IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁘ Peer-reviewed & Refereed journal ⁘ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15331

MULTI-STAGE DOCKER BUILDS AND DEPLOYMENT ON AWS ECS

Nandhini S[1], Dr. B. Narasimhan[2]

III BCA, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore 641006, Tamil Nadu, India[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore-641006, Tamil Nadu, India[2]

Abstract: Modern software applications require scalable, portable, and efficient deployment mechanisms. Traditional deployment methods often involve manual configuration of servers and dependencies, which increases complexity and deployment time. Containerization technologies such as Docker address these issues by packaging applications along with their dependencies into lightweight containers that can run consistently across different environments. However, single-stage container builds may create large images containing unnecessary dependencies.

This project presents a cloud-native deployment architecture that uses multi-stage Docker builds to create optimized container images and deploy them using Amazon Elastic Container Service (ECS). The application is containerized using Docker, stored in a container registry, and executed in a serverless container environment using AWS Fargate. The system also integrates monitoring and logging capabilities through centralized logging services to observe runtime performance and system behavior.

The proposed architecture demonstrates how optimized container images can be efficiently deployed and managed in a cloud environment while ensuring scalability, reliability, and security.

Keywords: Docker, Multi-Stage Builds, Containerization, AWS ECS, Cloud Deployment, DevOps

1. INTRODUCTION

Modern software systems demand portability, scalability, reliability, and efficient deployment processes. Traditional application deployment methods require manual server configuration, installation of dependencies, and environment-specific adjustments. These manual processes often result in inconsistencies between development, testing, and production environments.

Containerization has emerged as a powerful solution to overcome these challenges. Containers package an application along with its dependencies and runtime environment, allowing the software to run consistently across different systems. Docker is one of the most widely used containerization platforms that enables developers to build, package, and distribute applications efficiently.

However, traditional Docker builds sometimes produce large images containing unnecessary files and dependencies. Large images increase storage requirements and slow down deployment time. To address this issue, multi-stage Docker builds are used. This technique separates the build environment from the runtime environment, ensuring that only required components are included in the final image.

For large-scale container execution, container orchestration platforms are required. Amazon Elastic Container Service (ECS) is a managed container orchestration platform that allows users to run and manage containers without managing underlying servers. ECS combined with AWS Fargate enables serverless container execution, eliminating infrastructure management overhead.

The objective of this project is to design and implement a cloud-based architecture that integrates optimized Docker builds with AWS ECS deployment. The system demonstrates container packaging, deployment, monitoring, and automatic scaling in a secure cloud environment.

ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15331

2. LITERATURE SURVEY

Several studies have highlighted the importance of containerization and cloud-based deployment for modern software systems. Traditional deployment architectures rely heavily on manual infrastructure management, which increases operational complexity and reduces deployment speed.

Earlier research indicates that virtualization technologies were widely used to isolate applications and provide resource management. However, virtual machines consume significant system resources because each instance runs a full operating system. Containers provide a lightweight alternative by sharing the host operating system kernel while maintaining application isolation.

Docker has become a widely adopted platform for containerization due to its ability to package applications and dependencies into portable images. Research studies have demonstrated that Docker containers significantly reduce deployment time and improve consistency across environments.

Multi-stage Docker builds have been introduced as an optimization technique that reduces container image size by separating the build stage from the runtime stage. This approach minimizes unnecessary files in the final container image, improving performance and security.

Cloud orchestration platforms such as AWS ECS, Kubernetes, and Docker Swarm have also gained popularity for managing large numbers of containers. These platforms automate container scheduling, resource allocation, monitoring, and scaling. Among them, AWS ECS provides a fully managed service that integrates seamlessly with other AWS cloud services.

Recent studies also emphasize the importance of centralized logging and monitoring for cloud-based systems. Monitoring tools allow administrators to track performance metrics such as CPU usage, memory consumption, and container health, enabling efficient resource management.

The proposed system builds upon these concepts by integrating multi-stage Docker builds with AWS ECS to create a scalable and efficient deployment architecture.

3. METHODOLOGY

The methodology used in this project focuses on designing a cloud-native deployment architecture using containerization and managed orchestration services. The system follows a structured workflow that includes application development, container image creation, registry storage, cloud deployment, monitoring, and scaling.

The application is developed using Python and packaged into a Docker container using a multi-stage build process. This process separates dependency installation from runtime execution to minimize image size. The container image is then stored in a secure container registry.

