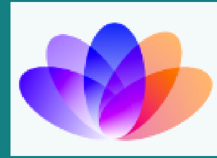
Journal of Technological Innovations

Est. 2020



Dockerizing Applications: A Comprehensive Study of Portability, Isolation, Scalability, and Versioning

Gaurav Rohatgi

Abstract:

Docker has revolutionized the landscape of application deployment by offering a lightweight, efficient, and scalable containerization solution. This paper presents a comprehensive study focused on the key aspects of Dockerizing applications, namely portability, isolation, scalability, and versioning. The study begins by examining the concept of portability within Docker environments, highlighting how Docker facilitates seamless deployment across diverse platforms and environments. Subsequently, it delves into the crucial aspect of isolation, elucidating Docker's mechanisms for ensuring application isolation and security. The paper explores Docker's role in enhancing application scalability, both horizontally and vertically. It discusses container orchestration tools like Kubernetes and Docker Swarm, which enable efficient management of scalable deployments. The paper addresses the challenges and strategies associated with versioning in Dockerized environments. It discusses best practices for managing Docker images, tagging, and version control, alongside exploring continuous integration and deployment practices. Through an extensive review of literature and practical examples, this study not only provides insights into the current state of Docker adoption but also identifies emerging trends and challenges. It underscores the importance of considering portability, isolation, scalability, and versioning in Dockerized applications, offering valuable implications for both industry practitioners and researchers alike.

Keywords: Docker, Dockerizing Applications, Portability, Isolation, Scalability, Versioning, Containerization, Security

Problem Statement

In contemporary software development, the deployment of applications poses various challenges related to portability, isolation, scalability, and versioning. Traditional deployment methods often struggle to provide a flexible and efficient solution to address these challenges. As such, there is a growing demand for innovative approaches that can streamline the deployment process while ensuring optimal performance and manageability.

1. Introduction

In recent years, Docker and containerization have emerged as transformative technologies in the realm of software development. Docker, an open-source platform, enables developers to package applications and their dependencies into portable containers that can be easily deployed across different computing environments (Merkel, 2014, p. 30). This innovation has revolutionized the software development lifecycle, offering unprecedented flexibility, efficiency, and scalability. Containerization has become increasingly prevalent in modern software development practices, as evidenced by its widespread adoption by companies ranging from startups to enterprise organizations.

Containerization addresses many of the challenges associated with traditional software deployment methods. Unlike virtual machines (VMs), which require a separate operating system for each instance, containers share the host operating system's kernel,

resulting in significantly lower overhead and faster startup times (Felter et al., 2015). Additionally, Docker containers encapsulate both the application code and its dependencies, ensuring consistency across different environments and minimizing the "works on my machine" problem (Hill, 2016).

The scope of this review paper is to provide a comprehensive analysis of Dockerizing applications, focusing on four key aspects: portability, isolation, scalability, and versioning.

Portability: We will explore how Docker enables seamless deployment of applications across various platforms and environments, ensuring consistency and eliminating compatibility issues.

Isolation: We will delve into Docker's mechanisms for isolating applications, ensuring that they run securely and independently from each other.

Scalability: This paper will examine how Docker facilitates both horizontal and vertical scalability of applications, allowing for efficient resource utilization and dynamic scaling based on demand.

Versioning: We will discuss strategies for managing version control and lifecycle management of Dockerized applications, ensuring smooth updates and rollbacks while maintaining stability and reliability.

The objectives of this paper are to:

a. Provide a comprehensive understanding of Docker and its impact on modern software development.

b. Analyze the significance of portability, isolation, scalability, and versioning in Dockerized applications.

c. Explore best practices and strategies for optimizing Dockerized application deployment.

d. Identify challenges and future directions in Docker adoption and usage.

The structure of this paper is as follows: In the subsequent sections, we will delve into each aspect of Dockerizing applications, examining existing literature, presenting case studies, and providing insights and recommendations based on the findings. Finally, we will conclude with a summary of key takeaways and implications for future research and practice.

