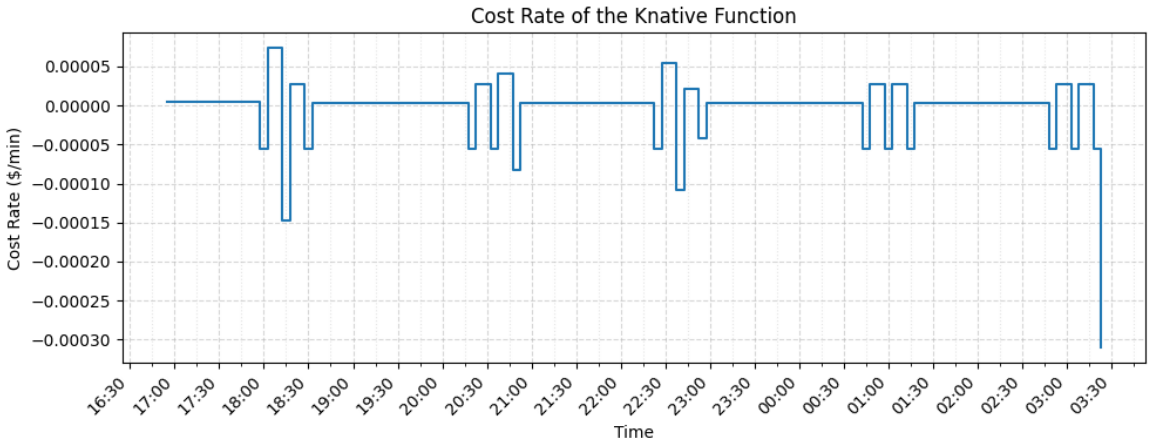

Figure 5.8: YoYo - Cost Rate.

Total cost: 0.02434$.

As shown in Table 5.2, the total cost nearly doubled compared to the baseline of 0.013$, reflecting an 87.2 % increase, showing how effective this attack was in terms of cost.

Scenario	Cost ($)	Increase
Baseline	0.013	–
YoYo Attack	0.02434	+87.2 %

Table 5.2: Cost comparison between Baseline and YoYo Attack Scenario

5.3.2 Performance Metrics

To understand how this attack affected the baseline behavior, we are going to compare the Max, Min, average metrics while the YoYo attack was happening.

It is important to understand that the primary goal of this attack is not to impact response times. On the contrary, the less noticeable the performance degradation, the better. This allows the attack to remain stealthy while gradually increasing operational costs, making it harder to detect. In Figure 5.9 we can clearly see it affected the average response time of the baseline when the YoYo was happening compared to the Figure 5.4 that we just saw before.

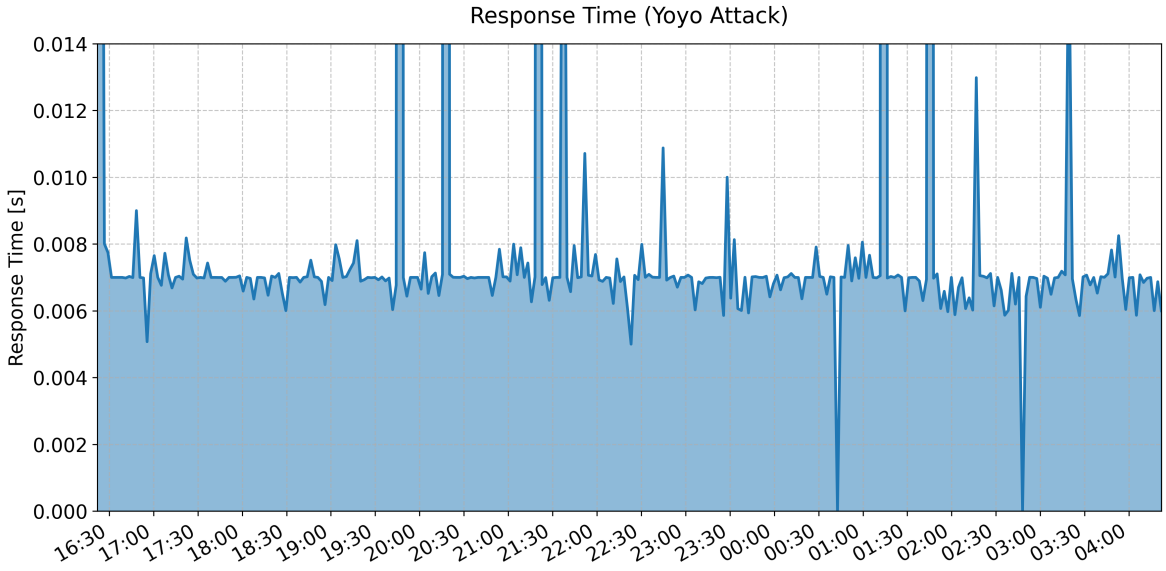

Figure 5.9: YoYo - Response Time.

As shown in Table 5.3, the maximum response time increased significantly from 0.021s to 2.208s, while the average response time went up by 86.12 %.

Metric	Baseline (s)	Baseline under Attack (s)
Minimum	0.004	0.002
Maximum	0.021	2.208
Average	0.005972	0.011116

Table 5.3: Baseline Response Time Comparison

In Figure 5.10, we see that the distribution shifted from the old median of 0.006s to 0.007s and the values are way more spread out. Overall it had a 16.7 % increase in median response which since is not that much, it could be easily go unnoticed.

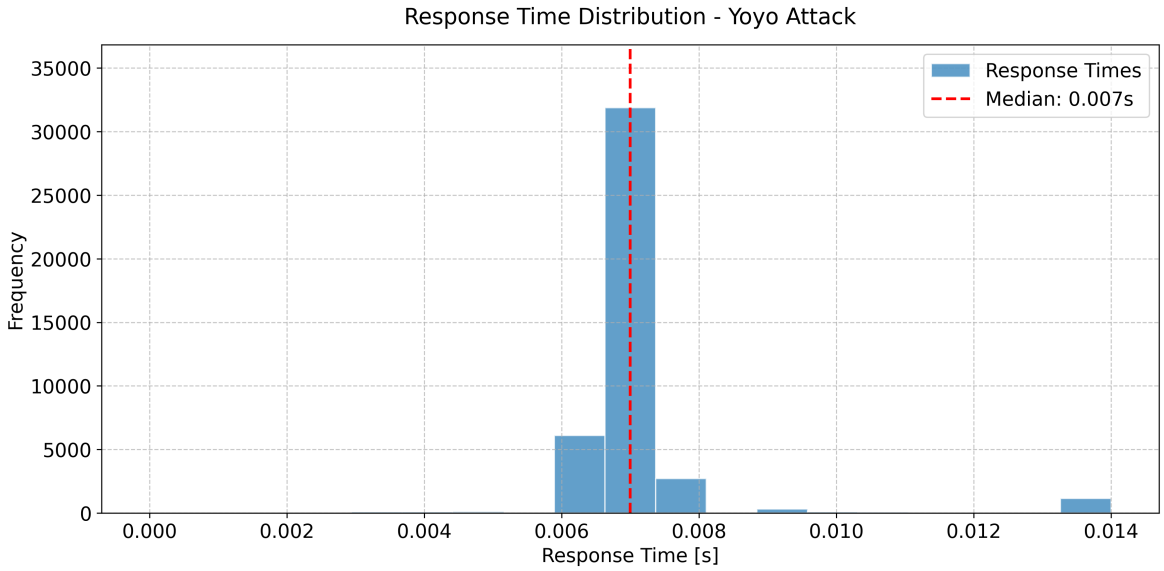

Figure 5.10: YoYo - Response Time Distribution.

