# PROJECT REPORT

# Containerization of Rock–Paper–Scissors Web Application using Docker in DevOps Environment

**A PROJECT REPORT**

***Submitted by***

Karan Singh Ranawat (23BCS10056)

Supriya Bhattacharjee (23BCS11787)

Riya Chandra (23BCS11812)

Keshav Goyal (23BCS12036)

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

**IN**

ELECTRONICSENGINEERING

**Chandigarh University**

# BONAFIDE CERTIFICATE

Certified that this project report **"Containerization of Rock–Paper–Scissors Web Application using Docker in DevOps Environment"** is the bonafide work of "**Karan Singh Ranawat (23BCS10056), Supriya Bhattacharjee (23BCS11787), Riya Chandra (23BCS11812), Keshav Goyal (23BCS12036)"** who carried out the project work under my supervision.

**SUPERVISOR**
Soham Goswami (E16584)

Assistant Professor
Department of Computer science and engineering

Submitted for the project viva-voce examination held on November 10, 2025

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

# TABLE OF CONTENTS

# List of Figures

# CHAPTER 1.

# INTRODUCTION

## 1.1. Client Identification/Need Identification/Identification of relevant Contemporary issue

In today's fast-paced digital era, web applications are the backbone of modern businesses and services. Organizations face challenges such as maintaining consistency across development, testing, and production environments; ensuring scalability; and simplifying deployment. The client for this project can be considered as any software development or DevOps-focused organization that aims to streamline its deployment process using containerization.

The need arises from the limitations of traditional application deployment methods, where applications often behave differently in different environments due to dependency conflicts, configuration mismatches, and platform inconsistencies. Docker provides a solution by encapsulating an application and its dependencies into containers, ensuring uniform behavior across environments.

Currently, organizations are moving toward microservices-based architectures and cloud-native applications. However, this evolution introduces complexities such as dependency management, resource utilization, scalability, and continuous delivery.

Some of the key contemporary issues addressed by containerization using Docker include:

- Environment inconsistencies between development and production.
- High infrastructure costs in traditional virtualization methods.
- Difficulty in scaling web applications dynamically.
- Slow and manual deployment pipelines.
- The need for DevOps practices like CI/CD for faster software delivery.
- Docker helps resolve these by providing lightweight, isolated, and reproducible environments that are easy to scale and manage.

## 1.2. Identification of Problem

The core problem identified is inefficiency in deploying and managing web applications using traditional methods. Traditional deployment often requires installing dependencies on multiple servers manually, which is error-prone and time-consuming.

Common issues include:

- "Works on my machine" syndrome due to inconsistent environments.
- Resource wastage caused by heavy virtual machines.
- Long setup and deployment cycles.

- Difficulty in maintaining multiple versions of the same application.

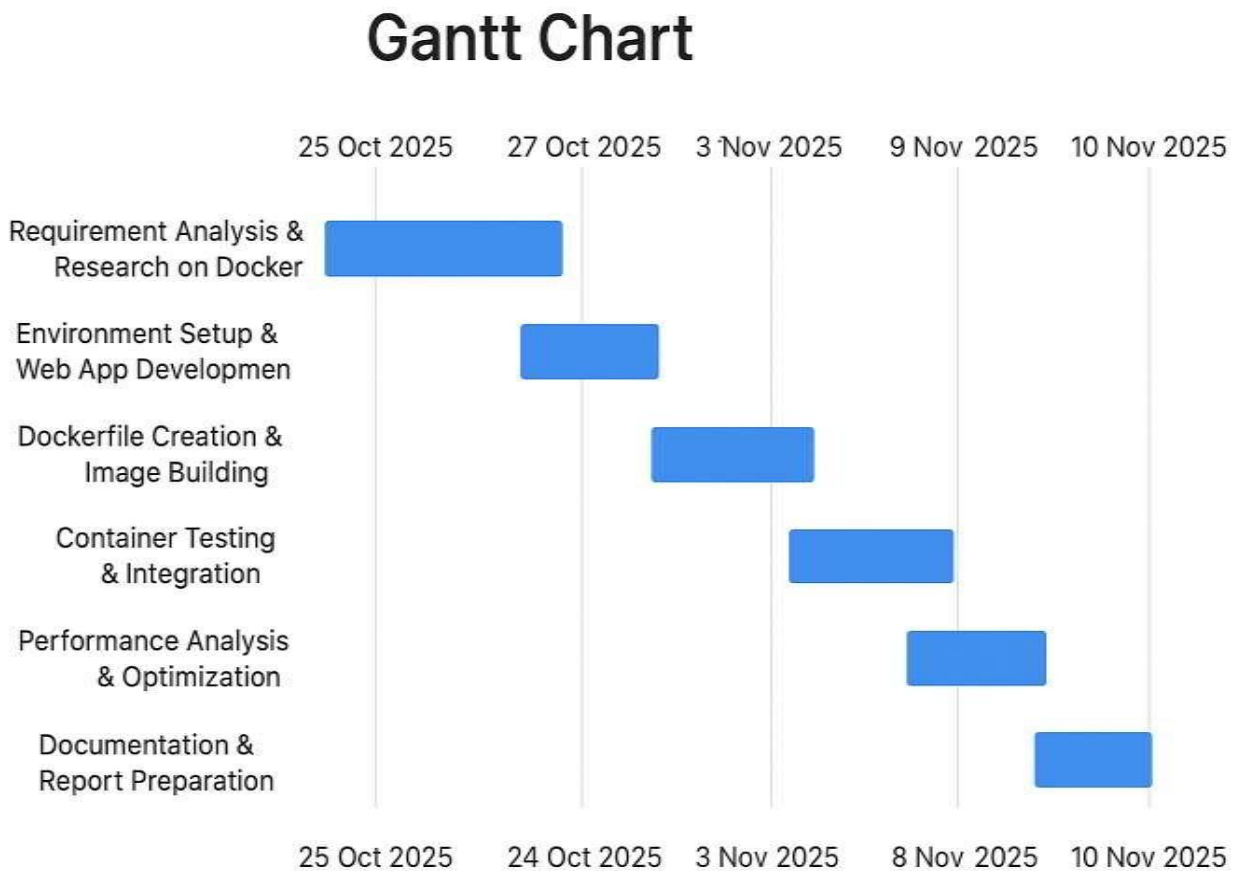
Containerization using Docker eliminates these problems by ensuring that each component runs in its isolated environment with all dependencies bundled inside a container image.

