10th International Scientific Conference
Technics, Informatics, and Education – TIE 2024
20-22 September 2024

Session: *Computer Sciences and Information Technology*
Professional paper
DOI: 10.46793/TIE24.034L

Horizontal Scaling with Session Preservation of PHP Applications with MVC Architecture

Veljko Lončarević[1*] [0009-0007-4296-2709], Željko Jovanović[2] [0000-0001-5401-8634], Vanja Luković[2] [0000-0002-1887-6102], Marina Milošević[2] [0000-0001-7927-1169], Savo Šućurović[1] [0009-0007-4950-5375] and Aleksa Ričanin[3] [0009-0006-8145-403X]

[1] University of Kragujevac, Faculty of Technical Sciences, Čačak, Serbia
[2] University of Kragujevac, Faculty of Technical Sciences, Department of Computer and Software Engineering, Čačak, Serbia
[3] University of Kragujevac, Faculty of Technical Sciences, Department of Information Technologies, Čačak, Serbia
[*] veljkoloncarevicharry@gmail.com

Abstract: *This paper explores horizontal scaling of PHP MVC applications with session preservation for enhanced availability and resource efficiency. It covers theoretical aspects of MVC architecture, PHP in web development, session handling, and horizontal scaling methods, including load balancers, Docker, and Kubernetes. The practical methodology details environment setup, application development, session management, Dockerization, Kubernetes integration, and horizontal scaling configuration. Performance testing reveals significant improvements, showing a response time decrease from an unresponsive state at 1000 RPS (5111 ms) to 32 ms at 2500 RPS with horizontal scaling. The study contributes insights and practical guidance for highly available and scalable web applications.*

Keywords: *horizontal scaling; session preservation; Kubernetes; high availability; load balancing*

1. INTRODUCTION

In the dynamic landscape of web development, the escalating demand for high availability and optimal resource management has highlighted the imperative need for horizontal scaling of web applications. This research addresses a pivotal challenge faced by developers when scaling applications—striking a balance between performance enhancement and the intricate preservation of user sessions. As web applications expand horizontally to meet growing demands, developers grapple with several formidable issues. Challenges include mitigating server overloads, ensuring seamless user experience during server transitions, and maintaining data integrity across distributed instances.

This paper gives a brief overview of the multifaceted problems developers encounter during the scaling process. These challenges encompass the intricacies of session continuity, load balancing intricacies, and the efficient orchestration of containerized environments. To surmount these hurdles, the proposed solution leverages cutting-edge technologies such as Docker and Kubernetes. By navigating through these challenges, this study aims to contribute practical insights, offering a comprehensive solution for developers seeking to develop scalable and highly available web applications.

2. THEORETICAL FOUNDATIONS

2.1. MVC Architecture

Model-View-Controller (MVC) architecture is a design pattern widely employed in software development to enhance the organization and maintainability of applications. At its core, MVC separates an application into three interconnected components: the Model, View, and Controller (Fig. 1).

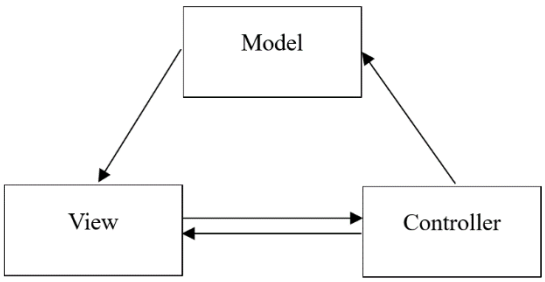

Figure 1. *MVC Architecture*

The Model represents the application's data and business logic, encapsulating the rules governing data manipulation and storage. It serves as the engine that manages and updates the application's state, ensuring a clean separation from the user interface. The View encompasses the user interface elements, presenting data to users and collecting input. It reflects the current state of the Model and displays information to users in a comprehensible

format. The View is responsible for visual representation and user interaction but remains detached from the underlying data logic. The Controller acts as the intermediary, facilitating communication between the Model and the View. It interprets user inputs from the View, processes them, and triggers corresponding actions in the Model. The Controller plays a pivotal role in managing the flow of information between the Model and the View, ensuring effective coordination and response to user interactions [1]. This architectural pattern promotes modularity and flexibility, making it easier to update and maintain different aspects of the application independently. MVC not only enhances code organization but also fosters a clear separation of concerns, enabling developers to focus on specific components without compromising the integrity of the entire system [2].