The results demonstrate that the YoYo attack was successful in achieving its primary objective, an EDoS attack. Throughout the 12 hour period, the attack was able to systematically manipulate the Knative autoscaler, forcing multiple rapid scale up events even though the workload didn't really need it, leading to a highly inefficient use of resources. This inefficiency is clearly reflected in the total efficiency dropping dramatically to 7.51 %, compared to 24.4 % in the baseline scenario.

From a cost perspective, the attack nearly doubled the total operational expense of the service, increasing the total cost by approximately 87.2 %. This confirms that even with relatively low intensity and stealthy bursts, can cause a significant economic cost over time without overwhelming the system's availability.

In terms of performance, while the attack introduced spikes in resource utilization, it did not visibly degrade the response time for legitimate traffic during the observation window, and maintaining service availability is a characteristic that aligns with the stealthy and sustainable damage goals of an EDoS attack.

Overall, the experimental results validate that the YoYo attack was effective in creating a significant financial problem in a serverless function environment.

5.4 Mitigation Scenario

5.4.1 POC 1

The mitigation scenario replicates the setup of the YoYo attacking scenario but introduces a custom mitigation Python script designed to counteract the effects of that attack using the first strategy. This script modifies the behavior of the Knative autoscaler in order to reduce unnecessary scaling events triggered by the attack traffic, and the goal is to reduce the

economic impact of the attack while preserving service availability and acceptable performance levels. As in previous scenarios, the same metrics were collected over the 12 hour period.

Autoscaling and Cost

Number of Active Pods: As expected, the mitigation strategy led to a more controlled autoscaling behavior. After the mitigation was applied, Knative only scaled up to 3 pods consistently throughout the 12 hour period. Initially, there was a peak of 9 pods, which can be explained by the need to apply the kubectl commands and update the new YAML configuration, which briefly triggered the scaling of additional pods.

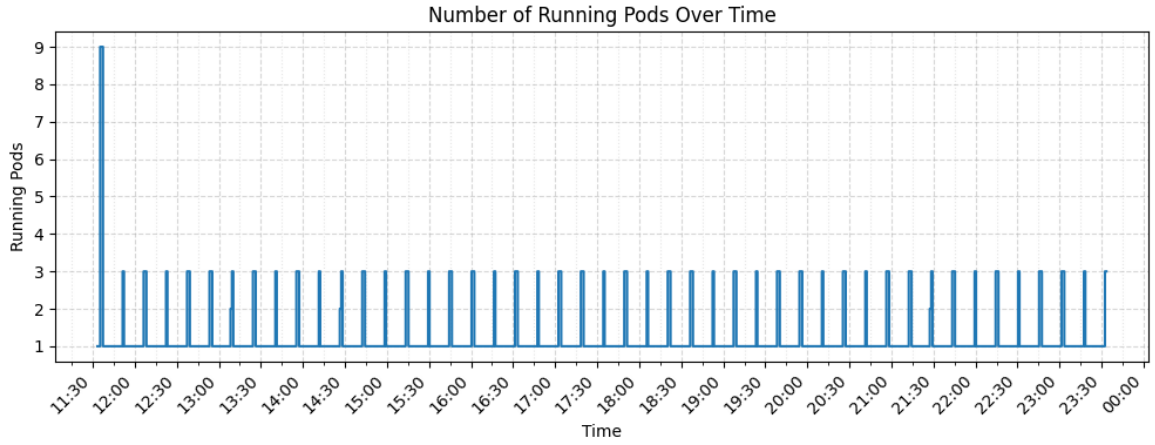

Figure 5.11: Randomized Autoscaling Defense - Number of Active Pods.

CPU usage: The graph shows an initial spike that reached approximately 818 % at the beginning. This burst is caused by the fact that the mitigation has to apply all those changes, which triggers the creation of new pods. Following this, the CPU usage stabilizes around 280 % or less, showing a more controlled usage across the active pods throughout the 12 hour window compared to the Figure 5.7 in the YoYo attack scenario.

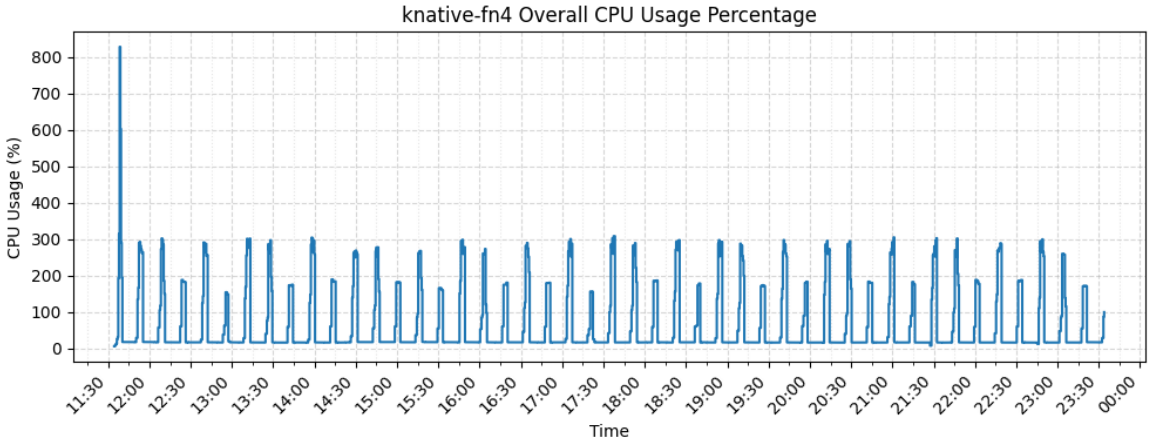

Figure 5.12: Randomized Autoscaling Defense - CPU Usage.

Cost rate: In Figure 5.13, we can see that the cost-rate curves shows a more stable variation than the YoYo attack scenario shown in Figure 5.8. In particular:

- Negative drops are much smaller. Rather than seeing drops like -0.00010$ to –0.00015$ per minute as idle pods shut down, these values now are around -0.00002$ per minute.
- The spike amplitude is much lower as well. Instead of +$0.00004–$0.00006/min bursts forcing four cold starts, the mitigation keeps positive peaks to well under +0.00001$ per minute, that results from fewer instant pod launches.
- Between the bursts, on the silence phase, the cost rate is again zero, just like the other two scenarios.

Overall, these smoother changes show that the mitigation strategy successfully reduced the YoYo spikes in both directions, limiting cold starts and reducing pod shutdowns. This made the cost rate more stable, without aggressive spikes in cost.

