



Managing Build Artifacts Using Maven and Nexus in CI/CD Workflows

Anil Kumar Manukonda

E-mail: anil30494@gmail.com

Nalluri Varun Kumar

E-mail: Nallurivarun@gmail.com

Abstract

Build artifacts which include all compiled results from binaries to packages and images and other deployment objects function as the core components of continuous integration/continuous delivery (CI/CD). Build artifact management practices help speed up delivery cycles while maintaining system dependability. This paper examines the joint operation of Apache Maven as a build and dependency management tool with Sonatype Nexus as an artifact repository manager to resolve artifact management problems. This work combines prevalent practices from literary sources to create CI/CD pipeline architecture that combines Maven with Nexus then provides technical implementation details based on Jenkins for automation purposes. Artifacts managed effectively by Maven and Nexus reduce build time and automate deployments and streamlines version tracking and smoothly transfer artifacts from dev to test to production environments. The paper shows how correct artifact management enhances system stability and release speed through analysis of Nexus pipeline operations in Jenkins. The paper provides important lessons about version management techniques alongside storage protocols and system defense mechanisms before exploring upcoming developments in DevOps artifact maintenance.

Keywords: Maven, Nexus Repository, CI/CD Pipelines, Continuous Integration, Continuous Delivery, Artifact Management, Build Artifacts, Jenkins, Jenkinsfile, Declarative Pipeline, POM, Groovy, DevOps, Repository Management, Versioning Strategies, Semantic Versioning, SNAPSHOT Artifacts, Release Artifacts, Cleanup Policies, Nexus OSS, Nexus Pro, Artifact Promotion, Artifact Retention, Nexus Lifecycle, Nexus Firewall, RBAC, Security Credential Management, Dependency Management, Nexus REST API, Maven Deploy, Nexus Artifact Uploader, Jenkins Plugins, Git Integration, Source Control, Docker Images, Helm Charts, Sonatype, SBOM, DevSecOps, Infrastructure as Code, Pipeline Automation, Nexus Configuration, Artifact Caching, Build Optimization, Jenkins Pipeline, GitLab CI, GitHub Actions, YAML Pipelines, Artifact Storage, Artifact Traceability, Nexus Group Repositories, Maven Proxy, Jenkins Nexus Plugin, Jenkins CI Server, Nexus Hosted Repository, Build Metadata, Compliance, Governance, Nexus Cleanup Policy, Artifact Rollback, Nexus Blob Store, Artifact Promotion Workflow, Nexus RBAC Roles, Secure Deployment, Performance Optimization.

I. Introduction

During contemporary DevOps operations build artifacts describe the products generated through software compilation (including examples such as JARs, WARs and Docker images) [1]. The deliverables necessary for deployment are contained within build artifacts that also include application code as well as dependencies and configuration. The maintenance of build artifacts in CI/CD pipelines remains important because every code modification leads to new versioned artifacts that need storage facilities followed by testing before reaching production stages. Team productivity suffers when no artifact management strategy exists because teams struggle to determine artifact versions and perform strategic rollbacks while duplicate and old storage uses up valuable resources and lofty unhandled binaries pose security threats.

Apache Maven stands as one of the most popular tools for building Java applications while managing their dependencies in the programming space [6]. The tool implements a uniform structure for projects through its lifecycle (compile, test, package, etc.) while using Project Object Model software to manage metadata and dependency control. During build processes Maven retrieves necessary dependencies from Maven Central as well as other remote repositories which get stored locally within a cache directory. One of its functions includes packaging project artifacts as JAR files and deploying them to specified repositories. A managed repository becomes necessary when using Maven because each developer machine and CI agent that downloads

dependencies without one incurs redundant expenses in bandwidth usage and build delays.

The Sonatype Nexus Repository operates as a designed tool for storing and managing software artifacts through its repository management properties [6]. The system serves as a consolidated artifact storage solution for businesses which enables group members to publish their final software outputs and share these files within the organization boundaries while maintaining external dependencies within the data center. Within the network environment Nexus provides functionality for both proxying external public repositories and private repository hosting which makes Maven (and its related build tools) able to find dependencies and deploy artifacts effectively. Nexus delivers organizations a unified location for their binaries together with both authorization rules and search capabilities. The combination of Maven and Nexus within CI/CD workflows solves the majority of artifact management issues because Maven provides standard artifact creation practices as Nexus ensures artifact version control coupled with efficient distribution to all pipeline stages.

