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.2026.15305

Dockerize a Node.js Application and Deploy on Amazon EC2

Vighnesh Jayakumar[1], Dr. KS. Gowrilaksshmi[2]

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

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

Abstract: Cloud computing and containerization technologies have fundamentally transformed modern software application deployment. Traditional deployment methods required manual server configuration, dependency management, and environment maintenance—processes that were time-consuming and error-prone. This project demonstrates the complete, end-to-end process of Dockerizing a Node.js application and deploying it on an Amazon Elastic Compute Cloud (EC2) instance. Docker encapsulates the application and its runtime environment into a portable container image, ensuring consistent behavior across environments. The project covers EC2 instance provisioning on Amazon Linux 2023, Docker Engine installation, container lifecycle management, AWS Security Group configuration, port mapping, and comprehensive deployment verification through terminal commands and browser-based testing. Results confirm successful deployment with the application accessible via public IPv4, validating the effectiveness of containerized cloud deployment as a modern DevOps solution.

Keywords: Docker, Node.js, Amazon Web Services (AWS), Amazon EC2, Containerization, Cloud Deployment, DevOps, Docker Image, Dockerfile, Security Groups, Port Mapping, Amazon Linux 2023, CI/CD.

I. INTRODUCTION

In the rapidly evolving landscape of modern software engineering, deploying applications quickly, reliably, and consistently across diverse computing environments has become a critical competitive advantage. Traditional deployment workflows were largely manual, labor-intensive, and dependent on the specific configuration of individual servers. System administrators had to install programming runtimes, configure environment variables, install dependency libraries, and ensure all components were compatible with each other.

The notorious "works on my machine" problem—where an application functions correctly in development but fails in production due to environmental differences—became one of the most costly challenges in software deployment. The introduction of Docker containerization addressed this by encapsulating an application together with its entire execution environment into a portable container image, eliminating inconsistency issues entirely. Simultaneously, AWS transformed infrastructure provisioning from capital expenditure to on-demand operational resources.

This project brings together Docker containerization and AWS cloud infrastructure to demonstrate the complete process of deploying a Node.js web application in a production-like cloud environment. Node.js was selected due to its widespread adoption in modern web development, its efficient asynchronous I/O model, and its suitability for scalable network applications [1].

II. RELATED WORK

Several studies have examined containerization and cloud deployment methodologies. Traditional application deployment relied heavily on manual server configuration and system administration expertise, which proved inconsistent across environments [2]. Research in virtualization demonstrated that isolating application environments reduces deployment failures, but full virtual machines carry significant overhead in memory and startup time.

Docker's lightweight containerization model, based on Linux namespaces and control groups (cgroups), provides near-native performance with strong process isolation [3]. Studies comparing Docker containers to traditional VMs consistently show container startup times reduced from minutes to seconds, and memory overhead reduced by 40-60%.

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.2026.15305

Cloud-based deployment on AWS EC2 has been shown to improve deployment repeatability and enable rapid horizontal scaling compared to on-premises infrastructure [4].

However, existing studies often focus on either containerization theory or cloud infrastructure separately. Few provide complete step-by-step documentation of integrating both technologies for production deployments. This project addresses that gap by providing a comprehensive, verified end-to-end deployment guide.

III. OBJECTIVES AND CHALLENGES

3.1 Primary Objectives

The primary objectives of this project are:

- To demonstrate the complete process of deploying a Dockerized Node.js application on Amazon EC2.
- To provide hands-on experience with AWS EC2, Docker Engine, and Amazon Linux 2023.
- To validate deployment through comprehensive testing using terminal commands and browser verification.
- To document each step with detailed explanations, creating a replicable deployment guide.

3.2 Development Challenges

Several challenges were encountered during implementation. Configuring AWS Security Group rules to correctly expose application ports while maintaining security required careful testing. The Docker daemon required manual enabling and starting post-installation on Amazon Linux 2023, as it does not start automatically. Container naming conflicts arose when attempting to re-run containers with existing stopped instances—resolved by distinguishing between docker run (creates new) and docker start (restarts existing). Additionally, ensuring the correct port mapping (0.0.0.0:3000→container:3000) for public internet accessibility required verification of both Security Group inbound rules and Docker run flags.

IV. SYSTEM ARCHITECTURE

The deployment architecture of this project is organized into three interconnected layers. The first layer is the AWS EC2 instance (t3.micro), which provides virtualized computing infrastructure including CPU, memory, EBS-backed storage, and up to 5 Gbps network bandwidth running Amazon Linux 2023. The second layer is the Docker Engine (version 25.0.14), which runs as a systemd service on the EC2 instance and manages the full container lifecycle. The third layer is the Docker container itself, which provides the isolated execution environment for the Node.js HTTP web server application.

Security is implemented at two levels: Docker container isolation using Linux namespaces provides process-level separation, while AWS Security Groups act as a stateful virtual firewall at the network level, controlling inbound and outbound traffic. The application is accessible over the public internet via the EC2 instance's public IPv4 address on TCP port 3000.