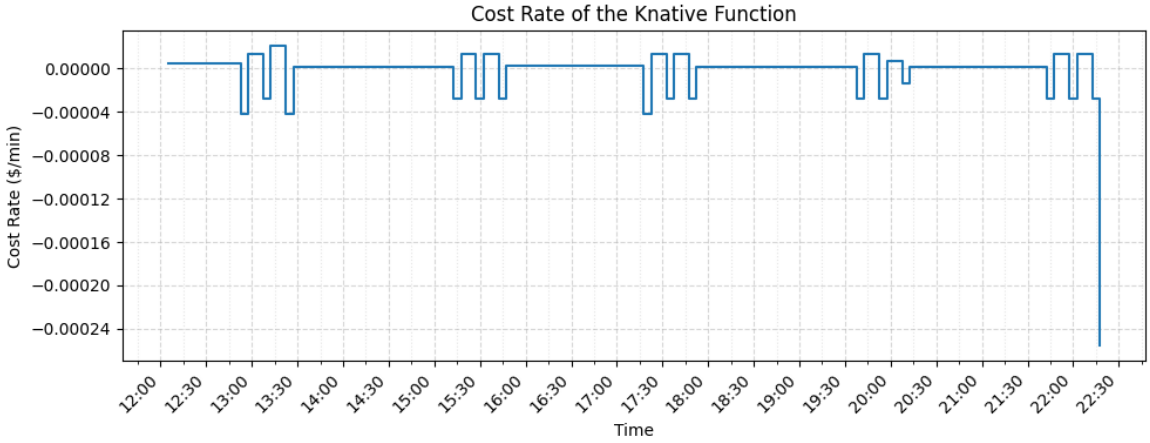

Figure 5.13: Randomized Autoscaling Defense - Cost Rate.

Total cost: $0.01846. The most critical metric for evaluating a good EDoS mitigation is the total cost during the 12 hour period. As previously shown in Table 5.2, the attack scenario led to a total cost of $0.02434, representing an increase of 87.2 %. With this mitigation scenario, the total cost dropped significantly to $0.01846, This represents a reduction of approximately 24.1 % compared to the unmitigated YoYo attack, demonstrating that the mitigation was successful in lowering the financial impact:

Scenario	Cost ($)	Increase from Baseline	Reduction from Attack
Baseline	0.01300	–	–
YoYo Attack	0.02434	+87.2 %	–
YoYo + Mitigation	0.01892	+45.5 %	-22.3 %

Table 5.4: Cost comparison between Baseline, YoYo attack, Mitigation

Performance Metrics

To evaluate the effectiveness of this mitigation strategy, we compare again the same metrics in the Baseline, YoYo Attack, and Mitigation scenarios.

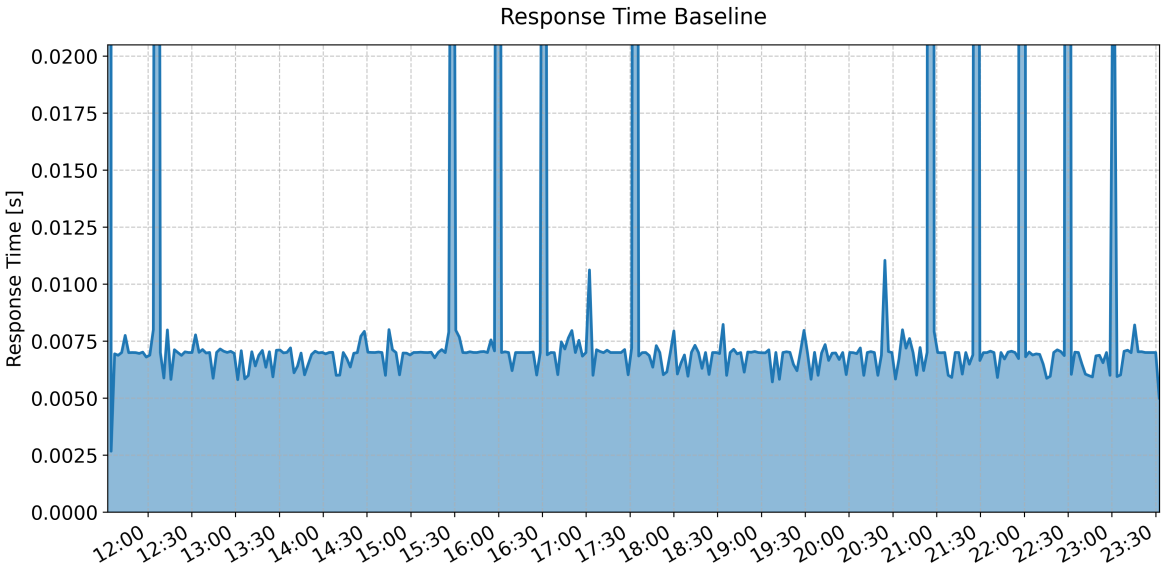

Figure 5.14: Randomized Autoscaling Defense - Response Time.

In Figure 5.14 and Table 5.5 it can be observed the response time behavior over the 12 hour scenario. The maximum response time had a slightly increase of 0.19s (about 8.6 % over the YoYo), this is because in the YoYo attack scenario there were the same amount of requests but spread among 5 different pods over the same time, which in this mitigation scenario, there were only 3 pods to respond to the same amount of traffic, leading to a higher increase on the response time. Despite this, the average response time decreases, showing that this mitigation successfully counters the YoYo attack while maintaining a good response time

over the 12 hours. The combination of a higher peak and lower average latency shows that when there are only 3 pods actively running, during each burst of 265 concurrent requests every 15 minutes, the first one or two responses take slightly longer with only three pods, but subsequent requests remain unaffected, keeping the overall response smoother.

Metric	Baseline (s)	Baseline under Attack (s)	With Mitigation (s)
Minimum	0.004	0.002	0.002
Maximum	0.021	2.208	2.398
Average	0.005972	0.011116	0.010855

Table 5.5: Baseline Response Time Comparison

In Figure 5.15, the median stays at 0.007s, the same as in the YoYo scenario, but the majority of those responses are more tightly clustered around that value. Although there are still occasional higher-latency outliers, the overall spread is noticeably reduced compared to the unmitigated attack. This shows that the mitigation keeps the median response time at the same level while reducing the occasional slow responses.

