Journal of Parallel and Distributed Computing 204 (2025) 105130

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Mitigating DDoS attacks in containerized environments: A comparative analysis of Docker and Kubernetes

Yung-Ting Chuang *, Chih-Han Tu

Institute of Information Management, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

ARTICLE INFO

Dataset link: https://reurl.cc/xN3e8e

Keywords:
Containerization
DDoS attacks and defenses
Web services security
Docker
Kubernetes

ABSTRACT

Containerization has become the primary method for deploying applications, with web services being the most prevalent. However, exposing server IP addresses to external connections renders containerized services vulnerable to DDoS attacks, which can deplete server resources and hinder legitimate user access. To address this issue, we implement twelve different mitigation strategies, test them across three common types of web services, and conduct experiments on both Docker and Kubernetes deployment platforms. Furthermore, this study introduces a cross-platform, orchestration-aware evaluation framework that simulates realistic multi-service workloads and analyzes defense strategy performance under varying concurrency conditions. Experimental results indicate that Docker excels in managing white-listed traffic and delaying attacker responses, while Kubernetes achieves low completion times, minimum response times, and low failure rates by processing all requests simultaneously. Based on these findings, we provide actionable insights for selecting appropriate mitigation strategies tailored to different orchestration environments and workload patterns, offering practical guidance for securing containerized deployments against low-rate DDoS threats. Our work not only provides empirical performance evaluations but also reveals deployment-specific trade-offs, offering strategic recommendations for building resilient cloud-native infrastructures.

1. Introduction

In the era of digital transformation, managing and deploying applications has become increasingly complex due to rapid expansion and intricate architectures [1]. Containerization technology offers a flexible and efficient solution to these challenges [2].

Containerization encapsulates an application and its dependencies into a standalone unit, providing robust isolation, portability, and rapid deployment [3]. This modern approach simplifies deployment and management, but it alone cannot handle all complexities [3].

Docker, a leading containerization platform, addresses these challenges by ensuring consistency across environments and accelerating development through portable containers [4,5]. Its lightweight nature and rapid deployment capabilities make it ideal for microservices architectures, promoting agility and scalability [6]. Docker also optimizes resource utilization by running multiple containers on the same host efficiently [7,8].

As IT demands evolve, managing Docker containers has become increasingly complex [9]. To address these challenges, Kubernetes has emerged as a powerful tool for orchestrating and deploying containerized applications.

Kubernetes automates application deployment, scaling, and operations.[1] It overcomes the limitations of containerization by providing comprehensive management for complex applications. Kubernetes enhances scalability [10], automates deployment and management [11], and ensures high availability, fault tolerance, and effective load balancing [12–14]. These features make it an essential tool for adapting to market demands and optimizing application operations.

As Kubernetes clusters scale, they must maintain high stability and performance, as noted by Hardikar et al. [15]. Elastic scaling and automated management are crucial for handling heavy workloads, traffic spikes, and hardware failures. For instance, during peak events like flash sales, Kubernetes clusters must dynamically allocate resources to prevent slowdowns or downtime.

The increasing use of Kubernetes for web services brings significant security challenges. Previous research has focused on Kubernetes in web services [16,17] but often overlooks other uses. Distributed denial of service (DDoS) attacks, which flood systems with malicious traffic,

* Corresponding author.
E-mail address: ytchuang@nycu.edu.tw (Y.-T. Chuang).

[1] The Kubernetes Authors, *Kubernetes Documentation*, 2024. Available at: https://kubernetes.io/docs/.

https://doi.org/10.1016/j.jpdc.2025.105130
Received 14 January 2025; Received in revised form 1 May 2025; Accepted 5 June 2025
Available online 11 June 2025

pose severe risks such as service instability and unavailability [18]. The dynamic nature of containerized environments complicates defense strategies, making robust, adaptive measures essential [19,20]. Despite Kubernetes' self-healing capabilities, it remains vulnerable to large-scale DDoS attacks [21]. Therefore, implementing comprehensive security measures like firewalls, intrusion detection systems, and DDoS protection is vital for maintaining stable service delivery.

Recent efforts have explored container security from various perspectives, including vulnerability detection frameworks [22], auto-scaling intrusion detection mechanisms [23], simulation platforms for ML-based defenses [24], and resource profiling under attack [25]. However, these works primarily focus on detection capabilities, elastic resource adaptation, or system profiling, rather than systematically evaluating mitigation strategies under real-world workloads and container orchestration behaviors.

While containerization technology has simplified application deployment and management, securing and ensuring resilience in Docker and Kubernetes environments remains challenging. Research such as Patidar et al. [26] provides foundational insights into Docker, but there is a need to extend this understanding to Kubernetes and evaluate web applications under realistic DDoS scenarios.

Low-rate DDoS attacks, which use subtle traffic bursts to mimic legitimate behavior, are particularly dangerous as they often evade conventional monitoring tools. This study addresses the need to evaluate the resilience of web applications in Docker and Kubernetes against such threats.

We assess web application resilience in Docker and Kubernetes environments against DDoS attacks, applying 12 mitigation strategies from [26]. Our evaluation includes default scenarios, page separation, and resource allocation for whitelisted users. We test three web applications (Simple Nginx Sample Page, Simulated Ticket Booking System, and Simulated E-Commerce Website) across various attack scenarios, focusing on webpage access, database operations, and concurrency to simulate realistic usage patterns.

By simulating low-rate DDoS attack scenarios [27] on various web application types—from simple informational pages to complex e-commerce platforms—this study evaluates the effectiveness of mitigation strategies against these stealthy threats. Our findings enhance understanding of how Docker and Kubernetes deployments can defend against low-rate DDoS attacks [28], providing actionable insights for real-world application security. We assess strategies using metrics such as test completion time, response times, and failed requests, informing optimal defense mechanisms for different deployment scenarios.

Our contributions include:

1. **Extension to Kubernetes Environments:** We extend DDoS resilience evaluation from Docker to Kubernetes, enabling platform-aware performance analysis and revealing orchestration-level behavioral differences under attack.
2. **Comprehensive Multi-Service Testing:** We simulate three realistic web service types—Simple Nginx Sample Page, Simulated Ticket Booking System, and Simulated E-Commerce Website—each representing different workload patterns, including static content, database writes, and complex page rendering.
3. **Systematic Evaluation of 12 Defense Strategies:** We empirically evaluate a taxonomy of twelve mitigation strategies combining page separation, user-level resource prioritization, and traffic control across varying concurrency levels and service types.
4. **Client-Centric Performance Metrics:** We focus on service-level measurements such as response time, failure rate, and recovery time to reflect real-world user impact during low-rate DDoS attacks.
5. **Practical Deployment Guidance:** Our cross-platform comparison reveals how orchestration-specific behaviors (e.g., Kubernetes scheduling latency, Docker resource rigidity) affect strategy effectiveness, offering actionable recommendations for developers and system administrators.

Unlike prior works that either propose theoretical frameworks or evaluate isolated configurations, our study introduces a practical, orchestration-aware defense strategy framework tailored for modern containerized infrastructures. Although Kubernetes provides built-in features for scaling and fault tolerance, how these mechanisms respond under coordinated, application-layer DDoS attacks remains underexplored. Our findings help bridge this gap by highlighting the platform-dependent trade-offs of deploying service-level mitigation strategies in Docker and Kubernetes environments.

2. Related work

In the following section, we will review related work focusing on containerization, Docker and Kubernetes.

2.1. Containerization

Containerization has become crucial in cloud computing, transforming application development and deployment with several key benefits:

1. **Isolation and Lightweightness**: Containers offer strong isolation and are lightweight, reducing virtualization overhead [2].
2. **Portability**: Containers enable consistent application performance across various environments, facilitating smooth transitions from development to deployment [2].
3. **Rapid Deployment and Scaling**: Containers start quickly and scale easily, enhancing flexibility and scalability [2].

Junzo Watada et al. [2] overview containerization, noting its advantages, challenges, and comparing it with virtual machines. Zhi Li et al. [29] present a machine learning-based approach for detecting low-rate DDoS attacks in containerized environments, while Yu Liu et al. [30] assess performance gaps in edge-cloud architectures. Naweiluo Zhou et al. [31] survey container engines and orchestration for HPC systems.

Despite containers' advantages, they are often deployed on virtual machines [32,8] due to hybrid deployment needs, offering enhanced isolation and security [33,34]. This approach combines the benefits of both technologies, ensuring security and efficiency in cloud computing [35].

2.2. Docker

Docker[2] is a leading platform for containerization that revolutionizes how applications are developed, shipped, and deployed. It packages applications and their dependencies into standardized containers, ensuring consistent performance across different environments. Key advantages include:

1. Simplified Deployment: Docker simplifies packaging and deploying applications, making it consistent across environments. [7]
2. Enhanced DevOps Practices: Docker improves collaboration between development and operations teams, reducing environment-related issues and increasing efficiency. [7]
3. Efficient Resource Utilization: Docker enables multiple containers to run on the same host with minimal overhead compared to full virtual machines, enhancing performance. [7]

2.2.1. Docker application

Recent studies have explored Docker's application in resource allocation and performance. Xili Wan et al. [36] proposed a scalable resource allocation algorithm for Docker-based microservices. Minh Thanh Chung et al. [37] found Docker more suitable than virtual machines for data-intensive applications. Chuanqi Kan [38] developed an

[2] *Docker*. Available at: https://www.docker.com/.

elastic cloud platform using Docker with dynamic container scaling. These studies focus on hardware costs, while our research emphasizes client-side user experience.

2.2.2. Docker security

Studies on Docker security cover various practices. Delu Huang et al. [39] reviewed Docker's security mechanisms and proposed a threat detection framework. Jeeva Chelladhurai et al. [40] discussed DoS attack mitigation and Docker's impact on IT environments. Jiang Wenhao et al. [41] explored Docker's vulnerabilities at the infrastructure level. Patidar et al. [26] proposed container-based strategies to mitigate DDoS attacks, but their experiments were limited to Docker and did not address orchestration behavior or concurrency sensitivity.

Garcia [24] developed a Docker-based testbed to evaluate deep learning models under real-time DDoS attack simulations, focusing on model-level metrics such as accuracy and precision using synthetic Layer 4 traffic. Gamess and Parajuli [25] analyzed the resource consumption of Docker and Podman under attack on ARM-based devices, providing performance profiles but not comparative analysis of defense strategies.

In contrast, our study introduces a strategy-centric, platform-aware evaluation of twelve DDoS mitigation strategies, spanning both Docker and Kubernetes. We assess how page separation, resource prioritization, and traffic filtering interact with orchestration mechanisms under varying workloads and concurrency levels. Unlike prior works that focus on model testing or low-level resource behavior, we simulate realistic service patterns (e.g., ticketing and e-commerce systems) and analyze client-side impact, offering deployment-level insights applicable to modern containerized environments.

2.3. Kubernetes

Kubernetes[3] is a popular open-source container management platform with several key advantages:

1. Scalability: Kubernetes excels in scalability, enabling seamless expansion of applications to meet changing demands and maintain optimal performance. [10]
2. Automated Deployment: Kubernetes automates the deployment and management of applications, reducing operational workload and enhancing efficiency. [11]
3. High Availability and Load Balancing: With features for high availability, fault tolerance, and load balancing, Kubernetes ensures system stability and reliability, even under varying loads. [12] [13] [14]
4. Agile Operations: Kubernetes supports agile and reliable application operations, helping organizations stay competitive by quickly adapting to market changes. [42]

2.3.1. Kubernetes application

Firstly, we examine Kubernetes as a container management platform and its impact on application deployment:

Shah et al. [43] highlight the role of Docker, Kubernetes, and Google Cloud Platform in modern cloud infrastructure, focusing on Kubernetes' features for automated management, deployment, and scaling. They provide practical examples of Kubernetes architecture and deployment on Google Cloud Platform but do not address security issues.

Vayghan et al. [44] evaluate the availability of microservices deployed under Kubernetes. Their study offers a quantitative assessment of Kubernetes' availability in default configurations and identifies practical issues, though it does not cover other security settings.

Medel et al. [45] present a performance and resource management model for Kubernetes, using real-world data. Their work aids in designing scalable applications and understanding Kubernetes' system behavior but does not address client-side security issues.

[3] *Kubernetes*. Available at: https://kubernetes.io/.

2.3.2. Kubernetes security

Prior research on Kubernetes security has largely centered on system configurations and vulnerability detection. Bose et al. [46] conducted a quantitative analysis of manifest vulnerabilities, highlighting the rarity of explicit flaws but emphasizing the risk of configuration errors. D'Silva et al. [47] proposed a Zero Trust framework for Kubernetes, focusing on dynamic authorization but not addressing runtime defense against volumetric attacks. Shamim [48] explored misconfiguration-related vulnerabilities and advocated mitigation through security policies and hardware-level controls.

For DDoS-specific research, David et al. [49] investigated the YoYo attack on Google Kubernetes Engine, introducing an XGBoost-based detection method. However, their work emphasized detection in managed cloud services rather than mitigation at the service orchestration layer. Similarly, Koksal et al. [23] developed an IDPS auto-scaling defense for DNS and Yo-Yo attacks in Kubernetes-based MEC networks, yet their evaluation was limited to DNS services and did not explore platform-level orchestration dynamics. Tripathi [22] analyzed Kubernetes vulnerabilities through the Kubernetes Goat framework, aligning them with the MITRE ATT&CK model, but focused solely on exploit identification rather than resilience assessment during attack execution.

Despite these valuable contributions, existing studies tend to treat Kubernetes either as a configuration target or a passive environment for threat detection, with limited exploration of how orchestration mechanisms—such as pod scheduling, dynamic resource allocation, or load balancing—interact with mitigation strategies under active DDoS conditions. None of the surveyed literature systematically evaluates service-layer defenses from an orchestration aware perspective.

Our work addresses this gap by introducing a taxonomy of twelve mitigation strategies and evaluating them across both Docker and Kubernetes environments. By simulating realistic service workloads—including static content delivery, database transactions, and dynamic page generation—under varying concurrency levels, we assess how orchestration behavior influences the performance of each strategy. Metrics such as test completion time, response latency, and failure rates are used to reveal platform-specific trade-offs. To our knowledge, this is the first study to provide a comparative, orchestration-sensitive evaluation of DDoS mitigation strategies in Kubernetes environments under active attack scenarios.

2.4. Research gap and our contributions

While prior studies have explored the benefits of containerization in deployment and scalability, most security-oriented research—such as Chelladhurai et al. [40], Patidar et al. [26], David et al. [49], and Shamim [48]—has focused on infrastructure-level vulnerabilities or centralized detection mechanisms. Few have evaluated how mitigation strategies perform in dynamic, orchestrated container environments.

Recent works from 2024–2025, including Tripathi's vulnerability exploration [22], Koksal's IDPS auto-scaling framework [23], Garcia's ML-based testbed [24], and Gamess and Parajuli's container profiling study [25], provide valuable insights into system behavior and detection accuracy. However, they do not evaluate strategy-level mitigation performance across platforms, nor do they consider orchestration-specific behaviors such as resource scheduling and load balancing under attack.

Although Patidar et al. [26] proposed effective mitigation strategies in Docker, their evaluation was limited to a single platform and did not address concurrency variation, application diversity, or orchestration effects.

Our work addresses these gaps by:

- Proposing a taxonomy of twelve mitigation strategies that combine page separation, resource prioritization, and DDoS filtering.
- Evaluating these strategies under realistic service types—including static content and database-driven systems—on both Docker and Kubernetes platforms.

- Simulating low-rate DDoS scenarios with varying concurrency to model stealthy attack patterns.
- Measuring client-side metrics such as response time, failure rate, and recovery time for practical deployment insight.
- Analyzing how orchestration mechanisms (e.g., pod scheduling, autoscaling) influence the effectiveness of each strategy.

This orchestration-aware, strategy-centric approach fills a critical gap in current research and contributes both methodological and practical guidance for enhancing service resilience in cloud-native infrastructures.

3. Proposed strategies

In the study by Patidar et al. [26], a set of 12 strategies was proposed for countering malicious users through page separation and resource allocation. However, the paper lacked detailed explanations on specific configurations and methodologies. They also referenced HAProxy[4] without comprehensive details. The experimental scope was limited to Docker containers and specific scenarios like authentication, browsing, and signup pages.

In our research, we applied the same 12 strategies with our interpretations during implementation. We extended the study to include both Docker and Kubernetes, aiming to evaluate their performance in mitigating DDoS attacks and compare service stability. Our experiments covered additional scenarios, such as the Nginx Sample Page, Ticket Booking System, and E-Commerce Website, differing from the reference paper. Details of these scenarios are explained in the following paragraphs.

To evaluate different container-based DDoS mitigation strategies in a Kubernetes environment, we designed twelve strategies combining various traffic handling, resource allocation, and mitigation mechanisms. Each strategy was deployed using six pods, a backend controller, and separate servers simulating both benign users and attackers. Below, we summarize the core strategy types:

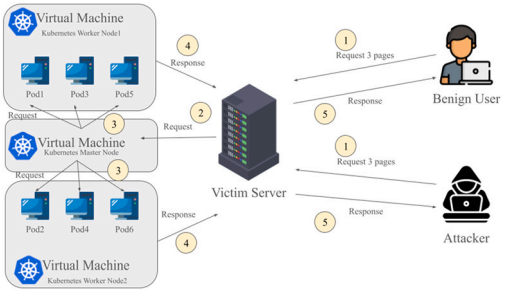

Fig. 1. Strategy 1 diagram.

- Strategy 1 (Baseline): All requests were processed without any separation or mitigation. All six pods handled traffic uniformly (see Fig. 1).
- Strategy 2 (Page-Based Separation): Pods were grouped by page type (e.g., Nginx, ticket booking, e-commerce) to reduce resource contention (See Fig. 2).
- User-Based Prioritization:
  - Strategy 3 (Whitelist Priority): Whitelisted users were allocated five pods; suspicious users were limited to one (See Fig. 3).
  - Strategy 4 (Suspicious Priority): The reverse of Strategy 3, favoring suspicious users with five pods (see Fig. 4).
  - Strategy 5 (Equal Allocation): Whitelisted and suspicious users were each allocated three pods (see Fig. 5).