2.2. PHP

PHP stands out as a popular server-side programming language in web development, playing a crucial role in crafting dynamic and interactive web applications. Its multifaceted capabilities empower developers to generate dynamic content on web pages, tailoring them to user demands and database inputs. PHP facilitates interaction with various databases, such as MySQL, PostgreSQL, and Oracle, enabling real-time data storage, retrieval, and updates. Additionally, it excels in processing data from web forms, validating inputs, and executing actions based on user submissions. Furthermore, PHP supports the creation and management of sessions, tracking user states during web page visits for functionalities like authentication, shopping carts, and personalization. Its utility extends to constructing APIs (Application Programming Interfaces) for seamless communication between diverse software components, including web and mobile applications. PHP's capacity to dynamically generate HTML code based on data or logic enhances the creation of dynamic components like lists, tables, menus, and more on web pages. With control over application flow, security features to counteract threats like SQL injection and XSS attacks, seamless integration with other technologies, and compatibility with popular web servers, PHP stands as a versatile and powerful server-side language, making it a cornerstone in the world of web development [3].

Sessions in PHP serve as a mechanism for maintaining state and tracking user information during their visit to a website. This functionality enables applications to temporarily store data on the server and utilize it throughout the user's session, facilitating personalization and interaction [4]. Session data is commonly employed for preserving details about a logged-in user, shopping cart contents, language preferences, or other settings crucial to retain during a website visit. In PHP applications, a session acts as a tool for preserving state across various HTTP requests that a user sends to the server during their visit to the website. Stored on the server, PHP maintains session data in temporary files or memory space, depending on the configuration. Each session possesses a unique identifier (session ID), typically transmitted to the user as a cookie or added to the URL. The mentioned basic concepts of sessions enable PHP applications to uphold states and track user information dynamically, facilitating personalization and interaction throughout their visit to the website. In PHP, sessions are implemented using the associative array global object $_SESSION. To begin a session, the session_start() function is utilized, initiating or resuming a session and allowing access to the $_SESSION array. This array serves as a container to store various session variables. Sessions are typically ended by calling session_destroy(). Accessing stored session values is achieved by referencing the $_SESSION array with the corresponding key. For example, to access a session variable named "username," one would use $_SESSION['username']. To unset or remove a specific session variable, the unset($_SESSION['variable_name']) function is employed, effectively removing the designated value from the session array. This mechanism provides a flexible and straightforward approach to managing session data in PHP applications.

2.3. Horizontal Scaling

Horizontal scaling of applications involves adding multiple instances (copies) of the same application to increase its capacity for handling requests and loads (Fig. 2). This technique is commonly used to enhance the performance, availability, and resilience of applications [5].

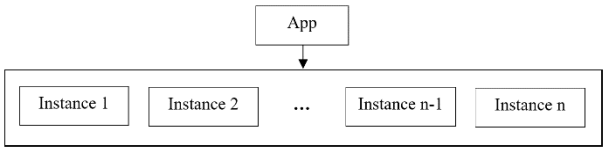

Figure 2. *Horizontal Scaling of an Application*

The key characteristic of horizontal scaling is that instead of overloading a single instance of the application, new instances are added to maintain a load balance among them. This is often achieved with the help of load balancing systems that direct incoming requests to available instances [6]. If one instance of the application becomes unavailable due to a malfunction or issue, other instances can take over the load, ensuring continuous availability. By adding more instances, the application can respond faster to requests and distribute the resources of each individual instance. In case the number of users or requests suddenly increases, horizontally scaled applications can quickly adapt to

the increased load. Horizontal scaling enables upgrading and maintaining the application without a complete service outage, as one instance can be updated while others continue to operate. It's important to note that horizontal scaling is not always the ideal solution for all applications and requires careful planning and resource management to achieve optimal performance and efficiency. Additionally, the application must be designed to support horizontal scaling, which may involve certain changes in the application's architecture.

