

# Examining the CI/CD migration process from GitLab to Jenkins



# Table of contents

<b>Abbreviations</b>	<b>3</b>
<b>Introduction</b>	<b>4</b>
<b>Business challenges</b>	<b>4</b>
<b>Problem statement</b>	<b>5</b>
<b>Methodology</b>	<b>5-23</b>
<b>Model implementation</b>	<b>24</b>
<b>Limitations</b>	<b>24</b>
<b>Conclusion</b>	<b>25</b>
<b>References</b>	<b>26</b>
<b>Author info</b>	<b>27</b>

# Abbreviations

CI	Continuous Integration
CD	Continuous Deployment
SSH	Secure Shell
SCM	Source Code Management
OAuth	Open Authorization
PAT	Personal Access Token
URL	Uniform Resource Locator
API	Application Programming Interface
SSH	Secure Shell
GPG	GNU Privacy Guard
RSA	Rivest-Shamir-Adleman
ecdsa	Elliptic Curve Digital Signature Algorithm
YAML	Yet Another Markup Language
DIND	Docker in Docker
PR	Pull Request
LOC	Lines of Code
DevOps	Development and Operations

# Introduction

Optimizing software development processes is paramount in an era of accelerating digital transformation. This paper explores the complex realm of transitioning critical technical components, focusing specifically on Continuous Integration and Continuous Deployment (CI/CD) pipelines. While cost-saving is a significant motivation, the primary focus is the complex technical process of migrating from GitLab to Jenkins. This comprehensive guide provides detailed insights, best practices and practical steps for organizations grappling with their CI/CD workflow transition. The narrative begins with migrating the Source Code Management (SCM) repository from GitLab to GitHub and culminates with integrating the CI feature into Jenkins.

During this journey, we encountered a range of challenges, including Secure Shell (SSH) authentication, the complex task of rewriting the entire 'GitLab -ci.yaml' configuration, the installation of essential plugins, extensive testing over a period of seven months and the establishment of webhooks. The paper provides in-depth solutions, strategies and valuable insights to overcome these technical hurdles. By the end of this paper, readers will gain a comprehensive understanding of the complexities involved in the migration process. Armed with knowledge and solutions, they will be well-prepared to navigate the intricacies of transitioning from GitLab to Jenkins, optimizing their CI/CD pipelines for the future.

## Business challenges

In the dynamic realm of software development, the management of CI/CD pipelines stands as a linchpin of progress. These pipelines are the gears that drive efficiency, facilitating the seamless progression of projects. As organizations increasingly seek cost-effective solutions and performance enhancements, refining their CI/CD workflows becomes paramount. This whitepaper embarks on a transformative journey, unveiling the intricate process of migrating CI/CD pipelines from GitLab to Jenkins.

But why undertake this transition? Beyond the quest for technical excellence, cost-effectiveness emerges as a driving force. The annual renewal costs associated with GitLab's enterprise edition have posed substantial financial challenges for organizations. With each developer's yearly renewal fee of around \$125, the cumulative expense becomes a significant burden, totaling approximately \$25,000. This paper explores how transitioning to Jenkins can alleviate this financial strain while providing a technical roadmap for migrating from GitLab to an in-house Jenkins server.

# Problem statement

In the dynamic field of software development, organizations constantly strive to improve efficiency, streamline processes and enhance collaboration. However, transitioning critical technical components, particularly CI/CD pipelines, poses significant challenges. The migration from one CI/CD service to another, such as GitLab to Jenkins, is a complex endeavor fraught with obstacles. Challenges encompass configuring services, ensuring seamless integration, managing artifacts and maintaining consistency across environments. Organizations grapple with issues like SSH authentication, rewriting configuration files, plugin installation and rigorous testing over prolonged periods. Additionally, the need to optimize resource usage, balance historical data retention and address limitations in service functionalities adds layers of complexity. Amidst these challenges, the overarching problem emerges how to successfully migrate CI/CD workflows while minimizing disruptions and maximizing efficiency.

## Methodology

### GitLab to GitHub SCM change – Phase 1

Our strategic transition to Jenkins initiated with the migration of our SCM from GitLab to GitHub.

#### GitLab to GitHub SCM migration

Our initial task involved transferring the SCM repository from GitLab to our in-house GitHub server, maintained by an in-house team. While this internal GitHub server served as our new source code hub, it lacked an integrated CI feature. This meant that we continued to rely on Git Lab's CI feature until the complete transition to Jenkins, which was a more time-consuming process.

#### Continued to use Git Lab's CI Feature

We quickly migrated the source code to Git within a month, but the migration of the CI feature was a more intricate process spanning several months. Throughout this transition, we continued to depend on GitLab for the CI feature. Therefore, our next task was to re-establish connectivity between GitLab and GitHub after completing the SCM migration. This reconnection enabled us to continue utilizing GitLab 's CI feature until we finalized the migration to Jenkins for CI.

#### Mirroring the repository

To ensure continuity, we established a mirroring process for the repository to continue running CI pipelines on GitLab during the transition period. This process required several code adjustments for external repositories and a connection between GitHub and GitLab. Our GitLab `-.ci.yaml` required code modifications to accommodate the transition from GitLab's built-in SCM to an external SCM. GitLab provides the inbuilt variables (`GitLab`) that can be used if your SCM is an external repository.

- UPSTREAM: origin/\$CI\_MERGE\_REQUEST\_TARGET\_BRANCH\_NAME
- ~~UPSTREAM: origin/\$CI\_EXTERNAL\_PULL\_REQUEST\_TARGET\_BRANCH\_NAME~~
- only:
  - refs:
- merge\_requests
- + only:
  - refs:
- external\_pull\_requests

With these code changes in place, now we moved to establishing the connection between GitLab and GitHub.

## OAuth and PAT selection

We had to select between OAuth and Personal Access Token (PAT) methods to establish mirroring. Initially, we explored OAuth, an automated method for mirroring, but it had limitations, allowing mirroring for up to five repositories. The constraint on mirroring up to five repositories means that even if more repositories were added through mirroring, the process would fail and the pipeline would not be triggered beyond this limit. This limitation was due to GitHub's constraint on the number of unique tokens (10) allowed to remain active. The choice between tokens depended on the number of concurrently configured projects, which posed challenges.