The combination of Maven and Nexus in CI/CD frameworks enables software delivery processes to reach new heights by providing uniformity alongside strong accountability alongside enhanced operational productivity. The practice of deploying artifacts to Nexus after every successful build enables teams to prevent “works on my machine” issues which allows identical artifacts that passed CI tests to be automatically deployed to production for immutable delivery. Safe tracking of artifacts becomes possible through Nexus by linking each component to version information and checksums and deployment source records which track where artifacts get deployed. The use of caching alongside artifact reuse delivers improved efficiency because Nexus repositories keep dependency caches active and deployment speed increases when working with previously produced artifacts. When Maven and Nexus integrate into CI/CD systems they enable both accelerated release periods as well as enhanced trust in released artifacts.

II. Literature Review

Artifact management represents a fundamental component of DevOps continuous delivery practice where multiple authoritative sources within the industry establish its importance. The Apache Maven project declares repository manager usage to be the basic requirement for essential Maven use. Organizations achieve better build reliability while cutting external downloads by directing dependency retrieval and artifact deployment through Nexus or JFrog Artifactory. Nearly every practitioner piece and community blog recommends the use of repository managers. Sonatype indicates that the Nexus repository manager held 120,000 active installations together with millions of users across many DevOps toolchains during 2017 [9]. Software development teams with high performance now use specialized artifact management tools regularly since they have become standard practice.

Multiple current developments in artifact management emerge from both industry reports and academic research. The increase in supported software formats demands multi-format repository solutions to be implemented. Modern applications require different formats than just back-end JARs and need access to both front-end bundles and container images and Helm charts. Nexus Repository functions as an all-in-one repository solution that accepts Java components together with NPM packages Python packages (PyPI) and Docker images and maintains proxy caching and hosted (internal) repositories for these supported formats. The free version of Nexus OSS provides access to more repository formats than Artifactory OSS since Artifactory OSS needs paid tiers for supporting Docker registries. Nexus provides an affordable option that supports all artifact types enabling teams to find a cost-efficient solution.

Much literature focuses on artifact management systems as they implement both DevSecOps principles and facilitate traceability of components. The controlled repository system enables teams to track exactly which artifact resulted from a particular build committed from a specific repository and which environment received it. The increasing usage of artifact repositories serves as a means to implement quality gateways supported with security assessments. The Nexus platform from Sonatype delivers automated open-source dependency security checks through Nexus Lifecycle and Nexus Firewall prior to promotion. The implementation of Software Bill of Materials (SBOM) documents now requires storage as artifact objects. According to a Sonatype community post published in late 2023 an organization should treat SBOMs as build artifacts and store them in Nexus alongside binaries [2][3]. The process secures an SBOM document that contains a list of release components and dependencies while also delivering it with its artifacts for compliance and security audit purposes which are relevant to DevSecOps practices today.

Technical blog reports compare practical aspects of Maven with Nexus against other alternatives for the reader's understanding. Nexus and Artifactory for various criteria like repository format support, access control, and cleanup policies. This evaluation finds Artifactory superior in terms of user interface and organizational features but Nexus OSS produces a robust platform without needing licenses [5]. Research findings indicate that artifact maintenance systems with storage strategies become vital because artifact collections inevitably expand in size. The lack of policies enables a CI server to fill its repository with countless snapshot builds. The cleanup policies within Nexus enable users to delete unused artifacts by setting a time-based threshold for purging.

According to this body of literature artifact management goes beyond mere storage because it encompasses the complete lifecycle that starts with artifact creation and proceeds through different stages until obsolete artifacts need archival or deletion.

Multiple industrial case studies demonstrate how organizations face substantial benefits from implementing mature artifact management systems. The State of DevOps reports which include DORA reports consistently relate high-performing teams to automated builds and deployments that artifact repositories enable. Through their internal system Spinnaker Netflix reduced deployment times by four times with more straightforward rollback capabilities. The core concept of artifact-first pipeline citizenship which Netflix developed forms the foundation of off-the-shelf technology Nexus. A large eCommerce firm achieved solution when they added a Nexus repository between their production environment and Jenkins to stop version inconsistencies which caused deployments to fail. Academics and practitioners agree that targeted artifact management through Maven and Nexus and their equivalents forms the essential base for reliable CI/CD which delivers faster more secure and automated software deployment [10].

Architecture & Workflow

CI/CD pipelines implement Maven and Nexus with a basic yet effective architectural design. The main objective for this process is to direct all build artifacts through Nexus beforehand so they achieve centralized access throughout pipeline stages. The general process operates at this elevated level:

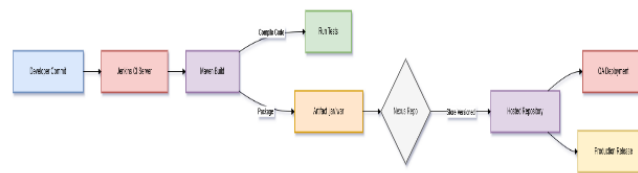


Figure 1: CI/CD Pipeline Artifact Flow with Jenkins, Maven, and Nexus.