[4] HAProxy, 2024. Available at: https://www.haproxy.com/.

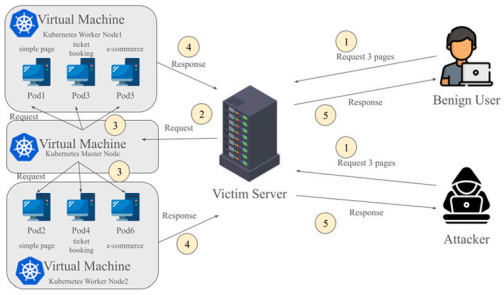

Fig. 2. Strategy 2 diagram.

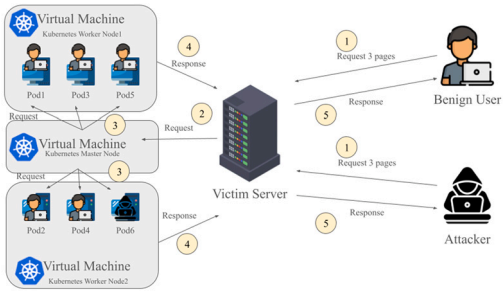

Fig. 3. Strategy 3 diagram.

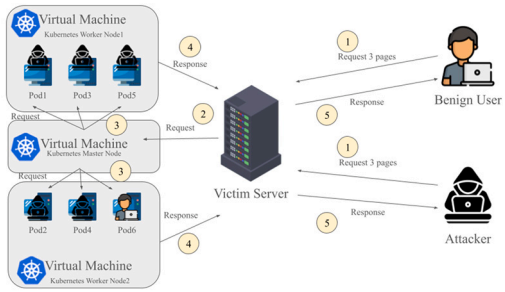

Fig. 4. Strategy 4 diagram.

- DDoS Mitigation with Connection Limit (DDoS connection limit): This category applies a uniform DDoS protection rule across all included strategies: each client IP is limited to a maximum of 150 connections; requests exceeding this limit are blocked.
  - Strategy 6 (Strategy 1 + DDoS Connection limit): Enhances the baseline (Strategy 1) by adding the DDoS connection limit. (See Fig. 6).
  - Strategy 7 (Strategy 2 + DDoS Connection limit): Extended Page-Based Separation (Strategy 2) with DDoS connection limit.
  - Strategy 8 (Strategy 3 + DDoS Connection limit): Whitelist Priority (Strategy 3) plus DDoS connection limit.
  - Strategy 9 (Strategy 4 + DDoS Connection limit): Suspicious Priority (Strategy 4) plus DDoS connection limit.
  - Strategy 10 (Strategy 5 + DDoS Connection limit): Equal Allocation (Strategy 5) with added DDoS connection limits.
- Hybrid Strategies:
  - Strategy 11 (Strategy 2 + Strategy 3): Combined Strategy 2 (Page-based separation) with Strategy 3 (Whitelist Priority) to allocate five pods to whitelisted users based on requested page type and reserves one pod for suspicious users.
  - Strategy 12 (Strategy 11 + DDoS Connection limit): Strategy 11 enhanced with the DDoS connection limit.

In all strategies, incoming traffic was first directed to a backend server, which routed it to the Kubernetes master node. The master node assigned traffic to idle pods based on predefined rules (by page type or

Table 1
Comparative Summary of the 12 DDoS Mitigation Strategies.

Strat	Sep. Type	User Priority	DDoS Mitigation	Strength	Weakness
S1	None	None	No	Simple baseline for comparison	No protection or traffic control
S2	Page-based	None	No	Reduces resource contention across pages	No user-level prioritization or DDoS protection
S3	User-based	Whitelist	No	Prioritizes trusted users	Attackers can still consume limited resources
S4	User-based	Suspicious	No	Stresses system under attack	May degrade service for benign users
S5	User-based	Equal	No	Fair resource distribution	No dynamic adaptation or protection
S6	None	None	Yes	Adds basic DDoS protection via connection limits	No traffic or user separation
S7	Page-based	None	Yes	Protects services across pages with DDoS rule	Does not prioritize user types
S8	User-based	Whitelist	Yes	Prioritizes whitelisted users and blocks excess traffic	One pod may still be overwhelmed by attackers
S9	User-based	Suspicious	Yes	Tests under high-volume attacks with protection	Limited benefit to benign users
S10	User-based	Equal	Yes	Balanced handling with basic protection	May not optimize for either group
S11	Page + User-based	Whitelist	No	Combines service separation and trusted user prioritization	Vulnerable to DDoS without limits
S12	Page + User-based	Whitelist	Yes	Comprehensive strategy with prioritization, separation, and DDoS protection	More complex setup and overhead

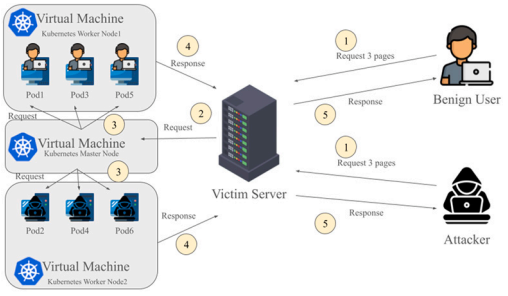

Fig. 5. Strategy 5 diagram.

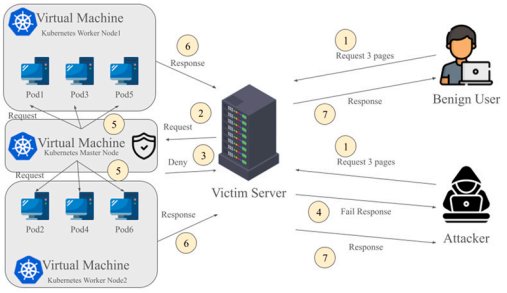

Fig. 6. Strategy 6 diagram.

user type). For strategies with DDoS mitigation, requests exceeding the connection limit were dropped early in the backend.

To improve clarity and facilitate comparison among the proposed strategies, Table 1 summarizes all 12 strategies mentioned above. In addition, this table also highlights the main strength and weakness of each strategy, providing a concise overview of their design characteristics and potential impact.

4. Experiment

4.1. Environment

This study aims to compare the survivability of Kubernetes and Docker container platforms under Distributed Denial of Service (DDoS) attacks. To accomplish this, we designed a series of experiments to evaluate the performance of these two platforms under different application scenarios.

4.1.1. Kubernetes

First, we set up a Kubernetes cluster with one master node and two worker nodes.[5] Fig. 7 shows the cluster architecture, including these nodes [50]. This study uses Kubernetes 1.28.1,[6] which offers improvements like enhanced resource management and better scalability.[7] This research employs `kubectl` for cluster management and scheduling. The master node comprises API Server, Scheduler, Controller Manager, and etcd. The worker node includes kubelet, kube-proxy, and Pod.

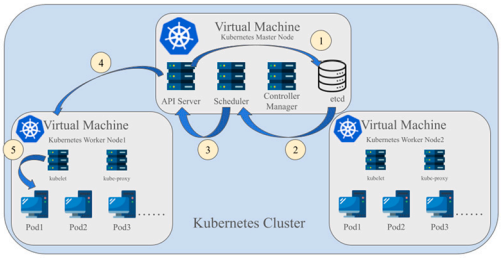

Fig. 7. Kubernetes architecture.

To deploy a service, a user initiates a Pod creation request via `kubectl`.[8] The process involves:

1. The API Server writes the request to etcd and updates the cluster state.
2. The Scheduler assigns the Pod to a Worker node based on resource usage and policies.
3. The Scheduler informs the API Server of the assignment.
4. The API Server notifies the kubelet on the chosen Worker node.
5. The kubelet pulls the container image and starts the container, running the Pod.

[5] *How to Install Kubernetes 1.26 on Ubuntu 22.04 LTS*. Available at: https://www.youtube.com/watch?v=7k9Rdlx30OY.
[6] *Kubernetes 1.28.1 change log*. Available at https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.28.md.
[7] IBM, *Kubernetes 7 advantages*. Available at https://www.ibm.com/blog/top-7-benefits-of-kubernetes/.
[8] *kubectl*. Available at: https://kubernetes.io/docs/reference/kubectl/.

4.1.2. Docker

We also set up an environment using Docker and configured the web service.[9] Fig. 8 shows the Docker architecture used, including Docker Daemon and Docker Registry. Users deploy services by issuing commands to the Docker Daemon via the Docker CLI. Containers are the smallest unit for deploying and managing applications in Docker.

We used Docker version 26.1.2, which offers performance and security improvements.[10] This study uses Docker to run a single container environment and compare its performance under DDoS attacks. These setups provide the foundation for comparing Kubernetes and Docker performance in DDoS scenarios, ensuring reliable, scalable, and reproducible results[11] [37].

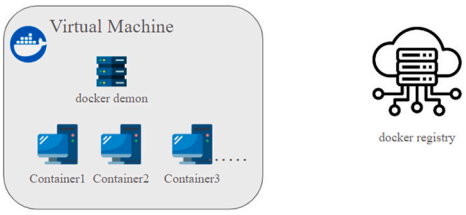

Fig. 8. Docker architecture.

4.2. Real-world web applications

To evaluate Kubernetes and Docker's resilience and performance, we devised three experimental scenarios simulating typical network applications under DDoS attacks. We containerized these websites into custom images and deployed them on both platforms. The scenarios are:

1. **Simple Nginx Page**: Deployed a basic Nginx page on Kubernetes and Docker to benchmark static content handling against DDoS attacks. Data totals 772 bytes.
2. **Simulated Ticket Booking System**: Modeled a ticket booking system to test performance with frequent database writes and DDoS resilience. Each visit triggers a write operation. Data totals 775 bytes.
3. **Simulated E-Commerce Website**: Emulated an e-commerce site with high database read demands, including product searches and order processing. Evaluates performance under heavy read loads and DDoS attacks. Data totals 4036 bytes.
   The e-commerce site uses HTML, JavaScript, and CSS, with a backend database containing diverse product data from Kaggle.[12,13] This setup reflects real-world e-commerce environments, providing a dynamic testing scenario.

4.3. Hardware configuration

4.3.1. Experiment platform

As a virtualization platform, we used Proxmox VE 8.1.3.[14] Proxmox VE, an open-source platform combining KVM (Kernel Virtual Machine) and LXC (Linux Containers), offers a web-based management interface with support for high availability clustering, storage replication, and backups, enhancing flexibility and efficiency in managing the experimental environment.[15]

4.3.2. Operating system

We used Ubuntu 22.04 for its stability, security, and support for container technologies. The x86-64-v2-AES architecture, with 8 GB RAM and a 4-core CPU, ensures compatibility with modern standards and efficient processing for web services and DDoS simulations. Ubuntu 22.04 serves as the base platform for Kubernetes and Docker environments, providing robustness and extensive package support for our experiments.

4.4. Image

An image is a self-contained, read-only operating system environment, used as a template for creating Containers. The image used in this experiment includes:

1. **MySQL 8.4.0**: An open-source RDBMS known for its performance, security, and scalability, suitable for handling large data and concurrent access.
2. **Nginx 1.17.4**: A high-performance HTTP and reverse proxy server, efficient in handling high traffic with load balancing features.
3. **Alpine:3.7 + Python Flask**: Alpine 3.7, a lightweight Linux distribution, is used to build our Python Flask app (Flask 1.0.2) due to its security and minimal footprint.

4.5. DDoS attack simulation tool

When evaluating DDoS simulation tools, we focused on:

1. **Simulation Capability**: Tools must generate high volumes of requests or packets to stress the target system.
2. **Flexibility**: Tools should allow customization of attack parameters like size and frequency.
3. **Accuracy**: Tools must provide precise results that mimic real-world DDoS impacts.
4. **Usability**: Tools should be user-friendly and readily available for easy deployment.

Based on these criteria, we selected Apache Bench for our DDoS attack simulations.

4.5.1. Apache bench

Apache Bench (ab) [16] is a command-line tool for testing HTTP server performance by simulating multiple concurrent users and analyzing server load [51,52]. Key features include:

- **High Concurrent Access**: Simulates multiple users with concurrent HTTP requests.
- **Customizable Load**: Allows specifying total requests and concurrency levels.
- **Detailed Reports**: Provides metrics like request processing time and success rate.
- **User-Friendly**: Simple command-line interface for quick performance testing.

Commands used:

1. Attacker: **ab -n 10000 -c [50,100,200,500] URL**

[9] *Install Docker Engine on Ubuntu*. Available at: https://docs.docker.com/engine/install/ubuntu/.
[10] Dimensiona, *What is Docker and what are its advantages*. Available at: https://www.dimensiona.com/en/what-is-docker-and-what-are-its-advantages/.
[11] *Why Docker*. Available at https://www.docker.com/why-docker/.
[12] KVPRATAMA, *Pokemon Images Dataset*, 2020. Available at: https://www.kaggle.com/datasets/kvpratama/pokemon-images-dataset/data.
[13] ROUNAK BANIK, *The Complete Pokemon Dataset*, 2017. Available at: https://www.kaggle.com/datasets/rounakbanik/pokemon/data.
[14] *Proxmox VE Administration Guide*. Available at: https://pve.proxmox.com/pve-docs/pve-admin-guide.html.
[15] *Proxmox Official Website*. Available at: https://www.proxmox.com/en/.
[16] *Apache HTTP server benchmarking tool*. Available at: https://httpd.apache.org/docs/2.4/en/programs/ab.html.

2. Benign user: **ab -c 1 -n 1000 URL**

Here, *n* is the total number of requests, *c* is concurrency, and URL is the target web page.

4.5.2. hping3

hping3[17] is a command-line tool for network testing and security auditing, useful for simulating various network attacks, including low-rate DDoS attacks. Key features include:

- **Custom Packet Creation**: Allows crafting packets with specific flags, sizes, and intervals.
- **Protocol Support**: Supports TCP, UDP, ICMP, and RAW-IP protocols.
- **Advanced Attack Simulation**: Simulates attacks like SYN floods and ACK floods.
- **Flexible Timing and Transmission**: Controls packet timing and data volume for accurate DDoS simulations.
- **Real-time Monitoring**: Provides feedback on responses from the target.

Commands used: **hping3 -c 100000 -i u250 -S -p 80 URL**

Here, *c* is the count of response packets, *i* is the packet interval, *S* is for SYN packets, *p* specifies the target port, and URL is the web page location.

In experiments, varying numbers of attackers (1 to 4) simulate different DDoS scenarios. A custom Python script simulates benign traffic by sending requests every 0.1 seconds. A lack of response indicates server overload due to hping3 attacks.

5. Result

In this section, we discuss the experimental methods described in Section 4 and the twelve strategies outlined in Section 3 to conduct the experiments. We tested Docker and Kubernetes under the same settings and recorded detailed data to compare the two. The provided tables include the following details: experiment names, different concurrency levels (50, 100, 200, 500), various strategies (1 to 12), and requests from benign user and attacker. The requests are further divided into three pages for testing: nginx, dbwrite, and dbread. The data includes the total time taken to complete the 'ab' command, the maximum and minimum response times, and the number of failed packet transmissions.

5.1. Apache bench result

5.1.1. Performance metrics

In this section, we describe the following performance metrics which were used in the experiment: 1) test completion time, 2) minimum response time, 3) maximum response time, and 4) number of failed packet transmissions. Each metric is explained below:

1. Test Completion Time: As mentioned in Section 4.5.1, this experiment simulates real-world scenarios with two types of users: attackers and benign users. The test completion time records the total duration for both users to execute the Apache Bench (ab) command mentioned in Section 4.5.1, including the time taken to send all packets under varying concurrency levels and receive all responses from the victim server.
2. Minimum Response Time: The minimum response time, recorded using the Apache Bench (ab) command, is the shortest time taken to receive a response during the process of sending request packets. In other words, it represents the quickest response among all packets upon completing the command.
3. Maximum Response Time: The maximum response time, recorded using the Apache Bench (ab) command, is the longest time taken to receive a response during the process of sending request packets. It represents the slowest response among all packets upon completing the command.
4. Number of Failed Packet Transmissions: This metric indicates the count of request packets that did not receive a response due to network traffic being blocked by DDoS mitigation strategies, specifically those used in strategies 6 through 10 and 12. When the Apache Bench (ab) command sends request packets, some packets may fail to receive responses as a result of these mitigation measures. The ab command records and tallies these failed responses, providing a measure of the total failed packet transmissions out of the total packets sent.

To further investigate whether the observed differences in performance are statistically significant, we applied a two-way ANOVA (analysis of variance) to assess the effects of both Strategy and PageType on the measured outcomes. This method allows us to examine: (1) whether each factor independently influences performance (main effects), and (2) whether their combination produces an additional effect (interaction effect). Given that the experimental data include multiple groups of Strategy and PageType with continuous performance metrics, two-way ANOVA is an appropriate and rigorous statistical tool for this analysis.

5.1.2. Docker environment

Test completion time Tables 2 and 3 display the test completion times, and Fig. 9 provides a visual summary of these results.

In these tables, the dbwrite page requires more time compared to the other two pages. This is due to the complexity of the database write operations involved. In addition, as observed from these two tables, when the concurrency level is set to 500, some entries are marked as NA (not available). This occurs because at this level of concurrency, the command on the attacker side fails. Recording the benign user data under these conditions would imply the absence of an attacker, thus NA is used to indicate this scenario. Furthermore, the attention should be given to the total time for benign user, as they represent real users; the shorter the total time, the better. For attacker, due to the DDoS mitigation strategy, the total completion time may appear to be shortened, but in reality, their malicious packets do not arrive.

As shown in Tables 2 and 3, for the same webpage, if the response time for a benign user request is longer than that for an attacker request, it indicates that the DDoS attack has effectively impacted the benign user. In this scenario, the network bandwidth is fully utilized by the attacker requests until they are completely processed, causing a delay in handling benign user requests. Conversely, if the response time for benign user requests is shorter, it signifies that the strategy in place can prioritize processing benign user requests before gradually addressing the attacker requests.

At concurrency levels of 50, 100, and 200, strategy 11 (page separation with maximum resources to benign users) and 12 (page separation with maximum resources to benign users with DDoS mitigation) demonstrate comparable optimal performance. They result in shorter test completion times for benign user across all three pages while requiring longer test completion times for attacker. Conversely, strategy 2 and 7 exhibit poorer performance. This indicates that simply employing page separation does not effectively reduce the required request time. Instead, combining page separation with strategies such as allocating maximum resources to known benign user and implementing DDoS mitigation techniques is necessary to effectively reduce test completion times.