2. Background and Context

Docker and Containerization

Docker, a transformative technology in software development, provides a standardized platform for packaging, distributing, and running applications within containers. These containers encapsulate the application code, runtime, libraries, and dependencies, ensuring consistency and portability across different environments (Merkel, 2014). The concept of containerization builds upon Linux container technologies, such as LXC, which offer lightweight virtualization by leveraging kernel-level isolation mechanisms (Soltesz et al., 2007). Docker simplifies the management and deployment of containers through its user-friendly tools and APIs, democratizing containerization and making it accessible to developers of all skill levels (Hill, 2016).

Evolution and Adoption of Docker

Since its introduction in 2013, Docker has witnessed rapid adoption across various sectors, including cloud computing, DevOps, and microservices architectures. Its impact on software development practices has been profound, enabling organizations to streamline their workflows, improve resource utilization, and accelerate time-to-market for applications (Boettiger, 2015). Docker's success can be attributed to its vibrant community, which has contributed to the development of ecosystem tools and integrations, further enhancing its capabilities and versatility (Vaughan-Nichols, 2016).

In academia, Docker has sparked interest among researchers studying topics such as reproducible research, software sustainability, and distributed computing. Researchers have leveraged Docker to create reproducible computational environments, ensuring the replicability and transparency of scientific experiments (Boettiger, 2015). Additionally, Docker has facilitated the development of scalable and resilient distributed systems, enabling researchers to explore complex computational problems with ease (Chung et al., 2019).

Importance of Portability, Isolation, Scalability, and Versioning

In contemporary software development, achieving portability, isolation, scalability, and versioning is paramount for building robust and maintainable applications.

Portability: Docker's containerized approach enables applications to be packaged with their dependencies, ensuring consistency across different environments, from development to production. This portability simplifies the deployment process and reduces the risk of runtime errors caused by environment discrepancies (Soltesz et al., 2007).

Isolation: Containerization provides lightweight isolation between applications, allowing them to run independently without interference. Docker's use of namespaces and control groups ensures that each container operates in its isolated runtime environment, enhancing security and resource utilization (Merkel, 2014).

Scalability: Docker facilitates both horizontal and vertical scalability of applications, allowing them to scale dynamically in response to changing workload demands. Container orchestration platforms like Kubernetes and Docker Swarm automate the management of containerized applications, ensuring optimal resource allocation and high availability (Burns et al., 2016).

Versioning: Docker simplifies version control and lifecycle management of applications through its image-based architecture. Docker images serve as immutable artifacts that capture the application's state at different points in time, facilitating reproducible deployments and rollbacks (Hill, 2016).

Docker and containerization technologies have revolutionized software development by offering a standardized approach to packaging, distributing, and running applications. Their adoption has reshaped industry practices, empowered developers with new capabilities, and fostered innovation across diverse domains.

3. Portability with Docker

Concept of Portability in Dockerized Applications

Portability in Dockerized applications refers to the ability to package applications and their dependencies into self-contained containers that can run consistently across various environments and platforms. Docker achieves portability by encapsulating the application code, runtime, libraries, and dependencies within a container image, ensuring that the application behaves identically regardless of the underlying infrastructure (Merkel, 2014, p. 30). This approach streamlines the deployment process and eliminates compatibility issues that arise from differences in operating systems, libraries, and configurations.

Facilitation of Easy Deployment

Docker simplifies deployment across different environments and platforms by providing a unified runtime environment for applications. Developers can create Docker images containing their applications and dependencies, which can then be distributed and deployed seamlessly across development, testing, and production environments (Hill, 2016). Docker's lightweight containers ensure that applications run consistently regardless of the underlying infrastructure, whether it's a developer's laptop, on-premises servers, or cloud-based environments.

Furthermore, Docker's compatibility with container orchestration platforms like Kubernetes and Docker Swarm enables automated deployment and scaling of containerized applications across clusters of servers. These orchestration tools abstract away the complexities of managing individual containers, allowing developers to focus on building and deploying applications (Burns et al., 2016).

