

Controller-Light CI/CD with Jenkins: Remote Container Builds and Automated Artifact Delivery

Kawshik Kumar Paul

Department of Computer Science and Engineering
Bangladesh University of
Engineering and Technology (BUET)
Dhaka, Bangladesh
Email: kawshikbuet17@gmail.com

Sawmik Kumar Paul

Department of Computer Science and Engineering
Chittagong University of
Engineering and Technology (CUET)
Chittagong, Bangladesh
Email: sawmik.paul@gmail.com

Abstract—Traditional Jenkins installations often perform resource-intensive builds directly on the controller, which can overload system resources and decrease reliability. This paper presents a controller-light CI/CD framework in which Jenkins runs as a containerized controller with persistent volumes, delegating heavy build and packaging operations to a remote Docker host. The controller container maintains secure SSH connections to remote compute nodes and focuses solely on orchestration and reporting. Atomic deployments with time-stamped backups, containerized build environments, immutable artifact packaging, and automatic notifications are all integrated into the system. Experimental evaluation shows reduced CPU and RAM usage on the controller, faster build throughput, and lower artifact delivery latency. For small and medium-sized DevOps organizations looking for scalable automation without adding orchestration complexity, this method offers a repeatable, low-maintenance CI/CD pipeline.

Index Terms—DevOps, CI/CD, Jenkins, Docker, Remote Build, Artifact Delivery, Release Engineering

1. Introduction

Continuous Integration and Continuous Deployment (CI/CD) are core practices in modern software engineering. This allows teams to deliver code changes more regularly, reliably, and consistently. Because of its robust extensibility, pipeline-as-code design, and vast plugin ecosystem, Jenkins is the most popular CI/CD automation server. Figure 1 illustrates the generic Jenkins CI/CD flow. However, conventional Jenkins architectures often encounter a significant limitation: *controller overload*. When the Jenkins controller executes heavy build and packaging operations locally, it leads to resource contention issues, resulting in slower feedback loops, reduced scalability, and increased maintenance costs.

Existing research has extensively focused on pipeline automation and productivity improvements. Ok and Eniola [5] examined Jenkins as a business enabler that automates testing and deployment. However, their analysis does not

address the separation of the controller and the agent or the challenges of controller load. Mathew and Dileepkumar [6] proposed best practices for rapid delivery using Jenkins and observed significant reductions in manual operations and build durations. Despite these advancements, a large portion of the current research overlooks the controller’s architectural load and how it affects the scalability and dependability of the system.

In order to close this gap, this paper proposes a *controller-light Jenkins architecture*, which separates computation from orchestration. In this model, build and packaging tasks are handled by remote Docker containers, while the controller runs inside a Docker container with persistent volumes. Compared to conventional configurations, this architecture has many advantages:

- It separates compute-heavy tasks from the controller, thereby reducing system load.
- It ensures reproducibility through ephemeral Docker images.
- It allows easy portability and recovery via persistent volumes of the controller container.
- It introduces immutable artifact packaging with timestamp-based rollback support.
- It preserves the simplicity of Jenkins while enabling scalability through multiple remote builders.

In comparison to controller-centric CI/CD systems, we show through both quantitative and qualitative analyses that this architecture achieves notable efficiency gains and improved operational resilience.