At a concurrency of 500, many strategy data are marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Since some of the data for Strategy 11

[17] *hping3(8) - Linux man page*. Available at: https://linux.die.net/man/8/hping3.

Table 2
Test completion time in seconds for different strategies on different pages for attacker and benign user in Docker environment under 50 and 100 concurrency levels (NA: Not available).

Test completion time in seconds		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	2.278	35.914	2.355	1.584	23.946	1.635
	2	4.076	73.119	0.504	3.449	61.176	0.41
	3	1.078	14.764	1.139	6.396	115.775	6.458
	4	1.749	18.196	1.852	1.688	27.512	1.736
	5	1.344	16.005	1.392	2.426	41.755	2.508
	6	2.283	34.97	2.355	1.575	23.632	1.618
	7	4.161	72.329	0.499	3.536	60.775	0.409
	8	1.059	14.619	1.074	6.247	114.788	6.365
	9	1.792	18.244	1.837	1.72	27.436	1.773
	10	1.26	15.864	1.357	2.442	42.074	2.485
	11	0.949	14.39	0.479	6.213	113.966	0.394
	12	0.925	14.396	0.492	6.146	115.864	0.4
100	1	2.352	37.867	2.377	1.543	24.353	1.578
	2	4.088	74.243	0.485	3.389	60.917	0.384
	3	1.074	14.778	1.079	7.704	116.196	6.417
	4	1.931	19.072	1.961	1.669	28.186	1.688
	5	1.329	16.233	1.454	2.46	42.573	2.494
	6	2.293	37.597	2.285	1.507	24.101	1.507
	7	4.192	73.734	0.509	3.482	60.463	0.404
	8	1.07	14.718	1.094	7.029	115.252	6.372
	9	1.929	19.116	1.993	1.683	28.244	1.728
	10	1.331	16.085	1.35	2.475	42.752	2.525
	11	0.919	14.4	0.507	6.275	114.28	0.395
	12	0.955	14.416	0.493	6.327	115.952	0.374

Table 3
Test completion time in seconds for different strategies on different pages for attacker and benign user in Docker environment under 200 and 500 concurrency levels (NA: Not available).

Test completion time in seconds		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	2.392	38.998	2.372	1.517	24.647	1.54
	2	4.126	74.765	0.497	3.375	60.748	0.379
	3	1.127	15.074	1.074	7.125	139.348	7.755
	4	1.975	19.382	1.882	1.673	28.586	1.726
	5	1.438	16.271	1.421	2.476	42.549	2.509
	6	1.764	15.645	1.691	0.911	0.839	0.821
	7	1.279	15.906	0.514	0.481	1.225	0.38
	8	1.209	14.594	1.251	1.038	2.123	1.455
	9	1.6	14.282	1.402	0.954	1.034	0.799
	10	1.31	14.466	1.27	0.646	1.075	0.649
	11	0.926	14.481	0.501	7.82	135.541	0.39
	12	1.112	14.037	0.521	1.237	2.066	0.39
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	0.511	NA	NA	0.477
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	1.683	15.648	1.682	0.839	0.84	0.814
	7	1.306	15.841	0.518	0.513	1.243	0.535
	8	2.179	14.611	1.261	0.473	2.072	0.484
	9	1.474	14.22	1.447	0.8	0.891	0.777
	10	1.229	14.566	1.262	0.496	1.012	0.501
	11	NA	NA	0.509	NA	NA	0.567
	12	1.208	14.205	0.518	1.179	2.059	0.389

are marked as NA, Strategy 12 emerges as the best strategy, exhibiting relatively good performance and consistent results at lower concurrency levels (50, 100, 200). The worst-performing strategy is Strategy 8. The reason for this is that, at very high concurrency, attackers consume a significant amount of resources. Consequently, allocating substantial resources to benign users under such conditions leads to prolonged test completion times.

Table 4 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 2 and 3. The statistical results indicate that, for both benign users and attackers, Strategy and PageType each exhibit significant main effects. Moreover, there is a significant interaction effect between the two factors, suggesting that the effectiveness of certain strategies varies depending on the page type.

Speed and bandwidth analysis Tables 5, 6 display the speed of DDoS attacks, and Fig. 10 provides a visual summary of these results. Tables 8 and 9 display the bandwidth of the DDoS attacks, and Fig. 11 offers a graphical representation of these findings. These tables provide additional insights into the attack characteristics and the impact on the system.

Table 4
Two-Way ANOVA and Interaction Effects for Tables 2 and 3.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	3963.946523	11	8.871051406	1.49342E-10	Y
	PageType	15836.14931	2	194.9214289	8.32895E-34	Y
	Strategy : PageType	7842.484491	22	8.775482043	2.05127E-14	Y
	Residual	3737.212847	92			
Suspicious	Strategy	19107.506	11	4.844476649	7.28662E-06	Y
	PageType	58166.3135	2	81.1105024	4.94896E-21	Y
	Strategy : PageType	32401.55726	22	4.107511141	8.61886E-07	Y
	Residual	32987.71851	92			

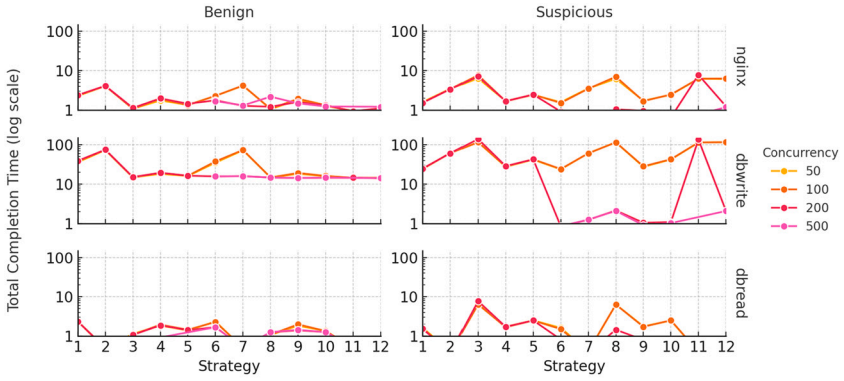

Fig. 9. Total test completion time (in seconds) for 12 mitigation strategies in a Docker environment, evaluated across three types of resource pages (nginx, dbwrite, dbread) and two user categories (benign and suspicious) under four concurrency levels (50, 100, 200, 500). The Y-axis uses a logarithmic scale (1–150) to highlight differences in completion time across strategies while preserving the visibility of low-latency cases.

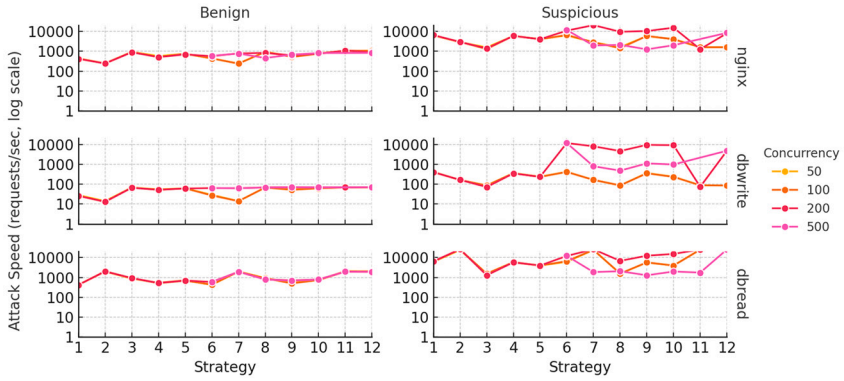

Fig. 10. DDoS attack speed (requests per second) across 12 mitigation strategies in a Docker environment, measured on three types of resource pages (nginx, dbwrite, dbread) for benign and suspicious users under four concurrency levels (50, 100, 200, 500). The Y-axis uses a logarithmic scale (1–21000) to better visualize wide-ranging request rates across different strategies and concurrency settings.

At concurrency levels of 50 and 100, Strategy 11 (page separation with maximum resources to benign users) and Strategy 12 (page separation with maximum resources to benign users with DDoS mitigation) demonstrate optimal performance in terms of speed. They achieve the highest requests per second for benign users across all three webpages, while maintaining lower request rates for attackers. Conversely, Strategy 2 and Strategy 7 exhibit poorer performance in terms of speed. This indicates that simply employing page separation does not effectively optimize request handling times. Instead, combining page separation with strategies such as allocating maximum resources to known benign users and implementing DDoS mitigation techniques is necessary to achieve higher speed efficiency.

At a concurrency level of 200, Strategy 11 demonstrates the best performance in terms of speed, achieving the highest requests per second for benign users. Strategy 2, on the other hand, performs the worst, indicating that simple page separation without additional mitigation measures is ineffective in optimizing request handling times at higher concurrency levels.

At a concurrency level of 500, many strategy results are marked as NA due to the rapid transmission rate of packets by the suspicious server, leading to disconnections from the victim server and incomplete experimental data transmission. While some data for Strategy 11 are marked as NA, Strategy 12 emerges as the best strategy with consistently high performance in terms of speed at lower concurrency levels (50, 100, 200). Conversely, Strategy 8 performs the worst in terms of speed. This is attributed to the significant resource consumption by attackers at very high concurrency levels, resulting in lower request handling times despite substantial resource allocation to benign users.

Table 7 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 5 and 6. The analysis shows that both Strategy and PageType individually have significant effects on

Table 5
DDoS Attack Speed (Requests per Second) in Docker environment under 50 and 100 concurrency levels.

DDoS Attack Speed		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	439	28	425	6313	418	6116
	2	245	14	1984	2899	163	24390
	3	928	68	878	1563	86	1548
	4	572	55	540	5924	363	5760
	5	744	62	718	4122	239	3987
	6	438	29	425	6349	423	6180
	7	240	14	2004	2828	165	24450
	8	944	68	931	1601	87	1571
	9	558	55	544	5814	364	5640
	10	794	63	737	4095	238	4024
	11	1054	69	2088	1610	88	25381
	12	1081	69	2033	1627	86	25000
100	1	425	26	421	6481	411	6337
	2	245	13	2062	2951	164	26042
	3	931	68	927	1298	86	1558
	4	518	52	510	5992	355	5924
	5	752	62	688	4065	235	4010
	6	436	27	438	6636	415	6636
	7	239	14	1965	2872	165	24752
	8	935	68	914	1423	87	1569
	9	518	52	502	5942	354	5787
	10	751	62	741	4040	234	3960
	11	1088	69	1972	1594	88	25316
	12	1047	69	2028	1581	86	26738

Table 6
DDoS Attack Speed (Requests per Second) in Docker environment under 200 and 500 concurrency levels.

DDoS Attack Speed		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	418	26	422	6592	406	6494
	2	242	13	2012	2963	165	26385
	3	887	66	931	1404	72	1289
	4	506	52	531	5977	350	5794
	5	695	61	704	4039	235	3986
	6	567	64	591	10977	11919	12180
	7	782	63	1946	20790	8163	26316
	8	827	69	799	9634	4710	6873
	9	625	70	713	10482	9671	12516
	10	763	69	787	15480	9302	15408
	11	1080	69	1996	1279	74	25641
	12	899	71	1919	8084	4840	25641
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	594	64	595	11919	11905	12285
	7	766	63	1931	1949	805	1869
	8	459	68	793	2114	483	2066
	9	678	70	691	1250	1122	1287
	10	814	69	792	2016	988	1996
	11	NA	NA	1964	NA	NA	1763
	12	828	70	1931	8482	4857	25707

the outcomes, and a clear interaction effect is also observed between the two factors.

Tables 8 and 9 shows the bandwidth of the attack, measured in bytes per second. This metric indicates the volume of data transmitted during the attack. Higher values suggest a larger amount of data being sent, which can lead to higher network resource consumption. The byte data for each webpage (nginx: 772 bytes, dbwrite: 775 bytes, dbread: 4036 bytes) are sourced from 4.2.

At concurrency levels of 50 and 100, Strategy 11 (page separation with maximum resources to benign users) and Strategy 12 (page separation with maximum resources to benign users with DDoS mitigation) demonstrate optimal performance in terms of bandwidth. They achieve the highest data transmission rates (bytes per second) for benign users across all three webpages, while maintaining lower transmission rates for attackers. Conversely, Strategy 2 and Strategy 7 exhibit poorer performance in terms of bandwidth. This indicates that simply employing page separation does not effectively optimize data transmission rates. Instead, combining page separation with strategies such as allocating maximum resources to known benign users and implementing DDoS

Table 7
Two-way ANOVA Results for Table 5 and 6.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	7822881.945	11	100.8655181	5.37863E-46	Y
	PageType	24187407.24	2	1715.252075	5.75771E-73	Y
	Strategy : PageType	12115788.55	22	78.10838115	1.61668E-49	Y
	Residual	641612.4167	91			
Suspicious	Strategy	1191892858	11	5.126638069	3.30976E-06	Y
	PageType	1984326962	2	46.94305651	9.82286E-15	Y
	Strategy : PageType	2222487972	22	4.779746506	5.14534E-08	Y
	Residual	1923327612	91			

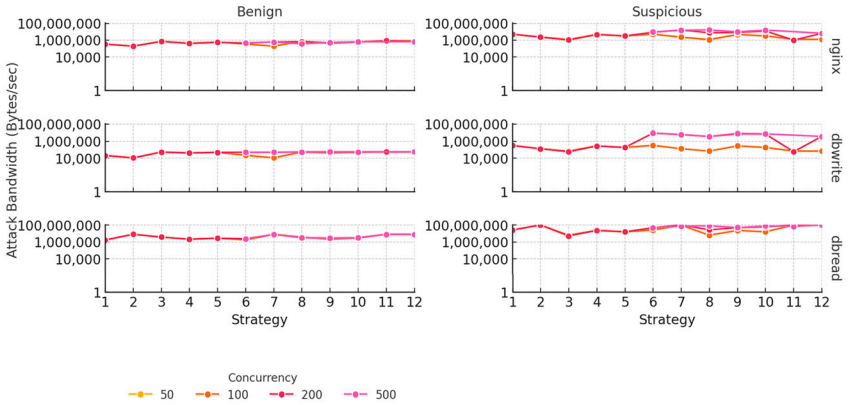

Fig. 11. DDoS attack bandwidth (Bytes per second) for 12 mitigation strategies in a Docker environment, evaluated across three types of resource pages (nginx, dbwrite, dbread) and two user categories (benign and suspicious) under four concurrency levels (50, 100, 200, 500). The Y-axis is plotted on a logarithmic scale (from 1 to 100,000,000) to clearly display bandwidth differences across strategies while preserving the visibility of lower-bandwidth cases.

Table 8
DDoS Attack Bandwidth (Bytes per Second) in Docker environment under 50 and 100 concurrency levels.

DDoS Attack Bandwidth		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	338894	21579	1713800	4873737	323645	24685015
	2	189401	10599	8007937	2238330	126684	98439024
	3	716141	52493	3543459	1207004	66940	6249613
	4	441395	42592	2179266	4573460	281695	23248848
	5	574405	48422	2899425	3182193	185607	16092504
	6	338152	22162	1713800	4901587	327945	24944376
	7	185532	10715	8088176	2183258	127520	98679707
	8	728990	53013	3757914	1235793	67516	6340927
	9	430804	42480	2197060	4488372	282476	22763677
	10	612698	48853	2974208	3161343	184199	16241449
	11	813488	53857	8425887	1242556	68003	102436548
	12	834595	53834	8203252	1256102	66889	100900000
100	1	328231	20466	1697939	5003240	318236	25576679
	2	188845	10439	8321649	2277958	127222	105104167
	3	718808	52443	3740500	1002077	66698	6289543
	4	399793	40635	2058134	4625524	274959	23909953
	5	580888	47742	2775791	3138211	182040	16182839
	6	336677	20613	1766302	5122760	321563	26781685
	7	184160	10511	7929273	2217117	128178	99900990
	8	721495	52657	3689214	1098307	67244	6333961
	9	400207	40542	2025088	4587047	274395	23356481
	10	580015	48182	2989630	3119192	181278	15984158
	11	840044	53819	7960552	1230279	67816	102177215
	12	808377	53760	8186613	1220168	66838	107914439

mitigation techniques is necessary to achieve higher bandwidth efficiency.

At a concurrency level of 200, Strategy 11 demonstrates the best performance in terms of bandwidth, achieving the highest data transmission rates for benign users. Strategy 2, on the other hand, performs the worst, indicating that simple page separation without additional mitigation measures is ineffective in optimizing data transmission rates at higher concurrency levels.

At a concurrency level of 500, many strategy results are marked as NA due to the rapid transmission rate of packets by the suspicious server, leading to disconnections from the victim server and incomplete experimental data transmission. While some data for Strategy 11 are marked as NA, Strategy 12 emerges as the best strategy with consistently high performance in terms of bandwidth at lower concurrency levels (50, 100, 200). Conversely, Strategy 8 performs the worst in terms of bandwidth. This is attributed to the significant resource consumption by attackers at

Table 9
DDoS Attack Bandwidth (Bytes per Second) in Docker environment under 200 and 500 concurrency levels.