The flow is managed by a CI server named Jenkins. The Maven build operates automatically after Jenkins detects every new commit in the codebase. Through Maven developers gain access to code compilation while testing runs afterward the system then packages an artifact such as .jar or .war. The artifact publishing process either uses Maven deployment tools or Jenkins plugins for release to Nexus where it gets saved in a versioned hosted repository. The artifact stored in Nexus functions as the source material during pipeline stages that follow the QA deployment until production release. The deployment of tested binaries from QA to production remains consistent since this process ensures exact replication. The deployment automation process uses Nexus as the authoritative artifact repository while Jenkins serves as the system that conducts artifact transfers across Nexus during the continuous integration process.

The configuration process for developers and CI jobs allows Maven to connect with Nexus for dependency resolution while using it for deployment purposes. Maven depends on a Nexus group repository to get dependencies through while the project POM specifies Nexus hosted repositories for its output deployment. A developer code push through Git activates the CI pipeline by triggering webhooks or Git hooks to Jenkins. The latest Git repository code gets retrieved by Jenkins as part of the process.

- **Jenkins (Continuous Integration Server):** The pipeline script (Jenkinsfile) implemented by Jenkins includes three stages that perform the build and then move onto testing before deployment. Jenkins activates Maven to execute mvn clean package commands for source code compilation in the Build stage. The Build stage allows Jenkins to provide Maven with settings about Nexus credentials and URL connections (we will explain this functionality through the implementation section).
- **Maven (Build Tool):** The Maven build process starts when it executes its lifecycle through source code compilation followed by running unit tests until successful completion leads to artifact packaging. Maven executes the deploy phase to move generated artifacts and their POMs to Nexus repositories when properly configured on release-build executions. Jenkins manages the upload as a standalone process if Maven deploy goal is not utilized [6].
- **Nexus Repository Manager:** The artifact storage location is a Hosted Maven repository of Nexus where the artifact enters along with its groupId:artifactId:version coordinate. During upload Nexus courses several checksum values for integrity purposes and adds the groupId:artifactId:version entry according to the Maven POM. A consumer with proper permissions can access the artifact in Nexus [6].

- **Continuous Deployment/Delivery:** Without code rebuild Jenkins (or its corresponding CD tool) retrieves the artifact content from Nexus during the later development stages. The A Deploy to QA stage implements a process which retrieves version 1.2.3-SNAPSHOT of the app from Nexus by using Maven or curl or a deployment plugin before deploying to the test server. The ability of Nexus to provide HTTP serving of artifacts enables pipeline tools and their administrators to access any build through its defined URL. The Promote to Production stage obtains the same artifact from Nexus release repository to deploy it to production servers after successful quality assurance tests. Users can implement artifact promotion through a straightforward procedure of repository “staging” to “releases” (Nexus enables staging operations that support repository-targeted release management). The above deployment process separates the operations of building from deployment purposes. The artifact creation process takes place just once before moving toward validation and storage steps. A single artifact built for deployment purposes can be used multiple times to different environments thus eliminating environmental factors from the build process. Nexus stores the previous version of code that enables easy rollback procedures because faulty new versions can be redeployed using the available backup version without requiring the recreation of older code. The architectural design leads to performance improvements because Nexus maintains dependency caches that shorten compilation and testing processes while deploying prebuilt artifacts is faster than building from scratch.

Some key elements must be incorporated for achieving a smooth flow. The Nexus system requires a repository structure that separates its artifacts by type and distributing them according to their lifecycle stages. One standard repository organization uses “maven- snapshots” for unstable and nightly builds alongside “maven-releases” for stable release builds. A Maven project with a “-SNAPSHOT” suffix in its version ends up at the snapshots repo yet versions without suffixes (releases) get deposited in the releases repo. The separation enables different retention policies because snapshots can be cleaned up frequently but releases stay long-term. The group repository feature of Nexus allows developers to specify one URL for dependency resolution which retrieves artifacts from releases, snapshots or proxy-based sources automatically. Multiple repositories should separate snapshots from releases because mixing them produces clutters and potential data loss from misconfiguration.