Transition to PAT method: Subsequently, we transitioned to the PAT method, offering manual setup for external CI/CD.

- **Procedure to establish PAT connection:** Before initiating the PAT connection between GitLab and GitHub, ensure that you have consistent access to both platforms using the same account. Successful logins to both GitHub and GitLab are essential prerequisites. On GitHub, confirm that the service account possesses the Owner role and has access to relevant projects. Similarly, on GitLab, create a group with the service account and grant it the maximum role, namely, the Owner role. These prerequisites set the foundation for a seamless PAT connection and subsequent GitLab -GitHub integration.
  - **Generate GitHub token:** Log in to GitHub with the service account. Generate a token for the service account, ensuring it has appropriate permissions. Document the token securely.
  - **Create GitLab project:** Log in to GitLab. Create a group if not already available (set it to Public). Within the group, go to "New Project" and choose "Run CI/CD pipelines for external repositories." Enter the GitHub repository URL, username and the PAT generated. Specify project details (name, slug, description), set the Visibility Level to private and click "Create."
- Enable mirroring:** Navigate to the mirrored project's Settings → Repository → Mirroring. Delete any existing failed mirroring configurations (necessary to add a new one). Add project details, including the project URL, PAT as the password and select "Trigger pipelines for mirror updates." Click "Mirror."

- **Integrate GitHub details in GitLab** : In GitLab , go to Project Settings → Integrations → GitHub using PAT. Generate a token for the same service account on the GitLab side with API scope.
- **Construct Webhook URL**: Generate a webhook URL for triggering CI on GitHub. The prototype is:

[https://GitLab.com/api/v4/projects/<NAMESPACE>%2F<PROJECT>/mirror/pull?private\\_token=<PERSONAL\\_ACCESS\\_TOKEN>](https://GitLab.com/api/v4/projects/<NAMESPACE>%2F<PROJECT>/mirror/pull?private_token=<PERSONAL_ACCESS_TOKEN>)

With these steps, you establish a secure and functional PAT connection between GitLab and GitHub, enabling seamless integration and triggering of CI pipelines.

Resolving PAT limitation: However, this method also encountered limitations, restricting mirroring to five repositories. This issue was linked to the use of an external\_pull\_requests condition, (GitLab) unsupported for manual setup. To overcome this limitation, we adjusted the pipeline configurations, eliminating external\_pull\_requests and opting for conditions like except: [branches] to avoid multiple pipelines running for each push. These changes necessitated code modifications but allowed effective mirroring for all eight product repositories.

- only:

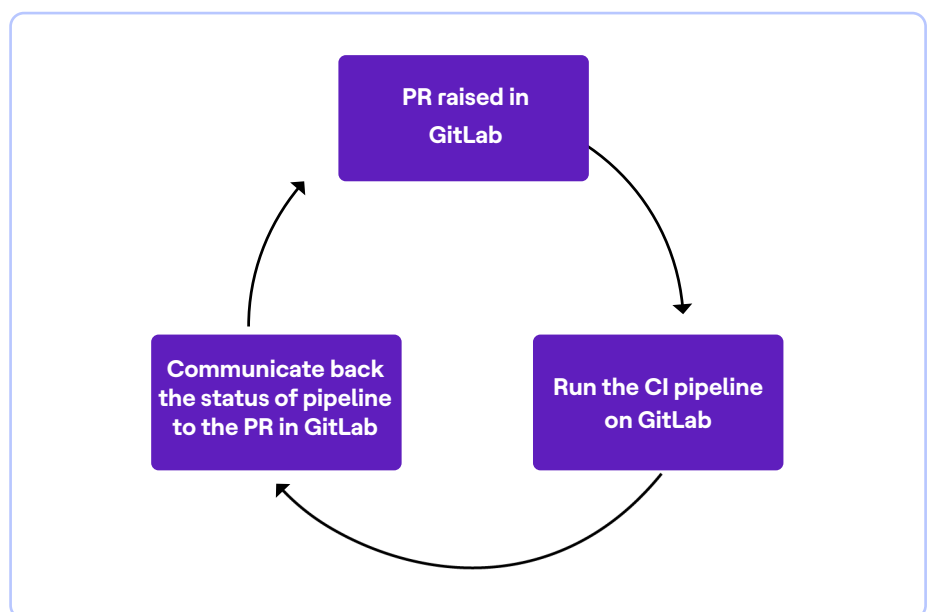
refs:

- external\_pull\_requests

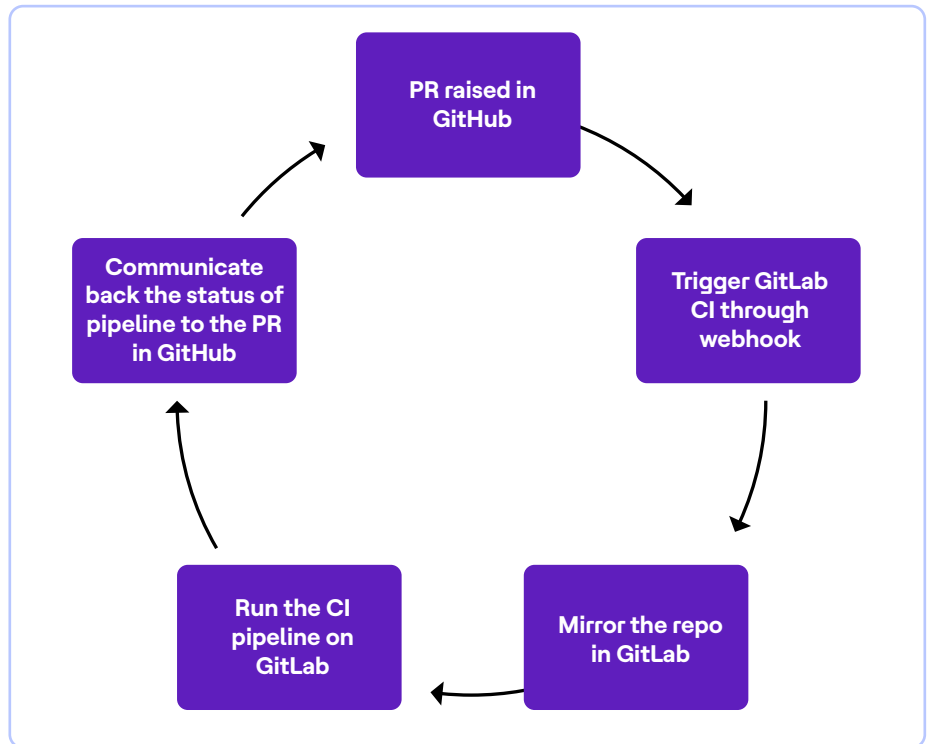
+ except:

- master

Throughout this phase, we documented the drawbacks of OAuth and outlined the reasons for selecting PAT as the preferred solution for our needs. A diagram illustrating the workflow of this method is included for reference. With the temporary solution in place, we shifted our focus toward the primary objective of “migrating the CI to Jenkins.”



Flow chart of the pipeline flow without mirroring



Flow chart of the pipeline flow with mirroring

## GitLab CI to Jenkins CI – Phase 2

This phase involved the implementation of the Jenkins CI.

### SSH configuration to the Jenkins server

SSH (Secure Shell Protocol) is a reliable authentication method for establishing a secure connection between Jenkins and GitHub. When connecting via SSH, the authentication process involves a private key file stored on your local machine. GitHub Enterprise Server recognizes specific public key filenames, such as `id_rsa.pub`, `id_ecdsa.pub` and `id_ed25519.pub`. However, due to GitHub's security updates that no longer accept RSA keys with SHA-1 from 2022, the `ecdsa` algorithm becomes crucial for establishing a connection and obtaining the necessary permissions for repository interaction.

To implement this algorithm for the service account, we generated SSH keys, resulting in two key files: `svc_ecdsa.pub` (public key) and `svc_ecdsa` (private key). The next steps involve adding the public key to the GitHub production instance. Access GitHub, log in with the service account, navigate to settings from the top right, proceed to SSH and GPG Keys and add the newly generated key.

On the Jenkins side, the private key must be added for seamless integration. In the Jenkins dashboard, access credentials, click on Global and add new credentials by choosing SSH with a private key. Enter an ID (an attribute/keyword for accessing these credentials in code), a description for clarity, the service account username as the login and the generated private key. Save the credentials and the SSH connection between Jenkins and GitHub will be securely configured.

### Testing and transitioning to Jenkinsfile

With the SSH connection successfully established between Jenkins and GitHub, our focus shifted to initiating pipeline execution. In the earlier

stages, Jenkins utilized scripted pipeline scripts to define pipelines. However, a pivotal shift occurred around 2016 with the introduction of Declarative Pipelines. This newer approach aimed to provide a more structured, user-friendly method for defining continuous delivery pipelines. Unlike Scripted Pipelines, which allowed for greater flexibility but demanded a deeper understanding of Groovy scripting intricacies, Declarative Pipelines sought to streamline the pipeline definition process. Declarative Pipelines facilitated easier comprehension, maintenance and collaboration by utilizing a more concise and structured syntax. The decision to opt for the Declarative Pipeline was influenced by its simplicity, ease of understanding and ability to define pipelines using a more declarative syntax. This aligns well to make CI/CD processes more transparent and accessible to a wider audience. Our journey commenced with testing a small Jenkinsfile, leveraging the capabilities of Groovy for pipeline configuration. This initial testing phase provided a valuable opportunity to become familiar with Groovy syntax, ensuring efficient pipeline definition and execution in the Jenkins environment.

With a solid foundation in Groovy, we transitioned to the most time-intensive task: rewriting the existing GitLab `-.ci.yaml` configurations for all repositories in Groovy Jenkinsfile format. This transition involved translating the logic and configurations from GitLab's YAML syntax to Groovy, ensuring a smooth migration of CI/CD processes.

## Choosing the conversion approach

In the process of converting GitLab CI to Jenkins CI, two methods presented themselves:

### **Jenkins YAML using the pipeline as YAML Plugin (Jenkinsfile.yaml):**

Conversion Flow: Input to the plugin is the GitLab `-.ci.yaml`, converted into an intermediate stage of Jenkinsfile.yaml. The plugin then converts it into a declarative pipeline.

#### **Decision factors:**

- **Limited support:** The plugin for YAML-based code in Jenkins does not support all keywords present in GitLab `-.ci.yaml`.
- **Missing post-build actions:** Post-build action options are not found in the plugin, restricting functionality.
- **Limited documentation:** Inadequate documentation exists for the Jenkins YAML approach.
- **Plugin stage:** The plugin is in an "incubation" stage, with insufficient details and documentation.
- **Transition costs:** Although it seems like a transition from YAML to YAML, it necessitates rewriting the entire CI pipeline, potentially more expensive than the Jenkins Groovy approach.
- **Production recommendation:** Due to the plugin's status and recommendation against using it in a production environment, the direct Groovy way was favored.

## Jenkins Groovy

**Conversion flow:** The manual way of converting GitLab `-.ci.yaml` into a Declarative Pipeline using Groovy.

### Advantages over the YAML approach:

- **Comprehensive support:** Allows handling complex logic, customizations and code reusability functions or targets more effectively.
- **No dependency issues:** Avoids dependency on plugins at the incubation stage, ensuring stability.
- **Plugin compatibility:** Provides better compatibility with other Jenkins plugins, ensuring full functionality.

Considering the limitations and challenges faced with the plugin approach, especially its incubation status and lack of documentation, the Jenkins Groovy method was chosen for its advantages in handling complex scenarios and providing better compatibility with Jenkins plugins.

Following the decision to proceed with the Direct Groovy approach for transitioning from GitLab CI to Jenkins, the focus shifted to the conversion of GitLab `-.ci.yaml` pipeline codes into Jenkinsfile using Groovy. This pivotal phase involved an in-depth exploration of various critical use cases. Given the vast landscape of potential scenarios, the whitepaper will selectively present the most important use cases to ensure clarity and relevance. These use cases were meticulously examined to capture the essence of the transition process, providing valuable insights into handling dependencies, promoting modularity, implementing conditional logic, addressing post-build actions, navigating complex logic scenarios and seamlessly integrating with GitHub. Each use case represents a key facet of the migration, contributing to a comprehensive understanding of the challenges and solutions encountered during the transition from GitLab CI to Jenkins in the Groovy environment.