DDoS Attack Bandwidth		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	322742	19873	1701518	5088991	314440	26207792
	2	187106	10366	8120724	2287407	127576	106490765
	3	685004	51413	3757914	1083509	55616	5204384
	4	390886	39986	2144527	4614465	271112	23383546
	5	536857	47631	2840253	3117932	182143	16086090
	6	437642	49537	2386753	8474204	9237187	49159562
	7	603597	48724	7852140	16049896	6326531	106210526
	8	638544	53104	3226219	7437380	3650495	27738832
	9	482500	54264	2878745	8092243	7495164	50513141
	10	589313	53574	3177953	11950464	7209302	62187982
	11	833693	53518	8055888	987212	57178	103487179
	12	694245	55211	7746641	6240905	3751210	103487179
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	458705	49527	2399524	9201430	9226190	49582310
	7	591118	48924	7791506	15048733	6234916	75439252
	8	354291	53042	3200634	16321353	3740347	83388430
	9	523745	54501	2789219	9650000	8698092	51943372
	10	628153	53206	3198098	15564516	7658103	80558882
	11	NA	NA	7929273	NA	NA	71181657
	12	639073	54558	7791506	6547922	3763963	103753213

Table 10
Two-way ANOVA and Interaction Effects Results for Table 8 and 9.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	1.03176E+14	11	411.1424613	1.74112E-72	Y
	PageType	5.14717E+14	2	11280.8778	9.5143E-110	Y
	Strategy : PageType	1.87766E+14	22	374.109493	1.69722E-79	Y
	Residual	2.07605E+12	91			
Suspicious	Strategy	1.78239E+16	11	13.81680543	3.60807E-15	Y
	PageType	6.98019E+16	2	297.6022247	1.19605E-40	Y
	Strategy : PageType	3.45036E+16	22	13.37336259	7.16508E-20	Y
	Residual	1.06719E+16	91			

very high concurrency levels, resulting in lower data transmission rates despite substantial resource allocation to benign users.

These additional metrics of speed and bandwidth complement the test completion time results, providing a more comprehensive view of the DDoS attack impact. The speed metric helps to understand the request density, while the bandwidth metric provides insights into the volume of data being transmitted during the attack. Both metrics are crucial for evaluating the overall strain on the system and the effectiveness of different mitigation strategies.

Table 10 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 8 and 9. In the Docker environment, the bandwidth usage under DDoS attacks is significantly influenced by both Strategy and the targeted PageType. For both benign and attack traffic, Strategy and PageType each cause significant variations in bandwidth consumption, and a notable interaction effect is observed between the two factors.

Minimum response time Tables 11 and 12 display the minimum response times, and Fig. 12 offers a graphical representation of these findings. Some values are shown as 0 because the ab command's request and response time is extremely fast, less than 0.001 seconds, and thus cannot be displayed to four decimal places in the table. This does not imply that the request was not sent. As observed in these tables, the dbwrite page has a significantly higher minimum response time compared to the other two pages, while the nginx and dbread pages have similar minimum response times. This again indicates the high resource consumption of the dbwrite page.

In addition, when the concurrency is at 500, many strategy data are marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data.

Across all concurrency levels (50, 100, 200, 500), the minimum response times for each strategy are comparable, with no strategy standing out as particularly better. This implies that regardless of the strategy used, the speed at which the first response is received by the user remains unaffected.

Table 13 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 11 and 12. The results show the following: (1) For benign users, both Strategy and PageType significantly affect the system's minimum response time, and an interaction effect is also observed between the two factors. (2) For attackers, only PageType has a significant influence on response time, while *Strategy* does not show a statistically significant effect on response time control.

Maximum response time Tables 14 and 15 present the maximum response times, and Fig. 13 offers a graphical representation of these findings. First of all, it is evident that the maximum response time of the dbwrite page is significantly higher than the other two pages, while the nginx and dbread pages exhibit similar maximum response times. This underscores the high resource utilization of the dbwrite page. In

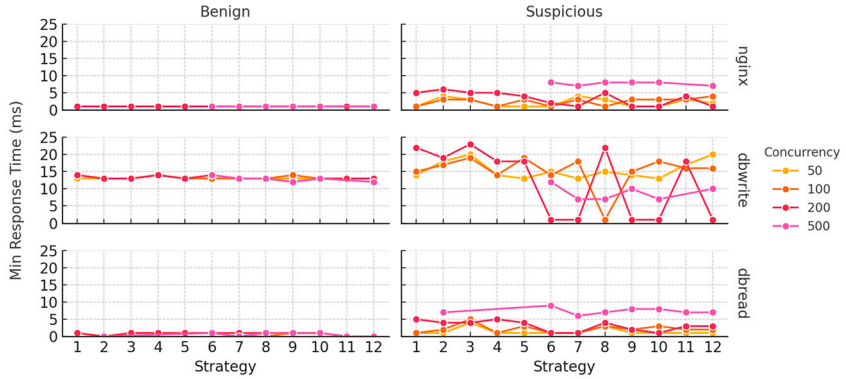

Fig. 12. Minimum response time across 12 mitigation strategies evaluated in Docker environment, for three page types (nginx, dbwrite, dbread), two user categories (benign and suspicious), and four concurrency levels (50, 100, 200, 500).

Table 11
Minimum response time values in ms for different strategies on different pages for attacker and benign user in Docker environment under 50 and 100 concurrency levels (NA: Not available).

Minimum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	1	13	1	1	14	1
	2	1	13	0	4	18	1
	3	1	13	1	3	20	4
	4	1	14	1	1	14	1
	5	1	13	1	1	13	1
	6	1	13	1	1	15	1
	7	1	13	0	4	13	1
	8	1	13	1	3	15	3
	9	1	13	1	1	14	1
	10	1	13	1	1	13	1
	11	1	13	0	3	17	1
	12	1	13	0	2	20	1
100	1	1	14	1	1	15	1
	2	1	13	0	3	17	2
	3	1	13	1	3	19	5
	4	1	14	1	1	14	1
	5	1	13	1	3	19	3
	6	1	13	1	1	14	1
	7	1	13	0	3	18	1
	8	1	13	0	1	1	3
	9	1	14	1	3	15	2
	10	1	13	1	3	18	3
	11	1	13	0	3	16	2
	12	1	13	0	4	16	2

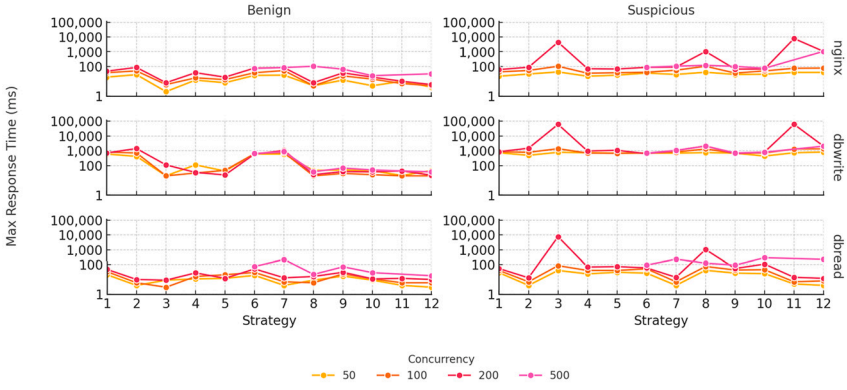

Fig. 13. Maximum response time (in milliseconds) across 12 mitigation strategies in a Docker environment, evaluated for three resource page types (nginx, dbwrite, dbread) and two user categories (benign and suspicious) under four concurrency levels (50, 100, 200, 500). The Y-axis is displayed on a logarithmic scale (1 to 100,000 ms) to illustrate both common and outlier delays, enabling fair comparison across all strategy configurations.

Table 12
Minimum response time values in ms for different strategies on different pages for attacker and benign user in Docker environment under 200 and 500 concurrency levels (NA: Not available).

Minimum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	1	14	1	5	22	5
	2	1	13	0	6	19	4
	3	1	13	1	5	23	4
	4	1	14	1	5	18	5
	5	1	13	1	4	18	4
	6	1	14	1	2	1	1
	7	1	13	1	1	1	1
	8	1	13	1	5	22	4
	9	1	12	1	1	1	2
	10	1	13	1	1	1	1
	11	1	13	0	4	18	3
	12	1	13	0	1	1	3
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	0	NA	NA	7
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	1	14	1	8	12	9
	7	1	13	0	7	7	6
	8	1	13	1	8	7	7
	9	1	12	1	8	10	8
	10	1	13	1	8	7	8
	11	NA	NA	0	NA	NA	7
	12	1	12	0	7	10	7

Table 13
Two-way ANOVA Results for Table 11 and 12.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	6.696699134	11	8.401313459	4.74472E-10	Y
	PageType	4267.532341	2	29445.97315	7.5942E-130	Y
	Strategy : PageType	7.223214286	22	4.530925325	1.37918E-07	Y
	Residual	6.666666667	92			
Suspicious	Strategy	204.1091811	11	1.106945835	0.364953122	N
	PageType	2973.241071	2	88.68632181	3.45128E-22	Y
	Strategy : PageType	387.4200397	22	1.050548038	0.414578586	N
	Residual	1542.166667	92			

addition, the data shows significant fluctuations across all strategies, which is due to network congestion. When a packet is sent under these conditions, the response time increases. Because this is highly dependent on the specific network congestion at the time, it is challenging to identify a consistent trend.

At a concurrency level of 50, strategy 3 and 8 perform well, showing the smallest maximum response times for benign user requests. This indicates that employing a white-list mechanism to allocate more resources to benign user can effectively reduce their waiting time. Conversely, strategy 1, 2, 6, and 7 have larger maximum response times, suggesting that without resource allocation and relying on six containers to provide services, regardless of page separation, can lead to prolonged maximum response times.

At concurrency levels of 100 and 200, strategy 3, 8, 11, and 12 demonstrate favorable performance. This suggests that apart from employing a white-list mechanism to allocate more resources to benign user for reducing their waiting time, employing page separation strategy can effectively reduce maximum response times, particularly under higher concurrency conditions. Conversely, strategy 1, 2, 6, and 7 exhibit larger maximum response times, mirroring the situation at a concurrency level of 50. The reason is that if resources are not allocated and instead six containers are used to provide the service, the maximum response time will still be extended, regardless of whether page separation or DDoS mitigation is implemented.

At a concurrency level of 500, many strategy data are marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Therefore, it can only be concluded that strategy 10 and 12 perform better under these conditions. This indicates that employing a strategy of evenly distributing traffic between benign and attacker can effectively reduce maximum response times under high concurrency. Due to the NA data for strategies 1 and 2, only strategies 6 and 7 show poor performance, consistent with the results for concurrency levels 50, 100, and 200. The reason is that if resources are not allocated and instead six containers are used to provide the service, regardless of whether page separation is implemented, it will still lead to an extension in maximum response time.

Table 16 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 14 and 15. The results show the following: (1) For benign users, both Strategy and PageType significantly affect the system's maximum response time, and an interaction effect is also observed between these two factors. (2) For attackers, Strategy does not have a statistically significant effect, while differences in PageType alone lead to significant variation in maximum response time.

Number of failed request Tables 17 and 18 show the number of failed requests, and Fig. 14 offers a graphical representation of these findings. When the concurrency is 50 or 100, requests from benign user

Table 14
Maximum response time values in ms for different strategies on different pages for attacker and benign user in Docker environment under 50 and 100 concurrency levels (NA: Not available).

Maximum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	19	604	21	22	729	29
	2	28	417	4	31	492	4
	3	2	20	9	43	783	41
	4	12	110	11	22	733	24
	5	8	42	12	26	686	31
	6	25	612	19	36	707	27
	7	26	602	4	29	682	4
	8	5	46	9	41	754	42
	9	12	51	17	29	691	27
	10	5	44	9	30	436	25
	11	10	21	4	40	730	5
	12	4	46	3	40	816	4
100	1	38	806	34	43	855	41
	2	50	707	6	54	780	7
	3	6	20	3	104	1377	86
	4	17	31	16	35	694	41
	5	13	47	21	38	672	41
	6	38	629	30	41	693	53
	7	53	740	7	56	815	7
	8	5	20	6	104	1320	76
	9	23	29	24	36	681	44
	10	14	24	11	50	705	46
	11	7	20	6	76	1314	7
	12	5	20	6	78	1334	8

Table 15
Maximum response time values in ms for different strategies on different pages for attacker and benign user in Docker environment under 200 and 500 concurrency levels (NA: Not available).

Maximum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	48	701	48	62	865	55
	2	86	1404	10	89	1474	13
	3	8	110	9	4415	61190	7751
	4	38	34	28	71	953	69
	5	19	23	12	68	1073	74
	6	78	590	52	87	633	61
	7	84	1047	13	91	1032	14
	8	8	24	16	1034	2103	1049
	9	37	39	31	67	629	55
	10	19	37	11	67	817	110
	11	10	43	12	7816	61026	14
	12	6	22	10	1053	2046	12
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	16	NA	NA	229
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	74	654	75	89	692	94
	7	85	869	223	105	1056	232
	8	104	38	22	121	2054	127
	9	66	67	72	102	713	93
	10	24	50	29	76	798	291
	11	NA	NA	24	NA	NA	236
	12	32	37	18	1059	2041	227

are correctly processed, resulting in no failed requests across all strategies. However, with a concurrency of 200 or 500, strategy with DDoS mitigation (strategy 6-10, 12) begin to block, leading failed requests for attacker. Our work, having different implementations, shows different results compared to Patidar et al. [26]. Our DDoS mitigation method involves not responding to requests from an IP address if more than 150 packets are received from it. Therefore, the maximum number of failed requests appears to be 9,850, as the first 150 packets are processed before identifying and blocking the IP.

From Table 18, it is observed that at concurrency levels of 200 and 500, strategy 8 can handle the highest number of. This occurs because the strategy allocates fewer resources to attacker, reducing the server's ability to respond to their requests and leading to a higher count of failed requests. Conversely, strategy 6 and 9 handle fewer. This suggests that

Table 16
Two-way ANOVA Results for Table 14 and 15.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	1793516.199	11	21.01765909	1.37184E-20	Y
	PageType	1740569.903	2	112.1845884	2.11499E-25	Y
	Strategy : PageType	2946588.574	22	17.26507799	9.38507E-24	Y
	Residual	713700.6667	92			
Suspicious	Strategy	896767915.8	11	1.528373703	0.134687993	N
	PageType	355383220.5	2	3.331264394	0.04108544	Y
	Strategy : PageType	1229417469	22	1.04765642	0.417795625	N
	Residual	4907334336	92			

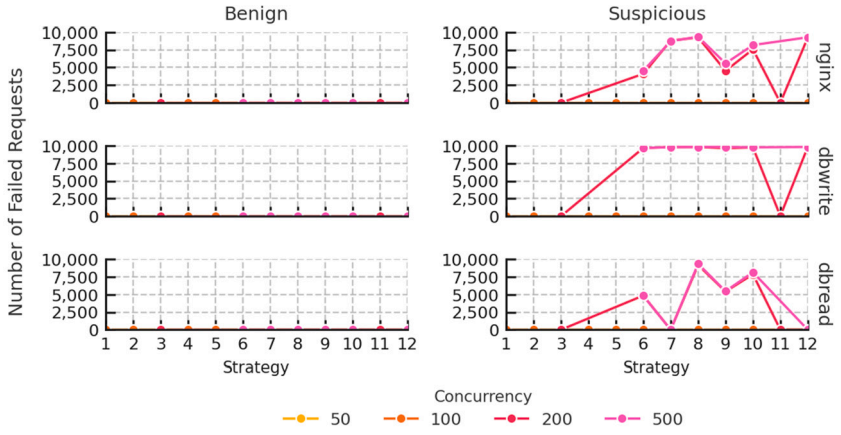

Fig. 14. Number of failed requests recorded across 12 mitigation strategies in a Docker environment, evaluated by page type (nginx, dbwrite, dbread) and user category (benign and suspicious) under four concurrency levels (50, 100, 200, 500).

Table 17
Number of failed requests for different strategies on different pages for attacker and benign user in Docker environment under 50 and 100 concurrency levels (NA: Not available).

Number of failed requests		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	0	0	0
	7	0	0	0	0	0	0
	8	0	0	0	0	0	0
	9	0	0	0	0	0	0
	10	0	0	0	0	0	0
	11	0	0	0	0	0	0
	12	0	0	0	0	0	0
100	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	0	0	0
	7	0	0	0	0	0	0
	8	0	0	0	0	0	0
	9	0	0	0	0	0	0
	10	0	0	0	0	0	0
	11	0	0	0	0	0	0
	12	0	0	0	0	0	0

using default settings or allocating more resources to attacker tends to consume a significant portion of resources with their requests.

Moreover, Table 18 highlights that both Strategy 7 and Strategy 12 exhibit 0 failed requests for attacker on the dbread page. This anomaly occurs despite the activation of the DDoS mitigation method. The asynchronous nature of the function responsible for pulling the API for the dbread page explains this phenomenon [53]. Consequently, browsers do not experience blocking when making API requests, thereby reducing synchronous connections. This, in turn, prompts Nginx to handle requests as batch arrivals, effectively diminishing the number of instan-

Table 18
Number of failed requests for different strategies on different pages for attacker and benign user in Docker environment under 200 and 500 concurrency levels (NA: Not available).

Number of failed requests		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	5	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	4090	9680	4844
	7	0	0	0	8810	9805	0
	8	0	0	0	9258	9822	9178
	9	0	0	0	4526	9647	5410
	10	0	0	0	7575	9761	7858
	11	0	0	0	0	29	0
	12	0	0	0	9241	9830	0
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	0	NA	NA	0
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	0	0	0	4531	9682	4820
	7	0	0	0	8722	9803	0
	8	0	0	0	9367	9827	9360
	9	0	0	0	5578	9833	5465
	10	0	0	0	8176	9776	8156
	11	NA	NA	0	NA	NA	0
	12	0	0	0	9281	9825	0

Table 19
Two-way ANOVA Results for Table 17 and 18.

Sheet	Source	sum_sq	df	F	p	Significant
Suspicious	Strategy	438471573.2	11	3.238006985	0.000909229	Y
	PageType	45227959.38	2	1.836986741	0.165091511	N
	Strategy : PageType	107187625.6	22	0.395777402	0.99222911	N
	Residual	1132553701	92			

taneous concurrent connections. As a result, the concurrency threshold of 150 is not surpassed, preventing the DDoS mitigation method from blocking.

Table 19 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 17 and 18. For attacker traffic, Strategy alone shows a statistically significant effect on the number of failed requests, while there is no significant interaction between Strategy and PageType. As for benign users, a two-way ANOVA could not be conducted. This is because: (1) ANOVA is used to assess whether there are statistically significant differences between group means, which requires variability among the data. In this case, all values are identical (i.e., zero), making it impossible to calculate variance and the F-statistic. (2) When there is no variation across all groups, the test has no statistical meaning, and comparison is unnecessary.

