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.2026.15312

Continuous Deployment using GitLab CI/CD and Docker

R. K. Selvavishnu[1], Dr. B. Narasimhan[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: Continuous Deployment (CD) is a modern DevOps practice that enables automatic building, testing, and deployment of applications whenever changes are pushed to the source code repository. This project focuses on implementing Continuous Deployment using GitLab CI/CD pipelines and Docker container technology to automate the software delivery process. The system integrates version control, automated pipeline execution, container image creation, testing, and deployment into a seamless workflow. When a developer pushes code to the main branch, the GitLab CI/CD pipeline is automatically triggered. The application is built into a Docker image, tested, stored in a container registry, and deployed to a target server without manual intervention. By containerizing the application using Docker, consistency across development, testing, and production environments is ensured. GitLab CI/CD manages the automation stages including build, test, and deploy, reducing human error and improving deployment speed and reliability. The implementation demonstrates how automated pipelines improve software quality, enhance team productivity, and support faster release cycles, providing a scalable foundation for future enhancements such as Kubernetes-based deployment and automated monitoring systems.

Keywords: Continuous Deployment, GitLab CI/CD, Docker, DevOps, Containerization, Automation, Pipeline, Software Delivery.

I. INTRODUCTION

In modern software development, delivering applications quickly and reliably is essential. Traditional development methods often require manual steps for building, testing, and deploying software, which are time-consuming and error-prone. Continuous Deployment (CD) has emerged as a key DevOps practice to overcome these challenges by automating the entire software delivery lifecycle.

Continuous Deployment is a software development approach where code changes are automatically built, tested, and deployed to production without manual intervention. This process helps developers release updates faster and ensures that new features, improvements, or bug fixes reach users quickly.

Continuous Deployment typically works in conjunction with Continuous Integration (CI), forming a pipeline that automates the entire software delivery process. Tools such as GitLab provide built-in CI/CD pipelines that help manage the process efficiently. Developers push their code to a repository, and the system automatically triggers tasks like building the application, running tests, and deploying it to the server.

Technologies such as Docker are widely used in Continuous Deployment to package applications into containers. These containers ensure that the application runs consistently across different environments including development, testing, and production. The main goal of Continuous Deployment is to increase development speed, improve software quality, and reduce manual work.

In this project, a Continuous Deployment system is designed and implemented using GitLab CI/CD and Docker to demonstrate how code changes can automatically move from development to deployment through an automated pipeline, with a Python Flask application hosted on AWS EC2.

II. RELATED WORK

Several deployment methodologies have been studied and adopted in the industry prior to automated CI/CD approaches. Traditional in-place deployment involves manually replacing the running application on a production server. While

DOI: 10.17148/IJARCCE.2026.15312

straightforward to execute, this approach results in significant downtime per deployment cycle and makes rollback slow and error-prone.

Research on containerization demonstrates that Docker significantly improves deployment reliability and reduces environment-related failures. By packaging the application alongside its dependencies, Docker eliminates the infamous "works on my machine" problem, ensuring behavioral consistency across all environments.

GitLab CI/CD has been widely studied as an integrated platform that combines source code management with automated pipeline execution. Unlike standalone tools such as Jenkins, GitLab provides a unified interface for version control, pipeline orchestration, container registry management, and deployment automation in a single platform.

Studies on DevOps adoption consistently demonstrate that organizations implementing automated CI/CD pipelines achieve higher deployment frequencies, shorter lead times for changes, and lower change failure rates. The DORA (DevOps Research and Assessment) metrics framework confirms that automation reduces human error and accelerates recovery from failures.

Existing literature identifies a gap in practical, documented implementations of CI/CD pipelines using GitLab and Docker deployed on cloud infrastructure. This project addresses that gap by providing a complete, reproducible implementation with comprehensive documentation and tested results.

III. OBJECTIVES AND CHALLENGES

The primary objectives of this project are: (1) to implement a fully automated Continuous Deployment pipeline using GitLab CI/CD; (2) to containerize a Python Flask web application using Docker for consistent cross-environment deployment; (3) to configure GitLab Runner and AWS EC2 for end-to-end automated deployment; (4) to demonstrate automated testing and deployment validation within the pipeline; and (5) to provide a reproducible reference architecture with comprehensive documentation for DevOps practitioners.

Development Challenges

The primary technical challenge was coordinating Docker container lifecycle management with GitLab Runner execution. Configuring the Docker-in-Docker (DinD) service within the pipeline required careful management of Docker socket permissions and network bridge modes. This was resolved by specifying the overlay2 driver and granting appropriate service dependencies within the. Gitlab-ci.yml configuration.

Managing SSH key authentication between the GitLab Runner and the AWS EC2 deployment server required secure storage of private keys as CI/CD environment variables. The deployment script was designed to programmatically write the SSH key, set correct file permissions, and execute remote Docker commands without interactive prompts, ensuring non-interactive deployment.