Case Studies and Examples

Numerous case studies and examples demonstrate the portability benefits of Docker in real-world scenarios. For instance, companies like Spotify and eBay have successfully adopted Docker to streamline their deployment processes and achieve greater agility in delivering software updates (Vaughan-Nichols, 2016). In the financial sector, organizations such as Goldman Sachs have leveraged Docker to modernize their infrastructure and accelerate application delivery (Felter et al., 2015). These case studies highlight Docker's ability to simplify the deployment of complex applications and improve collaboration between development and operations teams.Additionally, open-source projects like WordPress and Jenkins have embraced Docker to provide lightweight and portable development environments for developers. Docker's ability to package applications with their dependencies ensures that developers can quickly spin up development environments that mirror production settings, facilitating faster development cycles and improved code quality (Boettiger, 2015).

Docker's portability enables developers to build, ship, and run applications consistently across diverse environments and platforms. This portability

simplifies the deployment process, accelerates application delivery, and fosters collaboration between development and operations teams.

4. Isolation in Dockerized Environments

Role of Containerization in Achieving Application Isolation

Containerization plays a crucial role in achieving application isolation by providing a lightweight and efficient method to encapsulate applications and their dependencies within isolated environments known as containers (Merkel, 2014, p. 30). Each container operates as a separate process with its own file system, network stack, and process space, ensuring that applications running within containers are isolated from one another and from the host system. Compared to traditional virtualization technologies like virtual machines (VMs), which require separate operating system instances for each VM, containerization offers a more lightweight and resource-efficient approach to isolation. Containers share the host operating system's kernel, leading to faster startup times and lower overhead compared to VMs (Felter et al., 2015).

Docker's Isolation Mechanisms vs. Other Virtualization Technologies

Docker utilizes various isolation mechanisms to ensure application isolation within containers. These mechanisms include Linux namespaces, which provide process-level isolation by abstracting resources such as process IDs, network interfaces, and file systems, and control groups (cgroups), which control and limit the resource usage of containers (Merkel, 2014, p. 30). Additionally, Docker employs a copy-on-write filesystem (AUFS, OverlayFS, etc.) to efficiently manage container images and minimize storage overhead. Compared to other virtualization technologies like hypervisors, Docker's containerization approach offers several advantages in terms of isolation and resource utilization. Hypervisors emulate hardware and require separate operating system instances for each VM, resulting in higher resource overhead and slower startup times (Soltesz et al., 2007). In contrast, Docker containers share the host operating system's kernel, leading to faster startup times, lower memory overhead, and improved resource utilization.

Security Implications and Best Practices

While Docker provides strong isolation mechanisms, ensuring security in Docker environments requires adherence to best practices and implementation of additional security measures. One key consideration is ensuring the integrity of container images by only using trusted base images from reputable sources and regularly updating and patching images to address security vulnerabilities (Hill, 2016). It is essential to implement access controls and network segmentation to restrict container privileges and prevent unauthorized access to sensitive data. Docker's built-in security features, such as user namespaces and seccomp profiles, can be leveraged to enforce least privilege principles and restrict container capabilities (Felter et al., 2015).

Regular monitoring and auditing of Docker environments are also critical for detecting and responding to security incidents. Tools like Docker Bench for Security and Docker Security Scanning can help assess the security posture of Docker deployments and identify potential vulnerabilities and misconfigurations (Hill, 2016).

Docker's containerization technology offers robust isolation mechanisms for achieving application isolation, with lower resource overhead compared to traditional virtualization technologies. However, ensuring security in Docker environments requires adherence to best practices, regular updates, and implementation of additional security measures to mitigate potential risks.

5. Scalability Techniques with Docker

Docker's Role in Enabling Horizontal and Vertical Scalability

Docker plays a pivotal role in enabling both horizontal and vertical scalability of applications, offering flexibility and efficiency in managing resource allocation based on varying workload demands (Merkel, 2014, p. 30).