## Use cases explored

### 1. Parallel execution with `parallelsAlwaysFailFast`

In scenarios where parallel stage execution is essential, Jenkins provides the `parallelsAlwaysFailFast` option to expedite the build process.

When defining parallel stages in a Jenkinsfile, the `parallelsAlwaysFailFast` option can be set to true. With `parallelsAlwaysFailFast` enabled, the pipeline will abort as soon as any parallel branch encounters a failure. This accelerates the build duration by not waiting for all branches to complete.

Code snippet

```
options {
    parallelsAlwaysFailFast()
}
```

### 2. Switching user in Jenkins for command execution

Executing commands in Jenkins with a different user requires a user switch. The conventional methods, such as using the use of `'sudo su'` for the user to switch between commands in Jenkins, are found to be inconsistent due to separate shell instances.

- `sh 'sudo su - runner'`
- `sh 'cp source_path target_path'`

When we pass it this way, Jenkins will execute each line in separate shell instances. The user change in point 1 will not be consistent to the shell instance in point 2

Instead, we could use the following command to switch the user and execute the commands.

- `sh 'sudo su - runner -c "command_to_execute"'`

This ensures that the complete set of commands within the double-quoted section is executed with the specified user context. This approach is particularly useful when dealing with multiple commands that require a different user context.

### 3. Configuring GitHub Pull Request to trigger in Jenkins

Enabling Jenkins to trigger CI builds specifically for GitHub Pull Request (PR) poses a challenge. The process involves configuring GitHub webhooks and ensuring that Jenkins is responsive to relevant GitHub events.

To achieve a GitHub PR triggering pipeline in Jenkins, the following steps are implemented:

#### GitHub integration in Jenkins:

Utilize the "GitHub Pull requests" checkbox in the Jenkins pipeline job configuration. Select "Hooks with Persisted Data" for the Trigger mode. There are two checkboxes that would help run the duplicate builds.

- **Cancel queued builds:** If selected when a new PR build is triggered, pending builds for that PR will be removed from the queue in favor of this most recent build.
- **Abort running builds:** If selected when a new PR build is triggered, any ongoing builds for that PR on the executors will be aborted in favor of this most recent build.

Select the relevant Trigger events on which you would like to run the pipeline. In the GitHub Plugin Repository Provider, select the Repository requested permission as PUSH.

Once the Jenkins side configuration is set, we need to add the GitHub webhook.

#### GitHub webhook configuration:

Access GitHub repository settings and navigate to 'Hooks.'

- Add a webhook, specifying the Jenkins environment URL appended with `'/github-webhook/'`.
- Configure the webhook to trigger 'Pull Requests' and 'Pushes.'

By combining Jenkins pipeline configuration settings, GitHub webhooks and the GitHub plugin, Jenkins can seamlessly trigger CI builds for GitHub PRs. This integration ensures timely and automated validation of PR changes.

### 4. Utilizing downstream jobs to overcome LOC limitations

In our testing, we found that Jenkins imposes a Maximum Line of Code (LOC) limitation of 1047 for Jenkinsfiles. There is currently an open issue in Jenkins forums that limits the maximum code size within the pipeline

} block. Exceeding this limit results in a "method too large" error. In contrast to GitLab's 'include' keyword, which supports the inclusion of child Jenkinsfiles in the main Jenkinsfile, Jenkins lacks a direct alternative for this functionality.

The options that we explored are:

```
Use of LOAD keyword:
def codeObj pipeline {
  agent any
  stages {

    stage('Build') {
      steps {
        script {
          codeObj = load "testFun.groovy"
          codeObj.firstFun()
        }
      }
    }
    stage('Test') {
      steps {
        script {
          codeObj.secondFun()

        }
      }
    }
  }
}

#testFun.groovy def
firstFun(){ sh echo
"firstFun called"
}

def secondFun(){ sh echo
"secondFun called"
}
```

#### Limitations:

- Inability to use a separate Jenkinsfile as input for the load keyword, as the file with the word 'pipeline' in its name can only exist once per Jenkins job.
- Constraints on defining proper steps, Docker images and other configurations when loading functions.

#### Utilization of downstream jobs:

Advantages:

- Overcomes the limitations of the load keyword by allowing the main job to trigger downstream jobs.
- Enables the execution of a Jenkinsfile in a subdirectory, providing the flexibility to create a proper pipeline.

Drawbacks:

- Lack of visibility into downstream job stages from the Blue Ocean view, necessitating navigation to an additional stage for desired insights.

- Inability to view all stages simultaneously, as they are divided based on downstream jobs.

Considering the constraints of the load keyword, downstream jobs emerge as a practical solution, providing a workaround for LOC limitations in Jenkinsfiles. Despite some drawbacks in visualization, this approach facilitates the creation of comprehensive pipelines by executing subdirectory Jenkinsfiles in downstream jobs.

### Resolving downstream limitations:

Blue Ocean was not consistent in displaying the downstream URL. It failed in so many combinations.

We can overcome this by manually generating the downstream URL. We can assign a variable to the downstream build call and once the downstream job is done executing, it will store the information regarding it in the assigned variable. Then we can use the echo statements to print a bunch of information including the URL's.