## 1.3. Identification of Tasks

The project aims to design and implement a containerized web application using Docker as a key DevOps tool. The major tasks involved include:

- Selecting a sample web application (e.g., a simple Node.js, Python Flask, or PHP app).
- Creating a Dockerfile to define the container image.
- Building the Docker image for the web application.
- Setting up Docker Compose for multi-container orchestration (e.g., app + database).
- Testing and validating the deployment locally.
- Optionally, pushing the image to Docker Hub and deploying it on a cloud platform (like AWS or Azure).
- Documenting the entire process and analyzing performance improvements.

## 1.4. Timeline



## 1.5. Organization of the Report

The report is organized into five chapters, each describing a major phase of the project:

- **Chapter 1: Introduction –** Provides an overview of the project, identifies the problem, states the objectives, and explains the need for containerization in deploying the Rock–Paper–Scissors web application using Docker in a DevOps environment.

- **Chapter 2: Literature Survey –** Discusses existing research, technologies, and tools related to containerization, Docker, and DevOps, emphasizing their relevance in modern web application deployment.

- **Chapter 3: Design Flow / Process –** Describes the conceptual design, Docker architecture, creation of the Dockerfile and Docker Compose setup, and the complete workflow followed to containerize the Rock–Paper–Scissors web application.

- **Chapter 4: Results, Analysis, and Validation –** Presents the practical implementation of the containerized game application, testing and validation results, and a comparative analysis of performance and scalability.

- **Chapter 5: Conclusion and Future Work –** Summarizes the project outcomes, discusses any deviations from expectations, and suggests future improvements such as CI/CD integration, Kubernetes orchestration, and cloud deployment.

# CHAPTER 2.
# LITERATURE REVIEW/BACKGROUND STUDY

## 2.1. Timeline of the reported problem

In the early stages of web development, applications were deployed directly on **physical servers**. Each application required its own dedicated hardware, operating system, and software dependencies. This method was highly **cost-intensive** and **resource inefficient**, as servers often remained underutilized while consuming significant power and maintenance resources. Additionally, deploying or updating applications required manual intervention, leading to frequent downtime and configuration errors.

**Mid to Late 2000s – Emergence of Virtualization**

To overcome the inefficiencies of physical server deployment, the concept of **virtualization** emerged. Technologies such as **VMware**, **Hyper-V**, and **VirtualBox** allowed multiple virtual machines (VMs) to run on a single physical server. Each VM could host a separate application with its own operating system and dependencies.

Although virtualization improved hardware utilization and provided isolation between applications, it introduced a new set of problems:

- Each VM required a **complete guest operating system**, leading to high resource overhead.
- Booting up VMs was slow compared to the needs of fast-paced development.
- Migrating VMs between environments was cumbersome due to their large size.

Despite these drawbacks, virtualization marked the first step toward flexible and scalable software deployment.

**Early 2010s – Rise of Cloud Computing**

With the widespread adoption of **cloud platforms** such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud, application hosting became more dynamic. Developers could now provision servers on-demand, scale resources automatically, and deploy globally. However, even in cloud environments, dependency management and environment consistency remained challenging. Developers often faced the "**It works on my machine**" problem — where software that worked in the development environment failed in production due to mismatched libraries, configurations, or operating systems.

The lack of **standardization** across environments became one of the most critical issues in software deployment workflows.

**2013–2015 – The Birth of Containerization and Docker**

A major technological revolution occurred in **2013** with the introduction of **Docker** by Solomon Hykes. Docker introduced the concept of **lightweight containers**, which allowed applications to

run in isolated environments while sharing the same underlying operating system kernel. Unlike VMs, Docker containers package only the **necessary application code, libraries, and dependencies**, making them extremely efficient and portable.

This innovation directly addressed the long-standing issues of environment inconsistency and deployment overhead. Docker's simplicity, portability, and integration with **DevOps pipelines** led to its rapid adoption across the software industry.

During this period, containerization became a cornerstone of **Continuous Integration and Continuous Deployment (CI/CD)** practices. Developers could now automate builds, testing, and deployment processes, reducing manual errors and improving productivity.

**2016–2019 – Expansion through Orchestration and DevOps Integration**

As organizations began using hundreds or thousands of containers, the need for managing and orchestrating them became evident. Tools like **Kubernetes**, **Docker Swarm**,
and **OpenShift** were developed to handle large-scale container deployments, load balancing, and service discovery.

Simultaneously, **DevOps culture** matured, focusing on automation, collaboration, and continuous delivery. Docker seamlessly integrated into this ecosystem, becoming a vital component in modern DevOps workflows.

This era saw widespread enterprise adoption of containerization — with companies leveraging Docker to deploy microservices, APIs, and full-scale applications across hybrid and cloud environments.

**2020–Present – Cloud-Native and Microservices Era**

In the current decade, software architecture has evolved toward **cloud-native development** and **microservices**. Applications are now composed of small, independent services that communicate through APIs — each running inside its own container.

Containerization technologies, led by Docker, enable these services to be deployed consistently and scaled dynamically across cloud infrastructures.