2.4. Load Balancers

A **load balancer**, whether a physical device or software entity, plays a pivotal role in optimizing the performance, availability, and reliability of applications. Its main function is to evenly distribute incoming network requests among multiple servers, instances, or resources [7]. By preventing any single server from being overloaded and ensuring a balanced workload, load balancers contribute to faster response times and improved fault tolerance. In the context of horizontal scaling, load balancers are instrumental in accepting network requests, routing and distributing requests among available instances, ensuring proper load distribution, and monitoring the health and performance of each instance. This results in an efficient and resilient system, where workloads are evenly spread, and any potential disruptions are mitigated through automatic redirection of requests to healthy servers. Ultimately, load balancers are a crucial component in modern infrastructures, enhancing application availability and maintaining optimal performance.

The **IP hashing algorithm** is employed by load balancers to achieve session preservation, ensuring that requests from the same client are consistently directed to the same server. In this approach, the load balancer calculates a hash value based on the source IP address of the incoming request. This hash value determines the server (or instance) to which the request will be forwarded. By utilizing the source IP address, the algorithm ensures that all requests originating from a specific client IP are directed to a single server or instance, maintaining session continuity. Fig. 3 shows the principle with which the IP hashing algorithm operates.

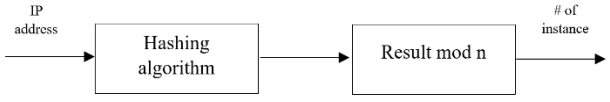

Figure 3. *IP Hashing Algorithm*

2.5. Docker

Docker is a containerization platform that empowers developers to package, distribute, and execute applications and their dependencies in isolated, lightweight, and portable containers. These containers encapsulate applications along with libraries and necessary resources, ensuring complete isolation from the host system and other containers [8]. This technology enables consistent and reliable application execution regardless of the environment. Key components and concepts within the Docker ecosystem include containers, Docker images, Dockerfiles, Docker Compose, and Docker Hub. Containers serve as the fundamental units, containing applications and associated resources. Docker images define the container's content and can be shared and reused. Dockerfiles are textual files specifying the steps for creating Docker images, providing precise configuration control. Docker Compose facilitates the definition and management of multiple containers as part of a single application. Docker Hub, a public registry, enables the sharing and retrieval of Docker images. Orchestration tools like Docker Swarm and Kubernetes work seamlessly with Docker, automating container management and scalability in diverse environments. Application containerization, as a technology, allows developers to isolate applications and their dependencies within containers—distinct environments known as containers. These containers, a form of operating system-level virtualization, package and execute applications along with all necessary resources, such as libraries and configurations, within isolated environments. This technology ensures consistent and reliable execution of applications irrespective of the executing environment. Containers remain isolated from the host system and other containers, preventing interference between applications. Portable and capable of running on various operating systems and cloud platforms without extensive adaptations, containers start quickly by sharing the host's operating system kernel [5].

2.6. Kubernetes

Kubernetes, often referred to as "K8s," is an open-source platform for container orchestration. Kubernetes automates the management, deployment, and scaling of containerized applications within a cluster. It enables automatic scaling and restarting of containers within a group of servers or a "cluster," ensuring applications adapt to variable resource needs [9]. Users define the desired system state through YAML or JSON files, and Kubernetes ensures the cluster maintains that state, taking automated actions to achieve it. Kubernetes possesses self-healing capabilities; in case of issues with an application instance, it can automatically replace it with a functional one. Dynamic scaling of applications is possible to meet changing resource demands, ensuring resilience to load changes. Kubernetes facilitates configuration management of applications and environments through configuration maps and secrets, simplifying the handling of sensitive data. The

