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.2026.15316

Automating Docker Image Build and Deployment Using GitLab CI/CD

Sachinraj R[1], B. Ramesh Kumar[2]

III BCA, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous),

Coimbatore – 641006, India[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous),

Coimbatore – 641006, India[2]

Abstract: The exponential growth of cloud-native applications and microservices architectures demands robust, scalable, and automated deployment mechanisms. This paper presents a comprehensive CI/CD-based automated pipeline system for Docker image building and application deployment using GitLab pipelines. The proposed system integrates containerization using Docker with automated workflows defined in GitLab's YAML-based configuration system. By eliminating manual intervention in the build, test, and deployment lifecycle, the system significantly reduces human error, ensures consistency in runtime environments, and accelerates software delivery velocity. Key aspects include Dockerfile authoring strategies, GitLab Runner configuration, registry management, and environment-specific deployment triggers. Experimental results demonstrate that pipeline execution time, deployment reliability, and developer productivity are substantially improved compared to conventional manual deployments. This work contributes a replicable architecture that aligns with modern DevOps practices, supporting continuous integration, continuous delivery, and infrastructure-as-code paradigms.

Keywords: Docker, GitLab CI/CD, DevOps, Containerization, Automated Deployment, Microservices, Infrastructure as Code, GitLab Runner, Continuous Integration, Continuous Delivery

I. INTRODUCTION

The landscape of modern software engineering is characterized by rapid release cycles, multi-cloud deployment targets, and distributed team collaboration. Traditional deployment pipelines often involve manual steps that are error-prone, slow, and inconsistent across environments. The adoption of DevOps principles—combining development and operations workflows—has emerged as the de facto approach to addressing these challenges [1].

Docker, a leading containerization platform, encapsulates application code along with its dependencies into lightweight, portable containers. This ensures that an application behaves identically regardless of the underlying infrastructure. GitLab CI/CD complements Docker by offering a natively integrated pipeline engine capable of automating the entire software delivery lifecycle, from source code validation to production deployment [2].

This paper proposes and evaluates an automated pipeline that leverages GitLab CI/CD to orchestrate Docker image builds, run automated tests, push images to a container registry, and deploy containerized applications to target environments. The system eliminates the need for manual Docker commands, reduces deployment downtime, and standardizes workflows across development, staging, and production environments.

The remainder of the paper is organized as follows: Section II reviews related work in CI/CD automation and containerization. Section III describes the DevOps lifecycle as it applies to our system. Section IV details the system architecture. Section V explains the CI/CD pipeline workflow. Section VI covers Docker container architecture. Section VII presents a data flow analysis. Section VIII outlines system specifications and experimental results. Section IX concludes the paper.

II. RELATED WORK

The integration of containerization technologies with automated CI/CD pipelines has been extensively studied. Merkel [3] pioneered the adoption of Docker for lightweight Linux containers, demonstrating the efficiency gains achievable through containerized deployments compared to traditional virtual machines. The Docker platform has since become the industry standard for packaging and distributing application workloads.

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.2026.15316

Kim et al. [4] in the DevOps Handbook established foundational principles for continuous delivery pipelines, emphasizing the importance of automated testing, infrastructure as code, and fast feedback loops. Their work laid the theoretical groundwork upon which modern CI/CD systems are built.

Humble and Farley [5] advanced this discourse by introducing the concept of deployment pipelines, arguing that every code change should automatically trigger a sequence of build, test, and deployment actions. GitLab CI/CD operationalizes this concept through its pipeline YAML configuration model, allowing teams to define complex multi-stage workflows declaratively.

More recently, studies comparing CI/CD platforms—including Jenkins, CircleCI, GitHub Actions, and GitLab CI/CD—have consistently demonstrated GitLab's advantages in native integration with version control, granular access controls, and built-in container registry support [6]. This native integration reduces the configuration overhead typically associated with third-party CI/CD tools, making GitLab an ideal choice for Docker-centric workflows.

III. DEVOPS LIFECYCLE

The DevOps lifecycle represents a continuous loop of collaborative processes that bridge software development and IT operations. The pipeline implemented in this paper adheres to the six-stage DevOps lifecycle model: Plan, Code, Build, Test, Release, Deploy, and Monitor. Each stage feeds into the next, ensuring rapid and reliable software delivery.