5.1.3. Kubernente environment

Test completion time Tables 20 and 21 show the test completion times, and Fig. 15 provides a visual summary of these results. From the Tables 20 and 21, it is evident that the dbwrite page requires more time compared to the other two pages due to its involvement in database write operations, which is a more complex task. At concurrency levels of 50 and 100, strategy 1, 4, 6, and 9 exhibit similar good performance, with strategy 6 showing the best performance. It achieves shorter test completion times for benign user across all three pages while requiring longer test completion times for attacker. Hence, in Kubernetes, there is no requirement for additional specialized configurations. Simply distributing requests evenly among all pods and integrating DDoS mitigation measures is adequate to attain the shortest test completion time. However, strategy 3, 8, 11, and 12 perform relatively poorer, with strategy 12 showing the worst performance. It appears that page separation and allocating maximum resources to known benign user in Kubernetes do not effectively reduce test completion times.

As shown in Tables 20 and 21, for the same webpage, if the response time for a benign user request is longer than that for an attacker request, it indicates that the DDoS attack has effectively impacted the benign user. In this scenario, the network bandwidth is fully utilized by the attacker requests until they are completely processed, causing a delay in handling benign user requests. Conversely, if the response time for benign user requests is shorter, it signifies that the strategy in place can prioritize processing benign user requests before gradually addressing the attacker requests.

At a concurrency level of 200, after DDoS mitigation begins to block some, strategy 6, 7, 8, 9, 10, and 12 exhibit improved performance, with strategy 8 showing the best performance. This indicates that in Kubernetes, when facing a high volume of traffic, DDoS mitigation can effectively reduce test completion times. However, strategy 3 performed the worst, indicating that the white-list mechanism alone is ineffective in reducing completion times during DDoS attacks on Kubernetes. Additionally, strategy 11, which builds upon strategy 3 by introducing page separation, also yielded poor results.

At a concurrency level of 500, without DDoS mitigation strategy, the data is all marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Moreover, among the strategy with available data (6-10, 12), the test

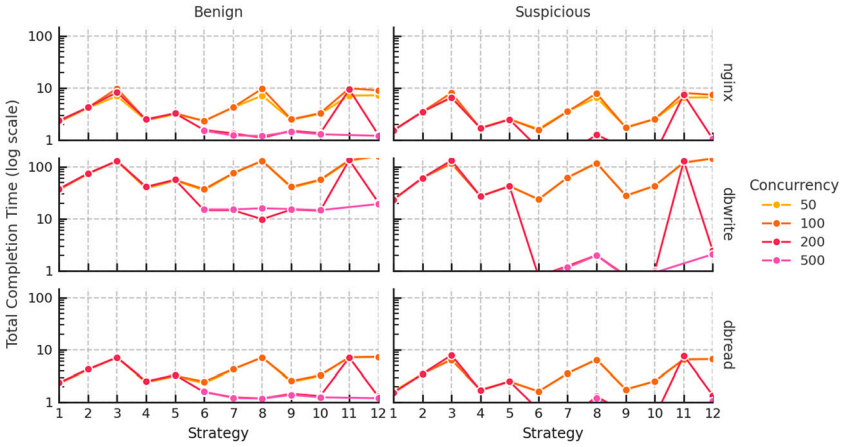

Fig. 15. Total completion time (in seconds) for different strategies in the Kubernetes environment, visualized on a logarithmic scale. The figure compares benign and suspicious user requests across three page types (nginx, dbwrite, dbread) under varying concurrency levels (50, 100, 200, 500). Y-axis ticks are explicitly set to 1, 10, and 100 to enhance readability.

Table 20
Test completion time in seconds for different strategies on different pages for attacker and benign user in Kubernente environment under 50 and 100 concurrency levels (NA: Not available).

Test completion time in seconds		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	2.232	36.711	2.309	1.539	23.246	1.614
	2	4.154	72.716	4.158	3.515	61.127	3.505
	3	7.049	129.408	7.139	6.476	118.083	6.558
	4	2.399	38.351	2.399	1.733	27.229	1.739
	5	3.148	54.523	3.174	2.513	42.704	2.524
	6	2.352	35.407	2.315	1.642	23.547	1.615
	7	4.232	74.505	4.221	3.596	62.748	3.574
	8	7.17	130.342	7.175	6.584	119.033	6.569
	9	2.461	39.735	2.451	1.772	27.741	1.777
	10	3.206	54.985	3.211	2.55	43.162	2.545
	11	7.18	130.695	7.228	6.58	119.34	6.614
	12	7.253	160.588	7.349	6.637	145.876	6.71
100	1	2.287	36.436	2.315	1.545	23.374	1.542
	2	4.218	74.463	4.182	3.496	61.3	3.472
	3	9.826	129.253	7.13	8.056	116.626	6.455
	4	2.438	40.554	2.475	1.677	27.267	1.715
	5	3.283	56.044	3.231	2.519	42.607	2.493
	6	2.328	37.308	2.486	1.558	23.796	1.618
	7	4.264	75.607	4.28	3.525	62.177	3.564
	8	9.742	130.761	7.13	7.922	117.988	6.466
	9	2.51	41.285	2.566	1.732	27.983	1.767
	10	3.299	56.647	3.299	2.541	43.241	2.537
	11	9.871	134.845	7.179	8.063	121.727	6.524
	12	9.024	160.598	7.358	7.39	144.141	6.645

completion times are nearly identical, with no strategy demonstrating notably shorter completion times.

Table 22 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 20 and 21. For both benign users and attackers, Strategy and PageType each have statistically significant effects on completion time. Additionally, the interaction between Strategy and PageType is also statistically significant, indicating that their combined influence contributes meaningfully to the variation in results.

Speed and bandwidth analysis Tables 23, 24 display the speed of DDoS attacks, and Fig. 16 offers a graphical representation of these findings. Tables 26 and 27 display the bandwidth of the DDoS attacks, and Fig. 17 provides a visual summary of these results. These tables provide additional insights into the attack characteristics and the impact on the system.

Tables 23 and 24 show the speed of the attack, measured in requests per second. This metric indicates the density of the attack traffic. Higher values suggest more intense traffic, which can lead to greater strain on the system. The request data for benign users is 1000 and for attackers is 10000. These data are sourced from 4.5.1.

At concurrency levels of 50 and 100, Strategy 1, 4, 6, and 9 exhibit similar good performance in terms of speed, with Strategy 6 showing the best performance. They achieve the highest requests per second for benign users across all three webpages, while maintaining lower request rates for attackers. Hence, in Kubernetes, there is no requirement for additional specialized configurations. Simply distributing requests evenly among all pods and integrating DDoS mitigation measures is adequate to achieve the fastest request handling times. However, Strategy 3, 8, 11, and 12 perform relatively poorer in terms of speed, with Strategy 12 showing the worst performance. It appears that page separation and

Table 21
Test completion time in seconds for different strategies on different pages for attacker and benign user in Kubernente environment under 200 and 500 concurrency levels (NA: Not available).

Test completion time in seconds		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	2.371	37.689	2.377	1.532	23.594	1.538
	2	4.259	74.859	4.247	3.493	61.013	3.471
	3	8.321	129.571	7.13	6.63	132.971	7.886
	4	2.516	41.299	2.511	1.693	27.288	1.695
	5	3.262	56.558	3.34	2.487	42.446	2.519
	6	1.567	14.651	1.57	0.699	0.795	0.721
	7	1.346	14.652	1.251	0.562	1.247	0.46
	8	1.109	9.863	1.184	1.264	2.015	1.298
	9	1.516	15.136	1.46	0.699	0.813	0.633
	10	1.356	14.331	1.323	0.54	0.996	0.529
	11	9.4	135.407	7.101	7.454	129.65	7.76
	12	1.269	20.659	1.291	1.048	2.435	1.356
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	1.515	15.213	1.609	0.664	0.779	0.861
	7	1.228	15.212	1.215	0.45	1.168	0.442
	8	1.2	15.993	1.181	0.438	1.994	1.206
	9	1.457	15.457	1.394	0.588	0.774	0.531
	10	1.297	14.711	1.251	0.484	0.932	0.44
	11	NA	NA	NA	NA	NA	NA
	12	1.209	19.326	1.201	1.059	2.1	1.069

Table 22
Two-way ANOVA Results for Table 20 and 21.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	20739.9482	11	4.212415383	4.97291E-05	Y
	PageType	94239.8702	2	105.2739435	2.72143E-24	Y
	Strategy : PageType	30043.18826	22	3.050981303	0.000105852	Y
	Residual	40283.41696	90			
Suspicious	Strategy	22147.80227	11	4.404937503	2.80961E-05	Y
	PageType	59329.1783	2	64.8993185	3.5464E-18	Y
	Strategy : PageType	32266.25246	22	3.208689147	5.17725E-05	Y
	Residual	41137.7667	90			

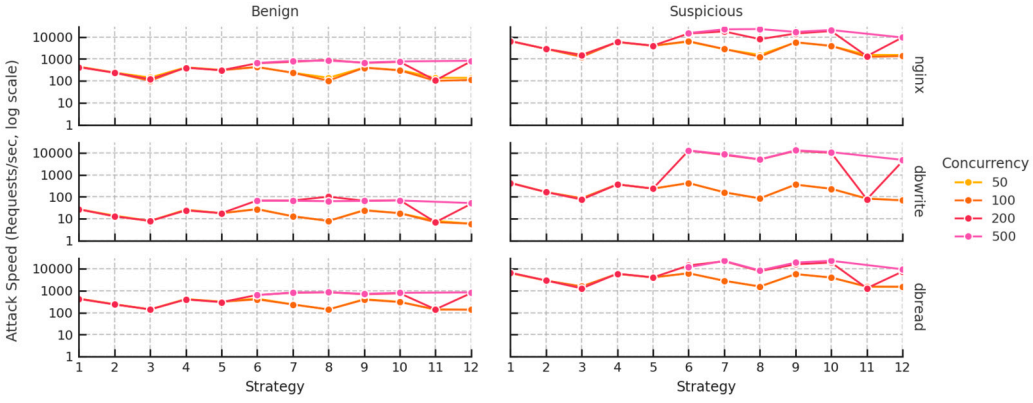

Fig. 16. DDoS attack speed (in requests per second) for different strategies in the Kubernetes environment, shown on a logarithmic scale. The figure compares benign and suspicious user requests across three page types (nginx, dbwrite, dbread) under various concurrency levels (50, 100, 200, 500). The y-axis uses logarithmic ticks at 1, 10, 100, 1,000, 10,000, and 100,000 to capture wide variance in attack throughput.

allocating maximum resources to known benign users in Kubernetes do not effectively optimize request handling times.

At a concurrency level of 200, after DDoS mitigation begins to block some traffic, Strategy 6, 7, 8, 9, 10, and 12 exhibit improved performance in terms of speed, with Strategy 8 showing the best performance. This indicates that in Kubernetes, when facing a high volume of traffic, DDoS mitigation can effectively optimize request handling times. However, Strategy 3 performed the worst, indicating that the whitelist mechanism alone is ineffective in reducing completion times during DDoS attacks on Kubernetes. Additionally, Strategy 11, which builds upon Strategy 3 by introducing page separation, also yielded poor results in terms of speed.

Table 23
DDoS Attack Speed (Requests per Second) in Kubernetes environment under 50 and 100 concurrency levels.

DDoS Attack Speed		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	448	27	433	6498	430	6196
	2	241	14	241	2845	164	2853
	3	142	8	140	1544	85	1525
	4	417	26	417	5770	367	5750
	5	318	18	315	3979	234	3962
	6	425	28	432	6090	425	6192
	7	236	13	237	2781	159	2798
	8	139	8	139	1519	84	1522
	9	406	25	408	5643	360	5627
	10	312	18	311	3922	232	3929
	11	139	8	138	1520	84	1512
	12	138	6	136	1507	69	1490
100	1	437	27	432	6472	428	6485
	2	237	13	239	2860	163	2880
	3	102	8	140	1241	86	1549
	4	410	25	404	5963	367	5831
	5	305	18	310	3970	235	4011
	6	430	27	402	6418	420	6180
	7	235	13	234	2837	161	2806
	8	103	8	140	1262	85	1547
	9	398	24	390	5774	357	5659
	10	303	18	303	3935	231	3942
	11	101	7	139	1240	82	1533
	12	111	6	136	1353	69	1505

Table 24
DDoS Attack Speed (Requests per Second) in Kubernetes environment under 200 and 500 concurrency levels.

DDoS Attack Speed		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	422	27	421	6527	424	6502
	2	235	13	235	2863	164	2881
	3	120	8	140	1508	75	1268
	4	397	24	398	5907	366	5900
	5	307	18	299	4021	236	3970
	6	638	68	637	14306	12579	13870
	7	743	68	799	17794	8019	21739
	8	902	101	845	7911	4963	7704
	9	660	66	685	14306	12300	15798
	10	737	70	756	18519	10040	18904
	11	106	7	141	1342	77	1289
	12	788	48	775	9542	4107	7375
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	660	66	622	15060	12837	11614
	7	814	66	823	22222	8562	22624
	8	833	63	847	22831	5015	8292
	9	686	65	717	17007	12920	18832
	10	771	68	799	20661	10730	22727
	11	NA	NA	NA	NA	NA	NA
	12	827	52	833	9443	4762	9355

At a concurrency level of 500, without a DDoS mitigation strategy, the data is all marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Moreover, among the strategies with available data (6-10, 12), Strategy 7, 8, and 12 demonstrate the highest speeds, achieving the most requests per second, while Strategy 6 and 9 exhibit the lowest speeds.

Table 25 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 23 and 24. For both benign users and attackers, Requests per Second is significantly influenced by both Strategy and PageType. However, no significant interaction effect is observed between the two factors, indicating that the effectiveness of each Strategy is consistent across different PageTypes.

Tables 26 and 27 show the attack bandwidth in bytes per second, indicating the data volume transmitted during the attack. Higher values suggest greater network resource consumption. The byte data for each webpage (nginx: 772 bytes, dbwrite: 775 bytes, dbread: 4036 bytes) are sourced from 4.2.

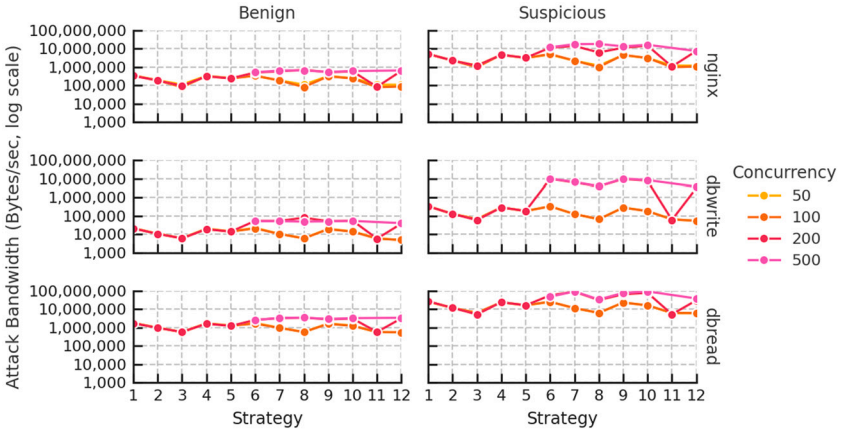

Fig. 17. DDoS attack bandwidth (in Bytes/sec) for Kubernetes environment under different concurrency levels and user types. The y-axis uses a logarithmic scale from 1,000 to 100,000,000 to represent bandwidth variations more clearly. Each line represents a concurrency level (50, 100, 200, 500), and the data is faceted by request page type (nginx, dbwrite, dbread) and user type (benign, suspicious).

Table 25
Two-way ANOVA Results for Table 23 and 24.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	1276965.151	11	3.159791424	0.001183513	Y
	PageType	4060203.444	2	55.25732483	2.20972E-16	Y
	Strategy : PageType	482005.2778	22	0.596349925	0.916185364	N
	Residual	3306514.667	90			
Suspicious	Strategy	1333345276	11	4.262924635	4.27962E-05	Y
	PageType	514342717.9	2	9.04441149	0.000263498	Y
	Strategy : PageType	136073718.5	22	0.217525054	0.99991779	N
	Residual	2559085500	90			

Table 26
DDoS Attack Bandwidth (Bytes per Second) in Kubernetes environment under 50 and 100 concurrency levels.

DDoS Attack Bandwidth		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	345878	21111	1747943	5016244	333391	25006196
	2	185845	10658	970659	2196302	126785	11514979
	3	109519	5989	565345	1192094	65632	6154315
	4	321801	20208	1682368	4454703	284623	23208741
	5	245235	14214	1271582	3072025	181482	15990491
	6	328231	21888	1743413	4701583	329129	24990712
	7	182420	10402	956172	2146830	123510	11292669
	8	107671	5946	562509	1172539	65108	6144010
	9	313694	19504	1646675	4356659	279370	22712437
	10	240799	14095	1256929	3027451	179556	15858546
	11	107521	5930	558384	1173252	64941	6102207
	12	106439	4826	549190	1163176	53127	6014903
100	1	337560	21270	1743413	4996764	331565	26173800
	2	183025	10408	965088	2208238	126427	11624424
	3	78567	5996	566059	958292	66452	6252517
	4	316653	19110	1630707	4603459	284226	23533528
	5	235151	13828	1249149	3064708	181895	16189330
	6	331615	20773	1623492	4955071	325685	24944376
	7	181051	10250	942991	2190071	124644	11324355
	8	79245	5927	566059	974501	65685	6241881
	9	307570	18772	1572876	4457275	276954	22840973
	10	234010	13681	1223401	3038174	179228	15908553
	11	78209	5747	562195	957460	63667	6186389
	12	85550	4826	548519	1044655	53767	6073740

At concurrency levels of 50 and 100, Strategy 1, 4, 6, and 9 exhibit similar good performance in terms of bandwidth, with Strategy 6 showing the best performance. They achieve the highest data transmission rates (bytes per second) for benign users across all three webpages, while maintaining lower transmission rates for attackers. Hence, in Kubernetes, there is no requirement for additional specialized configurations. Simply distributing requests evenly among all pods and integrating DDoS mitigation measures is adequate to achieve the highest bandwidth efficiency. However, Strategy 3, 8, 11, and 12 perform relatively poorer in terms of bandwidth, with Strategy 12 showing the

Table 27
DDoS Attack Bandwidth (Bytes per Second) in Kubernetes environment under 200 and 500 concurrency levels.