Security authentication mechanisms and access control requirements need evaluation for this architecture design. The Nexus system must have credentials assigned to Jenkins for publishing content. The automation account known as “jenkins-user” should get created inside Nexus with deploy authorization to the hosted repositories. There are two approaches to credential management: Jenkins Credentials to secure storage followed by pipeline reference to these credentials. Nexus provides two authentication options for read access (pulling artifacts) which includes anonymous access or credential requirements according to security protocols. The teams in typical production environments limit access to Nexus at both read and write levels since they utilize credentials within VPN environments to access the repository. The Role-Based Access Control in Nexus allows authorized systems/people to fetch or publish specific artifacts through its expanded Best Practices section.

Jenkins + Maven operate for CI purposes alongside Nexus functions as an artifact repository through well-defined server interaction where Jenkins/Maven transfers artifacts to Nexus then Jenkins (or deploy tools) obtains artifacts from Nexus. The integrated system produces reliable and auditable artifact flow starting from source code commit and continuing through to production deployment. The architecture's concrete deployment will be explained through detailed implementation instructions that present practical configurations as well as robust best practices.

Technical Implementation

A proper integration of Maven and Nexus for CI/CD workflow requires correct configuration for each element:

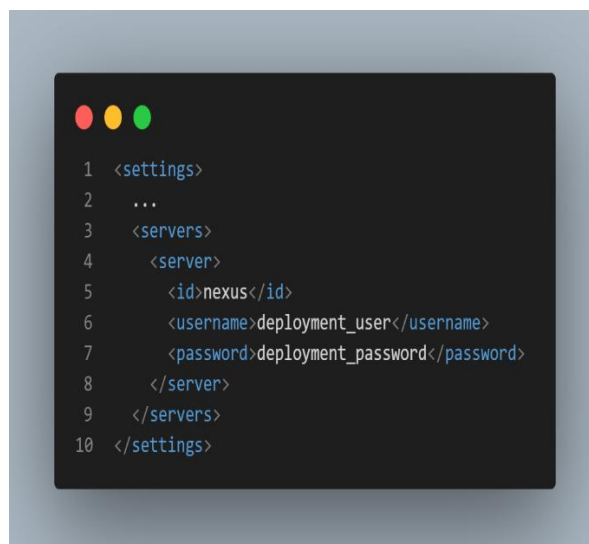
1. Maven Configuration for Artifact Generation and Deployment: Projects developed with Maven need to use a pom.xml document for configuration. A successful Nexus deployment requires adding the <distributionManagement> section with Nexus repository URLs inside the POM file. The Maven POM contains the following definitions:



```
1 <project>
2   ...
3   <distributionManagement>
4     <repository>
5       <id>nexus</id>
6       <name>Releases</name>
7       <url>http://nexus.example.com/repository/maven-releases</url>
8     </repository>
9     <snapshotRepository>
10      <id>nexus</id>
11      <name>Snapshots</name>
12      <url>http://nexus.example.com/repository/maven-snapshots</url>
13    </snapshotRepository>
14  </distributionManagement>
15  ...
16 </project>
```

Figure 2: maven-distribution-management-pom.xml.

Because of this configuration Maven identifies the appropriate Maven server repository for non-snapshot release versions (maven-releases) and snapshot versions (maven-snapshots). The `<id>` (nexus in this case) is a key that corresponds to credentials in Maven's settings. In the developer's or CI server's `~/.m2/settings.xml`, a server entry provides the username/password for that ID:



```
1 <settings>
2   ...
3   <servers>
4     <server>
5       <id>nexus</id>
6       <username>deployment_user</username>
7       <password>deployment_password</password>
8     </server>
9   </servers>
10 </settings>
```

Figure 3: maven-settings-authentication-nexus.

The secure credential storage system enables Maven to access Nexus through the credentials during an `mvn` deploy operation while the POM file remains in the public domain without VCS password inclusion. Executing `mvn clean deploy` performs three actions including project compilation and testing and subsequent upload of the artifact and its `.pom` file to Nexus. The deployment of a Java artifact to Nexus occurs when you include distribution Management in your configuration then execute `mvn deploy -DskipTests` – and Nexus will display “Artifact built and deployed to Nexus successfully” [6].

Users who prefer not to add repository URLs to POM files can trigger Maven's deploy plugin operations through parameterization. The project demonstrates harnessing `mvn deploy:deploy-file` goal by enabling manual artifact uploads through file path and coordinate specification in the command line input. This method enables deployment of external artifacts or allows alternate Maven building methods without complete POM use. The configuration of POM files in stable CI/CD frameworks occurs once before teams use the `mvn` deploy command.