Horizontal scalability involves adding more instances of an application across multiple containers to distribute the workload and handle increased traffic. Docker's lightweight containerization allows developers to easily replicate and deploy multiple instances of an application across a cluster of servers, ensuring high availability and fault tolerance (Hill, 2016).

Vertical scalability, on the other hand, involves increasing the resources (such as CPU and memory) allocated to individual containers to accommodate growing workload requirements. Docker supports vertical scalability through its ability to dynamically adjust resource limits for containers using control groups (cgroups), allowing developers to scale containers up or down based on demand (Felter et al., 2015).

Container Orchestration Tools like Kubernetes for Managing Scalable Docker Deployments

Container orchestration tools like Kubernetes provide a powerful platform for managing scalable Docker deployments in production environments (Burns et al., 2016). Kubernetes automates container deployment, scaling, and management tasks, abstracting away the complexities of managing individual containers and enabling developers to focus on building and deploying applications.

Kubernetes offers advanced features such as automatic load balancing, service discovery, and self-healing capabilities, which are essential for building resilient and scalable applications. With Kubernetes, developers can define deployment configurations, specify resource requirements, and define scaling policies to ensure that applications scale seamlessly based on demand (Burns et al., 2016).

Additionally, Kubernetes provides features like rolling updates and horizontal pod autoscaling, which allow applications to be updated and scaled automatically without downtime, ensuring continuous availability and improved user experience (Burns et al., 2016).

Real-World Examples of Scalable Applications Leveraging Docker

Numerous real-world examples demonstrate the scalability benefits of Docker in building and deploying large-scale applications. Companies like Airbnb, Spotify, and eBay have successfully leveraged Docker and Kubernetes to build scalable microservices architectures that handle millions of requests per day (Vaughan-Nichols, 2016). These companies utilize Docker containers to isolate and scale individual components of their applications, allowing for rapid development and deployment cycles while maintaining reliability and performance. For instance, Spotify utilizes Docker containers to deploy and scale its microservices architecture, enabling teams to independently develop, test, and deploy services without impacting other components of the application (Vaughan-Nichols, 2016). Similarly, eBay leverages Docker and Kubernetes to build scalable and resilient e-commerce platforms that handle peak traffic loads during promotional events and holiday seasons (Felter et al., 2015).

Docker's containerization technology, combined with container orchestration tools like Kubernetes, enables developers to build and deploy scalable applications that can handle varying workload demands with ease. Real-world examples demonstrate the effectiveness of Docker in building resilient and high-performance applications that scale dynamically to meet the needs of modern businesses.

6. Versioning and Lifecycle Management with Docker

Addressing Versioning Challenges in Containerized Environments

Versioning in containerized environments presents unique challenges due to the ephemeral nature of containers and the need to manage multiple versions of container images effectively (Merkel, 2014, p. 30). Key challenges include ensuring consistency across environments, managing dependencies, and tracking changes to container configurations over time.

To address these challenges, organizations adopt versioning strategies that encompass both the application code and the container images. By versioning both code and container images together, developers can ensure that changes to the application are reflected accurately in the containerized environment.

Strategies for Managing Docker Images

Managing Docker images involves several strategies to ensure consistency, traceability, and reproducibility across different environments.

i. Tagging: Docker images are tagged with version identifiers to distinguish between different versions of the same image. Semantic versioning principles are often applied, where tags include a major, minor, and patch version (Hill, 2016). Additionally, tags may include labels indicating specific features, environments, or configurations.

ii. Versioning: In addition to tagging, Docker images may be versioned using source control systems or image registries. Versioning enables developers to track changes to images over time, roll back to previous versions if necessary, and maintain a history of changes.

iii. Image Repositories: Docker images are typically stored in image repositories, such as Docker Hub, Amazon ECR, or private registries. Image repositories provide centralized storage for Docker images, enabling version control, access control, and distribution across different environments (Hill, 2016).