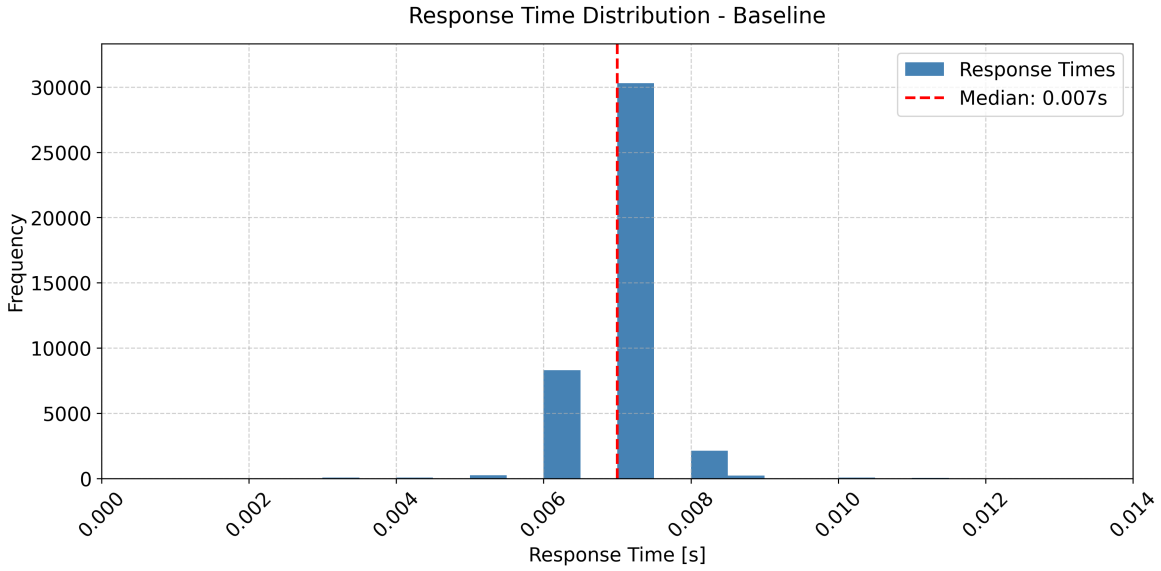

Figure 5.15: Randomized Autoscaling Defense - Response Time Distribution.

Evaluation of the Mitigation

As discussed before, each of the mitigation will be evaluated based on three key aspects, the baseline (normal user behavior) response time, total price, and the implementation complexity.

1. Baseline Response Time Delay:
   The mitigation introduces almost no additional delay during its normal operation. The average response time falls from 0.011116s under attack to 0.010855s, even though the maximum rises slightly from 2.208s to 2.398s.

2. Total Cost:
   The overall cost on those 12 hours went from 0.02434$ to 0.01892$. A good 22.3 % decrease in cost.
3. Implementation Complexity:
   This deployment required only a small Python script running with sudo on the control node to run kubectl apply commands and update the autoscaler configurations.

Overall, the mitigation achieves a good total cost decrease without disturbing the response time in normal user behavior. In terms of complexity, it is a really simple solution with only need to run a Python script in the background with privileges.

5.4.2 POC 2

This second mitigation scenario follows the same setup as before but applies an alternative approach that prioritizes faster response times while still mitigating cost impacts, as discussed in Chapter chapter 4.

Autoscaling and Cost

Number of Active Pods: Figure 5.16, we can observe multiple events that occurred throughout the 12 hour scenario. First, let's start with the expected 9 pod increase at the beginning. As seen in Figure 5.11 before, the mitigation was applied from the start, and each time it reapplies the Knative service configuration with the default values, the number of pods increases by around 8 to 9.

Following that, we can see that the attack triggered two separate scaling events of 3 pods each, which indicates that the mitigation had set a high target value due to its randomness. However, between 1:30 and 2:00, that did not happen, which suggests the mitigation had to apply new settings because the initial target value wasn't sufficient to maintain the expected traffic flow.

Then, after this period, the 2 hour cooldown window begins, and the default Knative settings are reapplied around 4:00, that's why we see another spike of 9 pods, as if the mitigation process were starting over. From that point until the end of the 12 hour window, we notice similar recurring behavior. Each time it scales by 8 to 9 pods, it means the 2 hour window ended, and the default settings had to be reapplied.

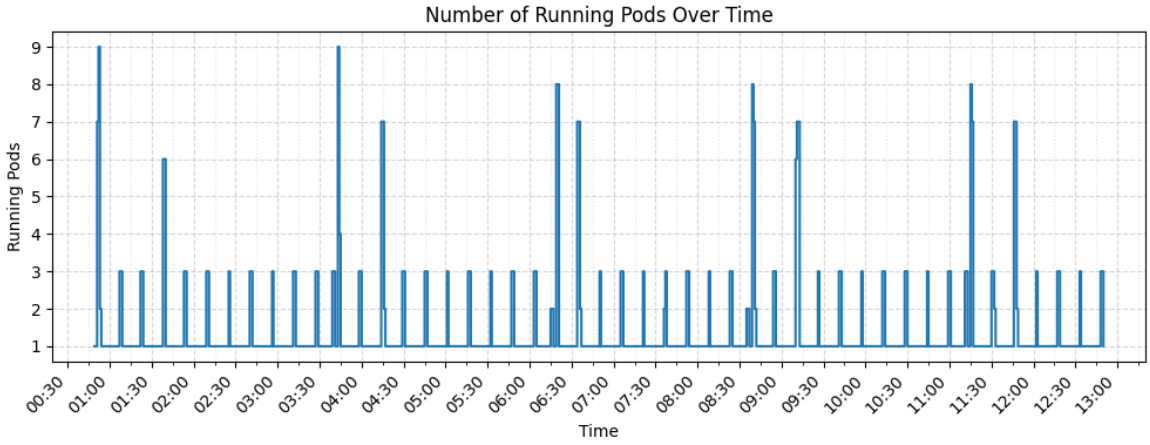

Figure 5.16: Dynamic Autoscaling for Response Time - Number of Active Pods.

CPU usage: Here we see something new compared to the previous mitigation graph. We see a lot more of CPU usage over time, and more spikes near 800-700 % than before. This is obviously explained by the fact that this mitigation slowly evolves with the traffic instead of pushing the target level to an extreme high value, and those 800 to 700 % spikes are due to the fact that every time after those 2 hour window, the knative service configuration is on the default values so it repeats the same behavior like the start.

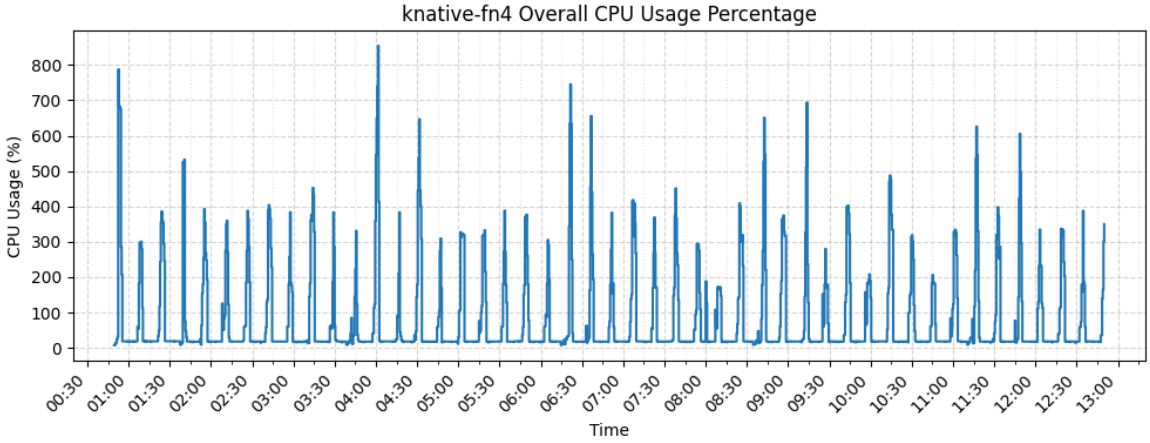

Figure 5.17: Dynamic Autoscaling for Response Time - CPU Usage.