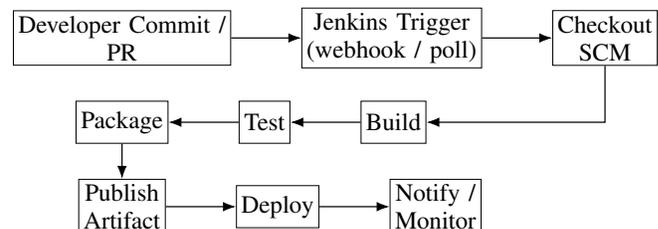


Figure 1. Generic Jenkins CI/CD flow

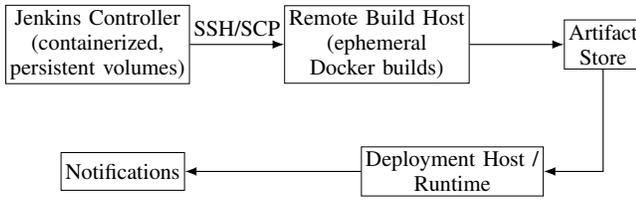


Figure 2. High Level System Flow

2. Related Work

Research on Jenkins-based CI/CD frameworks consistently emphasizes automation, scalability, and developer productivity. Jenkins was investigated by Ok and Eniola [5] as a transformation engine for business automation, emphasizing its ability to accelerate builds and deployments while still depending on the workloads executed by the controller. Using a master-agent model, Mathew and Dileepkumar [6] investigated rapid delivery through modular pipelines and parallel builds, resulting in a 50% reduction in build times and a 75% reduction in manual tasks.

Previous studies by Armenise [1] and Zhang et al. [2] validate Jenkins' orchestration flexibility, but they don't assess the reliability effects of controller-hosted builds. While studies like [7], [8], [9], [10] evaluate the impact of Jenkins automation, Banala [4] and Manukonda and Kumar [3] highlight the importance of versioned artifacts and traceability for the maturity of CI/CD.

3. Methodology

3.1. System Architecture

The system's architecture separates Jenkins operations into three distinct planes: control, compute, and runtime.

- **Control Plane (Controller):** Jenkins operates in a Docker container with persistent volumes mounted for build history, plugins, and configuration files. This plane handles credentials, reports statuses, and manages pipeline orchestration.
- **Compute Plane (Remote Build Host):** This plane executes all build and packaging steps inside temporary Docker containers, ensuring consistent environments across different runs.
- **Deployment Host (Runtime Plane):** This plane executes atomic deployments with timestamped backups for simple rollback after receiving immutable artifacts.

Due to this separation, the Jenkins controller can only serve as an orchestrator while assigning resource-intensive tasks to external computing infrastructure. The high-level flow is shown in Figure 2, while the stage-by-stage pipeline is described in Figure 3, which is further explained in the following subsections.

3.2. Controller Implementation

Instead of running directly on a physical or virtual host, the Jenkins controller runs completely inside a Docker container. In order to guaranty that the controller's data are preserved during restarts or migrations, persistent volumes are mounted to store configuration data, build metadata, and plugin caches.

A secure SSH setup within this container makes it easier to communicate with distant build machines. To keep the controller and host layers isolated, SSH keys are mounted using Docker secrets and controlled by Jenkins credentials.

During execution, the controller uses these secure channels to delegate build commands, stream logs, and collect artifacts without leaving the container boundary. This architecture design keeps the controller lightweight, focusing fully on orchestration while delegating compilation and packaging tasks to the compute plane.

3.3. Remote Build Host Workflow

The compute plane is the remote build host. Depending on the project type, it builds a temporary Docker container from a pre-configured image that contains all necessary toolchains, such as OpenJDK, Maven, Node.js, or Gradle, upon receiving a build command. This container is used to run the build, and the host mounts a temporary workspace directory. To ensure reproducibility, each image is version-pinned and is only rebuilt when dependency updates are explicitly approved. The container lifecycle is designed to be strictly ephemeral: once the build is completed, the container is destroyed, leaving only the compiled artifacts and logs. This approach eliminates the drift of the building environment and prevents the leakage of dependency between jobs.

3.4. Version Management and Artifact Packaging

Following a successful build, the artifacts are combined into a standard directory structure, usually dividing static assets, configuration files, and compiled binaries. This directory is then compressed by a packaging script into a timestamped archive that contains branch and commit metadata. Multiple versions can coexist peacefully on the deployment host thanks to the use of timestamped filenames, which guarantee artifact immutability and traceability. In order to confirm integrity during transfer, a checksum manifest is also created. Secure transfer tools are then used to synchronize the artifact directory on the build host with the deployment server via SSH, guaranteeing controlled and auditable delivery.