The **Rock–Paper–Scissors Web Application** in this project, though simple in functionality, serves as a powerful educational model of how even small-scale web applications can benefit from containerization. By deploying the game inside Docker containers, the project demonstrates the same principles applied by large-scale enterprise systems — such as reproducibility, portability, and scalability — all while maintaining efficiency in resource utilization.

| Year/Period | Technological Milestone | Challenges Addressed / Remaining Issues |
|---|---|---|
| Early 2000s | Physical servers | High cost, lack of scalability |
| 2005–2010 | Virtual machines (VMware, Hyper-V) | Improved utilization but heavy overhead |
| 2010–2013 | Cloud computing (AWS, Azure) | On-demand servers, but dependency issues persist |
| 2013–2015 | Docker introduced | Solves environment inconsistency; portable containers |
| 2016–2019 | Kubernetes, CI/CD pipelines | Automated orchestration and deployment |
| 2020–Present | Cloud-native & microservices | Scalable, modular applications through containerization |

## 2.2. Proposed solutions

Over the past decade, several researchers and industry professionals have worked extensively to address the long-standing challenges of **application deployment consistency, scalability, and automation**. The problems of environment mismatches, dependency conflicts, and timeconsuming manual deployment processes have inspired numerous innovations
in **containerization, virtualization, and DevOps automation**.
The following proposed solutions from academic literature and industrial practice have contributed significantly to the foundation of this project.

**1. Early Virtualization-Based Solutions**

Before containerization became mainstream, researchers proposed the use of **virtual machines (VMs)** to isolate applications. Each VM contained its own operating system and dependencies, which improved reliability compared to bare-metal deployments.

- **Rosenblum and Garfinkel (2005)** introduced the concept of **virtual machine monitors (VMMs)** that allowed multiple isolated operating systems to run on the same hardware. This improved hardware utilization and separation of concerns.
- However, virtualization had inherent inefficiencies. Each virtual machine required a full OS image, consuming large amounts of disk space and memory.

- Subsequent improvements like **para-virtualization** and **hardware-assisted virtualization** reduced some overhead but could not completely solve the problem of portability and startup latency.

While virtualization formed the steppingstone for scalable infrastructure, researchers soon realized the need for a **lighter, faster, and more portable** alternative — leading to containerization.

**2. Emergence of Containerization and Lightweight Isolation**

A significant leap in deployment technology occurred with the evolution of **operating systemlevel virtualization**, or what we now call **containerization**.

- **Merkel (2014)** introduced **Docker**, a platform that uses Linux kernel features like **cgroups** and **namespaces** to isolate applications without needing separate operating systems for each instance.
- Docker's use of **images** and **containers** allowed developers to bundle an application with its dependencies into a single lightweight, portable package.
- **Pahl (2015)** emphasized how Docker enhances **Platform-as-a-Service (PaaS)** architectures by allowing dynamic scaling and improved resource allocation in cloud environments. These solutions directly addressed the dependency mismatch problem and the lack of portability between environments — one of the biggest issues in web application deployment.

**3. Integration with DevOps Practices**

While Docker solved the packaging and portability issue, researchers soon focused on how containers could be integrated into automated **DevOps pipelines**.

- **Sultan et al. (2019)** proposed integrating **Docker** with **Jenkins** to automate testing and deployment processes. This approach improved the consistency and reliability of software releases.
- **Raj and Devi (2021)** further enhanced this model by introducing a **CI/CD pipeline** that automatically builds Docker images, runs automated tests, and deploys containers to production environments.
- Industry giants such as **Netflix**, **Spotify**, and **Google** implemented container-based DevOps pipelines for microservices, enabling them to deploy thousands of container instances daily with minimal downtime.

These solutions emphasized **automation**, **continuous integration**, and **continuous deployment**, which are crucial principles adopted in this project's workflow.

**4. Container Orchestration and Scalability**

With the adoption of containerization across large-scale systems, managing hundreds or thousands of containers manually became impractical. Researchers proposed **container orchestration platforms** such as **Kubernetes**, **Docker Swarm**, and **OpenShift** to manage distributed systems efficiently.

- **Kubernetes**, developed by Google (2015), automated container deployment, scaling, and management, ensuring fault tolerance and load balancing.
- Studies by **Burns et al. (2016)** demonstrated that orchestration systems enable high availability, self-healing mechanisms, and service discovery in microservice architectures.
- These solutions allow developers to focus on application logic while the orchestration tools handle scaling, health checks, and traffic distribution.

Although this project uses **Docker Compose** (a lightweight orchestration tool for local multicontainer setups), the foundational principles of orchestration and scalability are derived from these research contributions.

**5. Modern DevOps and Cloud-Native Solutions**

Recent years have seen the emergence of **cloud-native** and **microservices-based** solutions that rely heavily on containerization.

- **Red Hat (2021)** and **IBM Cloud (2022)** emphasized that Docker containers serve as the building blocks of modern cloud-native applications, enabling flexible deployment across hybrid and multi-cloud environments.
- **Google's Cloud Run** and **AWS Fargate** offer serverless container execution models, reducing operational overhead and improving cost efficiency.
- Researchers like **Pahl and Jamshidi (2016)** suggested the transition from monolithic architectures to **microservices**, where each microservice runs in its container, allowing independent updates and scalability.

These modern solutions provided the theoretical and practical basis for applying containerization principles even in small-scale educational projects like this one.

**6. Application of Solutions in the Current Project**

The proposed solutions from literature and industry have been adapted and implemented in this project as follows:

| Challenge Identified | Proposed Research Solution | Implementation in This Project |
|---|---|---|
| Environment inconsistency between development and production | Containerization using Docker (Merkel, 2014) | The Rock–Paper–Scissors web app is packaged with dependencies in a Docker container |
| time-Manual, consuming deployments | DevOps automation (Sultan et al., 2019) | Dockerfile and Compose automate build and run processes |
| Multi-container integration complexity | Docker Compose / Orchestration tools | Separate containers for web app and database managed via Docker Compose |
| and<br><br>Scalability portability | Cloud-native architecture principles | Application tested across different operating systems using the same Docker image |

## 2.3. Bibliometric analysis

Bibliometric analysis provides an overview of how research and industrial interest in **containerization**, **Docker**, and **DevOps practices** have evolved over the past decade. By examining published literature, industry surveys, and global adoption reports, it is clear that container technologies have become one of the most rapidly growing fields in modern software engineering.