Table 1: Hardware Configuration Requirements

Component	Minimum Requirement	Recommended
Processor	Intel Core i3 / AMD Ryzen 3	Intel Core i5 or higher
RAM	4 GB DDR4	8 GB DDR4 or higher
Storage	20 GB (HDD)	50 GB SSD
Network	10 Mbps broadband	50 Mbps or higher
Display	1024 x 768	1920 x 1080 Full HD

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.2026.15305

V. IMPLEMENTATION

5.1 EC2 Instance Provisioning

The deployment begins with provisioning an EC2 t3.micro instance in AWS region us-east-1 (US East, N. Virginia), Availability Zone us-east-1f, running Amazon Linux 2023. The instance was accessed via EC2 Instance Connect, a browser-based SSH terminal that requires no local SSH client or key pair management. System packages were updated using sudo yum update -y to ensure security patches and software compatibility.

5.2 Docker Installation and Configuration

Docker Engine was installed using sudo yum install docker -y followed by sudo systemctl start docker and sudo systemctl enable docker to ensure the service starts automatically on reboot. The installation was verified with docker –version, confirming Docker version 25.0.14 (build 0bab007). The pre-built Node.js application image node-app:latest (127 MB) was verified available using docker images.

5.3 Container Deployment

The application was deployed with: docker run -d –name node-container -p 3000:3000 node-app:latest. This command runs the container in detached mode (-d), assigns the name node-container, maps host port 3000 to container port 3000 (-p 3000:3000), and uses the pre-built image. The AWS Security Group (sg-009814cec70f681d7) was configured to allow TCP inbound traffic on port 3000 from 0.0.0.0/0, making the application accessible at http://32.192.90.7:3000 [5].

Table 2: Software Components and Versions

Software Component	Version	Purpose
Amazon EC2	t3.micro	Virtual server hosting
Amazon Linux	2023 (AL2023)	Server operating system
Docker Engine	25.0.14	Container runtime
Node.js	v18.x LTS	Application runtime
EC2 Instance Connect	Browser-based	Secure server terminal
systemctl / systemd	252	Service management

VI. EVALUATION RESULTS AND DISCUSSION

The deployment was validated through five comprehensive test cases covering all aspects of the deployment. All test cases passed successfully, confirming that the containerized application is fully operational in the cloud environment.

Table 3: Deployment Test Results

Test Case	Description	Result
Docker Installation	Verify Docker version on EC2	PASSED
Image Availability	Confirm node-app:latest present	PASSED
Container Running	docker ps shows active container	PASSED
Localhost Access	curl http://localhost:3000 responds	PASSED
Public IPv4 Access	Browser access via 32.192.90.7:3000	PASSED

The deployment demonstrates that Docker containerization solves the environment inconsistency problem: the same container image deployed on the EC2 instance behaves identically to the local development environment. The Security Group configuration validated that cloud network access control can be precisely managed to expose only required ports while restricting all other access. The entire deployment lifecycle—from EC2 provisioning to running application—was completed in approximately 15–20 minutes, compared to several hours for equivalent traditional manual server setup.

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.2026.15305

Container-based deployments also demonstrated clear advantages in reproducibility. Rolling back to a previous version requires only stopping the current container and running the previous image tag, a process that takes under a minute. This immutability is a fundamental improvement over traditional deployments where state changes to server configurations accumulate and are difficult to reverse.

VII. CONCLUSION

This project successfully demonstrated the complete process of Dockerizing a Node.js application and deploying it on Amazon EC2. The system was built around three interconnected layers—EC2 infrastructure, Docker Engine, and the application container—providing a secure, consistent, and scalable cloud deployment architecture. All five deployment test cases passed, confirming that the containerized application is publicly accessible and performs as expected.

The results confirm that combining Docker containerization with AWS cloud infrastructure eliminates the key limitations of traditional deployment: environment inconsistencies, dependency conflicts, slow deployment processes, and poor scalability. The deployment guide created through this project serves as a practical, replicable reference for developers and students seeking to acquire modern DevOps skills demanded by the current technology industry.

VIII. FUTURE ENHANCEMENTS

Several enhancements can further strengthen this deployment. Implementing a CI/CD pipeline using GitHub Actions or AWS CodePipeline would automate building, testing, and deploying new container images on every code push. Migrating from a single EC2 instance to Amazon ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service) would provide container orchestration, automated health checks, and horizontal scaling. Adding an Application Load Balancer with multiple EC2 instances across Availability Zones would ensure high availability and fault tolerance. Integrating AWS CloudWatch for log aggregation and metrics monitoring, and AWS Secrets Manager for secure credential management, would bring the deployment to production-grade standards.

REFERENCES

[1]. Node.js Foundation, "Node.js: A JavaScript Runtime Built on Chrome's V8 Engine," Node.js Documentation, 2024.
[2]. Turnbull, J., "The Docker Book: Containerization is the New Virtualization," Docker Inc. Technical Press, 2023.
[3]. Merkel, D., "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, 2014.
[4]. Amazon Web Services, "Amazon EC2 Instance Types," AWS Documentation, 2024. [Online]. Available: https://aws.amazon.com/ec2/instance-types/
[5]. Amazon Web Services, "Security Groups for Your VPC," AWS VPC User Guide, 2024. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html
[6]. Kim, G., Humble, J., Debois, P., Willis, J., "The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security," IT Revolution Press, 2021.
[7]. Poulton, N., "Docker Deep Dive," Independently Published, 2023.