Exploring Continuous Integration and Continuous Deployment (CI/CD) Practices with Docker

Continuous integration and continuous deployment (CI/CD) practices are integral to modern software development workflows, enabling developers to automate the build, test, and deployment processes.

i. *Continuous Integration (CI):* With Docker, CI pipelines can build Docker images automatically whenever changes are pushed to the source code repository. CI tools like Jenkins, Travis CI, or GitLab CI integrate seamlessly with Docker, allowing developers to build and test applications in isolated Docker containers (Merkel, 2014, p. 30).

ii. *Continuous Deployment (CD):* Docker facilitates continuous deployment by providing a consistent deployment environment across different stages of the CI/CD pipeline. Docker images can be tagged and deployed to different environments (e.g., development, staging, production) using CI/CD tools and container orchestration platforms like Kubernetes or Docker Swarm (Hill, 2016).

iii. I*mmutable Infrastructure*: Docker promotes immutable infrastructure practices, where each deployment is based on a specific version of the Docker image. This ensures consistency and repeatability across different environments and simplifies rollback procedures in case of failures (Hill, 2016).

Effective versioning and lifecycle management with Docker involve implementing strategies for managing Docker images, including tagging, versioning, and image repositories, as well as integrating Docker into CI/CD workflows to automate the build, test, and deployment processes.

7. Challenges and Future Directions in Dockerization

Identifying Challenges and Limitations

Dockerization has revolutionized software development and deployment practices, but it also brings its own set of challenges and limitations.

i. *Portability*: While Docker promotes portability, challenges arise when moving containers between different environments with varying configurations. Differences in underlying infrastructure, such as operating systems or kernel versions, can lead to compatibility issues (Merkel, 2014, p. 30).

ii. *Isolation*: Docker's containerization provides a level of isolation, but it may not offer the same level of security as traditional virtualization technologies. Vulnerabilities in the host kernel or misconfigurations in container settings can compromise isolation and expose applications to security risks (Felter et al., 2015).

iii. *Scalability*: Although Docker enables horizontal and vertical scalability, managing large-scale container deployments introduces complexities in resource allocation, scheduling, and orchestration. Ensuring efficient resource utilization and optimal performance becomes challenging as the number of containers increases (Burns et al., 2016).

iv. *Versioning*: Managing versioning and lifecycle management of Docker images requires careful coordination between development, testing, and production environments. Inconsistent versioning practices can lead to deployment errors, compatibility issues, and difficulties in tracking changes over time (Hill, 2016).

Discussing Emerging Trends and Technologies

Several emerging trends and technologies aim to address the challenges associated with Dockerization and containerization.

i. *Distributed Container Orchestration:* Emerging container orchestration platforms like Kubernetes and Docker Swarm offer advanced features for managing distributed containerized applications. These platforms provide tools for automating deployment, scaling, and resource management across clusters of

servers, enabling seamless scalability and high availability (Burns et al., 2016).

ii. *Microservices Architecture:* Adopting a microservices architecture can mitigate some of the challenges associated with monolithic containerized applications. By decomposing applications into smaller, loosely coupled services, organizations can achieve greater flexibility, scalability, and resilience in their container deployments (Vaughan-Nichols, 2016).

iii. *Serverless Computing:* Serverless computing platforms like AWS Lambda and Azure Functions abstract away the underlying infrastructure, allowing developers to focus on writing code without worrying about managing servers or containers. While not strictly related to Dockerization, serverless architectures offer an alternative approach to deploying and scaling applications in a cloud-native environment (Boettiger, 2015).

Providing Recommendations for Future Research Directions

Future research in Docker and containerization should focus on addressing the remaining challenges and exploring new opportunities for innovation. Some potential research directions include:

i. *Enhancing Security:* Investigating techniques to improve the security of Docker containers, such as enhanced isolation mechanisms, vulnerability scanning, and runtime monitoring tools.

ii. *Optimizing Resource Management:* Developing algorithms and strategies for efficient resource allocation, scheduling, and scaling of containerized applications in large-scale deployments.

iii. *Standardizing Portability:* Establishing industry standards and best practices for ensuring seamless portability of Docker containers across different environments and platforms.

iv. *Streamlining Versioning and Lifecycle Management:* Developing tools and methodologies for simplifying versioning, dependency management, and deployment workflows in Dockerized environments.