2. Nexus Repository Setup: Maven requires you to establish the repositories which it will use on your Nexus platform. The configuration process for Nexus Repository OSS 3.x starts by creating (a) a Hosted repository named “maven-releases” with Maven type and Release version policy and then (b) another Hosted repository dubbed “maven-snapshots” with Maven type and Snapshot version policy and (c) optionally builds a Group

repository called “maven-public” to merge hosted and proxied repositories including Maven Central. You must specify deployment rules during the process of adding a hosted Maven repository within Nexus. The deployment policy for snapshots allows repeated redeployments of the same version but releases usually disable redeploy to prevent version replacements. Each Nexus repository has its own distinct URL which is usually presented as `http://<host>:8081/repository/<repo-name>`. Both these URLs are required within the Maven POM configuration. Add Nexus credentials together with roles to the configuration by setting up a user account which CI will use to deploy artifacts. The application must have a user profile named “jenkins” which carries a role that enables nx-repository-view rights to perform edit (deploy/delete) functions within hosted repository access. At present some organizations choose to assign the Jenkins user the pre-defined “deployment” role to handle artifact upload capabilities directly. Production work should not utilize Nexus admin account while it remains acceptable for testing purposes. The Nexus user credentials for username and password must be saved within Jenkins system or stored within Maven settings located on CI agent under the described method.

The requirement for setting up proxy functionality through Nexus remains minimal. The default configuration of Nexus includes a pre-existent Maven Central proxy repository known as “maven-central.” The creation of a new Proxy repository of Maven type pointing to `https://repo1.maven.org/maven2/` (Maven Central URL) falls within the administrative capabilities. Developers can use Nexus group URL as their mirror in Maven settings to access all dependencies including internal and external ones through the “maven-public” group. The implementation allows for accelerated build processes that maintain the caching of both inner and external dependencies inside company Nexus which enhances public repository resilience [7].

3. Jenkins Pipeline Configuration: Jenkins functions as the automation tool for integrating Maven with Nexus once both platforms are prepared. The “Nexus Artifact Uploader” plugin must be added to Jenkins in addition to Maven installation on Jenkins nodes or the use of Maven container solutions. The integration of Nexus can be accomplished through two main methods in Jenkins pipelines.

- **Using Maven Deploy in Pipeline:** Execution of the “`sh 'mvn clean deploy'`” command in the Jenkinsfile (pipeline script) allows Maven to deploy to Nexus while adhering to the POM configuration. This process requires Maven to be available. Using this method depends on the built-in settings from Maven and it remains easy to execute. The `settings.xml` file containing Nexus credentials becomes the only requirement Jenkins needs to pass to Maven for successful deployment. The Config File Provider plugin together with a `withMaven` step enables you to provide custom settings files which pre-prioritize the `mvn` deploy command. Most CI setups contain a Maven global settings that automatically reflects all external repositories to Nexus through the `<mirrorOf>*</mirrorOf>` reference to the group repository and includes deployment server credentials.
- **Using Jenkins Nexus Plugin:** The Nexus Artifact Uploader plugin can be used independently during a dedicated step. This Jenkins Declarative Pipeline data segment for release processes would appear as follows:



```
1 stage('Publish to Nexus') {
2   steps {
3     script {
4       def pom = readMavenPom file: 'pom.xml'
5       def artifactId = "target/${pom.artifactId}-${pom.version}.${pom.packaging}"
6       nexusArtifactUploader {
7         nexusVersion: 'nexus',
8         protocol: 'http',
9         nexusUrl: 'nexus.example.com:8081',
10        groupId: pom.groupId,
11        artifactId: pom.artifactId,
12        version: pom.version,
13        repository: 'maven-releases',
14        credentialsId: 'nexus-credentials-id',
15        artifacts: [
16          [artifactId: pom.artifactId, file: artifactFile, type: pom.packaging],
17          [artifactId: pom.artifactId, file: 'pom.xml', type: 'pom']
18        ]
19      }
20    }
21  }
22 }
```

Figure 4: Jenkins groovy nexus artifact upload stage

The example snippet follows Sonatype’s pipeline template to read POM coordinates automatically before uploading Nexus both artifact files and POM [1]. The `credentialsId` value indicates to Jenkins that it should access Nexus with stored authentication data that consists of username/password combination or API token authentication. With this approach managers can bypass `mvn` deploy commands because Jenkins takes full control of explicit publishing tasks. The plugin serves as a valuable solution when users need to manually upload artifacts beyond Maven formats to Nexus. Maven implementation in our context supports both deploy methods although running Maven deploy directly is straightforward when POM settings are reliable or the plugin provides additional control in Jenkins pipeline scripts.

The Jenkins workspace needs to have the artifact built prior to initiating the upload process regardless of what upload method you use. A typical pipeline begins with checking out Git code then proceeds to Build followed

by Unit Test and finally Publish. The publish process needs to execute only after the successful build/test phase. When dealing with multi-module Maven projects you should determine how you will retrieve artifacts either by using `${project.build.directory}` or attaching artifacts.

