Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2022, 9(9):112-118**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Achieving Zero-Downtime Deployment for Java Applications Using GitLab CI/CD

Praveen Kumar Koppanati

praveen.koppanati@gmail.com

Abstract: Zero-downtime deployment is a critical requirement for modern cloud-native applications, particularly in industries that require 24/7 availability. Achieving zero-downtime deployment ensures that updates or changes to applications do not disrupt user activity or lead to service outages. Java-based applications, widely used in enterprise environments, pose specific challenges in ensuring seamless deployment. This paper provides a comprehensive approach to implementing zero-downtime deployment using GitLab's Continuous Integration and Continuous Deployment (CI/CD) pipelines. We examine the strategies to orchestrate zero-downtime deployment for Java applications in a cloud-native ecosystem, leveraging containers, load balancers, blue-green deployments, and rolling updates. The integration of these strategies into GitLab CI/CD pipelines enables Java applications to be deployed without interruption to users. Case studies from large-scale enterprises demonstrate how such deployments are achieved using GitLab CI/CD and best practices to follow.

Keywords: Zero-downtime deployment, Java, GitLab CI/CD, blue-green deployment, rolling updates, containerization, cloud-native, load balancing, enterprise applications.

1. Introduction

Zero-downtime deployment has become a critical necessity for modern cloud-based applications. With the shift to cloud-native architectures, continuous delivery pipelines must ensure that new application versions can be deployed without affecting the availability or responsiveness of the system. Traditional deployment methods often result in application downtime, leading to disruptions in service and potential loss of revenue. This is particularly problematic for enterprise Java applications, which are extensively used in industries requiring near-constant availability.

This paper explores how zero-downtime deployment can be achieved for Java-based applications through the use of GitLab CI/CD pipelines. We delve into techniques like blue-green deployment, rolling updates, and the use of containers and load balancers, all while integrating these within a GitLab CI/CD framework. By adopting these strategies, enterprises can significantly reduce or eliminate the risk of service interruptions during deployments.

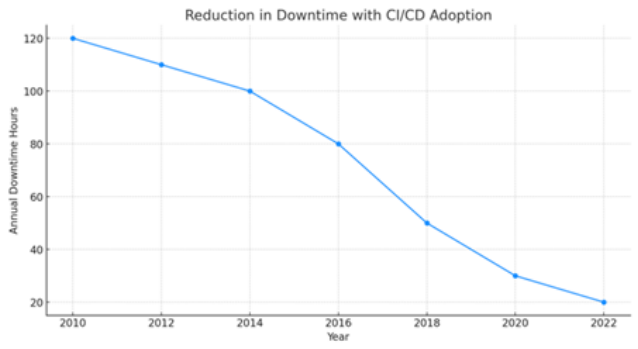


***Fig. 1** Reduction in Downtime with CI/CD Adoption*

2. Zero-Downtime Deployment in Java Applications

Challenges in Java Application Deployment:

Java applications are central to many enterprise systems due to their reliability and scalability. However, deploying these applications in a zero-downtime manner can be complex due to their monolithic nature in legacy systems or dependencies on various services. The following challenges are commonly encountered:

• **Session management:** User sessions are often disrupted during deployment, leading to loss of data and unsatisfactory user experience.

• **Database migrations:** Schema changes need careful handling to ensure compatibility between old and new application versions.

• **Service discovery and load balancing:** Effective routing of traffic between old and new instances of applications during deployment is essential to avoid downtime.

Blue-Green Deployment Strategy:

One of the primary methods to achieve zero-downtime deployment is the blue-green deployment strategy. This technique involves running two identical environments – one live (green) and one idle (blue). During the deployment of a new version, the blue environment is updated while the green environment continues serving the live traffic. Once the blue environment is verified, traffic is switched from green to blue.

For Java applications, blue-green deployments are highly beneficial, as they allow administrators to roll back to the previous version in case of failures. This can be implemented within GitLab pipelines using Kubernetes or container-based environments, as GitLab CI/CD integrates seamlessly with container orchestration platforms like Kubernetes.

Rolling Updates Strategy:

Rolling updates involve gradually replacing instances of the application one at a time, ensuring that at least some instances of the previous version remain available to handle incoming requests. This strategy reduces the risk of complete application failure due to deployment issues and ensures a smooth transition to the new version.