DDoS Attack Bandwidth		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	325601	20563	1697939	5039164	328473	26241873
	2	181263	10353	950318	2210135	127022	11627773
	3	92777	5981	566059	1164404	58283	5117931
	4	306836	18766	1607328	4559953	284008	23811209
	5	236665	13703	1208383	3104142	182585	16022231
	6	492661	52897	2570701	11044349	9748428	55977809
	7	573551	52894	3226219	13736655	6214916	87739130
	8	696123	78576	3408784	6107595	3846154	31093991
	9	509235	51202	2764384	11044349	9532595	63759874
	10	569322	54079	3050642	14296296	7781124	76294896
	11	82128	5723	568371	1035686	59776	5201031
	12	608353	37514	3126259	7366412	3182752	29764012
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	509571	50943	2508390	11626506	9948652	46875726
	7	628664	50947	3321811	17155556	6635274	91312217
	8	643333	48459	3417443	17625571	3886660	33466003
	9	529856	50139	2895265	13129252	10012920	76007533
	10	595220	52682	3226219	15950413	8315451	91727273
	11	NA	NA	NA	NA	NA	NA
	12	638544	40101	3360533	7289896	3690476	37754911

Table 28
Two-way ANOVA Results for Table 26 and 27.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	6.81295E+12	11	2.014258717	0.035846341	Y
	PageType	6.48865E+13	2	105.5108697	2.53518E-24	Y
	Strategy : PageType	7.58526E+12	22	1.121297324	0.340651324	N
	Residual	2.76739E+13	90			
Suspicious	Strategy	5.72959E+15	11	2.863648334	0.002900635	Y
	PageType	1.62033E+16	2	44.54127526	3.57697E-14	Y
	Strategy : PageType	5.28154E+15	22	1.319856874	0.181004975	N
	Residual	1.63702E+16	90			

worst performance. It appears that page separation and allocating maximum resources to known benign users in Kubernetes do not effectively optimize data transmission rates.

At a concurrency level of 200, after DDoS mitigation begins to block some traffic, Strategy 6, 7, 8, 9, 10, and 12 exhibit improved performance in terms of bandwidth, with Strategy 8 showing the best performance. This indicates that in Kubernetes, when facing a high volume of traffic, DDoS mitigation can effectively optimize data transmission rates. However, Strategy 3 performed the worst, indicating that the whitelist mechanism alone is ineffective in reducing completion times during DDoS attacks on Kubernetes. Additionally, Strategy 11, which builds upon Strategy 3 by introducing page separation, also yielded poor results in terms of bandwidth.

At a concurrency level of 500, without a DDoS mitigation strategy, the data is all marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Moreover, among the strategies with available data (6-10, 12), Strategy 7, 8, and 12 demonstrate the highest bandwidth, achieving the highest data transmission rates (bytes per second), while Strategy 6 and 9 exhibit the lowest bandwidth.

These additional metrics of speed and bandwidth complement the test completion time results, providing a more comprehensive view of the DDoS attack impact. The speed metric helps to understand the request density, while the bandwidth metric provides insights into the volume of data being transmitted during the attack. Both metrics are crucial for evaluating the overall strain on the system and the effectiveness of different mitigation strategies.

Table 28 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Table 26 and 27. Both Strategy and PageType have statistically significant effects on bandwidth usage for both benign users and attackers. However, no significant interaction effect is observed between the two factors, suggesting that the performance of each Strategy is generally consistent across different PageTypes.

Minumim response time Tables 29 and 30 present the minimum response times, and Fig. 18 provides a visual summary of these results. Some values are shown as 0 because the ab command's request and response time is extremely fast, less than 0.001 seconds, and thus cannot be displayed to four decimal places in the table. This does not imply that the request was not sent. In addition, it is evident that the minimum response time of the dbwrite page is significantly higher than the other two pages, while the nginx and dbread pages exhibit similar minimum response times. This further underscores the high resource utilization of the dbwrite page.

Across all concurrency levels (50, 100, 200, 500), it can be observed that the minimum response time for each strategy is comparable, with no strategy standing out as particularly superior. This indicates that regardless of the strategy employed, it does not affect the speed at which the fastest user receives a response. When at a concurrency of 500, many

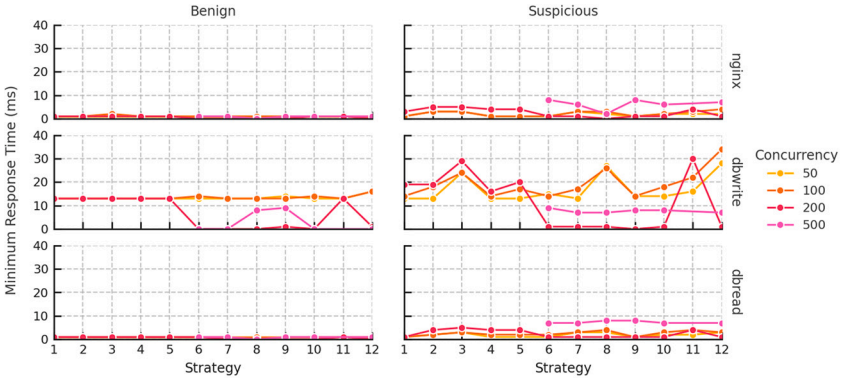

Fig. 18. Minimum response time (in milliseconds) for different strategies in a Kubernetes environment. Results are categorized by request type (nginx, dbwrite, dbread) and user type (benign vs. suspicious). Each line represents a concurrency level (50, 100, 200, 500).

Table 29
Minimum response time values in ms for different strategies on different pages for attacker and benign user in Kubernente environment under 50 and 100 concurrency levels (NA: Not available).

Minimum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	1	13	1	1	13	1
	2	1	13	1	3	13	2
	3	1	13	1	3	24	3
	4	1	13	1	1	13	1
	5	1	13	1	1	13	1
	6	1	13	1	1	15	1
	7	1	13	1	3	13	3
	8	1	13	1	2	27	3
	9	1	14	1	1	14	1
	10	1	13	1	2	14	2
	11	1	13	1	2	16	2
	12	1	16	1	2	28	3
100	1	1	13	1	1	14	1
	2	1	13	1	3	18	2
	3	2	13	1	3	24	3
	4	1	13	1	1	14	2
	5	1	13	1	1	17	2
	6	1	14	1	1	14	2
	7	1	13	1	3	17	3
	8	1	13	1	3	26	4
	9	1	13	1	1	14	1
	10	1	14	1	2	18	3
	11	1	13	1	3	22	4
	12	1	16	1	4	34	3

strategy data points are marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data.

Table 31 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Table 29 and 30. For both benign users and attackers, minimum response time is significantly affected by PageType. However, no statistically significant effect is observed for Strategy, and no interaction effect is found between Strategy and PageType.

Maximum response time Tables 32 and 33 display the maximum response times, and Fig. 19 offers a graphical representation of these findings. From the Tables 32 and 33, it is evident that the maximum response time of the dbwrite page is significantly higher than the other two pages, while the nginx and dbread pages exhibit similar maximum response times. This further highlights the high resource consumption of the dbwrite page.

At a concurrency level of 50, strategy 4 performs well, with the smallest maximum response time for benign user requests. This occurs because allocating fewer resources to the white-list mechanism suffices for managing benign user traffic, as allocating more resources may lead to longer waiting times for some benign user. Conversely, strategy 1 exhibits a larger maximum response time, indicating that relying solely on six pods to provide services can lead to prolonged maximum response times, necessitating the use of other strategies to mitigate this issue.

At concurrency levels of 100 and 200, strategy 4 performs well, which is similar to the situation at a concurrency level of 50. This occurs because allocating fewer resources to the white-list mechanism suffices for managing benign user traffic, as allocating more resources may lead to longer waiting times for some benign user. However, strategy 3, 11, and 12 exhibit larger maximum response times. This is because overallocating resources to benign user in Kubernetes can paradoxically lead to greater maximum response times, suggesting that such overallocation is inefficient and may have adverse effects.

At a concurrency level of 500, many strategy data points are marked as NA. This is because when concurrency becomes excessively high, the suspicious server transmits packets at such a rapid rate that it causes disconnection from the victim server, resulting in the inability to transmit all packets and receive experimental data. Therefore, it can only be concluded that strategy 6 performs better, indicating that processing

Table 30
Minimum response time values in ms for different strategies on different pages for attacker and benign user in Kubernente environment under 200 and 500 concurrency levels (NA: Not available).

Minimum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	1	13	1	3	19	1
	2	1	13	1	5	19	4
	3	1	13	1	5	29	5
	4	1	13	1	4	16	4
	5	1	13	1	4	20	4
	6	0	0	1	1	1	1
	7	0	0	0	1	1	1
	8	0	0	0	0	1	1
	9	0	1	0	1	0	1
	10	1	0	0	1	1	1
	11	1	13	1	4	30	4
	12	0	1	0	1	1	1
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	1	0	1	8	9	7
	7	1	0	1	6	7	7
	8	0	8	0	2	7	8
	9	1	9	1	8	8	8
	10	1	0	1	6	8	7
	11	NA	NA	NA	NA	NA	NA
	12	1	0	1	7	7	7

Table 31
Two-way ANOVA Results for Table 29 and 30.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	125.9365079	11	1.028675152	0.428295069	N
	PageType	2310.206349	2	103.7863085	4.25823E-24	Y
	Strategy : PageType	195.8492063	22	0.799868185	0.718011442	N
	Residual	1001.666667	90			
Suspicious	Strategy	399.1626984	11	1.387029534	0.192731028	N
	PageType	3953.587302	2	75.55962822	5.50147E-20	Y
	Strategy : PageType	667.468254	22	1.159675221	0.304044716	N
	Residual	2354.583333	90			

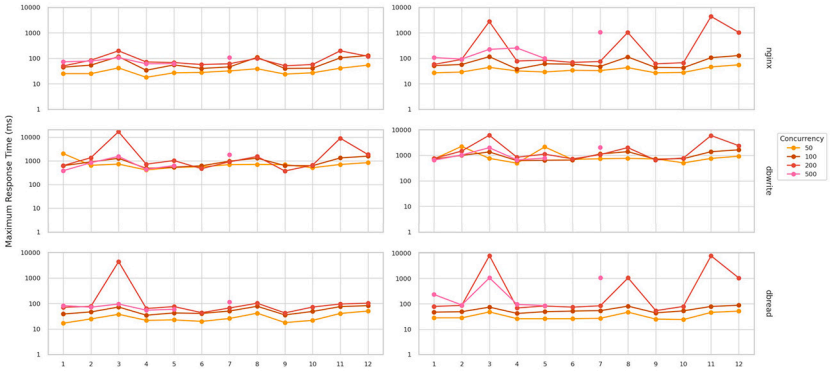

Fig. 19. Maximum response time (logarithmic scale) across 12 strategies in the Kubernetes environment, categorized by request type (nginx, dbwrite, dbread) and user type (benign vs. suspicious). Each subplot uses a log-scale y-axis and shows response trends under concurrency levels of 50, 100, 200, and 500, with consistent color coding across charts.

traffic with six pods and implementing DDoS mitigation can effectively reduce maximum response times under high concurrency. Due to the appearance of NA data for strategies 3 and 11, only strategy 12 exhibits poorer performance. This is because overallocating resources in Kubernetes is inefficient and can increase maximum response time for benign users.

Table 34 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Tables 32 and 33. The results show the following: (1) For benign users, the primary factor affecting maximum response time is PageType, while Strategy has a relatively lower impact, and no interaction effect is observed between the two. (2) For attackers, both Strategy and PageType significantly influence

Table 32
Maximum response time values in ms for different strategies on different pages for attacker and benign user in Kubernente environment under 50 and 100 concurrency levels (NA: Not available).

Maximum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	25	2045	17	27	722	28
	2	25	649	25	29	2228	28
	3	42	729	38	44	756	48
	4	18	413	22	32	492	26
	5	27	528	23	29	2119	26
	6	28	559	20	34	691	26
	7	32	693	26	33	731	27
	8	39	707	42	43	760	47
	9	24	693	18	27	721	25
	10	27	513	22	28	510	24
	11	41	705	41	46	762	46
	12	54	839	51	56	933	52
100	1	45	624	39	52	717	47
	2	54	902	47	57	988	49
	3	116	1281	73	116	1349	74
	4	34	483	35	38	617	42
	5	55	542	43	61	641	49
	6	40	619	41	59	651	52
	7	46	971	51	48	1125	54
	8	109	1297	78	113	1384	81
	9	40	631	36	44	703	44
	10	41	626	49	43	747	53
	11	104	1331	76	106	1400	79
	12	129	1584	83	129	1652	88

Table 33
Maximum response time values in ms for different strategies on different pages for attacker and benign user in Kubernente environment under 200 and 500 concurrency levels (NA: Not available).

Maximum response time in ms		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	50	632	71	58	754	79
	2	84	1356	78	90	1455	87
	3	196	17070	4395	2754	6102	7881
	4	72	726	64	78	838	69
	5	68	1029	76	85	1109	82
	6	57	469	44	69	722	74
	7	61	909	67	75	1060	84
	8	100	1534	103	1031	1995	1050
	9	51	373	43	61	652	54
	10	57	659	73	67	784	79
	11	196	9059	96	4374	6000	7756
	12	120	1850	103	1038	2403	1045
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	73	383	83	107	643	233
	7	78	834	72	93	1030	90
	8	106	1546	96	225	1974	1056
	9	61	438	55	253	657	95
	10	62	633	60	99	778	85
	11	NA	NA	NA	NA	NA	NA
	12	108	1824	116	1054	2036	1060

maximum response time; however, no significant interaction effect is found between them.

Number of failed request Tables 35 and 36 present the number of failed requests, and Fig. 20 provides a visual summary of these results. From Tables 35 and 36, it can be observed that at concurrency levels of 50 and 100, requests from benign user are correctly processed, resulting in no failed requests. However, at concurrency levels of 200 and 500, both benign and attacker begin to experience failed requests. This is due to the implementation of strategy with DDoS mitigation (strategy 6-10, 12), which start blocking, leading to the appearance of these data.

At concurrency levels of 200 and 500, strategy 8 can block the highest number of. This strategy 8 allocates fewer resources to attacker, which reduces the server's capacity to respond to requests, thereby leading to an increase in failed requests. Conversely, strategy 6 and 9 block fewer. This indicates that both strategy 6 and 9, either by applying default settings or allocating more resources to attacker, can lead to a substantial consumption of resources due to.

Table 34
Two-way ANOVA Results for Table 32 and 33.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	52146788.12	11	1.845216099	0.057724222	N
	PageType	50932081.54	2	9.912285128	0.000128657	Y
	Strategy : PageType	57767508.46	22	1.022052752	0.44711909	N
	Residual	231222532.4	90			
Suspicious	Strategy	53717394.74	11	3.167666752	0.001155602	Y
	PageType	22615454.33	2	7.334872198	0.001119336	Y
	Strategy : PageType	7563742.556	22	0.223013569	0.999898591	N
	Residual	138747536.1	90			

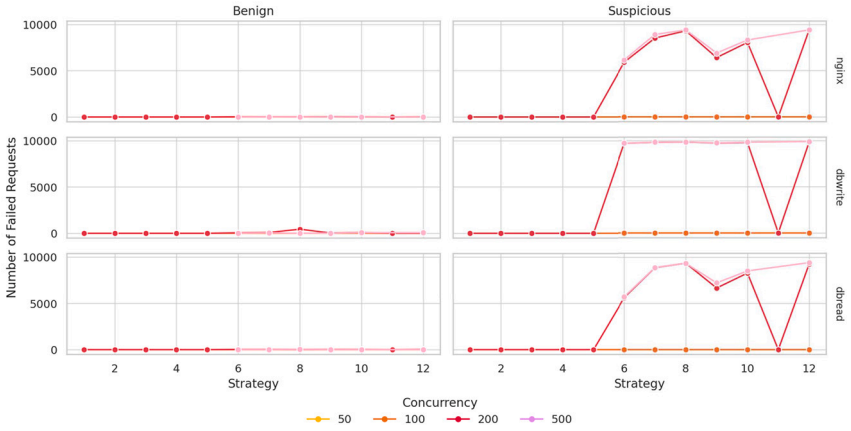

Fig. 20. Number of failed requests in the Kubernetes environment for benign and suspicious users across different strategies and concurrency levels (50, 100, 200, and 500).

Table 35
Number of failed requests for different strategies on different pages for attacker and benign user in Kubernente environment under 50 and 100 concurrency levels (NA: Not available).

Number of failed requests		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
50	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	0	0	0
	7	0	0	0	0	0	0
	8	0	0	0	0	0	0
	9	0	0	0	0	0	0
	10	0	0	0	0	0	0
	11	0	0	0	0	0	0
	12	0	0	0	0	0	0
100	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	0	0	0	0	0	0
	7	0	0	0	0	0	0
	8	0	0	0	0	0	0
	9	0	0	0	0	0	0
	10	0	0	0	0	0	0
	11	0	0	0	0	0	0
	12	0	0	0	0	0	0

Table 37 presents the results of the two-way ANOVA and interaction effect analysis based on the data from Table 35 and 36. The results show the following: (1) For benign users, there are no statistically significant differences in the number of failed requests across different Strategies or PageTypes. (2) For attackers, Strategy is the primary factor influencing failed requests, while PageType and the interaction between the two factors do not show significant effects.

5.1.4. Discussion

Docker environment In the Docker environment, Strategy 12—combining page separation, maximum resource allocation to known benign users, and DDoS mitigation—consistently demonstrates superior performance at concurrency levels of 50, 100, and 200. It leads to shorter test completion times for benign users and longer times for attackers, outperforming strategies that rely on only one or two mitigation compo-

Table 36
Number of failed requests for different strategies on different pages for attacker and benign user in Kubernente environment under 200 and 500 concurrency levels (NA: Not available).