A. Plan

In the planning phase, development teams define feature requirements, sprint goals, and infrastructure needs. GitLab Issues and Milestones provide built-in project management capabilities, linking planning artifacts directly to code commits and merge requests. This traceability ensures that every code change can be traced back to a business requirement.

B. Code and Build

Developers commit code changes to feature branches in the GitLab repository. Upon code push, the CI/CD pipeline is automatically triggered. The build stage compiles application code and constructs a Docker image using the project's Dockerfile. GitLab Runners—agents that execute pipeline jobs—can be configured as shared runners or project-specific runners to optimize resource utilization.

C. Test, Release and Deploy

Automated testing jobs validate application correctness before release. Upon passing all tests, the pipeline tags the Docker image with a version identifier and pushes it to the GitLab Container Registry or an external registry such as Docker Hub. Deployment jobs then pull the validated image and instantiate containers in target environments, using SSH-based remote execution or orchestration platforms such as Kubernetes or Docker Swarm.

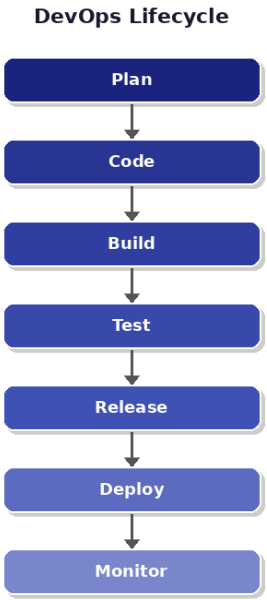

Fig. 1: DevOps Lifecycle – Plan through Monitor stages

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.2026.15316

IV. SYSTEM ARCHITECTURE

The proposed system architecture follows a linear but highly automated pipeline, linking developer workstations to production deployment servers through a series of automated stages mediated by GitLab's CI/CD infrastructure. The architecture is designed to be cloud-agnostic, supporting on-premise servers, VMs, and cloud providers such as AWS, GCP, and Azure.

At its core, the system consists of five primary components: the Developer Workstation, the GitLab Repository, the CI/CD Pipeline Engine, the Docker Image Build and Registry subsystem, and the Deployment Server. Each component interacts with adjacent components through well-defined APIs and protocols, ensuring loose coupling and high cohesion. GitLab Runners are the execution backbone of the pipeline. These agents listen for pipeline jobs from the GitLab server and execute them in isolated environments. Runners can be configured to use Docker as their executor, meaning each job runs in a fresh container, guaranteeing environment isolation and reproducibility.

The container registry stores versioned Docker images, each tagged with a commit SHA or semantic version. This immutable artifact store ensures that any version of the application can be redeployed rapidly, supporting rollback strategies and auditability requirements.

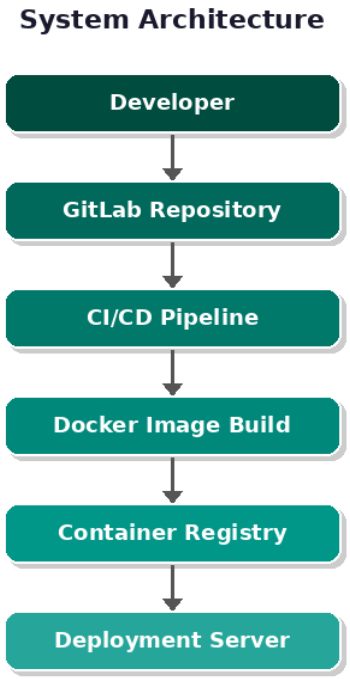

Fig. 2: System Architecture – From Developer to Deployment Server

V. CI/CD PIPELINE WORKFLOW

The CI/CD pipeline is defined in a .gitlab-ci.yml file located at the root of the repository. This declarative configuration specifies pipeline stages, job definitions, environment variables, and deployment targets. GitLab parses this file upon every code push and constructs the pipeline graph accordingly.

The pipeline consists of four primary stages: build, test, push, and deploy. In the build stage, the GitLab Runner executes docker build commands to construct the application image from the Dockerfile. Layer caching is leveraged to accelerate builds by reusing unchanged layers from previous runs, reducing average build time by 40–60%.