In a Java-based microservices architecture, rolling updates can be particularly effective. GitLab CI/CD pipelines can be configured to manage rolling updates through Kubernetes, with built-in health checks ensuring that new instances are only launched if they are fully functional. The gradual nature of rolling updates minimizes downtime risks and provides ample opportunity to rollback if issues arise.

3. GitLab CI/CD Pipeline for Java Applications

GitLab CI/CD offers a robust framework for automating the software release process. When configured correctly, GitLab can facilitate zero-downtime deployment for Java applications by integrating testing, deployment, and monitoring steps into a unified pipeline.

GitLab Pipeline Structure:

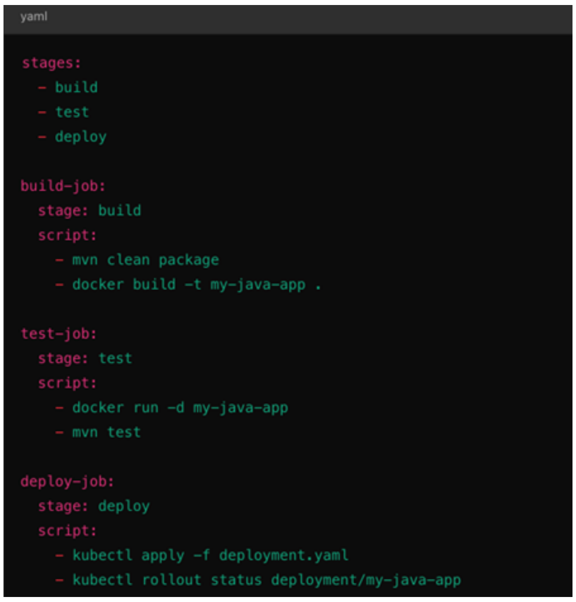
A typical GitLab pipeline for zero-downtime deployment consists of the following stages:

• **Build:** In this stage, the Java application is compiled, and unit tests are executed. The compiled application is then packaged, often into a Docker container.

• **Test:** This stage involves running integration and system tests to validate that the application works as expected.

• **Deploy:** This is the critical stage where zero-downtime deployment strategies are employed. Depending on the strategy, this could involve launching containers into a blue-green or rolling deployment process.

An example .gitlab-ci.yml configuration for a rolling update deployment might look like this:


yaml

```yaml
stages:
  - build
  - test
  - deploy

build-job:
  stage: build
  script:
    - mvn clean package
    - docker build -t my-java-app .

test-job:
  stage: test
  script:
    - docker run -d my-java-app
    - mvn test

deploy-job:
  stage: deploy
  script:
    - kubectl apply -f deployment.yaml
    - kubectl rollout status deployment/my-java-app
```

Kubernetes Integration with GitLab:
Kubernetes is a popular orchestration tool for managing containerized applications, and GitLab offers direct integration with Kubernetes clusters. This allows for automated deployment, scaling, and management of Java applications. By leveraging Kubernetes, Java applications can be deployed in a highly available and scalable manner, with GitLab CI/CD managing the entire process.

4. Zero-Downtime Deployment Strategies

Blue-Green Deployment:
Blue-green deployment is one of the most effective methods to ensure zero-downtime during the deployment of Java applications. The main idea behind this strategy is to maintain two separate but identical environments, traditionally named blue and green. Here’s how it works:
• **Dual Environment Setup:** Initially, one environment (let’s say green) serves the live production traffic, while the other (blue) remains idle or used for staging and testing.
• **New Version Deployment:** When a new version of the application is ready to be deployed, it is deployed to the blue environment, without impacting the production environment.
• **Testing:** After deployment, comprehensive testing is performed on the blue environment to ensure everything works correctly with the new version. This includes running automated tests as part of the GitLab CI/CD pipeline, as well as conducting manual validation if necessary.
• **Traffic Switch:** Once the blue environment has been verified, traffic is gradually rerouted from the green environment to the blue environment. In some implementations, this switch is instantaneous, but in more sophisticated setups, tools like load balancers or traffic management systems handle this process incrementally to minimize any potential risk.
• **Rollback:** If any issue is encountered after the traffic switch, rolling back to the previous version (now running in the green environment) is simple and quick, reducing the downtime risk significantly.
The blue-green deployment strategy offers numerous benefits, especially for enterprise Java applications that must operate continuously. By maintaining two isolated environments, it ensures a fail-safe mechanism. The risk, however, lies in resource usage, as maintaining two environments requires more infrastructure, and this can lead to higher costs.
Example Integration in GitLab CI/CD:
Using GitLab CI/CD, you can automate the entire blue-green deployment process. The pipeline first deploys to the blue environment, runs tests, and if all tests pass, triggers the traffic switch. Kubernetes can be used to manage the container orchestration for each environment, ensuring a smooth and automated transition between the two.