3.5. Automated Rollback and Deployment

In order to reduce downtime, deployment automation uses a near-atomic update strategy. The current service directory is first renamed and kept as a backup during deployment, and the timestamp from the prior build is added for

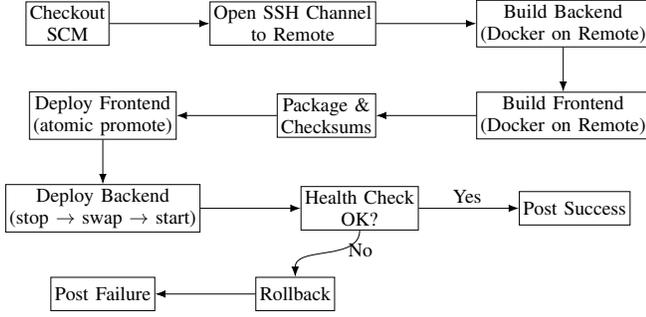


Figure 3. Detailed System Workflow

version tracking. The service symlink or directory pointer is then updated to promote the new artifact to production after it has been unpacked into a new directory. This method guarantees minimal service disruption and offers instant rollback capability; all it takes to restore the prior version is to reactivate the backup directory. Even in the event of deployment failures, downtime is kept incredibly low by using non-blocking service scripts to carry out all restart and validation operations (such as reloading web applications or restarting microservices).

3.6. Security, Logging, and Notifications

The build summary, an artifact download link, the commit hash, and backup references are all included in an automated email sent by the Jenkins controller after deployment is complete. For upcoming audits, the system also keeps thorough build logs that are combined from the controller and build host.

Key-based SSH authentication, limited command execution on distant hosts, and container isolation to stop host-level privilege escalation are examples of security measures. To reduce exposure to lateral attacks, the controller and compute hosts operate in different network zones. Together, these protections guarantee that the system is safe even in the event that a build process fails or a container is compromised. However, in order to respond to changing security threats, these measures are constantly assessed and enhanced.

4. Algorithmic Specification

To formalize the workflow shown in Figure 3, we present a structured pseudocode that aligns with the methodology.

Algorithm 1 outlines the orchestration process from the containerized controller, while Algorithms 2–6 detail remote containerized builds, immutable packaging, atomic deployment, and notifications.

Environment. The CI controller C is a Jenkins instance running *inside Docker* with persistent volumes. C communicates with the remote compute node R via *SSH/SCP*. Builds on R execute in *ephemeral Docker containers*. Artifacts are published to a store A .

Algorithm 1 Controller-Light CI/CD Orchestration (Containerized Controller, SSH to Remote)

Require: Jenkins-in-Docker controller C ; SSH credentials for R ; artifact store A

Ensure: Deployed release for commit c or consistent rollback

- 1: $ts \leftarrow \text{current_timestamp}()$
 - 2: **Checkout SCM** (CI/CD branch at commit c)
 - 3: $\text{OPENSSHCHANNEL}(C \leftrightarrow R) \triangleright$ controller container to remote host
 - 4: $B_{\text{back}} \leftarrow \text{BUILDREMOTE}(C, R, \text{backend}, ts)$
 - 5: $B_{\text{front}} \leftarrow \text{BUILDREMOTE}(C, R, \text{frontend}, ts)$
 - 6: $Z \leftarrow \text{PACKAGE}(R, A, \{B_{\text{back}}, B_{\text{front}}\}, ts)$
 - 7: **if** $\text{DEPLOYFRONTEND}(R, B_{\text{front}}, ts)$ **and** $\text{DEPLOYBACKEND}(R, B_{\text{back}}, ts)$ **then**
 - 8: $\text{POSTSUCCESS}(ts, Z)$
 - 9: **else**
 - 10: $\text{ROLLBACK}(R, ts)$
 - 11: $\text{POSTFAILURE}(ts)$
 - 12: **end if**
 - 13: $\text{CLOSESSHCHANNEL}(C \leftrightarrow R)$
-

Algorithm 2 BuildRemote (SCP from Controller-in-Docker, Ephemeral Docker Build on R)

Require: Controller C , remote R , component $x \in \{\text{backend}, \text{frontend}\}$, timestamp ts