Number of failed requests		Benign users requests			Suspicious users requests		
Concurrency	Strategy	nginx	dbwrite	dbread	nginx	dbwrite	dbread
200	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0
	6	28	58	25	5903	9701	5614
	7	20	90	32	8518	9806	8860
	8	21	441	17	9302	9831	9344
	9	40	30	30	6411	9725	6663
	10	29	94	24	8047	9781	8276
	11	0	0	0	0	9	0
	12	29	1	33	9405	9834	9257
500	1	NA	NA	NA	NA	NA	NA
	2	NA	NA	NA	NA	NA	NA
	3	NA	NA	NA	NA	NA	NA
	4	NA	NA	NA	NA	NA	NA
	5	NA	NA	NA	NA	NA	NA
	6	10	18	12	6151	9715	5706
	7	10	41	16	8905	9816	8882
	8	12	6	23	9401	9833	9337
	9	16	10	13	6902	9751	7216
	10	21	68	11	8311	9818	8511
	11	NA	NA	NA	NA	NA	NA
	12	15	72	8	9404	9888	9407

Table 37
Two-way ANOVA Results for Table 35 and 36.

Sheet	Source	sum_sq	df	F	p	Significant
Benign	Strategy	20754.53968	11	0.99681759	0.455640356	N
	PageType	7372.68254	2	1.947559842	0.148585706	N
	Strategy : PageType	25017.31746	22	0.600777047	0.913126999	N
	Residual	170352	90			
Suspicious	Strategy	591030662	11	3.507396938	0.000413017	Y
	PageType	6765070.619	2	0.220805522	0.802305706	N
	Strategy : PageType	11638599.88	22	0.034533902	1	N
	Residual	1378716325	90			

nents. This is because page-level separation reduces service contention, resource prioritization ensures benign users are not throttled, and DDoS mitigation filters out malicious traffic preemptively. Strategies 2 and 7, which apply only page separation and DDoS mitigation, are less effective at low concurrency, as they lack resource prioritization. In contrast, Strategy 8, which prioritizes resources for benign users with DDoS mitigation but without page separation, is not recommended under high concurrency, since attackers can still consume excessive resources, leading to prolonged test completion times.

In terms of minimum response time, strategy choice does not significantly affect the speed of the first response. Across all concurrency levels (50, 100, 200, 500), the minimum response times remain consistent. This suggests that most strategies can maintain a base level of responsiveness for at least one user.

For maximum response time, which affects the service quality perceived by the majority of users, Strategy 3 (maximum resources to white-listed users) is most effective at moderate concurrency levels. At higher concurrency, Strategies 8, 11, and 12 are recommended. These strategies combine resource prioritization with either page separation or DDoS mitigation, allowing better handling of benign traffic under stress. In contrast, Strategies 1, 2, 6, and 7 are less effective due to insufficient prioritization, resulting in longer response times under load. Additionally, their reliance on default configurations and six pods without adequate mitigation logic leads to performance degradation.

Considering failure rates, Strategy 8 performs well in rejecting suspicious requests and reducing failed connections by leveraging whitelist-based resource prioritization and strong DDoS filtering. Strategy 6 and 9, although equipped with DDoS mitigation, underperform because they do not pair it with intelligent resource control. Strategy 6 lacks prioritization for benign users, increasing the likelihood of failed requests. Strategy 9 allocates minimal resources to benign users, which is insufficient under heavy attack conditions.

Taken together, our findings highlight that mitigation strategies which integrate multiple mechanisms—resource allocation, page separation, and traffic filtering—consistently outperform simpler configurations. Strategies like 6 and 9 fail not because of a lack of DDoS protection, but because they do not differentiate resource allocation between benign and suspicious traffic.

This underscores that strategic resource management is essential not only for fairness but also for maintaining service quality at scale. Intelligent prioritization effectively reduces maximum response times and failure rates, and reveals the trade-offs in Docker's static resource model. By analyzing performance alongside strategic design, we demonstrate why integrated defenses yield more stable and resilient service delivery under attack.

Kubernetes environment In a Kubernetes environment, strategy performance varies significantly depending on concurrency and system behavior. At low concurrency, Strategy 6 (default scenario with DDoS mitigation) performs best, achieving shorter completion times for benign users while delaying attackers. This is largely due to its simplicity—distributing traffic evenly among pods combined with basic mitigation

suffices in preserving low completion times. In contrast, Strategy 12, which adds page separation and heavy resource prioritization, introduces overhead without corresponding benefits under light load. These results suggest that when the system is not under stress, complex mitigation mechanisms such as page separation or whitelist-based resource allocation may not only be unnecessary, but even detrimental due to increased orchestration overhead. At high concurrency, Strategy 8 (maximum resources to benign users with DDoS mitigation) is most effective, maintaining system responsiveness while filtering attacker traffic. In contrast, Strategy 3 (maximum resources to white-list users) underperforms, as its over-prioritization of benign traffic leads to pod saturation and inefficient scheduling. This illustrates that Kubernetes' dynamic pod scheduling and real-time load balancing are sensitive to unbalanced resource allocation, and over-committing to certain traffic categories can trigger unexpected performance penalties.

For minimum response time, the choice of strategy has negligible impact. Across all concurrency levels, minimum response times are comparable, indicating that all strategies maintain a base level of responsiveness for at least one user.

In terms of maximum response time, Strategy 4 (minimum resources to white-listed users) is effective at low to moderate concurrency, as it balances availability without overcommitting. Strategies 3, 11, and 12 are less effective under high concurrency due to over-allocation to benign traffic, which leads to scheduling contention and performance degradation. Our analysis confirms that successful strategies in Kubernetes must avoid both extremes—under-allocation causes delays, and over-allocation saturates pods and destabilizes service.

For failure rates, Strategy 8 again outperforms others by allocating fewer resources to attackers while maintaining service quality for benign users. Strategies 6 and 9 are not recommended: Strategy 6 lacks benign-user prioritization, and Strategy 9 provides insufficient resources to sustain services under attack.

These findings underscore the need for mitigation strategies to be not only traffic-aware but also orchestration-aware. Unlike Docker, Kubernetes introduces system behaviors—such as pod scheduling, control plane resource consumption, and autoscaling—that can amplify or undermine strategy performance. Designing mitigation strategies that respect these orchestration behaviors is central to ensuring real-world effectiveness.

This deeper platform-specific interpretation helps explain why some strategies excelled in Docker but failed in Kubernetes, and vice versa. Our work provides practical guidance on how mitigation logic must be aligned with the deployment environment's operational characteristics—offering a novel orchestration-sensitive framework for container-based DDoS defense.

Cross-platform strategy evaluation insights Our cross-platform evaluation framework reveals that the effectiveness of DDoS mitigation strategies is not solely determined by their individual mechanisms—such as page-level separation, user prioritization, or traffic filtering—but also by the behavioral characteristics of the underlying orchestration platform. In Docker, which features a more static resource management model, strategies that combine all three mechanisms (e.g., Strategy 12) consistently deliver superior performance by minimizing contention and ensuring reliable resource allocation. Conversely, Kubernetes' dynamic pod scheduling introduces new trade-offs: strategies that aggressively prioritize benign users can suffer from resource saturation or scheduling overhead, particularly under low or high concurrency.

Interestingly, simpler strategies like Strategy 6, which uses uniform traffic distribution and lightweight DDoS control, prove highly effective in Kubernetes under low concurrency. This highlights that mitigation effectiveness is context-dependent and that no single strategy is universally optimal across platforms and conditions. Our evaluation demonstrates that high-concurrency scenarios benefit from balanced designs (e.g., Strategy 8), while lower-concurrency settings may not require extensive separation or prioritization.

Beyond validating the comparative performance of mitigation strategies, our framework provides novel methodological insights. By integrating realistic multi-service workloads, varying concurrency levels, and orchestration-aware analysis across Docker and Kubernetes, we move beyond traditional experimental designs and offer deployment-level guidance for designing resilient, adaptable defense strategies in modern cloud-native environments.

To further contextualize these findings, we examine how architectural characteristics at the orchestration layer fundamentally contribute to the observed differences between Docker and Kubernetes. The observed differences between Docker and Kubernetes can be further explained by their underlying architectural characteristics. Docker, operating as a single-node architecture, can directly utilize all available hardware resources without needing to allocate portions for cluster management. Its relatively simple network stack and direct resource access allow for faster reaction to sudden surges of packet-based attacks, minimizing processing delays under high-traffic conditions. Moreover, Docker's static container runtime management maintains fixed resource allocations and simpler scheduling policies, resulting in minimal latency and predictable behavior under load. This enables resource prioritization and page-level separation strategies to be fully effective without frequent orchestration interference.

In contrast, Kubernetes introduces a more complex networking model and dedicates a portion of system resources to manage the cluster, including components like the API server, etcd, and control planes. Although this architecture offers superior flexibility, scalability, and resilience, it also introduces scheduling overhead and load-balancing delays during high traffic surges. Kubernetes' dynamic pod scheduling and real-time resource redistribution can lead to transient inefficiencies when strategies heavily prioritize certain users or services, causing pod saturation or scheduling bottlenecks. Consequently, simpler, load-distributed strategies (e.g., Strategy 6) perform better under low concurrency, while balanced prioritization (e.g., Strategy 8) is crucial at high concurrency to maintain system stability. These architectural and operational differences fundamentally shape the effectiveness of DDoS mitigation strategies across the two platforms.

5.2. Hping3 result

5.2.1. Performance metrics

In this section, we describe the following performance metrics which were used in the experiment: 1) test completion time, 2) service down time, 3) service recovery time. Each metric is explained below:

1. Test Completion Time: As mentioned in Section 4.5.2, this experiment simulates real-world scenarios with two types of users: attackers and benign users. Test completion time is recorded based on the number of packets sent by the attacker command mentioned in 4.5.2. The time is noted when the attacker finishes executing the hping3 command.
2. Service Down Time: Service down time is recorded using the Python script simulating benign user requests mentioned in 4.5.2. If no response is received from the webpage, the service is considered terminated. The total duration of the service outage is recorded within the test completion time.
3. Service Recovery Time: In scenarios with multiple attackers, if the Python script continues to receive failed responses after the attacker has finished executing the hping3 command, the script will keep recording data until it receives consistent successful responses. The time between the end of the attack and the first successful response is considered the service recovery time. If the service recovers before the attack ends, it is noted as "NS" (no stop) in the data, indicating that the service recovered before the attack was completed.

Table 38
Test completion time in seconds in Docker and Kubernetes environments.

Test completion time in seconds	Docker			Kubernetes		
Number of Attackers	nginx	dbwrite	dbread	nginx	dbwrite	dbread
1	26.654	27.194	26.874	28.492	27.682	28.318
2	27.086	27.67	27.026	27.794	28.191	28.482
3	27.338	27.422	27.458	28.194	28.142	28.178
4	27.25	27.934	27.01	28.322	28.638	28.544

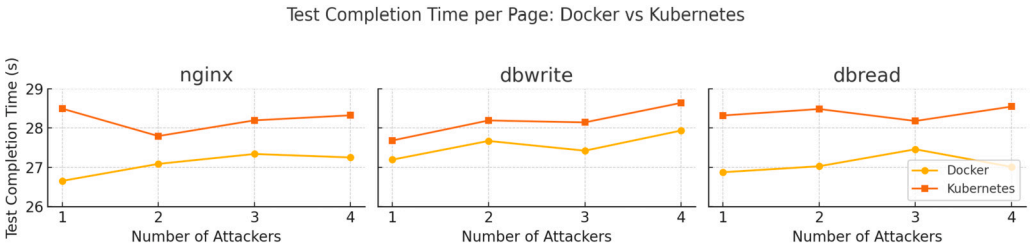

Fig. 21. Test completion time for each page type (nginx, dbwrite, dbread) under varying numbers of attackers (1, 2, 3, 4), based on the results of hping3-based stress testing in Docker and Kubernetes environments.

To evaluate whether the observed differences in test completion time and service down time across page types (nginx, dbwrite, dbread) are statistically significant within the same platform, we applied the Kruskal–Wallis test. This non-parametric method is suitable for small sample sizes, non-normal distributions, or the presence of outliers, and does not assume homogeneity of variances, making it appropriate when one-way ANOVA is not applicable.

5.2.2. Test completion time

Table 38 describes the test completion times, and Fig. 21 provides a visual summary of these results. As observed in Table 38, regardless of whether the environment is Docker or Kubernetes, the hping3 commands target the hardware layer. Consequently, the complexity of the web page does not affect the attack duration, which remains consistent across different tests. Another notable observation is that the test completion time does not increase with the number of attackers. This suggests that the attacked component is the network interface card (NIC) at the hardware level. Unlike the ab command, which simulates real users and waits for responses, thereby extending the test completion time, hping3 continuously sends attack packets regardless of whether a response is received. This difference in behavior further supports the notion that the NIC is the primary target of the hping3 attacks.

Table 39
Kruskal-Wallis Test Results for Table 38.

Platform	H	p	Significant
Docker	4.1923	0.1229	N
Kubernetes	0.9615	0.6183	N

As shown in Table 39, the Kruskal–Wallis test was used to compare the test completion times across three services (nginx, dbwrite, and dbread) within each platform. For both Docker and Kubernetes, the p-values are greater than 0.05, indicating no statistically significant differences in completion times across the three service types on the same platform. This suggests that performance remained relatively consistent regardless of the service being tested.

5.2.3. Service down time

Table 40 describes the service down times, and Fig. 22 offers a graphical representation of these findings. As observed in Table 40, there are no instances of service downtime in the Docker environment, whereas in the Kubernetes environment, the service downtime increases with the number of attackers. Moreover, the service down time for the three different web pages is approximately the same, indicating that the complexity of the web page does not impact the duration of service downtime. Under the conditions of this experiment, Docker's performance is significantly better than Kubernetes. When facing the same network attacks, the web service in Docker continues to function normally, whereas Kubernetes experiences periods of service unavailability.

As shown in Table 41, only Kubernetes data was included in the Kruskal–Wallis analysis, as Docker values were all zero and exhibited no variability, making statistical comparison infeasible. The results show that there is no statistically significant difference in service down time across the three services (nginx, dbwrite, dbread) on Kubernetes ($p = 0.8724$). This indicates that the impact of the attack on service availability was relatively consistent across different service types.

5.2.4. Service recovery time

Table 42 describes the service recovery times, and Fig. 23 provides a visual summary of these results. As observed in Table 42, since the service down time in the Docker environment is consistently 0, there are no recorded instances of service recovery time. However, in the Kubernetes environment, when the number of attackers exceeds 3, it takes an additional 2-3 seconds for the service to fully recover even after the hping3 command has finished executing. Additionally, the service recovery time for the three different web pages is approximately the same, indicating that the complexity of the web page does not impact the duration of service recovery.

As for Table 42, statistical analysis could not be conducted. Docker had no available data, while Kubernetes, although containing numerical values, did not have three or more valid continuous data groups. As a result, the Kruskal–Wallis test was not applicable, and no meaningful statistical comparison could be made.

5.3. Summary

5.3.1. Apache bench experiment summary

Experiments indicate that Docker outperforms Kubernetes in data performance under identical conditions, due to Docker's lighter-weight nature. Kubernetes, however, excels in complex environments with many pods, offering better resource management at the cost of some performance. As noted by Dewi et al. [10], Kubernetes' flexibility in capacity management and widespread adoption by cloud providers [54,43] are its key advantages.

For DDoS attack mitigation, Docker performs better by managing traffic from white-listed sources and delaying attacker responses. In Docker, white-listing strategies show superior performance. In contrast,

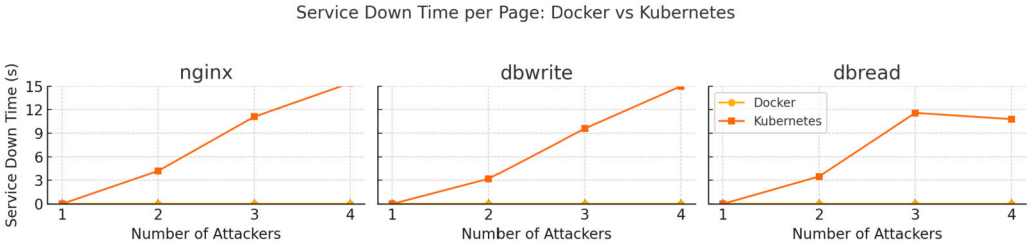

Fig. 22. Service down time (in seconds) for each page type (nginx, dbwrite, dbread) under varying numbers of attackers (1, 2, 3, 4), based on hping3 testing results in Docker and Kubernetes environments.

Table 40
Service down time in seconds in Docker and Kubernetes environments.

Service down time in seconds	Docker			Kubernetes		
Number of Attackers	nginx	dbwrite	dbread	nginx	dbwrite	dbread
1	0	0	0	0	0	0
2	0	0	0	4.2	3.2	3.5
3	0	0	0	11.1	9.6	11.6
4	0	0	0	15.4	15	10.8

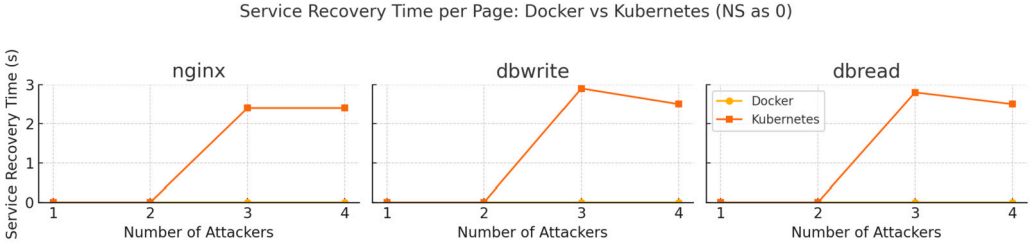

Fig. 23. Service recovery time (in seconds) for each page type (nginx, dbwrite, dbread) under varying numbers of attackers (1, 2, 3, 4), based on hping3 testing results in Docker and Kubernetes environments.

Table 41
Kruskal-Wallis Test Result for Table 40.

Platform	H	p	Significant
Kubernetes	0.273	0.8724	N

Kubernetes handles all requests impartially; even with dedicated pods for white-listed traffic, overall completion time is minimally affected. Therefore, optimizing Kubernetes to process all requests efficiently is more effective in reducing completion time.

5.3.2. Hping3 experiment result

Experiments show that Docker outperforms Kubernetes in resilience against DDoS attacks due to its lighter-weight nature, which enhances service availability.