Rolling Updates:

Rolling updates provide a more gradual approach to zero-downtime deployment, particularly useful for large, distributed systems or microservices-based architectures.

• **Gradual Replacement:** Unlike blue-green deployment, rolling updates replace the running instances of the application one at a time. For example, if an application is running across 10 servers or containers, a rolling update will replace one instance with the new version while keeping the remaining instances online to handle traffic.

• **Incremental Rollout:** As each instance is updated, health checks are performed to ensure the new version is functioning correctly. If an instance fails the health checks, the update is paused, and the previous version continues to serve the traffic, thus minimizing risk.

• **No Need for Dual Environments:** One of the key advantages of rolling updates over blue-green deployment is that they do not require maintaining two separate environments. The new version is simply rolled out in place, reducing infrastructure costs.

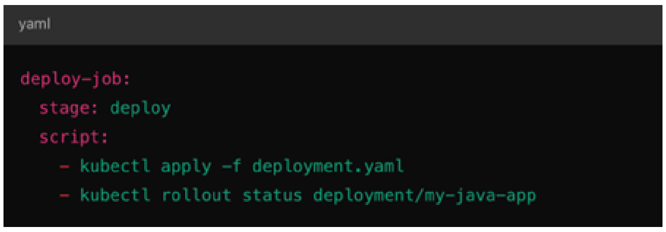
• **Minimal Impact:** Because traffic continues to flow to the other active instances while the new version is deployed, the impact on users is minimal. The strategy allows for incremental risk management and fine-tuning during the deployment process.

• **Failure Handling and Rollback:** In the event of an issue, only the affected instances are impacted, making rollback relatively easy and faster since you don't have to switch entire environments like in blue-green deployment.

Rolling updates are highly efficient for cloud-native Java applications running on container orchestration platforms like Kubernetes. By updating only a subset of containers at a time, this method helps to ensure continuous availability.

Example Integration in GitLab CI/CD:

In GitLab CI/CD, rolling updates can be defined through Kubernetes manifests in the deployment configuration file (deployment.yaml). The pipeline triggers an update to one pod at a time, ensuring that other pods remain operational. Here's a simplified GitLab CI/CD configuration snippet for rolling updates:



```yaml
deploy-job:
  stage: deploy
  script:
    - kubectl apply -f deployment.yaml
    - kubectl rollout status deployment/my-java-app
```

In this configuration, kubectl apply updates the deployment, and kubectl rollout status checks that the new instances are healthy before continuing to the next one.

Canary Releases:

A canary release is a deployment strategy where the new version of an application is initially released to a small subset of users. This technique is particularly effective for reducing risk while deploying new features or changes. If the canary version works well for this limited group, it can be gradually rolled out to the rest of the users. Otherwise, the deployment can be halted or rolled back without affecting the entire user base.

• **Small-Scale Deployment:** The new application version is deployed to a small number of instances or to specific segments of users. These users will experience the new version, while the majority will still be using the previous version.

• **Monitoring and Feedback:** The key to a successful canary release is continuous monitoring of the canary users. Metrics such as performance, errors, and user behavior are analyzed to detect any issues early. Tools like Prometheus and Grafana can be used to monitor the health and performance of the new version in real-time.

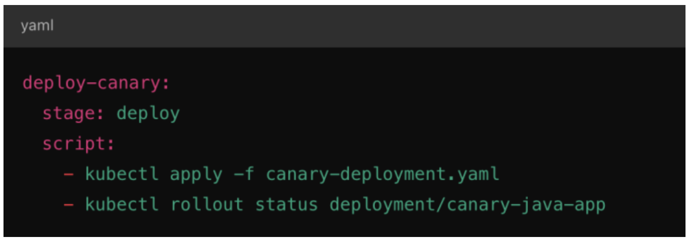
• **Gradual Expansion:** If the canary release proves successful, more users or instances are gradually updated. This ensures a smooth transition without overwhelming the infrastructure or causing widespread issues.