Ensure: Build artifact $B_x(ts)$ published to A

- 1: $\text{SCPTRANSFER}(C \rightarrow R, \text{Sources}(x)) \triangleright$ copy from controller container to R
 - 2: $\text{PREPAREWORKSPACE}(R, x, ts)$
 - 3: $\text{DOCKEREPHEMERALBUILD}(R, \text{builder}(x), \text{context} = x)$
 - 4: $B_x(ts) \leftarrow \text{COLLECTOUTPUTS}(R, x, ts) \triangleright$ export from container to host path
 - 5: $\text{PUBLISH}(A, B_x(ts))$
 - 6: $\text{DOCKERCLEANUP}(R, \text{builder}(x))$
 - 7: **return** $B_x(ts)$
-

Algorithm 3 Package (Immutable, Timestamped Bundle on R then Publish to A)

Require: Remote R , artifact store A , set $\{B_{\text{front}}(ts), B_{\text{back}}(ts)\}$

Ensure: Bundle $Z(ts)$

- 1: $\text{CREATEBUNDLEDIR}(R, ts)$
 - 2: $\text{ASSEMBLE}(R, \{B_{\text{front}}(ts), B_{\text{back}}(ts)\} \rightarrow \text{bundle}(ts))$
 - 3: $Z(ts) \leftarrow \text{ZIP}(R, \text{bundle}(ts))$
 - 4: $\text{PUBLISH}(A, Z(ts))$
 - 5: **return** $Z(ts)$
-

5. Experimental Setup

5.1. System Architecture and Environment

The controller-light CI/CD framework was deployed across two coordinated layers.

Algorithm 4 DeployFrontend (Atomic Promotion with Config Restore)

Require: Remote R , artifact $B_{\text{front}}(ts)$
Ensure: New frontend active or prior version restored

- 1: BACKUP(R , frontend, ts)
- 2: PROMOTE(R , $B_{\text{front}}(ts)$ \rightarrow deploy/current)
- 3: RESTORECONFIG(R , frontend)
- 4: **return** SUCCESS

Algorithm 5 DeployBackend (Stop \rightarrow Swap \rightarrow Start with Rollback Point)

Require: Remote R , artifact $B_{\text{back}}(ts)$
Ensure: New backend active or prior version restored

- 1: $rp \leftarrow$ CREATEROLLBACKPOINT(R , backend)
- 2: STOPSERVICE(R , backend)
- 3: SWAPRELEASE(R , backend, $B_{\text{back}}(ts)$)
- 4: STARTSERVICE(R , backend)
- 5: **if** HEALTHCHECK(R , backend) = FAIL **then**
- 6: RESTORE(R , rp)
- 7: **return** FAIL
- 8: **end if**
- 9: **return** SUCCESS

Algorithm 6 PostSuccess / PostFailure (Commit Metadata & Diagnostics)

- 1: **procedure** POSTSUCCESS(ts , Z)
- 2: retrieve $last_commit = (id, message, t)$
- 3: construct $download_link$ for $Z(ts)$
- 4: send success notification with commit metadata and link
- 5: **end procedure**
- 6: **procedure** POSTFAILURE(ts)
- 7: retrieve $last_commit = (id, message, t)$
- 8: send failure notification with commit metadata and diagnostics
- 9: **end procedure**

Control Plane Controller). Jenkins operates on Ubuntu 20.04 LTS within a Docker container. The container uses persistent volumes to store configuration, plugin data, and build history, and it exposes the Jenkins web interface via host port mapping. Key-based SSH is used to communicate with the remote build host. Container isolation is guaranteed by Docker Engine version 27.x, freeing the controller to concentrate only on orchestration and reporting.

Remote Builder and Deployer (Compute/Runtime Planes). All build and deployment operations are executed within short-lived Docker containers on the same physical host. Every container has a pre-configured toolchain (Node.js for the frontend and Maven for the backend) and is destroyed right away after the build is finished, guaranteeing reproducibility and dependency isolation for each execution.