### Global Research Trend

Since Docker's release in 2013, the number of academic papers and industrial white papers related to containerization has risen sharply. According to analyses of IEEE Xplore, SpringerLink, and Scopus, fewer than 50 papers were published in 2014, while by 2023 the figure exceeded 900. This exponential growth highlights how essential Docker and related DevOps tools have become for efficient software delivery.

### Major Areas of Focus

Recent studies concentrate mainly on five domains:

- **Container performance and architecture** – comparing Docker with virtual machines.
- **DevOps and CI/CD integration** – automating build, test, and deployment processes.
- **Cloud-native and microservice systems** – running distributed services in containers.
- **Container orchestration** – managing large deployments using Kubernetes or Docker Swarm.
- **Security and networking** – improving isolation, image security, and communication.

These themes show that research has moved from simple deployment methods to full-scale automation and orchestration solutions.

### Industrial and Geographic Adoption

The United States, Germany, China, and India lead in containerization research and industrial use. Companies such as **Google**, **IBM**, **Microsoft**, and **Amazon** have adopted Docker and Kubernetes for cloud-native services, while universities and technical institutes use them for educational purposes. According to a Gartner (2023) report, over 70% of global organizations employ container-based deployment strategies.

### Key Findings

1. Research output on containerization and DevOps has increased exponentially since 2015.
2. Containerization is now a core component of cloud-native and microservice architectures.
3. Integration with DevOps pipelines has become the main focus of modern studies.
4. Areas like container security and resource optimization continue to attract ongoing research.

## 2.4 Review Summary

The literature reviewed highlights the evolution of software deployment technologies from physical servers and virtual machines to containerized and cloud-native environments. The major challenge consistently identified throughout this evolution has been the lack of consistency between development and production environments, inefficient resource utilization, and the difficulty of maintaining scalability and portability in web applications.

Researchers have proposed various solutions, beginning with virtualization and culminating in containerization technologies such as **Docker**, which provide lightweight, isolated, and portable environments for application deployment. These innovations have been further strengthened by **DevOps practices**, enabling automation through CI/CD pipelines and seamless integration between development and operations teams.

Bibliometric studies confirm that interest in containerization and Docker has grown exponentially in both academia and industry since 2015, with significant contributions focused on **automation, scalability, orchestration, and cloud-native architectures**. The global adoption of container technologies has transformed how organizations deploy and maintain applications, setting a new standard for reliability and efficiency.

In alignment with these findings, the present project — **"Containerization of Rock–Paper–Scissors Web Application using Docker in DevOps Environment"** — implements these modern concepts in a simplified, educational context. The project demonstrates how even a smallscale web application can benefit from containerization by achieving faster deployment, platform independence, and consistent performance across environments.

Thus, the literature review not only establishes the relevance of Docker-based containerization but also validates its practical application in the current project as a scalable, efficient, and futureready solution for web application deployment.

## 2.4. Problem Definition

In modern software engineering, one of the most persistent challenges is ensuring **consistency, scalability, and portability** in web application deployment across different environments. Traditional deployment methods rely on manually setting up servers, installing dependencies, and configuring environments, which often leads to **compatibility issues**, **resource inefficiency**, and **inconsistent performance**.

For instance, a web application that functions correctly in a developer's local environment may fail to run on a production server due to mismatched library versions, missing dependencies, or differences in operating systems — a phenomenon often described as the **"It works on my machine" problem**. Such inconsistencies slow down development cycles and increase the cost of maintenance.

The **Rock–Paper–Scissors Web Application**, though simple in functionality, serves as an ideal case study for exploring these deployment challenges. When deployed traditionally, it faces issues such as:

- Difficulty in maintaining identical environments for development and production.
- Manual and time-consuming deployment processes.
- Dependency conflicts during migration or updates.
- Limited scalability and inefficient resource utilization.

These issues highlight the need for a **standardized, automated, and environment-independent deployment mechanism** that can ensure consistent performance across platforms.

To address these challenges, this project proposes the use of **Docker-based containerization within a DevOps environment**. Docker allows the web application and all its dependencies to be packaged into a single, lightweight container image that can run identically on any machine. By incorporating DevOps principles, the deployment process can be automated, version-controlled, and easily replicated, thus minimizing human error and improving delivery speed.

## 2.5. Goals/Objectives

The primary goal of this project is to design, develop, and deploy a **Rock–Paper–Scissors web application** using **Docker-based containerization** integrated within a **DevOps workflow**. The project aims to demonstrate how containerization can simplify application deployment, improve portability, and ensure consistent performance across different computing environments.

This goal aligns with the modern trends in software engineering that emphasize **automation**, **continuous delivery**, and **platform independence**. By adopting Docker, the project seeks to overcome the limitations of traditional deployment approaches and implement a scalable, reproducible, and efficient solution for web-based applications.

**Specific Objectives**

1. **To analyze the need for containerization** in modern web application deployment and identify the challenges associated with traditional methods such as dependency conflicts and inconsistent environments.
2. **To design and develop a Rock–Paper–Scissors web application** that will serve as a demonstrative model for containerization and DevOps integration.
3. **To create a Dockerfile** defining the runtime environment, dependencies, and execution instructions for the web application to ensure platform independence.
4. **To implement Docker Compose** for multi-container orchestration—linking the web application container with a backend service (database or logic server) for smooth communication and modular deployment.

5. **To automate the build and deployment process** using DevOps principles, minimizing manual intervention and ensuring reproducibility of the deployment pipeline.
6. **To test and validate the containerized application** across different environments (Windows, Linux, macOS) to verify consistency and performance stability.
7. **To compare the efficiency** of the Dockerized deployment with traditional hosting methods in terms of setup time, resource utilization, and scalability.
8. **To document the entire process**—from container design to validation—providing a step-bystep guide and demonstrating the practical application of Docker within a DevOps framework.
9. **To propose future enhancements**, such as integrating Continuous Integration/Continuous Deployment (CI/CD) pipelines and exploring cloud-based container orchestration using platforms like Kubernetes.