Ensuring that the GitLab Container Registry and Docker Hub credentials were securely managed required disciplined use of GitLab's masked CI/CD variables. Sensitive data such as registry passwords and server IP addresses were never committed to the repository, preventing credential exposure in pipeline logs.

IV. SYSTEM ARCHITECTURE

The system follows a pipeline-based architecture that integrates version control, automated build and test stages, container registry storage, and automated deployment. Each layer communicates through well-defined interfaces, ensuring loose coupling and independent scalability of each component.

The system architecture follows the principle of infrastructure immutability. Rather than modifying running containers in place, every deployment creates fresh container instances from versioned images stored in the registry. This eliminates configuration drift and ensures the running environment always reflects the documented configuration.

The CI/CD pipeline is defined entirely in the .gitlab-ci.yml file, which specifies three stages: build, test, and deploy. Each stage runs in isolation within Docker containers managed by the GitLab Runner. The build stage compiles the Docker image from the Dockerfile and pushes it to the registry. The test stage runs automated unit tests against the application. The deploy stage connects to the AWS EC2 server via SSH and executes Docker pull and run commands to update the running container.

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.2026.15312

Table 1. System Architecture Layers

Layer	Technology	Role
Developer Workstation	Python Flask, Docker, Git, VS Code	Application development and image build
Version Control	GitLab Repository	Source code management, branch control, merge requests
CI/CD Pipeline	GitLab CI/CD (.gitlab-ci.yml)	Automated build, test, and deploy orchestration
Pipeline Executor	GitLab Runner	Executes pipeline jobs on configured server
Containerization	Docker, Dockerfile	Packages app and runtime into portable images
Image Registry	GitLab Container Registry / Docker Hub	Versioned Docker image storage and pull
Compute Infrastructure	AWS EC2 (Ubuntu 22.04)	Hosts Docker Engine and deployed containers
Deployment	Docker Container, Port 5000	Running application served to users
Security	SSH Keys, GitLab CI/CD Variables	Encrypted access, masked credentials
End Users	HTTP via EC2 Public IP	Access to deployed Flask application

V. IMPLEMENTATION

The implementation is organized into five functional modules, each addressing a distinct layer of the Continuous Deployment architecture. This modular design ensures that individual components can be updated, tested, and replaced independently without disrupting the overall system.

A. Python Flask Web Application

The application is built using Python 3.11 and the Flask micro-framework, implementing a simple REST API that returns a greeting response. The application listens on port 5000 and provides a /health endpoint used by the CI/CD pipeline for deployment validation. The requirements.txt file specifies all Python dependencies, which are installed during the Docker image build process.

B. Docker Containerization

The application is packaged into a Docker container using a Dockerfile based on the official Python 3.11 slim image. The Dockerfile sets the working directory, copies application files, installs dependencies via pip, exposes port 5000, and defines the startup command. A .dockerignore file excludes unnecessary files such as local Python cache directories and environment files from the build context, reducing image size and improving build performance.

C. GitLab Repository Configuration

The project repository is hosted on GitLab and contains the application source code, Dockerfile, .gitlab-ci.yml pipeline configuration, and README documentation. Branch management follows a main-branch-centric workflow where pushes to the main branch automatically trigger the CI/CD pipeline. CI/CD variables including Docker registry credentials, SSH private key, and deployment server IP address are stored as masked variables in the GitLab project settings.

D. CI/CD Pipeline Configuration

The .gitlab-ci.yml file defines the three-stage automated pipeline. The build stage uses the docker:24.0.5 image with Docker-in-Docker service to build and push the container image. The test stage pulls the built image and executes the application's test suite using pytest. The deploy stage uses SSH to connect to the AWS EC2 server, pulls the latest Docker

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.2026.15312

image, stops and removes any existing container, and starts the updated container with the correct port mapping and restart policy.

E. AWS EC2 Deployment Server

The deployment target is an AWS EC2 instance running Ubuntu 22.04 LTS with Docker Engine installed. The instance is configured with a security group permitting inbound SSH (port 22) from the GitLab Runner IP and HTTP access (port 5000) for application users. The EC2 instance's public IP address is stored as a masked CI/CD variable, and SSH access is authenticated using an RSA key pair whose private key is securely stored in GitLab.

VI. RESULTS AND EVALUATION

The Continuous Deployment system was tested comprehensively across pipeline execution, containerization, deployment reliability, and application performance. All pipeline stages executed successfully in automated runs triggered by code commits to the main branch.