Cost rate: In Figure 5.18 we can see that the cost rate shows much more aggressive spikes and wide fluctuations compared to the previous mitigation, it has increased the maximum value of the positive spikes (increased cost), but it has a new minimum value of the negative spikes (decreased cost) :

- The negative drops are significantly deeper, reaching as low as -0.00042$/min, which results from the simultaneous shutdown of multiple idle pods. This is once more explained by the 7-9 pods shut down when applied the default configurations.

- Positive spikes exceed +0.00046$/min. This happens every time those 7-9 pods get launched at several moments, therefore the value is bigger then the actual attack.
- Between the spikes, there is some stability this time, due to when the number of pods are around 3 to 4, depending on the target value.

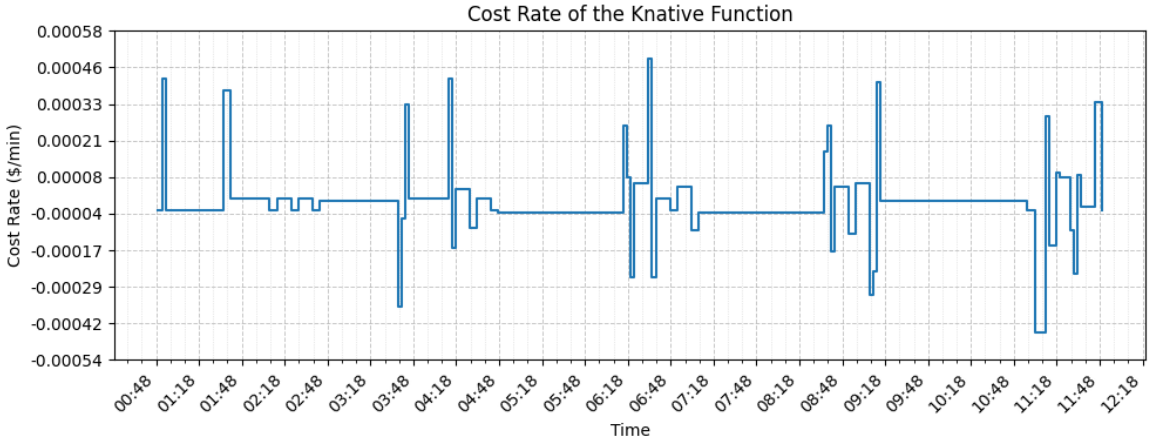

Figure 5.18: Dynamic Autoscaling for Response Time - Cost Rate.

Total cost: 0.02173$.

The total cost increased to 0.02173$ with the second mitigation strategy, as shown in Table 5.6. While this rise of 67.2 % above the baseline is not ideal, it was already expected.

This is because the alternative mitigation adopts a more dynamic approach, prioritizing service responsiveness and stability (see next section), even at the expense of a slightly higher overall cost. Despite this increase, it still achieves a 10.7 % cost reduction compared to the YoYo attack scenario, demonstrating a meaningful balance between performance and cost efficiency.

Scenario	Cost ($)	Increase from Baseline	Reduction from Attack
Baseline	0.01300	–	–
YoYo Attack	0.02434	+87.2 %	–
YoYo + Mitigation	0.01892	+45.5 %	-22.3 %
YoYo + Alternative Mitigation	0.02173	+67.2 %	-10.7 %

Table 5.6: Cost Comparison between Baseline, YoYo attack, Mitigation and Alternative Mitigation

Performance Metrics

Comparing the response time values once again in Table 5.7, we observe a slight decrease in the average response time, specifically a reduction of 0.009876 seconds, approximately 9.02 %. This indicates a positive, even if its a small value, it was an improvement overall.

However, the maximum response time increased slightly, because is likely due to the more frequent scaling between 7 and 9 pods, where the distribution of requests across different pods may occasionally introduce minor delays for some requests.

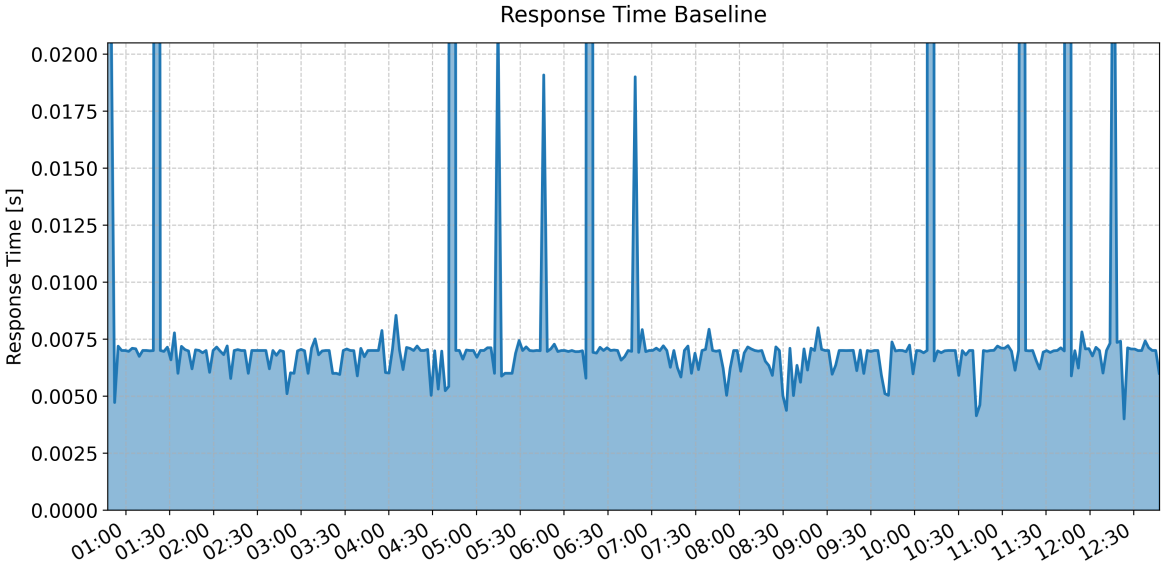

Figure 5.19: Dynamic Autoscaling for Response Time - Response Time.

Metric	Baseline (s)	YoYo (s)	With Mitigation (s)	Alternative Mitigation (s)
Minimum	0.004	0.002	0.002	0.002
Maximum	0.021	2.208	2.398	2.470
Average	0.005972	0.011116	0.010855	0.009876

Table 5.7: Baseline Response Time Comparison

In Figure 5.20, the median remains at 0.007s again, however, we can now clearly observe some progress. There are no visible bars beyond 0.008s, and there are more occurrences of lower values under 0.006s, such as increased counts at 0.005s, 0.004s, and even 0.003s.