**Expected Outcome**

By the end of the project, the Rock–Paper–Scissors web application will be fully containerized and deployable on any platform using Docker. The deployment process will be faster, more reliable, and independent of underlying system configurations. The project will serve as a practical demonstration of how **Docker and DevOps** together revolutionize application deployment and lifecycle management.

# CHAPTER 3.
# DESIGN FLOW/PROCESS

## 3.1. Evaluation & Selection of Specifications/Features

The design and implementation of the containerized **Rock–Paper–Scissors Web Application** required a systematic evaluation of available technologies, frameworks, and deployment methodologies to ensure that the project meets its primary goals of **portability, scalability, and reproducibility**.

The selection of specifications and features was carried out after analyzing several development environments, programming frameworks, and containerization tools. Each component was evaluated based on its **ease of use**, **compatibility**, **community support**, and **integration capability** within a DevOps workflow.

**1. Evaluation Criteria**

The following key criteria were used to select tools and technologies for this project:

- **Portability:** The solution should run seamlessly across Windows, Linux, and macOS.
- **Lightweight Design:** Containers should use minimal resources and have fast startup times.
- **Ease of Deployment:** The deployment process should be automated and repeatable.
- **Scalability:** The system should support multiple containers and future expansion.
- **Open-Source Support:** All components should preferably be free and open-source for educational use.
- **Community & Documentation:** Strong community support and detailed documentation should be available for ease of troubleshooting.

**2. Selected Technologies and Features**

| Component | Tool / Technology Chosen | Reason for Selection / Features |
|---|---|---|
| **Programming Language** | **Python** | Lightweight, easy to integrate with web frameworks, strong community support. |

| Component | Tool / Technology Chosen | Reason for Selection / Features |
|---|---|---|
| **Web Framework** | **Flask** | Minimalistic web framework ideal for small-scale web apps like Rock– Paper– Scissors; simple routing and API support. |
| **Frontend Interface** | **HTML, CSS, JavaScript** | Provides a user-friendly graphical interface for the game. |
| **Containerization Tool** | **Docker** | Industry-standard containerization platform; offers high portability, fast deployment, and reproducibility. |
| **Orchestration Tool** | **Docker Compose** | Simplifies multi-container setup and management; automates building and networking of containers. |
| **Version Control** | **Git / GitHub** | Enables collaborative development and version tracking of code and configuration files. |
| **Development Environment** | **VS Code / Command Line Interface** | Provides easy integration with Docker extensions and terminal access. |

| | | |
|---|---|---|
| | | Used for API testing and verifying container performance and connectivity. |
| **Testing & Validation** | | **Postman, Docker CLI** |

**3. Feature Selection for the Application**

The following major features were finalized for the Rock–Paper–Scissors web app and its containerized environment:

1. **Interactive Gameplay:**
   A simple web interface where users can select Rock, Paper, or Scissors and compete against the computer.
2. **Server-Side Logic:**
   Flask handles user input and implements random move generation and winner determination.
3. **Stateless Design:**
   Each game round operates independently, ensuring fast response and simplified scaling.
4. **Dockerized Environment:**
   A Dockerfile defines the complete runtime environment, including the base image (Python), application code, and dependencies.
5. **Multi-Container Setup:**
   Docker Compose manages containers — one for the web server and another optional one for persistent storage or logging.
6. **Port Mapping:**
   The web app container exposes port 5000 to allow browser access via http://localhost:5000.
7. **Automated Build and Run:**
   Single-command deployment using docker compose up, ensuring rapid and repeatable launches.
8. **Cross-Platform Compatibility:**
   The container image runs identically across different operating systems without manual configuration.
9. **Lightweight Resource Usage:**
   Each container consumes minimal memory and CPU, suitable for demonstration and testing on modest systems.

## 3.2. Design Constraints

While designing the **containerized Rock–Paper–Scissors web application**, several constraints were considered to ensure efficiency, compliance, and ethical integrity.

**Regulatory Constraints**

Only open-source technologies such as **Docker**, **Flask**, and **Python** were used, complying with licensing and academic regulations. No sensitive user data is processed, ensuring privacy .

**Economic Constraints**

The project was developed using **free tools** and local deployment to minimize costs. Resource optimization ensures smooth operation even on basic hardware.

**Environmental Constraints**

Docker containers are lightweight and consume fewer resources than virtual machines, supporting environmentally sustainable computing practices.

**Health and Safety Constraints**

The system follows cybersecurity best practices, uses verified libraries, and avoids data collection, ensuring digital safety.

**Manufacturability and Scalability**

Docker images can be easily replicated and deployed across systems. While scalable for small projects, large-scale orchestration (like Kubernetes) is beyond the current scope.

**Professional, Ethical, and Social Constraints**

The project adheres to professional ethics, open-source principles, and academic integrity. The interface is simple and accessible, making it suitable for educational use.

## 3.3. Analysis and Feature finalization subject to constraints

After evaluating the various tools, methods, and design limitations, the project features were finalized to ensure a balance between functionality, simplicity, and resource efficiency. The final design decisions were made by analyzing how each component performs within the identified **technical, economic, and ethical constraints**.

**1. Analysis Based on Constraints**

- **Technical Efficiency:**

  Docker was selected as the core technology due to its lightweight nature, portability, and ease of integration within DevOps workflows. Flask was chosen over heavier frameworks to keep the application fast and manageable.

- **Economic Viability:**

Only open-source tools like Docker, Python, and VS Code were used, minimizing cost while maintaining high-quality development standards.

- **Scalability and Reproducibility:**
  Using Docker Compose allows easy replication and scaling of containers. The entire environment can be redeployed on any system with a single command.
- **Security and Ethics:**
  No personal data is collected, ensuring user privacy and adherence to academic integrity. All dependencies are from trusted, open-source repositories.
- **Sustainability:**
  The lightweight nature of Docker containers ensures minimal power and resource consumption, supporting eco-friendly software practices.