Code snippet:

```
stage('Example') {
  steps {
    script {
      def DSJ = build job: 'test_Downstream_Child01'
      def downstreamJobURL = DSJ.getAbsoluteUrl()

      echo "Downstream Job URL: ${downstreamJobURL}"
      echo "Downstream Build Number: ${DSJ.number}"
      echo "Downstream Result: ${DSJ.result}"
      echo "Downstream Start Time: ${new Date(DSJ.startTimeInMillis)}"
      echo "Downstream Duration: ${DSJ.duration} milliseconds"
      echo "Downstream Full Display Name: ${DSJ.fullDisplayName}"
      echo "Downstream Absolute URL: ${DSJ.getAbsoluteUrl()}"

    }
  }
}
```

Output:

```
Downstream Job URL: https://jenkins.com/job/test_Downstream_Child01/5/
Downstream Build Number: 5
Downstream Result: SUCCESS
Downstream Start Time: Wed Nov 22 12:02:12 UTC 2023
Downstream Duration: 3538 milliseconds
Downstream Full Display Name: test_Downstream_Child01 #5
Downstream Absolute URL: https://jenkins.com/job/test_Downstream_Child01/5/
```

## 5. Setting pipeline status in GitHub PR

Setting the pipeline status in a GitHub PR involves utilizing the webhook and adding the `setBuildStatus` function in the Jenkinsfile. This function, used as a post-build action, communicates the Jenkins build status to the corresponding GitHub PR. Placing the `setBuildStatus` function as the first statement in the initial stage ensures the status reflects the pipeline's start on GitHub.

Available States for `setBuildStatus`:

- **PENDING:** Indicates the build is in progress or awaiting further processing.
- **SUCCESS:** Represents a successful build.
- **FAILURE:** Denotes a failed build.
- **ERROR:** Represents an error during the build process.
- **ABORTED:** Indicates a manually aborted build.
- **UNSTABLE:** Sets the overall build result and stage result to UNSTABLE if an exception is thrown.

Code snippet

```
void setBuildStatus(String message, String state) {
    step([
        $class: "GitHubCommitStatusSetter",
        reposSource: [$class: "ManuallyEnteredRepositorySource", url:
"GIT_URL"],
        contextSource: [$class: "ManuallyEnteredCommitContextSource",
context: "ci/jenkins/build-status"],
        errorHandlers: [[$class: "ChangingBuildStatusErrorHandler", result:
"UNSTABLE"]],
        statusResultSource: [ $class: "ConditionalStatusResultSource",
results: [[$class: "AnyBuildResult", message: message, state: state]] ]
    ]);

    #Need to call setBuildStatus function as post build action to reflect the
status of Jenkins build on GitHub PR
    post {
        success {
            setBuildStatus("Build succeeded", "SUCCESS");
        }
        failure {
            setBuildStatus("Build failed", "FAILURE");
        }
    }

    #we need to call setBuildStatus as first statement in first stage so, it can
set the status as pipeline started on GitHub PR
    setBuildStatus("Build is in progress", "PENDING");
}
```

## 6. Triggering Deploy Job Based on Master Branch Changes

In GitLab CI, triggering a deploy job upon changes to the master branch is straightforward. Using the only keyword, the job executes exclusively for the master branch, ensuring deployment occurs solely on relevant commits.

Code Snippet of GitLab `-.ci.yml`

```
deploy:
  stage: deploy
  tags:
    - ubuntu
  script:
    - echo "I'm in Deploy stage"
  only:
    - master
```

When attempting to replicate this behavior in Jenkins using the GitHub Integration plugin, challenges arise. Conditions based on `BRANCH_NAME` prove ineffective because, after a PR merge, the Jenkins job is triggered with the developer branch, not the master branch. As a result, the deploy job may erroneously trigger when there are no commits on the master.

To overcome this, conditions were tested based on `GITHUB_PR_STATE` and `GITHUB_PR_COND_REF`. These are the variables that the GitHub integration plugin is offering; however, there was ambiguity in the behavior of these variables. The combination that worked was:

- For PRs being raised:  
    `GITHUB_PR_STATE: OPEN` and `GITHUB_PR_COND_REF: head`
- For PRs being closed:  
    `GITHUB_PR_STATE: CLOSED` and `GITHUB_PR_COND_REF: merge`
- For PRs being re-opened:  
    `GITHUB_PR_STATE: OPEN` and `GITHUB_PR_COND_REF: head`
- For PRs being merged:  
    `GITHUB_PR_STATE: CLOSED` and `GITHUB_PR_COND_REF: head`

This approach ensures that the Test stage triggers when a PR is opened and the Deploy stage triggers when the PR is merged, aligning with the desired behavior of the GitLab CI configuration.

```
pipeline {
  agent any
  stages {
    stage('Print Variables') {
      steps {
        echo "GITHUB_PR_STATE: ${env.GITHUB_PR_STATE}"
        echo "GITHUB_PR_COND_REF: ${env.GITHUB_PR_COND_REF}"
      }
    }
    stage('Test') {
      when {
        expression { env.GITHUB_PR_STATE == 'OPEN' }
      }
    }
  }
}
```





replicate these functionalities in Jenkins using Groovy, three alternatives were explored:

#### **Using stash/un-stash:**

- **Pros:** Efficient for sharing artifacts within the same stage without running into space issues, as they are discarded after the build is done.
- **Cons:** Limited in duration; artifacts will be discarded once the Jenkins job is finished.

#### **Using manual methods to save artifacts:**

- **Pros:** Provides a comprehensive solution, allowing artifacts to be stored if build logs are available. Additional control is offered through the Build Discarder setting in Jenkins.
- **Cons:** Requires additional configuration of the plugin.

#### **Using manual methods to save artifacts:**

- **Pros:** Artifacts are only available if the workspace is not cleared.
- **Cons:** Prone to running into space issues, especially when running pipelines on a large scale.

This evaluation reveals that each approach comes with its own set of advantages and limitations. However, after careful consideration, the use of ArchiveArtifacts/CopyArtifacts emerged as the most suitable solution, offering a balance between functionality, configurability and ease of use.

#### **Setting for build logs/artifacts retention**

The "expires\_in" keyword in GitLab CI allows fine-grained control over how long artifacts are retained. In Jenkins, the equivalent is achieved through the Build Discarder setting, which dictates the duration for which build logs and artifacts are preserved.

The Build Discarder setting can be configured at both the global level and the job level. Configuring it at the job level allows for customization tailored to the needs of individual projects, providing flexibility for different retention policies across various jobs. However, in our setup, we have opted to configure the Build Discarder setting at the global level due to the considerable number of jobs we manage. This centralized approach simplifies administration and ensures a consistent retention policy across all projects.