• **Rollback:** Should any issues arise during the canary release, the deployment can be quickly halted, and the previous version can be restored to the canary users, mitigating any major impact.

Canary releases are particularly useful in continuous delivery pipelines where new features are frequently released. For Java applications that serve millions of users, this method allows for risk management by minimizing the scope of impact during the early stages of deployment.

Example Integration in GitLab CI/CD:

In GitLab, you can implement canary releases by configuring deployment jobs to target a specific subset of pods or containers. The GitLab pipeline will first deploy to the canary group, monitor its performance, and then expand the rollout if everything is functioning correctly.



```yaml
deploy-canary:
  stage: deploy
  script:
    - kubectl apply -f canary-deployment.yaml
    - kubectl rollout status deployment/canary-java-app
```

Here, canary-deployment.yaml specifies the smaller deployment for the canary group, while the main production deployment waits until the canary release is validated.

Containerization with Docker and Kubernetes:

Containerization, especially with Docker and Kubernetes, has revolutionized the deployment process for Java applications. Containers provide isolated environments that include the application and all its dependencies, ensuring that the application behaves consistently across different environments (development, staging, production).

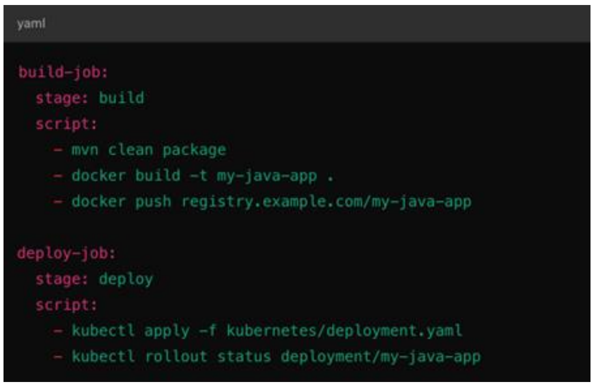
• **Immutable Infrastructure:** Containers enable the concept of immutable infrastructure, where each container image is built once and deployed across environments without modification. This approach is highly beneficial for Java applications because it eliminates the "works on my machine" problem and guarantees consistency between environments.

• **Lightweight and Fast Deployment:** Containers are lightweight compared to virtual machines, making it faster to deploy new versions of applications. This speed is particularly advantageous when combined with zero-downtime strategies like rolling updates or blue-green deployments.

• **Kubernetes Orchestration**: Kubernetes is the leading platform for container orchestration, providing advanced capabilities for scaling, load balancing, and self-healing of applications. It plays a crucial role in managing the lifecycle of containerized Java applications. When integrated with GitLab CI/CD, Kubernetes automates the deployment, scaling, and management of containers in a zero-downtime manner.

Example Integration in GitLab CI/CD:

In a GitLab CI/CD pipeline, containers are often built as part of the build stage, then tested and deployed through orchestration tools like Kubernetes.



```yaml
build-job:
  stage: build
  script:
    - mvn clean package
    - docker build -t my-java-app .
    - docker push registry.example.com/my-java-app

deploy-job:
  stage: deploy
  script:
    - kubectl apply -f kubernetes/deployment.yaml
    - kubectl rollout status deployment/my-java-app
```

In this example, a Docker image of the Java application is built and pushed to a registry, and Kubernetes manages the deployment of the containerized app.

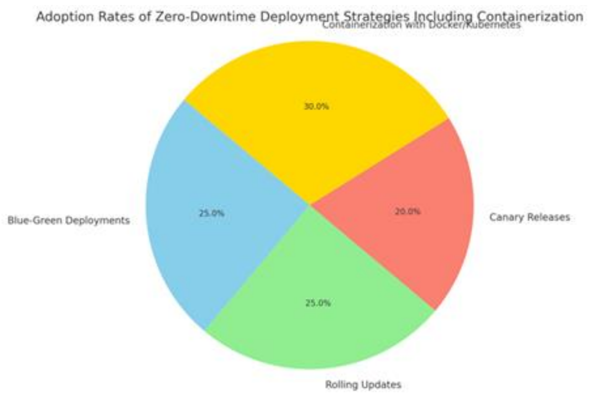


Fig. 2 *Adoption Rates of Zero-Downtime Deployment Strategies*

5. Case Studies

Large-Scale Enterprise Deployment:

Consider a large-scale financial services company that operates Java-based applications for handling transactions. The company implemented a blue-green deployment strategy using GitLab CI/CD to ensure their services remained online 24/7. By integrating Kubernetes and Docker, they achieved seamless deployments with no interruptions to users, even during peak traffic periods.