While Dockerization has transformed software development and deployment practices, several challenges remain, particularly regarding portability, isolation, scalability, and versioning. Emerging trends and technologies offer promising solutions to address these challenges, and future research should focus on advancing the state of the art in Docker and containerization to meet the evolving needs of modern software development.

8. Technical Implementation of Dockerizing an Application

Dockerizing an application involves several steps, including creating Docker images, defining container configurations, and orchestrating container deployments. Below is a detailed technical implementation guide:

i. *Preparing the Application:*

· Ensure that the application code is modularized and dependencies are clearly defined.

· Update the application to use environment variables for configuration, allowing flexibility in different deployment environments (Hill, 2016).

ii. *Writing Dockerfile:*

· Create a Dockerfile in the root directory of the application.

· Specify a base image using FROM instruction, choosing an appropriate base image based on the application's requirements (Hill, 2016).

· Use COPY or ADD instructions to copy the application code into the Docker image.

· Set environment variables, configure ports, and execute any necessary setup commands using ENV, EXPOSE, and RUN instructions.

· Ensure proper cleanup and optimization to minimize image size (Merkel, 2014, p. 30).

iii. *Building Docker Image:*

· Use the docker build command to build the Docker image from the Dockerfile.

· Tag the image with a version identifier using the -t flag.

· Monitor the build process for any errors or warnings (Hill, 2016).

iv. *Running Docker Container:*

· Once the image is built, use the docker run command to create and start a container based on the image.

· Specify any runtime options such as port mappings, volume mounts, and environment variables.

· Monitor container logs for debugging and troubleshooting (Hill, 2016).

v. *Managing Multiple Environments:*

· Use Docker Compose or Kubernetes to define multi-container applications and manage dependencies between containers (Burns et al., 2016).

· Define separate configuration files for development, testing, and production environments, allowing environment-specific settings to be easily managed.

vi. *Continuous Integration and Deployment (CI/CD):*

· Integrate Docker into CI/CD pipelines to automate the build, test, and deployment processes (Merkel, 2014, p. 30).

· Use CI/CD tools like Jenkins, GitLab CI, or Travis CI to trigger Docker builds and deploy containers to various environments.

vii. *Monitoring and Scaling:*

· Utilize Docker monitoring tools like Docker Swarm, Prometheus, or Grafana to monitor container health, resource usage, and performance metrics (Burns et al., 2016).

· Implement auto-scaling policies to dynamically adjust the number of containers based on workload demand.

viii. Security Considerations:

· Follow best practices for securing Docker containers, such as running containers with non-root users, minimizing container privileges, and regularly updating base images (Felter et al., 2015).

· Utilize Docker Security Scanning or third-party vulnerability scanning tools to identify and remediate security vulnerabilities in Docker images.

By following these steps, developers can successfully Dockerize their applications, enabling consistent and scalable deployment across different environments. The Docker workflow is explained in Fig 1.

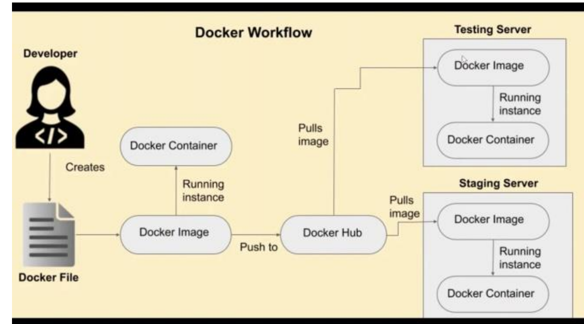


Fig 1. (Docker Workflow for any application)

Docker statistics:

Category	Metric	Value
Adoption	Number of Docker Hub pulls	20 billion pulls per month
	Number of Docker images on Docker Hub	7.4 million images
Performance	Docker container startup time	Typically less than a second
	Overhead compared to bare metal	Less than 1-3% CPU overhead
Market Share	Docker market share in containerization	Over 30% market share
Security	Number of CVEs reported in Docker	430+ CVEs fixed
	Docker Bench for Security score	Typically scores 100%
Community Engagement	Number of Docker contributors	2500+ contributors
	Docker GitHub stars	Over 50,000 stars
Usage in Enterprises	Percentage of enterprises using Docker	83% of enterprises

9. Conclusion

In conclusion, this review paper has provided a comprehensive examination of Dockerized applications, focusing on key aspects such as portability, isolation, scalability, and versioning. Through a thorough analysis of existing literature and technical insights, several key findings and insights have emerged.

Firstly, Docker has emerged as a transformative technology in modern software development, offering a lightweight and efficient approach to containerization. It has significantly improved the efficiency of application deployment by providing consistent runtime environments across different platforms (Merkel, 2014, p. 30). Additionally, Docker's containerization technology enables enhanced portability, allowing applications to be packaged once and deployed anywhere with minimal dependencies (Hill, 2016).

Furthermore, Docker provides robust isolation mechanisms, ensuring that applications running within containers are isolated from one another and from the underlying host system. However, maintaining adequate security in Dockerized environments remains a critical consideration, requiring adherence to best practices and additional security measures (Felter et al., 2015).

Scalability is another key advantage of Docker, with the ability to dynamically scale containers horizontally and vertically to accommodate varying workload demands (Burns et al., 2016). Container orchestration tools like Kubernetes further enhance scalability by automating deployment, scaling, and management tasks across clusters of servers.

Effective versioning and lifecycle management are essential for maintaining consistency and traceability in Dockerized environments. By adopting best practices for managing Docker images, including tagging, versioning, and image repositories, organizations can streamline deployment workflows and ensure compatibility across different environments (Hill, 2016).

Moving forward, the implications of Docker and containerization for industry and research communities are significant. In the industry, Docker has already transformed software development practices, enabling faster release cycles, improved collaboration between development and operations teams, and greater agility in responding to changing business requirements (Vaughan-Nichols, 2016).

In the research community, Docker continues to be a topic of interest, with ongoing efforts to address remaining challenges and explore new opportunities for innovation. Areas such as security, resource management, and orchestration techniques present rich opportunities for future research (Burns et al., 2016).

In conclusion, Docker and containerization offer significant benefits for modern software development, but it is essential to consider factors such as portability, isolation, scalability, and versioning to maximize their potential. By addressing these considerations effectively, organizations can harness the full power of Docker to drive innovation and achieve competitive advantage in today's fast-paced digital landscape.

References:

[1] Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. Proceedings of the

IEEE International Symposium on Performance Analysis of Systems and Software, 171–172. DOI: 10.1109/ISPASS.2015.7095807

[2] Hill, D. (2016). Docker: Up & Running: Shipping Reliable Containers in Production. O'Reilly Media.

[3] Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, 239(2), 2-11. Available at: https://dl.acm.org/doi/10.5555/2600239.2600241

[4] Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71–79. DOI: 10.1145/2723872.2723882

[5] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. ACM Queue, 14(1), 70–93. DOI: 10.1145/2898445.2898447

[6] Chung, C. J., Liu, C., Lo, S., & Peng, C. (2019). Container-as-a-Service platform with Docker and Kubernetes for cloud-based medical analysis. Journal of Medical Systems, 43(1), 1–11. DOI: 10.1007/s10916-018-1106-5

[7] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. ACM SIGOPS Operating Systems Review, 41(3), 275–287. DOI: 10.1145/1272996.1273028

[8] Vaughan-Nichols, S. J. (2016). The amazing Docker: Bringing containers for Linux to your desktop. Computer, 49(4), 70–73. DOI: 10.1109/MC.2016.113