While this global configuration provides a streamlined method for managing build log and artifact retention, caution is warranted. Adjustments to the global Build Discarder setting impact all builds across Jenkins. Therefore, it is crucial to configure this setting judiciously to align with project-specific requirements, ensuring an optimal balance between resource usage and historical build information retention.

#### **Build continue with empty artifacts**

In certain scenarios, it is essential for a build to progress to the next stage even if the previous stage does not generate any artifacts. This is crucial for ensuring that the pipeline does not fail unnecessarily due to the absence of artifacts. In Jenkins, this can be achieved using the 'archiveArtifacts' step with the 'allowEmptyArchive' option.

```
*archiveArtifacts artifacts: 'themes', allowEmptyArchive: true
```

Using this configuration, Jenkins will simply throw out a warning if no artifacts are found that match the specified pattern, but the build will continue to the subsequent stages. This ensures that the pipeline remains robust and can handle cases where artifacts are not always produced.

When the specified artifacts are not found, Jenkins will log a warning message like the following:

```
Archiving artifact: 'themes' does not match anything No artifacts are found matching the pattern - "themes". Configuration error?
```

This message indicates that no artifacts matching the 'themes' pattern were found, which might be due to a configuration error or because the artifacts were not generated in that run. However, the pipeline will proceed to the next stage despite this warning, maintaining the flow of the build process.

## 10. Replacement for interruptible keyword in GitLab CI

In GitLab CI, the interruptible keyword is crucial in managing pipeline cancellations. When dealing with a running pipeline, three scenarios arise: jobs pending to start (cancellable), jobs started with the interruptible set (cancellable) and jobs started without interruptible (non-cancellable). The non-cancellable jobs, often associated with critical stages like BUILD or Deploy, pose challenges in immediate cancellation, risking potential corruption or partial deployments. To address this in Jenkins, the GitHub PR plugin provides an alternative. The system intelligently cancels old jobs when a PR is updated by configuring the "GitHub Pull Requests" option in Jenkins job settings. For instance, if a new commit is pushed while a previous job is running, the older job is immediately aborted, triggering a new pipeline with the reason "Newer PR will be scheduled." Screenshots of the settings involved can be included for reference, illustrating the seamless handling of pipeline interruptions in Jenkins.

## 11. Configuration setting - Wipe Out Workspace and Force Clone

In Jenkins, a critical configuration setting impacts the behavior of workspaces and repository clones. Specifically, the option to "Wipe Out Workspace" with "Force Clone" combines two functionalities that are crucial in workspace management.

### Wipe out workspace with Force Clone:

- Enabling "Wipe Out Workspace" in the Jenkins job configuration deletes the entire workspace before a new build starts.
- Enabling "Force Clone" simultaneously mandates that Jenkins perform a fresh clone of the repository before starting a build.
- This combination ensures a clean environment for each build and guarantees an up-to-date repository state.

### Configuration steps:

Navigate to the Jenkins job configuration page.

- Under the "General" section, find and select "Wipe Out Workspace" to enable this option.

- Additionally, under the "Advanced Clone Behaviors," check "Force Clone" to enforce a fresh clone of the repository.

Resolving the limitation of Wipe Out Workspace and Force Clone:

Working on Wipeout and Force Clone:

- It will delete all the workspace associated with the job.
- It will not use sudo to delete it; rather, it will try to delete as a user on which the Jenkins slaves are configured.

So, if there is any workspace modification, such as a git clone, artifacts are generated. The files may be generated with root permissions, which will prevent the workspace from being cleaned. The workaround in such cases would be to change the workspace ownership after the job/stage/build.

- `chown -R Jenkins:Jenkins $WORKSPACE/`

## 12. Workspace naming convention in Jenkins based on executors and Docker Container Usage

In Jenkins, workspaces are named according to a structured convention that provides insights into the order of execution, the number of executors and the usage of Docker containers.

- The naming convention involves appending specific indicators to the workspace name:
- Ordinal indicators (e.g., 1st, 2nd, 3rd) denote the order of execution on the machine.
- '@' followed by the executor count signifies the executor associated with the workspace.
- '@tmp' is added to indicate the workspace's association with a Docker container.

Examples:

Test-1stExe:

Workspace for the first execution on the machine.

Test@2:

Workspace associated with the second executor.

Test@tmp:

Workspace created when a Docker container runs.

Test@2@tmp:

Workspace for the second executor and Docker container usage.

This convention helps distinguish workspaces, providing a quick overview of the execution order, executor count and whether a Docker container is involved. It facilitates easy identification and understanding of workspaces within the CI/CD environment.

## 12. Implementing Blue Ocean to match the visualization of GitLab CI

We had to rely on the Blue Ocean plugin to match the graphical representation of the CI pipeline flow like GitLab. Jenkins' standard

dashboard for CI pipelines does not offer the same level of graphical representation; it only has the classic view, which is a plain console stdout without any head and tail. Blue Ocean, on the other hand, provides a modern and intuitive interface that visualizes the pipeline's flow, making it easier to understand and manage complex CI/CD processes. It enhances the user experience by displaying the stages and steps more organized and visually appealingly, closely aligning with the visualization capabilities of GitLab CI.

### How to display the node or slave name in Blue Ocean

One drawback we observed with Blue Ocean is that it does not show the node name where a particular stage is running. While the classic view does show the slave machine on which the current job is running, Blue Ocean lacks this detail.

We added an extra stage in the Jenkinsfile to display the node name using Jenkins environment variables to address this. This can be done by incorporating the `${env.NODE_NAME}` variable into the pipeline script. Here is how you can do it:

```
pipeline {
  agent any
  stages {
    stage('Display Node Name') {
      steps {
        script {
          echo "Running on node: ${env.NODE_NAME}"
        }
      }
    }
  }
}
```

This approach helps bridge the gap between Jenkins and GitLab CI in terms of visualization and information accessibility.

The insights gained from addressing interruption handling, plugin integrations, artifacts management and other intricate scenarios have collectively empowered us to rewrite Jenkinsfile's for all eight product repositories. By navigating these challenges and implementing effective alternatives, we have not only optimized our CI/CD workflows but also demonstrated the flexibility and adaptability of Jenkins. This comprehensive journey provides valuable guidance for teams venturing into similar transitions, offering a practical roadmap to integrate Jenkins into their DevOps toolchain seamlessly.