platform also allows the definition of services and load balancing to provide access to applications and distribute the load among instances. Known for its flexibility, Kubernetes manages various application and resource types, including services, microservices, data storage, and more. Supporting multiple cloud providers, Kubernetes allows the management of multiple clusters from a single control center. Working with Kubernetes requires a set of tools to manage clusters, applications, and resources. These tools are crucial for development, testing, and management of Kubernetes environments. Some essential tools include kubectl for interacting with clusters, Helm for managing Kubernetes packages, kubeadm for fast cluster setup, k9s for interactive management through a Text User Interface (TUI), and minikube for running Kubernetes clusters locally during development. Additional tools like kubectx, kubens, kustomize, Velero, Kubeval, and Kube-hunter extend Kubernetes functionality, providing capabilities such as easy cluster switching, configuration management, backup, validation, and security testing. Kubernetes clusters are the fundamental infrastructure units for executing and managing containerized applications. Comprising nodes and control planes, clusters ensure high scalability, availability, and manageability of containerized applications. Nodes, either worker or master nodes, execute applications and manage the Kubernetes Agent (kubelet), maintaining the desired cluster state. Control planes, acting as the cluster's brain, include components like the API Server, etcd, Scheduler, and Controller Manager, responsible for managing and controlling the cluster. Worker nodes execute containers with applications, while master nodes manage and coordinate the cluster's operations.

3. METHODOLOGY

In the methodology section, comprehensive details regarding the development environment setup will be provided. This will encompass the initial configuration steps tailored for the Alpine Linux operating system, followed by the development of a PHP Model-View-Controller (MVC) application. The section will delve into the intricacies of session management within the application, elucidate the Dockerization process of the application, and subsequently outline the setup of a Kubernetes environment, complete with services and a load balancer. This holistic approach aims to offer a clear and detailed account of the entire process, ensuring a comprehensive understanding of the methodology employed in the development and deployment phases.

3.1. Preparing the Environment

Alpine Linux serves as the designated operating system for the research paper under consideration. The initial step involves installing the nginx web server by executing the command "apk add nginx" in the terminal. Subsequently, the nginx service is started, and its automatic initiation during the operating system startup is ensured by executing "service nginx start" and "rc-update add nginx default" commands in the terminal. Following that, the installation of Docker and related tools is performed, with the Docker service set to launch automatically during the operating system startup using the "apk add docker" and "rc-update add docker boot" commands. The process then continues with the installation of the "kind" tool and the subsequent creation of a Kubernetes cluster using specific commands in the terminal. Once the environment preparation is completed, it is essential to verify the success of each step and check the cluster's status, ensuring its activation using the "kubectl" tool.

3.2. Developing the PHP MVC Application

For this paper, the development of an application is required, encompassing a user login form and a user list with summarized data on HTTP requests sent to the server. This list should be accessible only to successfully logged-in users, with the recorded request types being GET upon accessing the list and POST after a successful login. Additionally, the application should store the user's name and the time of the last access in the session, displaying this information above the request list. It is necessary to define key components, including a model for managing user data, a model for managing user-request data, controllers for processing user requests, and multiple views responsible for the user interface and structured data presentation. The models for managing user and user-request data should include functions for logging in and reading summarized data from the database, respectively. Fig. 4 shows the entity-relationship diagram for the database created for this research paper.

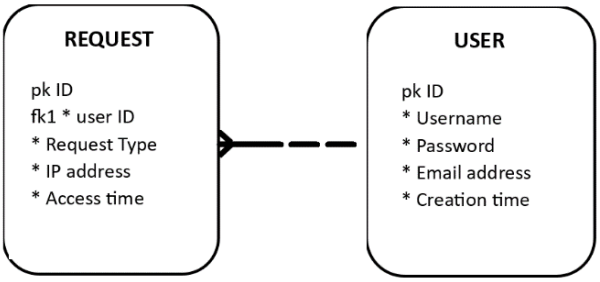

Figure 4. *Entity-Relationship Diagram*

The controller must handle incoming user requests, such as login and displaying summarized data, while the views include essential elements like header and footer for basic information, a login form view, and a data list view.

3.3. Leveraging Session Functionalities

In accordance with the specified specifications, sessions are employed to store information about the user's name and the time of their last access. At the beginning of each PHP script interacting with the session, including login and viewing previous visits pages, a session must be initiated. Typically, this is done by calling the session start function at the top of each PHP script, allowing PHP to access and manipulate the session. Upon user login through the form, their username can be stored in the session using the associative array $_SESSION. Additionally, after recording information about the sent request into the database, the time of the last access can be saved in the session. This facilitates later display of this information on other pages. On pages where it is necessary to display session information, a check can be performed to verify the existence of this information in the session, and if available, it can be displayed. This allows for personalized greetings and the display of the last access time if this data is accessible in the session. When the user finishes the session or logs out of the site (if it is part of the requirements of some other potential application), the session can be terminated by calling the session destroy function. This clears all data in the session and closes it. This process enables the preservation of user information and the time of the last access during their session on the site.