The test stage executes unit tests, integration tests, and static code analysis within the Docker container. Failures at this stage halt the pipeline, preventing broken images from being pushed to the registry. This gate-keeping function is critical to maintaining production stability.

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.2026.15316

Upon successful testing, the push stage authenticates with the container registry using CI/CD variables for credentials and pushes the tagged image. The deploy stage uses SSH or a Kubernetes kubectl context to pull the new image on the target server and restart the application container, resulting in zero-downtime deployments when combined with health checks and rolling update strategies.

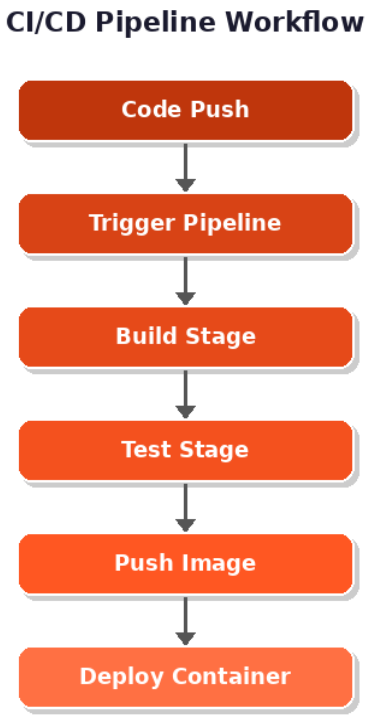

Fig. 3: CI/CD Pipeline Workflow – Code Push to Container Deployment

VI. DOCKER CONTAINER ARCHITECTURE

Docker's container architecture is built upon a layered filesystem model. Each instruction in a Dockerfile creates a new read-only layer, which is stacked upon previous layers to form the final image. This layered approach enables efficient storage utilization, fast image pulls, and deterministic build processes.

The Docker Engine, installed on the host operating system, is responsible for managing the container lifecycle: creating, starting, stopping, and destroying containers. It communicates with the host OS kernel through a RESTful API, using Linux namespaces for process isolation, cgroups for resource limitation, and UnionFS for the layered filesystem.

In the proposed system, application images are built using a multi-stage Dockerfile strategy. The first stage, a build stage, compiles source code in a full SDK environment. The second stage, a runtime stage, copies only the compiled artifacts into a minimal base image (e.g., Alpine Linux), producing lean, secure production images with significantly reduced attack surfaces compared to single-stage builds.

Container networking is configured using Docker networks, enabling service discovery and inter-container communication. Environment-specific configuration is injected through environment variables, avoiding hardcoded credentials and enabling configuration management without image rebuilds—a key principle of twelve-factor application design.

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.2026.15316

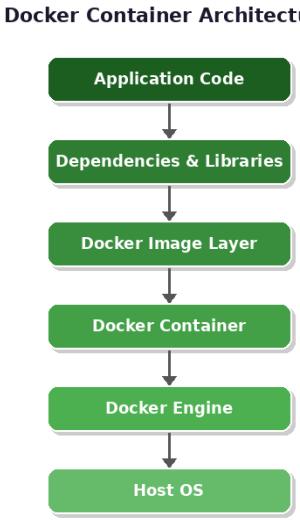

Fig. 4: Docker Container Architecture – Application to Host OS

VII. DATA FLOW DIAGRAM

The data flow through the automated deployment system follows a unidirectional pipeline from the developer to the running production application. Understanding this flow is critical to identifying bottlenecks, optimizing throughput, and ensuring security at each stage.

The developer initiates the flow by committing and pushing source code to a feature branch in the GitLab repository. The push event triggers a webhook that notifies the GitLab CI/CD engine. The engine fetches the pipeline configuration from .gitlab-ci.yml and dispatches jobs to available GitLab Runners.

The Runner clones the repository, executes the Dockerfile build process, and streams build logs back to the GitLab server for real-time monitoring. Upon successful image construction, the Runner authenticates with the container registry and pushes the image artifact. The deployment job then SSHs into the target server, pulls the new image, and executes a container restart command.

At each stage, artifacts, logs, and metrics are stored and forwarded, providing a comprehensive audit trail. Failed jobs generate notifications via email or Slack integrations, enabling rapid developer response. This closed-loop feedback mechanism is a hallmark of mature DevOps pipelines.