## Maintaining Docker consistency during Jenkins CI migration

During the migration of our CI pipeline from GitLab to Jenkins, we ensured that the Docker containers used for builds remained consistent. Here is a summary of our approach and the considerations involved:

### • Adding Jenkins-specific user accounts in Docker containers

Initially, we considered adding the `svc_jenkins` user, which registers the slaves in Jenkins, to all Docker containers. This approach would have defeated the primary advantage of dockerizing the built environment: its independence from the host environment running the container. Adding Jenkins-specific configurations to the container contradicts the principles of containerization. However, upon further analysis of the different user accounts present in the Docker containers, we realized that adding the `svc_jenkins` dependency was unnecessary.

### • Existing Docker user accounts

**Root user:** The root user is only used within containers, eliminating any concerns about changing this setup during the migration. No changes were needed here.

**Runner user:** Previously, the 'runner' user was utilized for GUI builds within GitLab CI due to its pre-existing home directory in the container. In Jenkins, we mapped this user to a new service account (`svc_nprunner`) to maintain functionality without creating local accounts on GitHub, as GitHub does not support "local" accounts.

**CI user:** This user is arbitrary and does not exist inside the container. It was primarily used to own files created during builds, ensuring the CI runner could manage these files without permission issues. The same principle was applied to Jenkins, so no changes were needed here.

Our strategy ensured a smooth transition without compromising the integrity of the built environment. Consistent user mappings and service accounts facilitated the transition from GitLab to Jenkins while maintaining the functionality and security of our CI/CD pipeline.

## Migration of the GitLab runners to Jenkins slaves

Migrating GitLab runners to Jenkins slaves was crucial to ensure seamless integration into our CI/CD infrastructure. The strategic reuse of existing infrastructure reflects the efficiency of our migration approach, promoting continuity in our build and deployment workflows. The following steps outline the process we followed:

### • Stop GitLab runners:

Halt the GitLab runners on the GitLab production and development environments. Cease the Puppet configuration.

### • Prepare Jenkins slave environment:

Install Java on the intended Jenkins slave machine using the command: `apt install default-jre`. Stop the GitLab Runner service and install the zip package.

### • Configuration changes and puppet management:

Implement necessary puppet configuration changes specific to the organization to adapt to the Jenkins environment. Execute puppet to apply the updated configuration and manage the remaining services seamlessly.

### **Jenkins Slaves and the number of executors post-migration**

During the migration, we encountered Git fetch errors when utilizing multiple executors on Jenkins slaves. This issue presented a challenge, as the root cause was unclear; it could have stemmed from GitHub, the Jenkins server or the Jenkins slaves themselves. After a thorough investigation, we discovered that reducing the number of executors to one per slave resolved the issue, ensuring stable and error-free operations. This adjustment was crucial in maintaining the reliability and efficiency of our CI pipeline in Jenkins post-migration.

### **Essential Jenkins Plugins that are needed for GitLab CI migration**

A pivotal aspect of the GitLab CI to Jenkins migration involves strategically using essential plugins to enhance functionality and replicate GitLab CI features. The following key plugins played a significant role in ensuring a seamless transition: Here is the summary of the plugins used:

- **GitHub integration plugin:**

Functionality: This plugin is the backbone for enabling CI from GitHub to Jenkins. By adding a Jenkins webhook on the GitHub side and enabling the "GitHub PR" checkbox, the integration facilitates efficient communication and seamlessly triggers Jenkins pipelines.

Installed version: 0.5.0

- **Blue Ocean:**

Functionality: Offering an advanced CI view, the Blue Ocean plugin provides a user interface that closely resembles the familiar "GitLab CI" layout. This intuitive interface enhances the visualization of CI pipelines, aiding developers in understanding and monitoring the CI/CD workflows. Installed version: 1.26.0

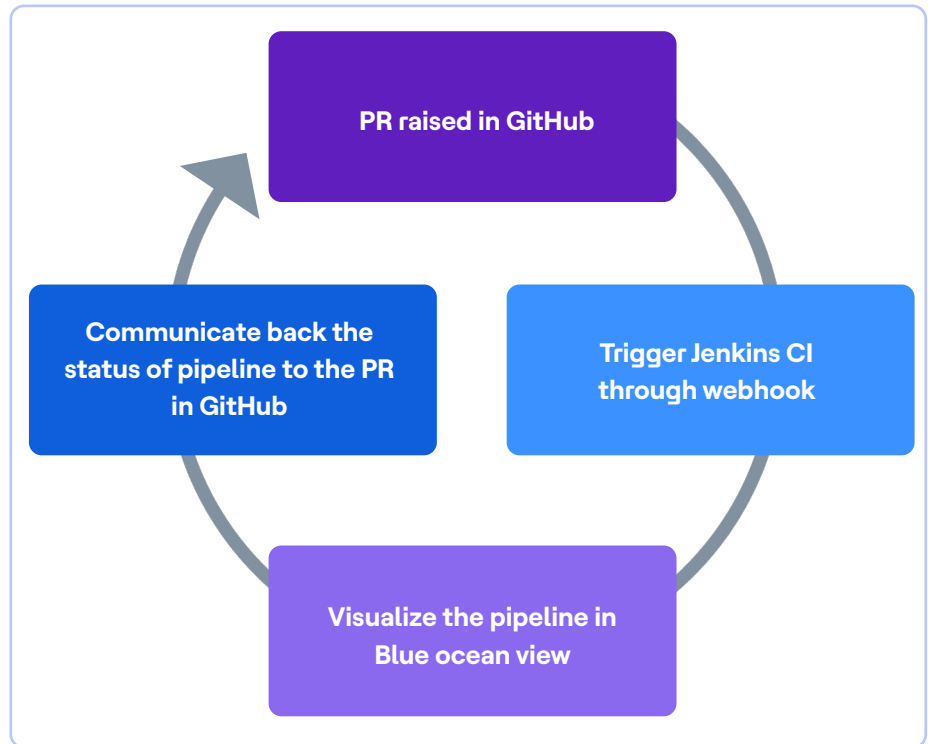
- **CopyArtifacts:**

Functionality: A critical plugin for handling artifact dependencies, CopyArtifacts ensures smooth referencing of artifacts generated during the build process. These artifacts are stored on the Jenkins server, enabling their use in subsequent stages of the pipeline.