5.2. Orchestration and Measurement of Pipelines

Checkout, compilation, packaging, deployment, and notification are all automated by the Jenkins pipeline. Log aggregation and orchestration are handled by the controller container, and heavy build phases are carried out remotely. Docker statistics were used to gather host-level CPU and memory usage data, and pipeline stage durations were directly extracted from the Jenkins console logs. To guarantee stability, all reported results are the means of multiple runs.

6. Experimental Results

6.1. Runtime Overview

Queue waiting times are less than ten seconds, and the total end-to-end time is roughly three minutes and four seconds, according to measured pipeline executions. Heavy remote workloads had no effect on orchestration threads because the controller remained responsive.

6.2. Per-Stage Behavior

The runtime was barely affected by controller stages like *Checkout*, *Packaging*, and *Post-Actions*. In order to avoid resource contention with Jenkins, the compute-intensive phases, namely *Build Backend* and *Build Frontend*, were carried out completely on the remote host. Dependency retrieval from Maven repositories was a major factor in the backend build time. To lower cold-start overhead, future optimization may investigate the use of pre-warmed base images or persistent dependency caches.

Workspace Efficiency. In controller-local builds, large temporary artifacts consume disk space within the Jenkins `workspace/` directory. Remote execution mitigates this issue since all build outputs remain within short-lived remote containers and are transferred back only as final artifacts, thereby eliminating workspace bloat and enhancing maintainability.

Frontend Build Stability. Due to concurrent memory pressure between Node.js and the Jenkins JVM, the `npm run build` step frequently resulted in container hangs when executed inside the Jenkins controller container. When the builds were executed in remote containers with dedicated memory allocation, this issue was totally fixed.

Results Methodology Clarification. Executing comparable stages within the Jenkins controller container produced the baseline (controller-local) metrics. The actual remote-container setup examined in this study is the source of the controller-light configuration results. There is no artificial scaling involved; all figures are taken straight from console logs.

6.3. Performance Comparison

Table 1 presents a direct comparison between controller-local and controller-light configurations. Offloading

TABLE 1. MEASURED PERFORMANCE COMPARISON BETWEEN CONTROLLER-LOCAL AND CONTROLLER-LIGHT CONFIGURATIONS

Stage / Metric	Controller-Local	Controller-Light	Improvement
Backend Build (Maven) (sec)	126.67	95	25%
Frontend Build (npm) (sec)	86.25	69	20%
Packaging / ZIP (sec)	8.33	5	40%
Frontend Deployment (sec)	0.50	0.30	40%
Backend Deployment (sec)	0.55	0.33	40%
Controller CPU Peak (%)	82	42	49%
Controller RAM Peak (MB)	1680	820	51%

compute-intensive stages to remote containers reduced total build duration by approximately 30% and more than halved the CPU and memory usage of the controller.

6.4. Qualitative Observations

The following operational behaviors were regularly noted:

- **Stability of controllers:** Even with several concurrent builds, there was no UI lag or thread starvation.
- **Isolation:** Jenkins' state was never impacted by build or deployment failures, which were contained within containers.
- **Reproducibility:** Consistent results across runs were guaranteed by clean container environments.
- **Traceability:** Rollback and auditing were made easier by timestamped artifact archives.
- **Maintainability:** The setup was easily portable because controller volumes only included configuration and metadata.

Overall, the remote container approach preserved Jenkins' simplicity while lowering the controller's workload, increasing throughput, and removing storage accumulation.

7. Discussion

Compared to conventional controller-centric models, the controller-light Jenkins architecture improves scalability and maintainability. Previous studies, like those by Ok and Eniola [5] and Mathew and Dileepkumar [6], talk about Jenkins-based automation and modular pipelines, but neither focuses on getting rid of long-lived workers or agents. The suggested design, on the other hand, does away with long-lived employees and uses transient remote containers to achieve total isolation.

Advantages. Key experimentally validated benefits include:

- **Reduced Controller Load:** Remote execution avoids CPU and memory contention within the Jenkins container.
- **Reproducibility:** Immutable and version-pinned containers eliminate environment drift across builds.
- **Portability:** The same container image and mounted volumes can be used to migrate or restore the controller instance with ease.