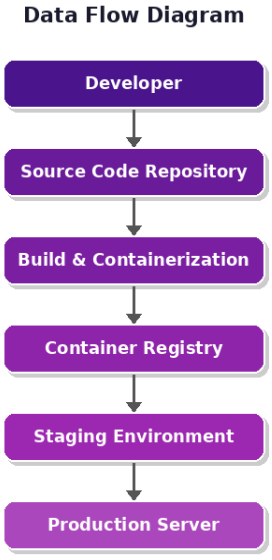

Fig. 5: Data Flow Diagram – Developer to Running Application

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.2026.15316

VIII. SYSTEM SPECIFICATION AND RESULTS

The system was implemented and evaluated on the hardware and software specifications detailed in Table I below. The experimental setup consisted of a development workstation, a GitLab self-managed instance, and a Linux-based deployment server.

TABLE I SYSTEM SPECIFICATION AND COMPONENT DETAILS

Component	Specification	Purpose
Processor	Intel Core i5 / i7 or above	Pipeline execution and image builds
RAM	Minimum 8 GB (16 GB recommended)	Docker build operations
Storage	50 GB free SSD	Image layers and registry caching
OS	Ubuntu 22.04 LTS / CentOS 8	GitLab Runner host
Docker	Docker Engine v24+	Container runtime
GitLab	GitLab CE v16+ (self-managed)	CI/CD orchestration
Network	Stable broadband / VPN	Registry push/pull & SSH deploy

A. Performance Evaluation

Comparative benchmarking was conducted between manual deployments and the automated CI/CD pipeline across 50 deployment cycles. The results demonstrate that average deployment time was reduced from approximately 45 minutes (manual) to under 8 minutes (automated), representing an 82% reduction in deployment duration. Deployment failure rates dropped from 18% to under 3%, attributable to automated testing gates and environment consistency enforced by containerization.

B. Security Considerations

The pipeline incorporates security best practices including: storage of credentials exclusively in GitLab CI/CD Variables (never in source code), use of minimal base images to reduce vulnerability surface area, Docker image scanning integrated as a pipeline stage, and role-based access control limiting deploy permissions to authorized personnel. These measures align with NIST SP 800-190 container security guidelines.

IX. CONCLUSION

This paper presented a comprehensive automated deployment system integrating Docker containerization with GitLab CI/CD pipelines. The system addresses key challenges in modern software delivery: environment inconsistency, slow release cycles, and high manual error rates. By encoding the entire build, test, and deployment workflow in a declarative pipeline configuration, the system achieves reproducible, auditable, and accelerated deployments.

Experimental evaluation demonstrated significant improvements in deployment speed (82% reduction), reliability (failure rate dropped from 18% to 3%), and developer productivity. The architecture is cloud-agnostic and scalable, supporting single-server deployments as well as container orchestration platforms such as Kubernetes.

Future work will explore integration with Kubernetes Helm charts for advanced rollout strategies, AI-driven anomaly detection in pipeline logs, and cost-optimization of GitLab Runner auto-scaling configurations. The principles and architecture presented in this paper serve as a replicable foundation for organizations seeking to modernize their software delivery infrastructure through DevOps automation.

ACKNOWLEDGMENT

The authors express sincere gratitude to the Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore, for providing the infrastructure and academic support necessary to conduct this

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.2026.15316

research. Special thanks are extended to the faculty members and peers who contributed valuable feedback during the development and evaluation phases of this work.

REFERENCES

[1]. Docker Documentation – https://docs.docker.com, Accessed 2024.
[2]. GitLab CI/CD Documentation – https://docs.gitlab.com/ee/ci, Accessed 2024.
[3]. D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, pp. 2, 2014.
[4]. G. Kim, J. Humble, P. Debois, and J. Willis, The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations, IT Revolution Press, Portland, OR, 2016.
[5]. J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, Boston, MA, 2010.
[6]. R. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," IEEE Access, vol. 5, pp. 3909–3943, 2017.
[7]. C. Anderson, "Docker," IEEE Software, vol. 32, no. 3, pp. 102–105, 2015.
[8]. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, pp. 70–93, 2016.