Installed version: 1.48

These plugins collectively contribute to a comprehensive and feature-rich CI/CD environment in Jenkins, replicating and enhancing the capabilities found in GitLab CI. Additionally, it is worth noting that these plugins have continuous support and come with newer versions. The versions mentioned here are the ones we had installed when we underwent migration.

# Model implementation



## Limitations

Navigating the landscape of GitHub and Jenkins, two formidable platforms in the realm of DevOps, is not without its intricacies. The process of integrating these giants for optimal CI/CD workflows brings forth challenges and limitations.

### Restarting a job in Jenkins

- Full stage restart: Unlike GitLab `-.ci.yml`, where job-level restart is possible, in Jenkins, the restart/resume functionality is at the stage level. Restarting a failed parallel job in each stage in Jenkins will restart the entire stage.
- Downstream restart impact:  
In Jenkins, Restarting a failed job in a downstream project will only restart the downstream project stage; it will not trigger subsequent stages in the upstream pipeline.

### Line of Code (LOC) limitations

In our testing, the maximum amount of code that can be written in a main Jenkinsfile is 1047 lines. If the code exceeds this limitation, a “method too large” error occurs.

Workaround: To overcome this limitation, downstream jobs are implemented. While this resolves the LOC challenge, it introduces an extra step for developers to view the status of jobs in the pipeline due to Jenkins design constraints.

# Conclusion

From migrating SCM repositories to integrating CI features, we encountered diverse challenges, such as SSH authentication, rewriting intricate GitLab `-.ci.yml` configurations, installing essential plugins and establishing webhooks. Through seven months of rigorous testing and continuous refinements, we have compiled valuable insights, best practices and detailed solutions to overcome these hurdles.

The journey unfolded with the establishment of a seamless connection between GitHub and GitLab using PATs and the subsequent integration into Jenkins. We explored use cases ranging from triggering Jenkins pipelines for GitHub pull requests, handling artifacts and build logs and implementing Docker in Docker (DIND) to managing downstream jobs and handling workspace limitations. Each use case addressed specific challenges, offering practical solutions that emerged from real-world scenarios.

Limitations were acknowledged and addressed, underscoring the importance of realistic expectations when dealing with two powerful yet distinct platforms. As we explored these facets, we discovered the necessity of rewriting Jenkinsfile's for eight distinct product repositories, reflecting the diverse nature of our organization's software projects.

In conclusion, this whitepaper is a comprehensive resource for organizations contemplating or undergoing a similar migration. By sharing our experiences, challenges and solutions, we aim to empower readers with the knowledge needed to successfully navigate the complexities of transitioning from GitLab to Jenkins. The presented strategies, insights and best practices outline a roadmap for organizations aiming to optimize their CI/CD pipelines and adapt to the future of software development.

# References

- GITHUB SCM POLLING <https://www.devopsschool.com/blog/how-to-build-when-a-change-is-pushed-to-github-in-jenkins/>
- <https://stackoverflow.com/questions/14274293/show-current-state-of-jenkins-build-on-github-repo/26910986#26910986>
- <https://stackoverflow.com/questions/43214730/how-to-set-github-commit-status-with-jenkinsfile-not-using-a-pull-requestbuilde/47162309#47162309>
- [https://docs.GitLab.com/ee/ci/ci\\_cd\\_for\\_external\\_repos/index.html#limitations](https://docs.GitLab.com/ee/ci/ci_cd_for_external_repos/index.html#limitations)
- <https://GitLab.com/GitLab-org/GitLab-/-/issues/323336>
- SSH KEY SETUP IN GITHUB - <https://docs.github.com/en/enterprise-server@3.11/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>
- <https://github.com/KostyaSha/github-integration-plugin/blob/master/docs/Configuration.adoc>
- <https://stackoverflow.com/questions/43214730/how-to-set-github-commit-status-with-jenkinsfile-not-using-a-pull-requestbuilde/47162309#47162309>
- GitLab . Limitations - [https://docs.GitLab.com/ee/ci/ci\\_cd\\_for\\_external\\_repos/index.html#limitations](https://docs.GitLab.com/ee/ci/ci_cd_for_external_repos/index.html#limitations).
- Predefined-variables-for-external-pull-request-pipelines - [https://docs.GitLab.com/ee/ci/variables/predefined\\_variables.html#predefined-variables-for-external-pull-request-pipelines](https://docs.GitLab.com/ee/ci/variables/predefined_variables.html#predefined-variables-for-external-pull-request-pipelines).

# Author information



## Sunil Kumar Raju Nadimpalli

Sunil completed his master's degree in mechanical engineering from IIT Madras in 2020. He has been with HCLTech since then, serving as a Technical Lead on the DELL VPLEX team for nearly four years. His current work, detailed in this whitepaper, involves the migration of CI/CD as part of the Tools-Rocket team.

He is also part of the team that filed for the patent, publicly available as USPTO Publication Number US-12112053-B1, titled "Automated Migration of Virtual Volumes Between Storage Virtualization Appliances," which has been published by the United States Patent and Trademark Office (USPTO) as a public document.

The official details are as follows:

Publication Number: US-12112053-B1

Publication Date: October 8th, 2024

# HCLTech | Supercharging Progress™

HCLTech is a global technology company, home to more than 220,000 people across 60 countries, delivering industry-leading capabilities centered around digital, engineering, cloud and AI, powered by a broad portfolio of technology services and products. We work with clients across all major verticals, providing industry solutions for Financial Services, Manufacturing, Life Sciences and Healthcare, Technology and Services, Telecom and Media, Retail and CPG and Public Services. Consolidated revenues as of 12 months ending December 2024 totaled \$13.8 billion. To learn how we can supercharge progress for you, visit [hcltech.com](https://hcltech.com).

[hcltech.com](https://hcltech.com)