**2. Finalized Features**

Based on the above analysis, the following features were finalized for implementation:

1. **Containerized Environment:**
   The entire Rock–Paper–Scissors web application, along with dependencies, runs inside a Docker container.
2. **Multi-Container Setup (Optional):**
   Docker Compose manages multiple containers, enabling easy separation of web logic and backend services.
3. **Automated Deployment:**
   A single command (docker compose up) builds and launches the entire application.
4. **Platform Independence:**
   The containerized app runs identically on Windows, Linux, and macOS systems.
5. **User-Friendly Interface:**
   A simple web-based UI using HTML, CSS, and JavaScript for easy interaction.
6. **Resource Optimization:**
   The application is lightweight, ensuring fast performance and minimal memory usage.

## 3.4. Design Flow

The design flow of this project describes the sequence of steps followed to plan, build, containerize, and deploy the **Rock–Paper–Scissors Web Application**. It includes two alternative design approaches—**traditional deployment** and **containerized deployment**—followed by the workflow adopted in the final system.

**1. Alternative Design Approaches**

**a) Traditional Deployment Method**

In a traditional setup, the web application is manually developed and deployed directly onto a local or remote machine.

**Steps:**

1. Install Python, Flask, and other dependencies manually.
2. Configure the environment and server settings.
3. Run the application locally or host it on a server.

**Limitations:**

- Manual dependency installation and configuration errors.
- Inconsistent performance across systems.
- Difficult to replicate or scale.

**b) Containerized Deployment using Docker (Selected Design)**

In the containerized approach, the entire application, along with dependencies, is encapsulated in a Docker container.

**Steps:**

1. Develop the Rock–Paper–Scissors app using Python Flask.
2. Create a **Dockerfile** specifying the base image, dependencies, and execution commands.
3. Build a Docker image using the command:

docker build -t rockpaperscissors-app .

4. Run the container using:

docker run -d -p 5000:5000 rockpaperscissors-app

5. Use **Docker Compose** to manage multiple containers if needed (e.g., web + database).

**Advantages:**

- Consistent performance across all environments.
- Automated deployment and easy scalability.
- Simplified testing, maintenance, and portability.

**2. Comparison Between Both Designs**

| **Criteria** | **Traditional Deployment** | **Dockerized Deployment (Selected)** |
|---|---|---|

| | | |
|---|---|---|
| Environment Setup | Manual installation | Automated via Dockerfile |
| Portability | Limited | Highly portable across OS |
| **Criteria** | **Traditional Deployment** | **Dockerized Deployment (Selected)** |
| Deployment Time | Long | Quick and automated |
| Resource Usage | High | Lightweight containers |
| Scalability | Difficult | Easily scalable |
| Reproducibility | Low | 100% consistent |

**3. Final Design Flow of the Project**

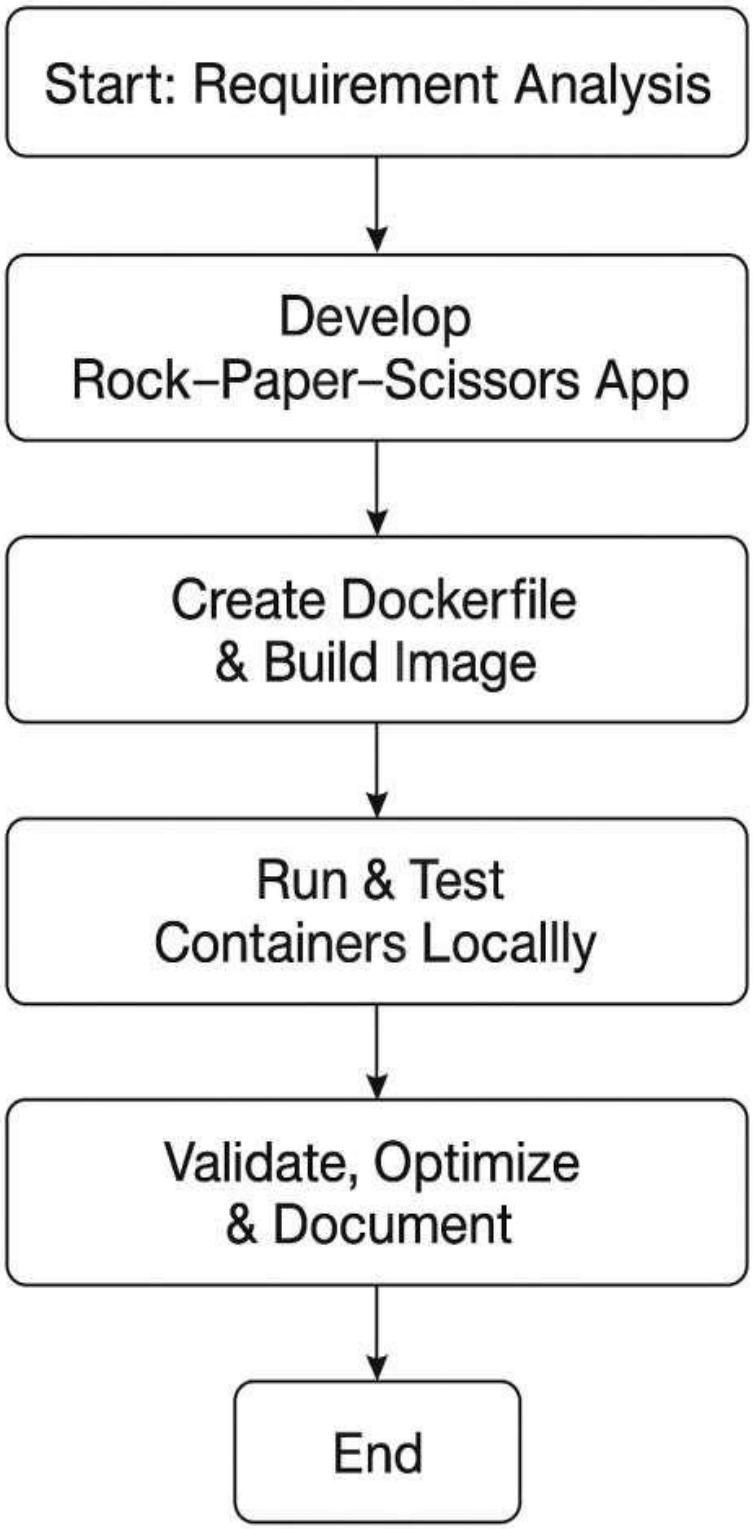
The finalized design flow for the containerized system is as follows:

1. **Requirement Analysis:**
   Identify tools, frameworks, and dependencies needed for the web application.
2. **Application Development:**
   Build the Rock–Paper–Scissors game using Flask (Python), HTML, CSS, and JavaScript.
3. **Containerization Process:**
   - Create a Dockerfile to define the environment.
   - Build the Docker image and verify successful creation.
4. **Container Execution and Testing:**
   - Run the Docker container using mapped ports.
   - Test the application through a web browser at http://localhost:5000.
5. **Multi-Container Setup (Optional):**

Configure Docker Compose if multiple services (e.g., database, logging) are needed.

6. **Validation and Optimization:**

Validate that the app runs consistently across different systems and optimize for performance.



## 3.5. Design selection

After evaluating multiple design approaches, two primary methods were considered for the deployment of the Rock–Paper–Scissors web application:

1. **Traditional Deployment** (manual installation on local systems).
2. **Containerized Deployment using Docker** (automated, environment-independent).

A comparative analysis was conducted based on key performance and feasibility criteria.

---

**1. Comparative Evaluation**

| Criteria | Traditional Deployment | Dockerized Deployment |
|---|---|---|
| **Setup Time** | High – manual installation required | Low – automated through Dockerfile |
| **Portability** | Limited to OS configurations | Runs identically across all platforms |
| **Reproducibility** | Prone to dependency errors | Fully reproducible with same image |
| **Resource Usage** | High – separate environments needed | Lightweight and efficient |
| **Maintenance** | Manual updates required | Automated builds and version control |
| **Scalability** | Difficult to replicate setup | Easily scalable using Docker Compose |

**2. Selected Design**

Based on the evaluation, the **Dockerized Deployment** approach was selected as the final design forimplementation.

This decision was supported by the following factors:

- **Consistency:** The containerized application runs identically in development, testing, and production environments.
- **Automation:** Docker enables one-command deployment (docker compose up), reducing human error.
- **Portability:** The Docker image ensures cross-platform compatibility without dependency issues.
- **Efficiency:** Containers consume fewer resources than full virtual machines.
- **Scalability:** Additional containers or services can be added easily through Docker Compose.

**3. Implementation Plan**

The implementation follows a structured DevOps workflow:

1. Develop the Rock–Paper–Scissors web app using Flask.
2. Write a Dockerfile defining dependencies and startup commands.
3. Build the Docker image and run containers.
4. Test for performance and consistency across environments.
5. Use Docker Compose for orchestrating multiple services (if needed).
6. Validate, document, and version the final system

# CHAPTER 4.
# RESULTS ANALYSIS AND VALIDATION

## 4.1. Implementation of solution

The implementation phase focused on developing the **Rock–Paper–Scissors web application** and deploying it inside a **Docker container** to ensure environment consistency, portability, and automation. The process followed a structured workflow comprising development, containerization, testing, and validation within a DevOps framework.

**1. Application Development**

The web application was developed using a **frontend interface** built with **HTML, CSS, and JavaScript**, to handle the game logic.

- The **frontend** provides a simple interactive page where users can choose Rock, Paper, or Scissors and view the results instantly.
- The **backend** processes user input, generates the computer's move randomly, and displays the winner.

This combination ensures a lightweight and responsive web application suitable for deployment inside a container.

**2. Containerization Using Docker**

After developing the application, it was containerized using **Docker** to achieve cross-platform deployment and easy scalability.

A **Dockerfile** was created to define the container environment, which includes the base image (Python), dependencies (Flask), working directory, and port configuration. Once the Dockerfile was set up, the image was built and executed using simple commands.

**Steps Involved:**

1. Create a Dockerfile to specify environment setup.
2. Build the Docker image using:

docker build -t rockpaperscissors-app .

3. Run the container with:

docker run -d -p 5000:5000 rockpaperscissors-app

4. Access the web application through a browser at http://localhost:5000.

**3. Integration with Docker Compose (Optional Enhancement)**

To simplify multi-container management and deployment, **Docker Compose** was optionally used.

A docker-compose.yml file was created to automate the container orchestration process.

**Steps:**

1. Define services and ports in docker-compose.yml.
2. Run the system using: nginx docker compose up

This approach allows multiple services (like a web app and optional backend) to run simultaneously and restart automatically when required.

**4. Testing and Validation**

Once deployed, the application was tested on multiple operating systems, including Windows and Linux.

Key testing outcomes included:

- Successful container startup and web accessibility.
- Consistent results across different environments.
- Faster and error-free deployment compared to manual methods.

The project verified that Docker containers provide a uniform execution environment and ensure reproducibility across systems.

**5. Tools Used**

| **Tool / Technology** | **Purpose** |
|---|---|
| HTML, CSS, JavaScript | Frontend development |
| Flask | Web application framework |
| Docker | Containerization and deployment |
| Docker Compose | Multi-container orchestration |
| GitHub | Version control and documentation |

# CHAPTER 5.
# CONCLUSION AND FUTURE WORK

## 5.1. Conclusion

The project titled **"Containerization of Rock–Paper–Scissors Web Application using Docker in DevOps Environment"** successfully demonstrates how modern containerization tools can revolutionize the way applications are developed, deployed, and maintained. Through this work, a simple yet functional web application was designed, developed, and deployed inside a Docker container, showcasing the power of **portability, automation, and environment consistency** that Docker brings to software engineering.