3.4. Dockerizing the Application

The Dockerization process of the given application involves several key steps. Initially, the choice of the base image plays a crucial role in this process. To ensure a compact size, the php:7.4.0-fpm-alpine image was selected, occupying a mere 25 megabytes of space. Subsequently, a Dockerfile must be created in the root directory of the application. This textual file outlines the steps for building the image, specifying the chosen base image, and copying the application code into the appropriate directory within the base image (typically /var/www/html). Adjustments to the Nginx configuration are imperative for seamless integration. The nginx.conf file, tailored to the needs of the web application (such as specifying the server definition), must be copied into the image within the Dockerfile. Fig. 5 shows the Dockerfile written for this research paper.

```
1   FROM php:8-fpm-alpine3.14
2   RUN apk update && apk upgrade
3   RUN apk add nginx --no-cache
4   RUN mkdir -p /var/log/nginx
5   RUN mkdir -p /var/www/html
6   COPY . /var/www/html
7   COPY nginx.conf /etc/nginx/nginx.con
8   RUN docker-php-ext-install mysqli pd
9   EXPOSE 8080
10  CMD php-fpm && nginx -g 'daemon off;
```

Figure 5. *Dockerfile*

Following this, the Docker image is constructed using the docker build command, which executes the instructions laid out in the Dockerfile. Lastly, in preparation for deployment, Kubernetes Deployment resources need to be created using YAML configuration. This involves specifying the desired number of pods to ensure the application's availability and scalability within the Kubernetes cluster. Through these steps, the application is effectively containerized and ready for deployment in a Dockerized environment, providing a streamlined and consistent execution across various platforms.

3.5. Setting Up Kubernetes Resources

Upon the creation of the Deployment resource in Kubernetes, its responsibility encompasses managing and scaling the application's pods within the cluster. However, this doesn't automatically expose the application to external users. To facilitate external access, a Service resource is typically created. The created Service will also act as a load balancer based on the IP hash algorithm. Defining a Service involves creating a YAML file specifying the resource type as a service, setting the service type to "LoadBalancer," and crucially, configuring the "sessionAffinity" field to "ClientIP." The "selector" field in the service definition is pivotal, pointing to the corresponding Deployment that the service should balance. After defining the YAML file, the resource needs to be applied using the kubectl apply command. Once the Service is created, its status can be verified to determine the external IP address of the load balancer through which the application will be accessed. A YAML configuration written for this research paper is shown on Fig. 6.