4. Example Jenkins Pipeline (Jenkinsfile): A simplified Jenkinsfile with declarative syntax can appear thus:



```
1 pipeline {
2   agent any
3   tools { maven "Maven_3.8.5" }
4   environment {
5     MAVEN_OPTS = "-Dmaven.repo.local=.m2/repository"
6   }
7   stages {
8     stage('Build') {
9       steps {
10        // Run Maven package (not deploying yet)
11        sh 'mvn -B clean package'
12      }
13    }
14    stage('Publish') {
15      when { branch 'main' } // only publish on main branch, for example
16      steps {
17        // Use Maven deploy or Nexus uploader
18        sh 'mvn -B deploy'
19      }
20    }
21    stage('Deploy to Dev') {
22      steps {
23        // e.g., call an Ansible playbook or subctl, which pulls artifact from Nexus by URL
24        sh 'ansible-playbook -i dev deploy.yml --extra-vars "artifactVersion=${ARTIFACT_VERSION}"'
25      }
26    }
27  }
28  post {
29    success {
30      echo "Pipeline completed successfully."
31    }
32  }
33 }
```

Figure 5: Jenkins declarative pipeline maven-nexus-ansible

In this script: Our first step in this script involves Maven build and packaging procedures. The execution of mvn deploy on main branches triggers a deployment to the Nexus server where the credentials and URLs exist as configured settings.xml on this Jenkins node. Following successful publication the deploy stage will start while using the artifact stored within Nexus. The Jenkinsfile obtains pom.version as an environment variable to let an Ansible script download [http://nexus/repository/maven-releases/com/example/app/\\${ARTIFACT_VERSION}/app-\\${ARTIFACT_VERSION}.jar](http://nexus/repository/maven-releases/com/example/app/${ARTIFACT_VERSION}/app-${ARTIFACT_VERSION}.jar) deployment content. CI and CD functions separately until their artifacts become accessible through the artifact repository.

5. Nexus Usage: Hosted, Proxy, Group Repositories: Users managing different artifact types can benefit from Nexus repository types due to their flexible operations design. Nexus positions the Hosted repository as a repository for artifact publication. Among our Maven configuration we maintain the “maven-releases” and “maven-snapshots” repositories. The Proxy repository serves as a local storage for remote repositories in which Nexus's Proxy for Maven Central enables Maven builds to retrieve dependencies from the caching platform. A Group repository acts as a practical combination of various repos through one convenient wrapper. The default maven-public group in Nexus contains the combination of maven-releases, maven-snapshots, and central (proxy) to handle retrieval from a single URL source. You have the ability to establish this group as a mirror for all repositories under your Maven settings so dependency requests consistently start with Nexus. The artifact caching functionality of Nexus enables your builds to continue without interruption from Maven Central network issues as long as you have the required dependencies stored locally in Nexus. Security increases through your ability to determine which external artifacts Nexus can access by means of whitelist/blacklist rules or firewall plugins.

6. YAML/Code Snippets for CI: The configuration of pipelines as code for Jenkinsfile takes place when using Jenkins in Docker/Kubernetes setups or Jenkins X installations (illustrated below). Any CI system including GitLab CI and GitHub Actions can access Nexus storage through command line interface functions. GitLab CI YAML employs the Maven image to perform publication tasks.



Figure 6: gitlab ci maven nexus deploy snippet

A container would execute this task which implements both build and deploy operations against Nexus. Both approaches follow the same basic principle whether through the use of Maven with proper configuration and script-based direct upload.

You can verify that your artifact appears under the hosted repository section after finishing the described setup process when you access Nexus UI. The Nexus interface shows group ID and artifact ID directories (for Maven users) to locate your uploaded artifact file through its directory path. The URL for direct access can be copied from this location while metadata options are also visible. The consistent naming and versioning of CI artifacts remains essential because Nexus maintains numerous versions yet semantic versioning or a clear snapshot scheme help users determine which version is current. Best Practices contains details about versioning conventions along with retention methods. (The upcoming section on Best Practices will focus on this information.).

The technical part of implementation requires changes to three components: Maven pom.xml and settings files together with Nexus repository configuration and user privileges setup and Jenkins pipeline setup. The CI/CD system maintains artifact management automatically through correct setup of deployment jobs which retrieve Nexus files by their version number following correct builds which cause new Nexus files to be created. The artifact repository becomes both quicker and more dependable because of this implementation which allows for audits and cleaning tasks.