Table 2. Pipeline Execution Performance Results

Metric	Measured Result	Benchmark	Status
Build Stage Duration	1 min 42 sec	< 5 min	✓ PASS
Test Stage Duration	38 sec	< 2 min	✓ PASS
Deploy Stage Duration	52 sec	< 3 min	✓ PASS
Total Pipeline Time	3 min 12 sec	< 10 min	✓ PASS
Docker Image Size	148 MB	< 500 MB	✓ PASS
Application Response Time	< 50 ms	< 200 ms	✓ PASS
Pipeline Success Rate	100%	> 95%	✓ PASS
Rollback Time (re-deploy)	< 4 min	< 10 min	✓ PASS

Table 3. Deployment Test Scenarios

Test Scenario	Expected Result	Outcome
Code push to main branch	Pipeline auto-triggered	PASS
Docker image build stage	Image built and pushed to registry	PASS
Automated test stage	Tests executed and passed	PASS
SSH deployment to EC2	Container deployed successfully	PASS
Application accessible via HTTP	Flask app returns 200 OK	PASS
Pipeline failure on test error	Deploy stage blocked	PASS
Container restart on EC2 reboot	Auto-restart via Docker policy	PASS
New commit re-triggers pipeline	Updated container deployed	PASS

Pipeline execution testing confirmed that every commit to the main branch automatically triggers the full build-test-deploy sequence without any manual intervention. The Docker build stage consistently completed within under two minutes, with layer caching reducing subsequent builds to under one minute for minor code changes.

Deployment testing confirmed that the Flask application was accessible via the EC2 public IP immediately following pipeline completion. The /health endpoint returned HTTP 200 responses, confirming successful container initialization. Container restart testing confirmed that the --restart unless-stopped Docker policy automatically recovers the application within 30 seconds of an EC2 reboot.

DOI: 10.17148/IJARCCE.2026.15312

The project demonstrates measurable improvements across all key DevOps metrics. Deployment frequency increased from manual ad-hoc releases to fully automated deployments on every commit. Lead time for changes decreased from hours to under four minutes. Human error was eliminated from the deployment process, and rollback capability was achieved by re-running the pipeline with the previous commit.

VII. CONCLUSION

This project successfully implemented a Continuous Deployment system using GitLab CI/CD pipelines and Docker containerization for a Python Flask web application hosted on AWS EC2. The architecture integrates source code management, automated pipeline execution, container image creation, and remote deployment into a seamless, fully automated workflow triggered by code commits.

Performance evaluation confirms that the total pipeline execution time averages under four minutes, meeting all defined benchmarks. All eight deployment test scenarios passed, demonstrating the reliability of the automated build, test, and deployment mechanisms. The implementation adheres to security best practices including masked CI/CD variables, SSH key authentication, and Docker registry credential management.

The results validate that combining GitLab CI/CD with Docker and AWS EC2 provides a scalable, reliable, and cost-efficient solution for modern Continuous Deployment pipelines. The documented architecture, configuration files, and procedures serve as a reproducible reference that DevOps practitioners can adapt for production implementations.

VIII. FUTURE ENHANCEMENTS

Several enhancements can extend the capabilities and maturity of this implementation. The most significant improvement would be migration to Kubernetes-based container orchestration using AWS EKS or a self-managed cluster, enabling automatic scaling, self-healing deployments, and advanced rollout strategies such as canary and Blue/Green deployments without service interruption.

Integration with AWS Application Load Balancer (ALB) and multiple EC2 instances would provide high availability and horizontal scaling, eliminating the current single point of failure at the deployment server. Automated monitoring using Prometheus and Grafana would provide real-time visibility into application performance, container health, and pipeline execution metrics, enabling proactive incident response.

Infrastructure as Code (IaC) using Terraform or AWS CloudFormation would automate the provisioning of all cloud resources, ensuring reproducible infrastructure across environments. Integration with automated security scanning tools within the CI/CD pipeline, including container image vulnerability scanning and static application security testing (SAST), would further improve the security posture of the deployment system.

REFERENCES

[1]. Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press.
[2]. Humble, J., & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional.
[3]. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 2014(239).
[4]. GitLab Documentation. (2024). CI/CD Pipelines. https://docs.gitlab.com/ee/ci/
[5]. Docker Documentation. (2024). Docker Engine Overview. https://docs.docker.com/get-started/
[6]. Amazon Web Services. (2024). Amazon EC2 User Guide for Linux Instances. https://docs.aws.amazon.com/ec2/
[7]. GitLab Documentation. (2024). GitLab Runner Setup and Configuration. https://docs.gitlab.com/runner/
[8]. Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press.
[9]. Sharma, D., et al. (2020). Continuous Integration and Deployment: Automation for Faster Software Delivery. International Journal of Computer Applications, 175(12), 1-6.
[10]. DigitalOcean Tutorials. (2024). Deploying Docker Containers on Ubuntu. https://www.digitalocean.com/community/tutorials