The container is deployed on AWS ECS using a serverless execution model. ECS automatically manages container scheduling, resource allocation, and task execution. Runtime logs are captured and stored in a centralized logging system to monitor application performance.

This approach ensures that the system remains scalable, reliable, and easy to maintain while reducing infrastructure management overhead.

3.1 Architectural Design

The system architecture consists of multiple interconnected components that work together to enable containerized application deployment.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15331

The architecture begins with the **application development environment**, where the Python application and configuration files are created. The application is then packaged into a container image using a Dockerfile with multi-stage build instructions.

The container image is pushed to a **container registry**, which acts as a secure storage location for versioned container images. The AWS ECS platform retrieves the container image from the registry and launches it within a managed execution environment.

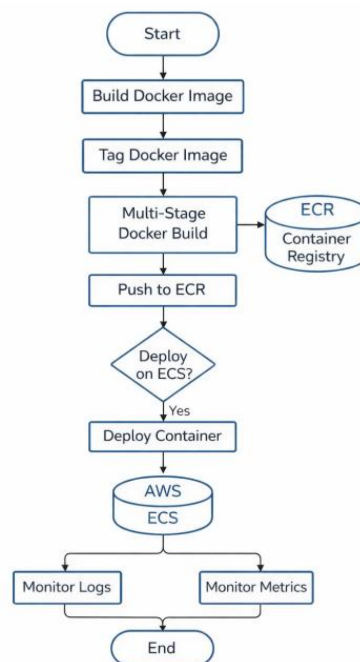
The application runs inside a container on AWS ECS using the **Fargate launch type**, which eliminates the need to manage servers or virtual machines. Monitoring services collect runtime logs and system metrics such as CPU utilization and memory usage.

This architecture provides a scalable and secure environment for executing containerized applications in the cloud.

3.2 Workflow / System Flow

The system workflow follows a sequence of deployment and execution steps:

- The application is developed using Python and necessary dependencies are defined in a requirements file.
- A Dockerfile is created to build the container image using a multi-stage build process.
- The Docker image is built locally and tagged with a version identifier.
- The container image is uploaded to a container registry.
- AWS ECS retrieves the image from the registry and deploys it as a running container task.
- The container executes the application and generates runtime system logs.
- Logs and performance metrics are collected through monitoring services.
- Auto-scaling policies adjust the number of container instances based on CPU utilization.
- This workflow demonstrates a complete cloud-native deployment pipeline from development to monitoring.



4. RESULTS AND EXPLANATION

After deployment, the containerized application successfully runs inside the AWS ECS environment. The application continuously generates runtime reports containing system information such as hostname, platform details, CPU usage, memory utilization, Python runtime version, and execution time.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15331

These reports are captured by the centralized logging system, which stores and displays the logs through the cloud monitoring console. Administrators can view logs in real time to confirm that the application is functioning correctly.

The monitoring dashboard displays key performance metrics including CPU utilization and container task count. When system load increases, the auto-scaling mechanism automatically launches additional container instances. When load decreases, unnecessary containers are terminated to conserve resources.

The results demonstrate that the system successfully achieves its objectives of optimized container deployment, centralized monitoring, and automatic scaling within a secure cloud environment.

5. CONCLUSION AND FUTURE SCOPE

This project demonstrates the implementation of a cloud-native deployment architecture using multi-stage Docker builds and AWS ECS. The system successfully packages the application into an optimized container image and deploys it in a scalable cloud environment.

The use of multi-stage builds significantly reduces container image size, improving deployment speed and security. AWS ECS provides automated container orchestration, eliminating the need for manual infrastructure management. Centralized logging and monitoring allow administrators to track application performance and diagnose issues efficiently.

The proposed system offers several advantages including portability, scalability, resource efficiency, and simplified deployment. These characteristics make containerized cloud deployment suitable for modern enterprise applications.

In the future, the system can be extended by integrating continuous integration and continuous deployment (CI/CD) pipelines for automated build and deployment processes. Additional features such as load balancing, advanced monitoring dashboards, and security automation can further enhance system performance and reliability.

REFERENCES

[1]. Docker Documentation. Available: https://docs.docker.com/get-started/
[2]. Docker Multi-Stage Builds Guide. Available: https://docs.docker.com/build/building/multi-stage/
[3]. Amazon Web Services. Amazon Elastic Container Service (ECS). Available: https://aws.amazon.com/ecs/
[4]. AWS Fargate Documentation. Available: https://aws.amazon.com/fargate/

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License