Best Practices

Maven and Nexus implementation for CI/CD necessitates more than simple initial setup because best practices ensure that the system stays efficient, secure and easy to maintain throughout its operational period. This discussion centers on major best practices which pertain to artifact versioning as well as retention and security and promotion management.

Versioning Strategies: Organizations should make artifact versioning strategies their top priority. A common approach is Semantic Versioning (SemVer) – using a MAJOR.MINOR.PATCH format (e.g., 2.5.0) for releases. Semantic versions carry meaning: Changes with incompatibilities rise the MAJOR version number while new features raise the MINOR version number and fixes trigger PATCH version number increases. The versioning system enables humans and tools to identify release importance effectively. Numerous organizations practice version immutability as a best practice intended to reduce ambiguities and the "it works on my machine" problems. Snapshot versions receive higher update priority checks from Maven for systems running nightly artwork compilation or Continuous Integration builds. A released version cannot be reused for new builds because artifact coordinate must always point to a single binary. Builds which must be restarted require new version numbering or addition of build metadata. Teams ensure version uniqueness by adding build numbers and commit hashes either to artifact versions or as part of their metadata system. Some development teams include build numbers and commit hashes in their artifact versions as metadata elements to establish unique IDs. A CI pipeline generates artifact version 2.6.1-build.45 during production. The artifact version 2.6.1-build.45 will receive promotion to the final release version 2.6.1. The success depends on a well-defined scheme which matches your release approach alongside proper version update training for the team through POM.

Handling SNAPSHOT vs Release Artifacts: Each Nexus repository must have its own lifecycle rules and developers should use SNAPSHOTs exclusively during development. SNAPSHOTs should be used only in development; The project POM should receive an updated version change when release time arrives before moving artifacts to releases repo through Nexus which enables staging repos that need manual closure to avoid unintended incomplete artifact releases. The recommended workflow for production deployments requires disabled snapshots because the production environment must retrieve fixed releases from the releases repository. The approach prevents repeatedly changing build results from appearing in production. Periodic removal of unnecessary snapshots is recommended under retention policies while allowing storage management of your system.

Artifact Retention and Cleanup: A large number of builds accumulate in artifact repositories over extended periods of time. Your organization needs to create procedures for artifacts deletion. Through Cleanup Policies in Nexus 3 you can set conditions to delete unused content in your system by defining rules. The experts at Sonatype indicate that applying cleanup policies is most vital for snapshot repositories [2][3]. Your policy may enforce retaining the latest 5 snapshot versions while discarding older artifacts that exceed 2 weeks without use. Release artifacts typically remain in your system (for audit purposes) although you may want to delete pre-release RC builds at a later stage. An official guideline suggests aligning retention with build frequency: The cleanup policy for components in a weekly snapshot repository managed by a team should remove artifacts that have reached more than ten days of age. When you delete artifacts through soft cleanup you should execute the Nexus “Compact blob store” task to recover storage capacity. The lack of proper cleanup in Nexus blob stores caused users to report storage consumption reaching tens of TBs. Running automated cleanup routines prevents repositories from becoming security risks. The Nexus Pro tool enables you to relocate artifacts that remain unused for long periods into cheaper storage systems or dedicated archive repositories (also known as “cold storage”).

Security and Access Control: Nexus implements strong role-based access control features (RBAC). The least privilege principle must be implemented. When deploying CI systems build unique roles and user accounts rather than relying on the default admin credential. The role “CI Deploy” enables artifact creation and update permissions in specific repositories but lacks permission to delete artifacts or access sensitive repos. Developers maintain read privileges to specified repos for dependency access yet release engineers own the exclusive authority to transport artifacts into the release repository. Enabling anonymous access must be disabled in all Nexus production instances. By default Nexus OSS gives anonymous read privileges during setup yet enterprises need to implement Nexus integration with LDAP/SSO authentication for secure authenticated access. Each repository inside Nexus maintains its own permissions system across both repositories and their respective actions using Active Directory, LDAP and precise authorization settings. According to Sonatype's security recommendations organizations need to perform two key security actions: they must replace the default admin password upon initial Enable your CI server to use SSL to connect with Nexus since Nexus can utilize SSL encryption or you can establish TLS through an Nginx reverse proxy.

Security management requires administrators to maintain control over all entries and exits from the repository. Nexus Firewall (Pro) enables content validation by blocking components that contain known vulnerabilities. Nexus OSS allows users to build custom scripts and utilize its REST API for scanning uploaded files as needed. Organizations must establish a process for artifact vulnerability assessment (either through CI pipeline tools like OWASP Dependency Check and Snyk or Nexus Lifecycle solutions are available). Artifacts gain this capability when stored in Nexus due to the available choke point for analysis.