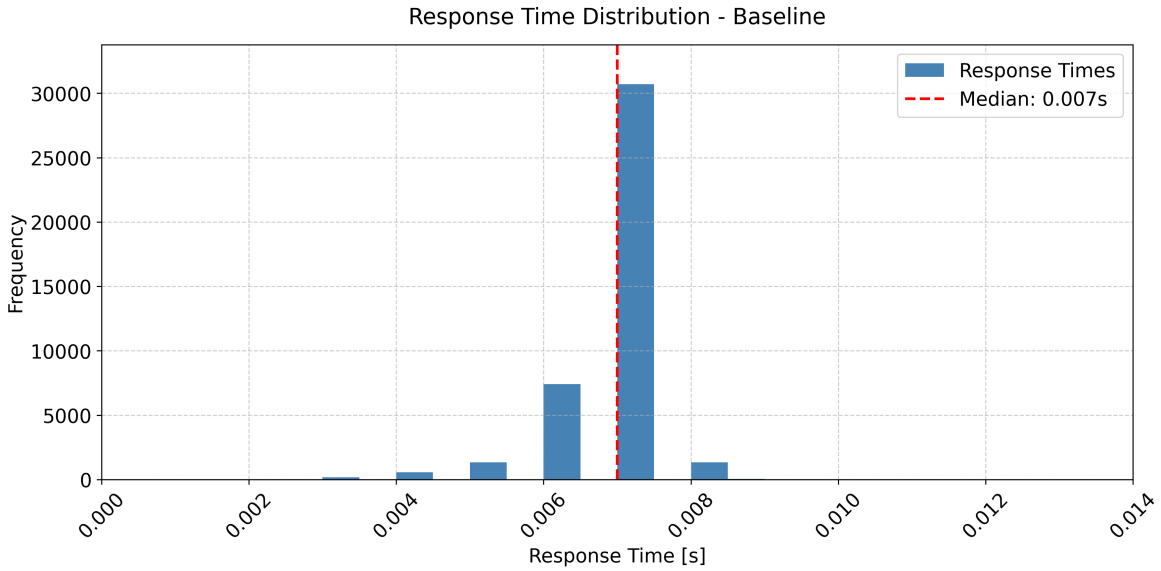

Figure 5.20: Dynamic Autoscaling for Response Time - Response Time Distribution.

Evaluation of the Mitigation

When comparing this mitigation to the previous one, we can see that the earlier approach delivered better results in terms of overall cost after the 12 hour period, making the Knative function less expensive, while still maintaining a reasonable response time for baseline (normal user behavior) traffic under the same YoYo attack. The alternative version, on the other hand, led to a 11.6 % increase in total cost but improved the baseline response time by around 9.02 %. This trade off, might be worth it for some use cases, who care more about performance and don't mind paying a bit extra for better service. The implementation complexity stayed the same since it was just some improvements on the code.

1. Baseline Response Time Delay:
   This mitigation, has a better average response time values, giving a more responsive and less delayed service even though the maximum value could be higher then expected.
2. Total Cost:
   The overall cost was 0.02173$ which had a -10.7 % reduction in mitigating the attack.
3. Implementation Complexity:
   This deployment involved running a simple Python script again, with sudo privileges on the control node to execute kubectl apply commands and update the autoscaler configurations.

Overall, this mitigation presents both strengths and weaknesses when compared to the previous approach. There is no definitive best choice, as each solution offers distinct advantages depending on the priorities of the use case. While one favors lower cost, the other provides better performance. The decision ultimately depends on whether minimizing expenses or maximizing responsiveness is more critical for the target application when being target by a YoYo attack.

CHAPTER 6

Conclusions

This chapter presents the overall conclusions of this dissertation and outlines potential areas for improvement in future work.

6.1 Conclusion

The evolution of mitigation strategies for DoS, DDoS, and EDoS attacks reflects the growing complexity and sophistication of threats in cloud environments. From early reactive approaches like SOS and WebSOS to proactive and multi-layered frameworks such as CLAD and DDoS-MS, researchers have continuously adapted to counter increasingly complex attack methods. However, these solutions often face trade-offs between security, performance, and user accessibility, highlighting the ongoing challenges in this domain and that will depend in various things. For EDoS attacks, innovative approaches like sPoW, EDoS-Shield, and EDoS-ADS have introduced mechanisms tailored to the economic dimensions of these threats. While these strategies demonstrate significant advancements, they still fall short in addressing adaptive and stealthy attacks like the YoYo attack, which exploit predictable scaling mechanisms. The need for better solutions or a real-time adaptive defenses and the incorporation of advanced analytics, such as machine learning-based detection, is evident.

In addition, the rapid adoption of serverless computing has revolutionized how applications are deployed and scaled, offering near infinite elasticity and a new pay-per-use model. However, this elasticity can easily become a vulnerability. As shown in this work, these emerging periodic YoYo traffic bursts can trick Knative's autoscaler into repeatedly scale up multiple pods, without actually using them, transforming it into an EDoS attack. The results show that, under a YoYo attack, the billing over those twelve hours increases rapidly without significantly degrading performance, making this a serious stealthy and economic problem.

Therefore, this Dissertation had two main objectives. One was to implement a PoC of an EDoS attack, in this case, the YoYo attack against a serverless function in a cloud-based environment, and quantify its economic and performance impact. The second one was to

design and evaluate two possible mitigation strategies capable of effectively mitigating this passive cost increase over time while maintaining acceptable service responsiveness.

Under a controlled evaluation, the YoYo attack successfully inflated the operational costs by 87.2 % compared to baseline, in the 12 hour window, without visibly degrading the response time for legitimate traffic that same time.

When the first mitigation was applied, the total cost under this same attack fell from 0.02434$ to 0.01892$ with a -22.3 % decrease in total cost, and improving the response time from an average of 0.011116s to 0.010855 with a +2.35 % increase. This shows that the mitigation was successful in mitigating the damage cost, without delaying the normal requests.

The second mitigation, an alternative version which had an improvement on the incrementation of the target value by being more dynamic, and going back to default configs after a short time, had some positive results as well. Besides the slight increase of the overall cost compared to the first mitigation, it still had a reduction from 0.02434$ to 0.02173$ with a -10.7 % decrease in total cost when being targeted by the YoYo attack. In terms of response time, this mitigation outperformed the first one, achieving a +9.02 % improvement in responsiveness making this mitigation an alternative by providing better service response at the cost of a little extra money, in this case, +11.6 %.

In summary, both mitigation approaches succeeded in reducing the economic impact of the YoYo attack, but with different trade offs. Depending on the use case one could be better then the other.

In conclusion, the solution presented in this Dissertation proved to be effective in demonstrating the real economic threat caused by the EDoS attacks, such as the YoYo attack, in serverless environments. The developed mitigation strategies, showed promising results in reducing the financial impact without compromising service performance, highlighting the potential of lightweight and adaptive strategies to protect cloud-native infrastructures.

That said, there are points that can be improved, which opens up opportunities for future developments.