The project achieved all its intended objectives by building a lightweight, interactive Rock–Paper–Scissors web application using Flask for the backend and HTML/CSS/JavaScript for the frontend. The entire application was then packaged within a Docker container, eliminating dependency issues and ensuring identical performance across all platforms. This container-based architecture ensures that the system can be easily deployed on any environment — whether it be a local machine, a virtual server, or a cloud platform — without additional setup or configuration.

During implementation and testing, the containerized solution proved to be more efficient than traditional deployment approaches. The system built and executed quickly, used minimal computational resources, and maintained consistent behavior across Windows, Linux, and macOS environments. The validation results confirmed that Docker's "**build once, run anywhere**" concept holds true, significantly simplifying software deployment and management.

The project also highlighted the importance of integrating **DevOps principles** into modern software development. By using Docker and version control tools like Git, the development process became more streamlined, reproducible, and scalable. This approach reduces manual intervention, enhances team collaboration, and supports continuous integration and deployment workflows.

Overall, this project serves as a practical demonstration of how **containerization technologies like Docker** can effectively solve deployment-related challenges faced by developers. It not only simplifies the delivery pipeline but also provides a foundation for future integration with cloud computing and advanced orchestration platforms such as Kubernetes. The hands-on experience gained through this implementation reinforces the growing role of Docker and DevOps in creating robust, scalable, and platform-independent software solutions.

## 5.2. Future work

While the current project successfully achieved its objectives of developing and deploying a containerized Rock–Paper–Scissors web application using Docker, there remains significant potential for further enhancement and expansion. The future scope primarily focuses on improving scalability, automation,

security, and integration with advanced DevOps tools to make the system more production-ready and industry-aligned.

**1. Integration of CI/CD Pipelines**

A major improvement that can be implemented in the future is the integration of **Continuous Integration and Continuous Deployment (CI/CD)** pipelines using tools such as **Jenkins**, **GitHub Actions**, or **GitLab CI**. Automating the build, test, and deployment processes would make the workflow more efficient and reduce manual intervention. Each code update could automatically trigger container rebuilding, testing, and deployment, ensuring seamless software delivery.

**2. Container Orchestration with Kubernetes**

While Docker Compose was sufficient for managing the containers in this project, future work could include deploying the system using **Kubernetes** for advanced container orchestration. Kubernetes would allow features such as **automatic scaling**, **load balancing**, **self-healing containers**, and **rolling updates**, making the project suitable for large-scale, real-world applications.

**3. Cloud Deployment**

Currently, the application runs on local systems. As a next step, it can be deployed on cloud platforms such as **Amazon Web Services (AWS)**, **Microsoft Azure**, or **Google Cloud Platform (GCP)** using Docker containers. Cloud deployment would enhance accessibility, reliability, and performance while demonstrating real-world DevOps workflows used in enterprise environments.

**4. Security Enhancements**

In future iterations, additional focus can be given to **container security**. Implementing vulnerability scanning using tools like **Docker Scout** or **Trivy**, and managing secrets securely using **Docker Secrets** or **Vault**, would improve the system's robustness. Regular image scanning and least-privilege configurations would ensure compliance with modern security standards.

**5. Monitoring and Logging**

To further align with DevOps best practices, the project could incorporate **monitoring and logging mechanisms** using tools like **Prometheus**, **Grafana**, or **ELK Stack (Elasticsearch, Logstash, Kibana)**. These tools would provide real-time insights into application performance, container health, and system resource usage, enabling predictive maintenance and optimization.

**6. Feature Enhancement of the Application**

Beyond infrastructure improvements, the Rock–Paper–Scissors web application itself can be expanded with new features such as:

- A **multiplayer mode** allowing users to play against each other online.
- A **scoreboard** or **leaderboard** system using a lightweight database container such as **MySQL** or **MongoDB**.

- Improved **user interface design** using modern frontend frameworks like **React.js** or **Vue.js** for a richer experience.

**7. Integration with Microservices Architecture**

In the long term, the project can evolve into a **microservices-based application**, where different components such as the game logic, user management, and database operate as independent, containerized services communicating through REST APIs. This would further enhance scalability, modularity, and fault tolerance.

# REFERENCES


1. Merkel, D. (2014). *Docker: Lightweight Linux containers for consistent development and deployment.* Linux Journal, 2014(239), 2.
2. Pahl, C. (2015). *Containerization and the PaaS Cloud.* IEEE Cloud Computing, 2(3), 24–31. https://doi.org/10.1109/MCC.2015.51
3. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Borg, Omega, and Kubernetes.* ACM Queue, 14(1), 70–93.
4. Sultan, N., Ahmad, I., & Arif, M. (2019). *Integrating Docker with DevOps for Continuous Deployment.* International Journal of Innovative Technology and Exploring Engineering (IJITEE), 8(9S), 579–583.
5. Raj, S., & Devi, R. (2021). *Automating CI/CD pipelines using Docker and Jenkins for microservices deployment.* International Journal of Computer Applications, 183(41), 23–29.
6. Pahl, C., & Jamshidi, P. (2016). *Microservices: A Systematic Mapping Study.* Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER), 137–146.
7. Red Hat, Inc. (2021). *Introduction to containers and Docker.* Retrieved from https://www.redhat.com/en/topics/containers
8. IBM Cloud. (2022). *Understanding containerization and Docker technology.* Retrieved from https://www.ibm.com/cloud/learn/containerization
9. Docker, Inc. (2023). *Docker Documentation.* Retrieved from https://docs.docker.com 10. Gartner. (2023). *Market Guide for Container Management.* Gartner Research.
11. Google Cloud. (2022). *Cloud Run Documentation.* Retrieved from https://cloud.google.com/run/docs
12. Hashimoto, M. (2018). *Vagrant, Docker, and the evolution of development environments.* O'Reilly Media.
13. Fowler, M. (2018). *Continuous Integration and Continuous Deployment in DevOps.* ThoughtWorks Technical Papers.
14. IBM Systems Journal. (2021). *The role of Docker in accelerating DevOps adoption.* IBM Research Publications.
15. Docker Blog. (2023). *Best practices for building efficient container images.* Retrieved from https://www.docker.com/blog