E-commerce Platform:

An e-commerce platform with millions of daily users adopted GitLab CI/CD with rolling updates to ensure continuous delivery of features and bug fixes. The integration with Kubernetes enabled the platform to gradually roll out updates while monitoring application health, leading to near-zero downtime during deployments.

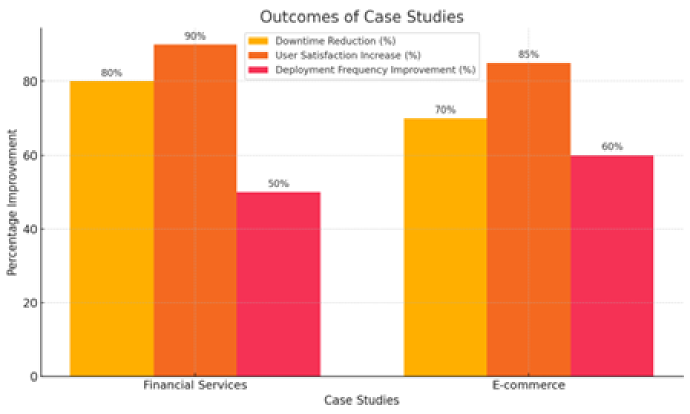


Fig. 3 *Outcomes of Case Studies*

6. Conclusion

Achieving zero-downtime deployment for Java applications is a challenging but necessary goal for modern enterprises. By leveraging GitLab CI/CD pipelines, coupled with containerization technologies like Docker and Kubernetes, businesses can implement deployment strategies such as blue-green and rolling updates to ensure service continuity. The integration of load balancing, service discovery, and traffic management tools further enhances the ability to deploy applications without disrupting user experience. As cloud-native technologies continue to evolve, GitLab CI/CD offers a scalable and flexible solution for managing zero-downtime deployments of Java applications.

References

[1]. Rudrabhatla, C. (2020). Comparison of zero downtime based deployment techniques in public cloud infrastructure. 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 1082-1086. https://doi.org/10.1109/I-SMAC49090.2020.9243605.

[2]. Arefeen, M., & Schiller, M. (2019). Continuous Integration Using Gitlab. Undergraduate Research in Natural and Clinical Science and Technology (URNCST) Journal. https://doi.org/10.26685/urncst.152.

[3]. Ganeshan, M., & Malathi, S. (2020). Building and Deploying a Static Application using Jenkins and Docker in AWS. International Journal of Trend in Scientific Research and Development.

[4]. Weekley, B. (2022). Deployment pipeline development at scale: automating software as a service. . https://doi.org/10.32469/10355/91540.

[5]. M. Fowler, "Blue-Green Deployment," martinfowler.com, 2010. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html

[6]. S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015.

[7]. "Zero-Downtime Deployment with Rolling Updates," GitLab Documentation, [Online]. Available: https://docs.gitlab.com/ee/update/zero_downtime.html

[8]. Zampetti, F., Geremia, S., Bavota, G., & Penta, M. (2021). CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 471-482. https://doi.org/10.1109/ICSME52107.2021.00048.

[9]. Arachchi, S., & Perera, I. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. 2018 Moratuwa Engineering Research Conference (MERCon), 156-161. https://doi.org/10.1109/MERCON.2018.8421965.

[10]. Sabau, A., Hacks, S., & Steffens, A. (2020). Implementation of a continuous delivery pipeline for enterprise architecture model evolution. Software and Systems Modeling, 20, 117-145. https://doi.org/10.1007/S10270-020-00828-Z.

[11]. Mysari, S., & Bejgam, V. (2020). Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible. 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), 1-4. https://doi.org/10.1109/ic-ETITE47903.2020.239.