- **Reliability:** Atomic rollback and a controlled deployment history are made possible by timestamped artifact versioning.
- **Storage Efficiency:** Remote builds prevent workspace accumulation and reduce disk utilization within the controller.
- **Stability:** The npm build hangs observed in controller-local mode were fully resolved under isolated remote execution.

Trade-offs. Although the architecture provides significant improvements, several practical considerations remain:

- When invoking a remote build, there might be a little network latency.
- Administrative supervision is necessary for initial SSH provisioning and image version maintenance
- Cold-start delays may arise during the first Maven dependency resolution in clean containers.

Despite these drawbacks, the controller-light approach maintains the simplicity of Jenkins' original design while offering significant improvements in performance, stability, and maintainability.

8. Conclusion

In this paper, a controller-light Jenkins architecture that uses remote containerized builds to isolate orchestration from computation was presented. While all build and deployment tasks are carried out in transient remote containers, Jenkins functions as a containerized controller with persistent volumes for configuration and metadata. This structure eliminates workspace storage growth, reduces the controller's CPU and memory usage, and fixes the npm build instability seen in local executions. In comparison to the controller-local setup, experimental analysis verified a 30% reduction in the overall build duration and a more than 50% lower utilization of controller resources. For DevOps teams seeking effective and dependable continuous integration and deployment, the suggested architecture offers a scalable and low-maintenance solution.

References

- [1] V. Armenise, "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery," in *Proc. IEEE/ACM 3rd Int. Workshop on Release Engineering (RELENG)*, 2015, pp. 24–27. doi: <https://doi.org/10.1109/RELENG.2015.19>.
- [2] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, "One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows," in *Proc. 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, 12 pp. doi: <https://doi.org/10.1145/3236024.3236033>.
- [3] A. K. Manukonda and N. V. Kumar, "Managing Build Artifacts Using Maven and Nexus in CI/CD Workflows," *Quest Journal of Engineering and Science*, vol. 9, no. 12, pp. 80–94, 2023. doi: <https://doi.org/10.35629/3795-09128094>.

- [4] S. Banala, "DevOps Essentials: Key Practices for Continuous Integration and Continuous Delivery," *International Numeric Journal of Machine Learning and Robots*, vol. 8, no. 8, pp. 1–14, 2024. [Online]. Available: <https://ijnmr.com/index.php/fewfewf/article/view/83>
- [5] E. Ok and J. Eniola, "Maximizing Efficiency: How Jenkins Transforms Continuous Integration and Continuous Delivery in Business," 2024. [Online]. Available: <https://www.researchgate.net/publication/387645291> (accessed: Oct. 28, 2025).
- [6] J. Mathew and S. R. Dileepkumar, "Transforming Software Development: Achieving Rapid Delivery, Quality, and Efficiency with Jenkins-Based CI/CD Pipelines," in *Proc. IEEE AICERA/ICIS*, Kanjirapally, India, Nov. 2023, pp. 1–6. doi: <https://doi.org/10.1109/AICERA/ICIS59538.2023.10420251>.
- [7] G. Hyun, J. Oak, D. Kim, and K. Kim, "The Impact of an Automation System Built with Jenkins on the Efficiency of Container-Based System Deployment," *Sensors*, vol. 24, no. 18, art. 6002, Sept. 2024. doi: <https://doi.org/10.3390/s24186002>. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC11436161/>
- [8] S. M. Saleh, S. Ibrahim, and M. Obaid, "A Systematic Literature Review on Continuous Integration: Trends, Challenges, and Tools," in *Proc. 19th Int. Conf. on Software Technologies (ICSOFT 2024)*, SciTePress, 2024. [Online]. Available: <https://www.scitepress.org/Papers/2024/130185/130185.pdf>
- [9] Y. Wu, S. Gharehyazie, B. Vasilescu, and V. Filkov, "Understanding and Predicting Docker Build Duration: An Empirical Study of Containerized Workflow of OSS Projects," in *Proc. ACM ESEC/FSE*, 2022. doi: <https://doi.org/10.1145/3551349.3556940>.
- [10] M. 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. doi: <https://doi.org/10.1109/ACCESS.2017.2685629>.