6.2 Future Work

There are several aspects of this dissertation that can be further improved in future work:

- A better exploration of the KPA ConfigMap parameters could provide additional improvements in both cost efficiency and performance. By trying to understand if some combination of those configurations work together, they could help even more mitigating this problem and probably achieve similar or better results, giving more alternatives. There were some with good potential to help mitigate this problem for example: target-burst-capacity, stable-window and panic-window-percentage, but an actual conclusion wasn't reached in this time frame. Going more deeply into those configurations and others could measure their individual and combined impacts on autoscaling behavior, cold-start frequency, resource utilization, request latency, and overall billing.

- Another thing is that both of the mitigation techniques could be tested against other kinds of attacks and understand if they would help in any way possible. By running controlled experiments under these different attack scenarios and measuring cost, latency, and resource usage, one could find out whether the same defenses work more generally or need adjustments for specific threats.
- Finally, in the second mitigation script, the fixated threshold jump could be replaced with a more adaptive threshold. One solution was computing the mean of the last N pod counts and then triggering the mitigation when the increase exceeded a fraction of that average, rather than relying on a fixed jump. This method could help the detection adjust to normal changes in workload and avoid false positives during regular traffic increases.

References

[1] Q. Zhang, L. Wang, and C. Fu, "The evolution of distributed denial of service (ddos) attacks: Nlp-based detection and strategic management countermeasures in modern networks", *Journal of Economic Theory and Business Management*, vol. 1, no. 4, pp. 28–37, Aug. 2024. DOI: 10.5281/zenodo.13232838. [Online]. Available: https://www.suaspress.org/ojs/index.php/JETBM/article/view/v1n4a04.

[2] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks", *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013. DOI: 10.1109/SURV.2013.031413.00127.

[3] F. S. Dantas Silva, E. Silva, E. P. Neto, M. Lemos, A. J. Venancio Neto, and F. Esposito, "A taxonomy of ddos attack mitigation approaches featured by sdn technologies in iot scenarios", *Sensors*, vol. 20, no. 11, 2020, ISSN: 1424-8220. DOI: 10.3390/s20113078. [Online]. Available: https://www.mdpi.com/1424-8220/20/11/3078.

[4] N. Dayal, P. Maity, S. Srivastava, and R. Khondoker, "Research trends in security and ddos in sdn", *Security and Communication Networks*, vol. 9, no. 18, pp. 6386–6411, 2016. DOI: https://doi.org/10.1002/sec.1759. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1759. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1759.

[5] A. Praseed and P. S. Thilagam, "Ddos attacks at the application layer: Challenges and research perspectives for safeguarding web applications", *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 661–685, 2019. DOI: 10.1109/COMST.2018.2870658.

[6] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: Methods, systems and tools", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 303–336, 2014. DOI: 10.1109/SURV.2013.052213.00046.

[7] Cloudflare, *Bigger and badder: How ddos attack sizes have evolved over the last decade*, Accessed: 2025-01-06, 2023. [Online]. Available: https://blog.cloudflare.com/bigger-and-badder-how-ddos-attack-sizes-have-evolved-over-the-last-decade/.

[8] Radware, *Threat report executive summary 2024*, https://www.cisco.com/c/dam/m/en_in/events/security-conclave-2024/radware-threat-report-summary-2024.pdf, Accessed: 2024-12-22, 2024.

[9] V. Zlomislić, K. Fertalj, and V. Sruk, "Denial of service attacks: An overview", in *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*, 2014, pp. 1–6. DOI: 10.1109/CISTI.2014.6876979.

[10] N. Innab and A. Alamri, "The impact of ddos on e-commerce", in *2018 21st Saudi Computer Society National Computer Conference (NCC)*, 2018, pp. 1–4. DOI: 10.1109/NCG.2018.8593125.

[11] M. Sachdeva, K. Kumar, G. Singh, and K. Singh, "Performance analysis of web service under ddos attacks", in *2009 IEEE International Advance Computing Conference*, 2009, pp. 1002–1007. DOI: 10.1109/IADCC.2009.4809152.

[12] F. Z. Chowdhury, L. B. M. Kiah, M. M. Ahsan, and M. Y. I. Bin Idris, "Economic denial of sustainability (edos) mitigation approaches in cloud: Analysis and open challenges", in *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, 2017, pp. 206–211. DOI: 10.1109/ICECOS.2017.8167135.

[13] Q. V. Ta and M. Park, "Economic denial of sustainability (edos) attack detection by attention on flow-based in software defined network (sdn)", in *2022 International Conference on Information Networking (ICOIN)*, 2022, pp. 183–185. DOI: 10.1109/ICOIN53446.2022.9687229.

[14] S. Bulla, B. Brao, K. Rao, and K. Chandan, "An experimental evaluation of the impact of the edos attacks against cloud computing services using aws", *International Journal of Engineering & Technology (IJET)*, vol. 7, pp. 202–208, Jan. 2018. DOI: 10.14419/ijet.v7i1.5.9147.

[15] T. H. H. Aldhyani and H. Alkahtani, "Artificial intelligence algorithm-based economic denial of sustainability attack detection systems: Cloud computing environments", *Sensors*, vol. 22, no. 13, 2022. DOI: 10.3390/s22134685. [Online]. Available: https://www.mdpi.com/1424-8220/22/13/4685.

[16] Amazon Web Services, *Aws lambda*, https://aws.amazon.com/pt/lambda/, Accessed: 2024-11-04.

[17] Google Cloud, *Google cloud functions*, https://cloud.google.com/functions, Accessed: 2024-11-04.

[18] A. Abdeladim, S. Baina, and K. Baina, "Elasticity and scalability centric quality model for the cloud", in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, pp. 135–140. DOI: 10.1109/CIST.2014.7016607.

[19] Amazon Web Services, *Amazon ec2*, https://aws.amazon.com/pt/ec2/, Accessed: 2024-11-03.

[20] H.-J. Ko, G.-Y. Wang, G. Horng, and S.-J. WANG, "A chaotic attack offering with improving mechanism in economic denial of sustainability", in *2020 International Conference on Pervasive Artificial Intelligence (ICPAI)*, 2020, pp. 41–45. DOI: 10.1109/ICPAI51961.2020.00015.

[21] S. Bulla, B. Brao, K. Rao, and K. Chandan, "An experimental evaluation of the impact of the edos attacks against cloud computing services using aws", *International Journa l of Engineering & Technology (IJET)*, vol. 7, pp. 202–208, Jan. 2018. DOI: 10.14419/ijet.v7i1.5.9147.

[22] H. Wang, Z. Xi, F. Li, and S. Chen, "Abusing public Third-Party services for EDoS attacks", in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/wang.

[23] B. Omoniwa, R. Hussain, M. A. Javed, S. H. Bouk, and S. A. Malik, "Fog/edge computing-based iot (feciot): Architecture, applications, and research issues", *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4118–4149, 2019. DOI: 10.1109/JIOT.2018.2875544.

[24] P. Soumplis, P. Kokkinos, A. Kretsis, P. Nicopolitidis, G. Papadimitriou, and M. Varvarigos, "Resource allocation challenges in the cloud and edge continuum", in Mar. 2022, pp. 443–464, ISBN: 978-3-030-87048-5. DOI: 10.1007/978-3-030-87049-2_15.