Test completion times are consistent for both Docker and Kubernetes, unaffected by attacker numbers or web page complexity, as hping3 targets the NIC at the hardware level.

Docker experiences no downtime, while Kubernetes shows increased downtime with more attackers, with no impact from web page complexity. Docker also exhibits no recovery time, whereas Kubernetes requires 2-3 seconds of recovery when attackers exceed 3, regardless of web page complexity.

Overall, Docker excels in maintaining service availability and minimizing recovery times compared to Kubernetes, which struggles more under higher attack conditions.

5.3.3. DDoS resilience strategies in Kubernetes and Docker environments

In practical scenarios, DDoS attacks can cause significant disruptions, overwhelming servers with malicious traffic and blocking legitimate users. Effective mitigation strategies are crucial to maintain service availability.

For Docker environments, Strategy 12 (page separation with maximum resources for known benign users and DDoS mitigation) is recommended based on experimental results from Section 5.1.2. This strategy allocates significant resources to verified users and isolates malicious traffic by dedicating containers to single pages, enhancing efficiency.

For Kubernetes environments, Strategy 8 (maximum resources for benign users with DDoS mitigation) and Strategy 6 (six pods with DDoS mitigation) are recommended based on findings from Section 5.1.3. Strategy 8 allocates resources to ensure prompt processing of requests, with each pod handling all pages, simplifying management. Strategy 6 ensures equitable request distribution across six pods without specific adjustments.

These strategies improve DDoS resilience in their respective environments. Effective resource management, real-time monitoring, and prompt detection of malicious activity are essential for optimal performance.

6. Conclusion

Containerization has become the primary method for deploying applications, especially web services. However, exposing server IP addresses increases vulnerability to DDoS attacks, which can deplete resources and block legitimate user access. This study addresses the challenge of low-rate DDoS attacks by evaluating twelve mitigation strategies across Docker and Kubernetes environments, incorporating different configurations of resource allocation and page separation. These strategies were tested using three representative applications: a Simple Nginx Sample Page, a Simulated Ticket Booking System, and a Simulated E-Commerce Website.

Based on the experimental results discussed in Section 5.1.4, Strategy 12 is recommended for Docker due to its integration of page separation, benign user resource prioritization, and DDoS mitigation, achiev-

Table 42
Service recovery time in seconds in Docker and Kubernetes environments. (NS: No Stop).

Service Recovery Time in seconds	Docker			Kubernetes		
Number of Attackers	nginx	dbwrite	dbread	nginx	dbwrite	dbread
1	NS	NS	NS	NS	NS	NS
2	NS	NS	NS	NS	NS	NS
3	NS	NS	NS	2.4	2.9	2.8
4	NS	NS	NS	2.4	2.5	2.5

ing consistently strong performance under varied workloads. For Kubernetes, Strategy 6 is preferred at low concurrency, while Strategy 8 becomes more effective at higher concurrency levels, leveraging Kubernetes' orchestration capabilities to maintain service stability. Docker demonstrated superior responsiveness in handling traffic from whitelisted sources, while Kubernetes achieved efficient request processing through its dynamic pod scheduling mechanisms.

Rather than proposing standalone algorithms or theoretical frameworks, our study offers a practical, orchestration-aware comparative analysis of mitigation strategies tailored to real-world deployment settings. By simulating realistic, multi-service workloads under varying concurrency levels and deploying identical strategies across both platforms, we uncover platform-specific behavioral trade-offs that influence mitigation outcomes. This cross-platform, service-level evaluation provides actionable insights to guide strategy selection and helps practitioners design robust, container-native defenses against stealthy DDoS attacks.

6.1. Limitations and future work

While our study offers practical insights into application-layer DDoS mitigation within containerized environments, several limitations remain. First, the use of Apache Bench and hping3—though widely adopted for controlled low-rate DDoS simulations—may not fully represent the complexity or distributed scale of real-world attacks. In particular, our experiments involve a small number of simulated nodes, whereas genuine DDoS attacks typically span thousands of distributed sources. More advanced simulation tools (e.g., Grafana k6) could provide richer modeling capabilities in future studies. Moreover, these tools primarily simulate traffic load but do not emulate adaptive attacker behavior, which is common in modern DDoS campaigns.

Second, our mitigation strategies center on resource allocation, page separation, and traffic filtering at the application layer. More advanced defenses—such as adaptive, AI-driven mechanisms or reputation-based filtering—were not explored. Additionally, while Kubernetes supports cluster-wide or ingress-level defenses, our study limits its scope to service-level behaviors within containers to maintain comparability with Docker's single-node model. Thus, our findings are most applicable to developers and administrators seeking container-level protection rather than cluster-wide enforcement policies.

Third, all experiments were conducted in a controlled, single-cluster environment. As such, we did not evaluate strategy performance under multi-node Kubernetes clusters, hybrid-cloud deployments, or heterogeneous production scale traffic. Furthermore, although service-level metrics such as completion time and response latency were analyzed, detailed system-level resource profiling (e.g., CPU, memory, and bandwidth usage) under high load was not fully incorporated. In extreme overload conditions, system monitoring tools often failed to return values due to service unresponsiveness. While we observed trends suggesting saturation behavior (e.g., resource usage spiking to 100% before dropping to 0%), we were unable to capture sustained metrics during peak failure.

Future research may extend this work in several directions. First, scaling the experimental design to include multi-node Kubernetes clusters and distributed attacker sources would allow testing under realistic cloud-native DDoS conditions. In addition, conducting systematic scalability assessments under increasing node counts and varying traffic patterns would further validate strategy robustness under diverse deployment scenarios. Second, investigating additional platforms—such as Podman and LXC/LXD—could provide further insight into orchestration-level trade-offs and performance differences. Third, the integration of AI-driven or anomaly-aware defense strategies could improve adaptability against evolving attacks. Future work should also analyze network bandwidth utilization efficiency for different mitigation strategies across Docker and Kubernetes environments, especially under high concurrency or resource contention. It is also worth exploring hybrid models that combine Docker's lightweight responsiveness with Kubernetes' orchestration flexibility, particularly in deployments requiring both low-latency response and scalable management. Moreover, Kubernetes-specific optimizations deserve deeper exploration. These include proactive auto-scaling policies, fine-grained pod-level resource configurations, and adaptive traffic redistribution mechanisms. In particular, future work may investigate: (1) aggressive auto-scaling policies triggered by abnormal traffic patterns, possibly detected through monitoring tools or integrated intrusion detection systems (IDS); (2) fine-grained tuning of pod-level resource requests and limits to prioritize mission-critical services during attack scenarios; and (3) pre-emptive horizontal scaling or dynamic resource reservation based on load prediction models to safeguard performance under stress. Lastly, establishing a more robust resource monitoring framework capable of capturing CPU, memory, and bandwidth usage trends even under service saturation conditions will allow for more precise evaluation of system overhead and performance degradation across strategies.

CRediT authorship contribution statement

Yung-Ting Chuang: Writing – original draft, Writing – review & editing, Validation, Supervision, Project administration, Formal analysis, Investigation, Methodology, Conceptualization, Visualization. **Chih-Han Tu:** Writing – original draft, Software, Conceptualization, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is supported by NSTC 110-2410-H-A49-017-MY2 and 112-2410-H-A49-025 of National Science and Technology Council, Taiwan.

Data availability

All experimental scripts, configuration files, and related materials used in this study have been made publicly available. Our dataset can be found on https://reurl.cc/xN3e8e.

References

[1] T. Borangiu, D. Trentesaux, A. Thomas, P. Leitão, J. Barata, Digital transformation of manufacturing through cloud services and resource virtualization, 2019.

[2] J. Watada, A. Roy, R. Kadikar, H. Pham, B. Xu, Emerging trends, techniques and open issues of containerization: a review, IEEE Access 7 (2019) 152443–152472.

[3] N.G. Bachiega, P.S. Souza, S.M. Bruschi, S.D.R. De Souza, Container-based performance evaluation: a survey and challenges, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2018, pp. 398–403.

[4] J. Chen, Q. Guan, X. Liang, L.J. Vernon, A. McPherson, L.-T. Lo, Z. Chen, J.P. Ahrens, Docker-enabled build and execution environment (bee): an encapsulated environment enabling hpc applications running everywhere, Available: arXiv:1712.06790, 2017.

[5] C. Boettiger, An introduction to Docker for reproducible research, Oper. Syst. Rev. 49 (1) (2015) 71–79.

[6] Z. Huang, S. Wu, S. Jiang, H. Jin, Fastbuild: accelerating Docker image building for efficient development and deployment of container, in: 2019 35th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2019, pp. 28–37.

[7] D. Merkel, et al., Docker: lightweight Linux containers for consistent development and deployment, Linux J. 239 (2) (2014) 2.

[8] A.M. Potdar, D. Narayan, S. Kengond, M.M. Mulla, Performance evaluation of Docker container and virtual machine, Proc. Comput. Sci. 171 (2020) 1419–1428.

[9] S.K. Mondal, R. Pan, H.D. Kabir, T. Tian, H.-N. Dai, Kubernetes in it administration and serverless computing: an empirical study and research challenges, J. Supercomput. (2022) 1–51.

[10] L.P. Dewi, A. Noertjahyana, H.N. Palit, K. Yedutun, Server scalability using Kubernetes, in: 2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON), IEEE, 2019, pp. 1–4.

[11] A. Poniszewska-Marańda, E. Czechowska, Kubernetes cluster for automating software production environment, Sensors 21 (5) (2021) 1910.

[12] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, Microservice based architecture: towards high-availability for stateful applications with Kubernetes, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2019, pp. 176–185.

[13] B. Johansson, M. Rågberger, T. Nolte, A.V. Papadopoulos, Kubernetes orchestration of high availability distributed control systems, in: 2022 IEEE International Conference on Industrial Technology (ICIT), IEEE, 2022, pp. 1–8.

[14] A.A. Khatami, Y. Purwanto, M.F. Ruriawan, High availability storage server with Kubernetes, in: 2020 International Conference on Information Technology Systems and Innovation (ICITSI), IEEE, 2020, pp. 74–78.

[15] S. Hardikar, P. Ahirwar, S. Rajan, Containerization: cloud computing based inspiration technology for adoption through Docker and Kubernetes, in: 2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC), IEEE, 2021, pp. 1996–2003.

[16] M. Jazayeri, Some trends in web application development, in: Future of Software Engineering (FOSE'07), IEEE, 2007, pp. 199–213.

[17] K. Nath, S. Dhar, S. Basishtha, Web 1.0 to web 3.0-evolution of the web and its various challenges, in: 2014 International Conference on Reliability Optimization and Information Technology (ICROIT), IEEE, 2014, pp. 86–89.

[18] S.T. Zargar, J. Joshi, D. Tipper, A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks, IEEE Commun. Surv. Tutor. 15 (4) (2013) 2046–2069.

[19] F. Lau, S.H. Rubin, M.H. Smith, L. Trajkovic, Distributed denial of service attacks, in: Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics, in: Cybernetics Evolving to Systems, Humans, Organizations, and Their Complex Interactions (Cat. No. 0), vol. 3, IEEE, 2000, pp. 2275–2280.

[20] K.N. Mallikarjunan, K. Muthupriya, S.M. Shalinie, A survey of distributed denial of service attack, in: 2016 10th International Conference on Intelligent Systems and Control (ISCO), IEEE, 2016, pp. 1–6.

[21] G.A. Jaafar, S.M. Abdullah, S. Ismail, et al., Review of recent detection methods for http ddos attack, J. Comput. Netw. Commun. (2019) 2019.

[22] A.A. Tripathi, Attacking and defending Kubernetes, Ph.D. thesis, Dublin Business School, 2024.

[23] S. Koksal, F.O. Catak, Y. Dalveren, Flexible and lightweight mitigation framework for distributed denial-of-service attacks in container-based edge networks using Kubernetes, IEEE Access (2024).

[24] L.D. Garcia, Real-time network simulations for ml/dl ddos detection using Docker, Master's thesis, California Polytechnic State University, 2024.

[25] E. Gamess, M. Parajuli, Image-processing workloads and ddos attack resilience: evaluating Docker and podman containers on raspberry pi and odroid, in: Proceedings of the 2024 ACM Southeast Conference, 2024, pp. 138–147.

[26] A. Patidar, G. Somani, Serving while attacked: ddos attack effect minimization using page separation and container allocation strategy, J. Inf. Secur. Appl. 59 (2021) 102818.

[27] W. Zhijun, L. Wenjing, L. Liang, Y. Meng, Low-rate dos attacks, detection, defense, and challenges: a survey, IEEE Access 8 (2020) 43920–43943.

[28] Y. Xiang, K. Li, W. Zhou, Low-rate ddos attacks detection and traceback by using new information metrics, IEEE Trans. Inf. Forensics Secur. 6 (2) (2011) 426–437.

[29] Z. Li, H. Jin, D. Zou, B. Yuan, Exploring new opportunities to defeat low-rate ddos attack in container-based cloud environment, IEEE Trans. Parallel Distrib. Syst. 31 (3) (2019) 695–706.

[30] Y. Liu, D. Lan, Z. Pang, M. Karlsson, S. Gong, Performance evaluation of containerization in edge-cloud computing stacks for industrial applications: a client perspective, IEEE Open J. Ind. Electron. Soc. 2 (2021) 153–168.

[31] N. Zhou, H. Zhou, D. Hoppe, Containerization for high performance computing systems: survey and prospects, IEEE Trans. Softw. Eng. 49 (4) (2022) 2722–2740.

[32] M. Nardelli, C. Hochreiner, S. Schulte, Elastic provisioning of virtual machines for container deployment, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, 2017, pp. 5–10.

[33] G. Somani, S. Chaudhary, Application performance isolation in virtualization, in: 2009 IEEE International Conference on Cloud Computing, IEEE, 2009, pp. 41–48.

[34] H. Zhao, J. Wang, F. Liu, Q. Wang, W. Zhang, Q. Zheng, Power-aware and performance-guaranteed virtual machine placement in the cloud, IEEE Trans. Parallel Distrib. Syst. 29 (6) (2018) 1385–1400.

[35] B. Varghese, L.T. Subba, L. Thai, A. Barker, Container-based cloud virtual machine benchmarking, in: 2016 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2016, pp. 192–201.

[36] X. Wan, X. Guan, T. Wang, G. Bai, B.-Y. Choi, Application deployment using microservice and Docker containers: framework and optimization, J. Netw. Comput. Appl. 119 (2018) 97–109.

[37] M.T. Chung, N. Quang-Hung, M.-T. Nguyen, N. Thoai, Using Docker in high performance computing applications, in: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), IEEE, 2016, pp. 52–57.

[38] C. Kan, Docloud: an elastic cloud platform for web applications based on Docker, in: 2016 18th International Conference on Advanced Communication Technology (ICACT), IEEE, 2016, pp. 478–483.

[39] D. Huang, H. Cui, S. Wen, C. Huang, Security analysis and threats detection techniques on Docker container, in: 2019 IEEE 5th International Conference on Computer and Communications (ICCC), IEEE, 2019, pp. 1214–1220.

[40] J. Chelladhurai, P.R. Chelliah, S.A. Kumar, Securing Docker containers from denial of service (dos) attacks, in: 2016 IEEE International Conference on Services Computing (SCC), IEEE, 2016, pp. 856–859.

[41] J. Wenhao, L. Zheng, Vulnerability analysis and security research of Docker container, in: 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE), IEEE, 2020, pp. 354–357.

[42] R. Dautov, H. Song, Towards agile management of containerised software at the edge, in: 2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS), vol. 1, IEEE, 2020, pp. 263–268.

[43] J. Shah, D. Dubaria, Building modern clouds: using Docker, Kubernetes & Google cloud platform, in: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2019, pp. 0184–0189.

[44] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, Deploying microservice based applications with Kubernetes: experiments and lessons learned, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 970–973.

[45] V. Medel, O. Rana, J.Á. Bañares, U. Arronategui, Modelling performance & resource management in Kubernetes, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, 2016, pp. 257–262.

[46] D.B. Bose, A. Rahman, S.I. Shamim, 'Under-reported' security defects in Kubernetes manifests, in: 2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS), IEEE, 2021, pp. 9–12.

[47] D. D'Silva, D.D. Ambawade, Building a zero trust architecture using Kubernetes, in: 2021 6th International Conference for Convergence in Technology (i2ct), IEEE, 2021, pp. 1–8.

[48] S.I. Shamim, Mitigating security attacks in Kubernetes manifests for security best practices violation, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1689–1690.

[49] R.B. David, A.B. Barr, Kubernetes autoscaling: yoyo attack vulnerability and mitigation, arXiv preprint, arXiv:2105.00542, 2021.

[50] A.E. Nocentino, B. Weissman, A.E. Nocentino, B. Weissman, Kubernetes architecture, in: SQL Server on Kubernetes: Designing and Building a Modern Data Platform, 2021, pp. 53–70.

[51] S. Pothuganti, M. Samanth, Comparative analysis of load balancing in cloud platforms for an online bookstore web application using apache benchmark, 2023.

[52] Q. Fan, Q. Wang, Performance comparison of web servers with different architectures: a case study using high concurrency workload, in: 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), IEEE, 2015, pp. 37–42.

[53] S. Gokhale, A. Turcotte, F. Tip, Automatic migration from synchronous to asynchronous javascript apis, Proc. ACM Program. Lang. 5 (OOPSLA) (2021) 1–27.

[54] S.P. Sinde, B. Thakkalapally, M. Ramidi, S. Veeramalla, Continuous integration and deployment automation in aws cloud infrastructure, Int. J. Res. Appl. Sci. Eng. Technol. 10 (2022) 1305–1309.

Dr. Yung-Ting Chuang is an associate professor in the Institute of Information Management at National Yang Ming Chiao Tung University, Taiwan. Yung-Ting received all of her BS, MS, and PhD degrees in Electrical and Computer Engineering from the University of California, Santa Barbara in 2006, 2008, and 2013, respectively. Her research interests include peer-to-peer networks, social networks, distributed systems, network security, edge computing, wireless sensor networks, and, information search and retrieval.

Chih-Han Tu received his MS degree in the Institute of Information Management at National Yang-Ming Chiao Tung University, Taiwan, in 2024. His research interests include Distributed Systems, DDoS attacks and defenses, Docker, and Kubernetes.