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: php-nginx-service
5   spec:
6     selector:
7       app: php-nginx-app
8     ports:
9       - protocol: TCP
10        port: 8080
11        targetPort: 8080
12    type: LoadBalancer
13    sessionAffinity: ClientIP
```

Figure 6. *Deployment YAML Configuration*

In the context of automatic horizontal scaling based on pod workload, the HorizontalPodAutoScaler (HPA) resource in Kubernetes is utilized. This resource enables automatic scaling of the number of pod replicas based on defined metrics or resources. The HPA offers a set of configurable fields in its YAML definition to precisely manage the horizontal scaling of the application. Crucial fields include "spec," which defines the HPA specification, "scaleTargetRef," identifying the Deployment to be scaled, "minReplicas" and "maxReplicas" to set the minimum and maximum number of replicas, and "metrics," a list defining the metrics or resources

used for scaling decisions. Frequently used metrics include processor and RAM load. The "type," "resource," "targetAverageUtilization," and "targetAverageValue" fields further refine the scaling behavior based on the chosen metric type. Once configured, the HPA dynamically adjusts the number of pod replicas to handle varying workloads effectively. A separate Deployment has been created for the database, using MySQL as the database management system. Ten different user accounts were created in the table "Users", and 200,000 instances were inserted into the table "Requests".

3.6. Testing Performance

Testing the performance of such an application is crucial to ensure that it can function efficiently and reliably under various load conditions. Application performance directly impacts user experience and the application's ability to handle user requests in real-time. Performance testing aids in identifying potential issues and bottlenecks in the application or infrastructure before deployment. This includes monitoring response times, request processing capacity, resource loads such as processor and memory usage, and identifying points where the application may become slower or unresponsive. Performance testing also helps optimize the application and infrastructure for efficient resource utilization and improved scalability. Informed decisions about scaling needs, code or infrastructure optimization can be made through this testing, ensuring that the application remains stable and responsive during user base growth or increased load. In the context of horizontal scaling testing for a web application, selected metrics include load and response time. The goal of testing is to determine how these metrics change with an increase in the number of application instances. It is expected that the requests per second (RPS) will increase with scaling, while response time remains stable or minimally increases. If response time significantly grows with scaling, it may indicate performance or resource issues that need resolution. The results of testing should demonstrate that the application scales efficiently, supporting a higher number of users without a significant degradation in response time. This is crucial to ensure that the application remains fast and responsive even under high loads, enhancing user experience and satisfaction. Locust, a Python library and load testing tool, can be employed to simulate a large number of users interacting with the web application. This enables load and response time testing to assess the application's behavior under various load conditions and horizontal scaling. Test scenarios can be defined using Python scripts, with each scenario representing simulated user behavior. Testing is then performed with the configuration of the number of users and other testing parameters. After running the test, performance can be monitored through Locust's web interface, displaying relevant metrics. This allows for the analysis of response times, requests per second, and other performance indicators to identify potential problems and optimize the application. For horizontal scaling testing, multiple Locust instances ("slaves") can be added to simulate increased load and evaluate the application's behavior under such conditions.

4. RESULTS AND DISCUSSION

4.1. Results

The performance testing results, utilizing the Locust tool, encompassed multiple load and response time tests, concurrently monitoring the number of pods created by the HorizontalPodAutoScaler. The consolidated results are presented in Table 1, showcasing the application's behavior in both horizontal scaling and non-scaling scenarios. Two types of requests—GET, returning HTML code and a list with summary data, and POST for user login—were employed, each representing 50% of the test load.

Table 1. *Test Results*

Requests Per Second (RPS)	Response Time (in ms)	Number of active instances
Application without Horizontal Scaling		
250	33	1
500	36	1
750	116	1
1000	5111	1
Application with Horizontal Scaling		
1000	27	3
1500	28	5
2000	28	5
2500	32	8

The tests revealed that the application could efficiently handle a small number of requests per second (RPS) without a significant increase in response time. However, under moderate load (750 RPS), the response time tripled, and at 1000 RPS, the application became nearly unresponsive, with response times exceeding 5 seconds and a notable percentage of failed requests. In the case of horizontal scaling, the tests showed positive outcomes. With three active pods, the application efficiently processed 1000 RPS, demonstrating its ability to react effectively to moderate loads. As the number of active pods increased to 5 and 8 in subsequent tests, the application supported higher RPS (1500, 2000, and 2500), indicating effective horizontal scaling. Notably, response times, measured in milliseconds, remained relatively low throughout all tests, demonstrating the application's responsiveness even under increased load. The gradual increase in active pods in each