[25] S. Mustafa, B. Nazir, A. Hayat, A. ur Rehman Khan, and S. A. Madani, "Resource management in cloud computing: Taxonomy, prospects, and challenges", *Computers & Electrical Engineering*, vol. 47, pp. 186–203, 2015, ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2015.07.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S004579061500275X.

[26] A. W. Services, *Aws pricing*, Accessed: 2025-01-02, 2025. [Online]. Available: https://aws.amazon.com/pricing/?aws-products-pricing.sort-by=item.additionalFields.productNameLowercase&aws-products-pricing.sort-order=asc&awsf.Free%20Tier%20Type=*all&awsf.tech-category=*all.

[27] A. W. Services, *Aws savings plans*, Accessed: 2025-01-02, 2025. [Online]. Available: https://aws.amazon.com/pt/savingsplans/.

[28] A. W. Services, *Amazon ec2 reserved instances*, Accessed: 2025-01-02, 2025. [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/?nc1=h_ls.

[29] A. W. Services, *Amazon ec2 spot instances*, Accessed: 2025-01-02, 2025. [Online]. Available: https://aws.amazon.com/pt/ec2/spot/.

[30] G. Cloud, *Google cloud pricing*, Accessed: 2025-01-02, 2025. [Online]. Available: https://cloud.google.com/pricing?%5C&hl=en.

[31] G. Cloud, *Signing up for committed use discounts*, Accessed: 2025-01-02, 2025. [Online]. Available: https://cloud.google.com/compute/docs/instances/signing-up-committed-use-discounts.

[32] G. Cloud, *Sustained use discounts*, Accessed: 2025-01-02, 2025. [Online]. Available: https://cloud.google.com/compute/docs/sustained-use-discounts.

[33] G. Cloud, *Preemptible virtual machine instances*, Accessed: 2025-01-02, 2025. [Online]. Available: https://cloud.google.com/compute/docs/instances/preemptible.

[34] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein, "Using graphic turing tests to counter automated ddos attacks against web servers", in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, Washington D.C., USA: Association for Computing Machinery, 2003, pp. 8–19, ISBN: 1581137389. DOI: 10.1145/948109.948114. [Online]. Available: https://doi.org/10.1145/948109.948114.

[35] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica, "Taming ip packet flooding attacks", *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 45–50, Jan. 2004, ISSN: 0146-4833. DOI: 10.1145/972374.972383. [Online]. Available: https://doi.org/10.1145/972374.972383.

[36] D. Cook, W. Morein, A. Keromytis, V. Misra, and D. Rubenstein, "Websos: Protecting web servers from ddos attacks", in *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, 2003, pp. 461–466. DOI: 10.1109/ICON.2003.1266234.

[37] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds", in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05, USA: USENIX Association, 2005, pp. 287–300.

[38] S. H. Khor and A. Nakao, "Daas: Ddos mitigation-as-a-service", in *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, 2011, pp. 160–171. DOI: 10.1109/SAINT.2011.30.

[39] H. Beitollahi and G. Deconinck, "Analyzing well-known countermeasures against distributed denial of service attacks", *Computer Communications*, vol. 35, no. 11, pp. 1312–1332, 2012, ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2012.04.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366412001211.

[40] S. H. Khor and A. Nakao, "Spow: On-demand cloud-based eddos mitigation mechanism", in *HotDep (Fifth Workshop on Hot Topics in System Dependability)*, 2009.

[41] J. Idziorek and M. Tannian, "Exploiting cloud utility models for profit and ruin", in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 33–40. DOI: 10.1109/CLOUD.2011.45.

[42] M. H. Sqalli, F. Al-Haidari, and K. Salah, "Edos-shield - a two-steps mitigation technique against edos attacks in cloud computing", in *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 2011, pp. 49–56. DOI: 10.1109/UCC.2011.17.

[43] F. Al-Haidari, M. H. Sqalli, and K. Salah, "Enhanced edos-shield for mitigating edos attacks originating from spoofed ip addresses", in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012, pp. 1167–1174. DOI: 10.1109/TrustCom.2012.146.

[44] W. Alosaimi, M. Zak, and K. Al-Begain, "Denial of service attacks mitigation in the cloud", in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, 2015, pp. 47–53. DOI: 10.1109/NGMAST.2015.48.

[45] M. Naresh Kumar, P. Sujatha, V. Kalva, R. Nagori, A. K. Katukojwala, and M. Kumar, "Mitigating economic denial of sustainability (edos) in cloud computing using in-cloud scrubber service", in *2012 Fourth International Conference on Computational Intelligence and Communication Networks*, 2012, pp. 535–539. DOI: 10.1109/CICN.2012.149.

[46] P. S. Bawa and S. Manickam, "Critical review of economical denial of sustainability (edos) mitigation techniques", *Journal of Computer Science*, vol. 11, no. 7, pp. 855–862, Oct. 2015. DOI: 10.3844/jcssp.2015.855.862. [Online]. Available: https://thescipub.com/abstract/jcssp.2015.855.862.

[47] W. Alosaimi, M. Alshamrani, and K. Al-Begain, "Simulation-based study of distributed denial of service attacks prevention in the cloud", in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, 2015, pp. 60–65. DOI: 10.1109/NGMAST.2015.50.

[48] N. Agrawal and S. Tapaswi, "A proactive defense method for the stealthy edos attacks in a cloud environment", *International Journal of Network Management*, vol. 30, Feb. 2020. DOI: 10.1002/nem.2094.

[49] Red Hat, Inc., *Kourier and istio ingresses*, Accessed: 2025-05-04, 2024. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_openshift_serverless/1.34/html/serving/kourier-and-istio-ingresses#serverless-kourier-and-istio-ingresses-solutions_kourier-and-istio-ingresses.

[50] Knative Project, *Serving architecture - networking layer and ingress*, Accessed: 2025-05-04, 2024. [Online]. Available: https://knative.dev/docs/serving/architecture/#networking-layer-and-ingress.

[51] The Knative Authors, *Knative serving*, Accessed: 2025-03-29, 2025. [Online]. Available: https://knative.dev/docs/serving/.

[52] The Knative Authors, *Knative serving*, Accessed: 2025-03-29, 2025. [Online]. Available: https://knative.dev/docs/serving/autoscaling/.

[53] Alibaba Cloud, *Enable auto scaling to withstand traffic fluctuations*, Accessed: 2025-05-1, 2024. [Online]. Available: https://www.alibabacloud.com/help/en/ack/serverless-kubernetes/user-guide/enable-auto-scaling-to-withstand-traffic-fluctuations.

[54] K. Authors, *About load balancing*, May 1, 2025. [Online]. Available: https://knative.dev/docs/serving/load-balancing/#activator-pod-selection (visited on 05/06/2025).

[55] K. Authors, *Kourier ingress controller design*, Accessed: 2025-05-06, 2023. [Online]. Available: %5Curl%7Bhttps://github.com/knative/net-kourier/blob/main/docs/architecture.md%7D.