test indicates successful horizontal scaling, highlighting Kubernetes' efficient resource management and ability to scale the application to support growing loads. The results suggest that the application successfully scaled horizontally to handle increased RPS. Response times remained acceptably low, and the number of active pods increased proportionally to accommodate the growing load. This implies that horizontal scaling, combined with session persistence in a PHP MVC application using Docker and Kubernetes, can be an effective strategy to enhance application performance and scalability. The absence of a sharp increase in response time indicates the application's ability to respond well to higher loads, emphasizing the efficacy of the proposed scaling approach.

4.2. Comparison with Similar Research

In comparison with similar research efforts, the findings of this study align with those presented in [10], where an increase in performance, measured in terms of Requests Per Second (RPS), was observed. The current research similarly demonstrates enhanced application scalability, allowing for the handling of higher RPS loads efficiently. However, a distinct approach is taken in [11], where the authors introduce a custom Kubernetes Horizontal Pod Autoscaler Algorithm (KPHA-A) resource. Notably, [11] managed to achieve a notable optimization in response times when compared to using the default horizontal autoscaling resource provided by Kubernetes. The custom KPHA-A resource, designed specifically for their context, resulted in response times that were consistently 1.5 to 2 times lower than those achieved with the default Kubernetes horizontal autoscaler. This discrepancy in performance outcomes emphasizes the significance of tailoring autoscaling strategies to the unique characteristics and requirements of the application or system under consideration. While both studies, including [10], indicate positive scalability results, the novel approach presented in [11] with the custom KPHA-A resource showcases the potential for even greater performance gains by fine-tuning autoscaling mechanisms to suit specific workloads and application architectures.

5. CONCLUSION

In conclusion, this paper delves into the realm of horizontal scaling for PHP MVC applications, emphasizing the preservation of sessions to enhance both availability and resource efficiency. The exploration covers crucial theoretical aspects, spanning MVC architecture, PHP in web development, session handling intricacies, and the implementation of horizontal scaling techniques using load balancers, Docker, and Kubernetes. The practical methodology, encompassing environment setup, application development, Dockerization, Kubernetes integration, and horizontal scaling configuration, is detailed comprehensively. Performance testing highlights substantial improvements, showcasing a noteworthy reduction in response time—from an initially unresponsive state at 1000 RPS (5111 ms) to an impressive 32 ms at 2500 RPS with horizontal scaling. This study contributes valuable insights and practical guidance, offering a roadmap for the development of highly available and scalable web applications. Future work may explore further optimizations in the PHP MVC application's horizontal scaling configuration, aiming to uncover additional performance enhancements and refine resource allocation strategies. Additionally, investigating the integration of emerging technologies or alternative frameworks could provide valuable insights into continuously improving the scalability and responsiveness of web applications.

ACKNOWLEDGEMENTS

This study was supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, and these results are parts of the Grant No. 451-03-66 / 2024-03 / 200132 with University of Kragujevac - Faculty of Technical Sciences Čačak.

REFERENCES

[1] Majeed, A., Rauf, I. (2018). MVC Architecture: A Detailed Insight to the Modern Web Applications Development*, Crimson Publishers Wings to the Research Peer Review Journal of Solar & Photoenergy Systems*.

[2] Fowler, M. (2003). Patterns of Enterprise Application Architecture*, Pearson Education*.

[3] *PHP Programming – Cross Site Scripting Attacks*. Wikibooks. [Online]. Available: https://en.wikibooks.org/wiki/PHP_Programming/Cross_Site_Scripting_Attacks. Accessed: Sep. 6, 2023.

[4] Mihret, E. (2021). *PHP Sessions and Cookies - Sci-Tech with Estif,* DOI: 10.13140/RG.2.2.31128.52482.

[5] Millnert, V., Eker, J. (2020). *HoloScale: horizontal and vertical scaling of cloud resources*, IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 196-205, doi: 10.1109/UCC48980.2020.00038.

[6] Bondi, B. (2000) *Characteristics of scalability and their impact on performance*. WOSP '00, 195.

[7] Afzal S., Kavitha, G. (2019). Load balancing in cloud computing – A hierarchical taxonomical classification, *J Cloud Comp*, 8, 22, https://doi.org/10.1186/s13677-019-0146-7.

[8] Docker Documentation, [Online]. Available: https://docs.docker.com/. Accessed on: January 16th, 2024.

[9] Bilgin, I. (2019). Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. *O'Reilly Media.*

[10] Barzu, A. P., Barbulescu, M., Carabas, M. (2017), *Horizontal scalability towards server performance improvement*, 16th RoEduNet Conference: Networking in Education and Research (RoEduNet), Targu-Mures, Romania, 1-6, doi: 10.1109/ROEDUNET.2017.8123729.

[11] Casalicchio, E. (2019). A study on performance measures for auto-scaling CPU-intensive containerized applications, *Cluster Comput*, 22, 995–1006. doi: 10.1007/s10586-018-02890-1.