Promoting Artifacts Across Environments: Multi-environment deployments (Dev → QA → Staging → Prod) use artifact promotion to transfer the identical artifact binary from stage to stage. Nexus supports segmenting repositories by environment stage with distinct “development” “staging” and “production” repositories. Artifact promotion triggers when builds succeed testing by performing a file copy or move operation from dev to staging repository in Nexus through scripts or Nexus REST API. The Nexus Repository Pro system includes stage-specific repositories and a REST API for automatic promotion features (the Maven Staging Plugin supports automated Central release promotion). With Nexus OSS, you might have to script it: The artifact retrieval starts from the source repository until successful deployment reaches the target repository. Using separated repositories allows you to define distinct access rules while keeping production-ready assets separate from staging assets. Teams that apply version-based artifact labeling to their single repository can improve their process by implementing Nexus promotion steps.

Some CI/CD systems reference artifacts directly through version numbers instead of using physical promotion. Both the QA deployment and prod jobs retrieve specific versions from the shared “ci-artifacts” repository so they can perform their functions. A release repository serves as protected storage for final artifacts that passed QA testing yet using the same repository as immutable storage after QA completion is more secure. As a rule: never rebuild the artifact for each environment; The rebuilding process for artifacts should occur only a single time since reuse of the built artifact is necessary. Understanding this is crucial to CI/CD because it prevents divergence between system environments.

Logging and Monitoring: Monitoring Nexus repository usage constitutes a standard management practice. Nexus generates logs which track repository events and download activities. The metrics about downloads and storage of artifacts enable you to create optimal cleaning strategies. Regular alerts alongside periodic checks should get triggered when your repositories approach their storage capacity limits. Nexus infrastructure stability requires a clustered or highly available setup (Nexus Pro supports multiple-node High Availability configurations) when Nexus stands as a vital part of your system. Regular backup operations for Nexus serve as a minimum procedure that protects artifact data and metadata besides utilizing Nexus built-in export features. A server crash that destroys your single copy of a released artifact will result in an invalid rebuild of the artifact. This is the worst possible scenario.

DevOps Process Integration: All stages of artifact promotion and release procedures need to have clear documentation. Establish guidelines to define every step of release candidate version number changes, Nexus deployment practices and authorization protocols and recovery protocols. The team requires training for Nexus interface or automation script usage to retrieve artifacts for testing purposes which should replace the current manual rebuilding process. Builds by organizations must obtain every dependency through their repository manager ensuring both whole library caching and use of approved open source libraries with no known license or security problems (controlled OSS library ingestion).

In summary, key best practices include: Semantic Versioning enables artifact versioning that incorporates snapshot-release disconnects together with automated deletion rules for accounts and anonymous control access and artifact promotion instead of rebuilds and security scanning capabilities. The operational efficiency of Maven and Nexus will be sustained by their combined configuration after deployment of these practices alongside improved scalability. A 30-day no-download cleanup policy for snapshots enabled a team to free hundreds of GB of storage without developer performance degradation. The organization deployed a final build release repository structure combined with complete lock protection to enable auditors trace production binaries. By implementing such practices organizations can transform their standard CI/CD pipelines into extensive software supply chains.

III. Challenges & Solutions

The combination of Maven and Nexus efficiently manages artifacts but teams face practical difficulties when using them. Storage bloat together with stale dependencies and artifacts and snapshot sprawl and problematic integrations between CI and the repository remain common issues. We review potential issues that arise along with proposed solutions.

- **Storage Bloat and Performance Degradation:** An active Nexus repository tends to build up its data collection with time. An oversized Nexus repository strains the storage resources and could slow down your system while doing operations on metadata. If cleanups were disabled a snapshots repository would quickly store and waste thousands of unused artifacts. An actual Nexus admin found that their saved builds reached 10 TB because the repository stored builds for multiple years. You need to put the best practice guideline into action for cleaning up your system and managing stored data. Additionally, consider repository partitioning: A large active maven-snapshots repository can benefit from dividing projects across multiple hosted repositories even when this practice becomes hard to maintain. Nexus lets you distribute storage space evenly between repositories or any related group you choose to allocate it. When Nexus slows down Sonatype proposes removing old content from service and enhancing the Nexus hardware configuration. Our Pro feature lets you spread workloads across multiple servers. Regularly remove unused products while watching Nexus performance to stop project slowdown.
- **Stale Dependencies (Caching Issues):** A problem can happen when a proxy repository caches external dependencies that receive updates in their original repositories. When Nexus gets snapshots from outside repos it delivers stored preceding snapshots instead. When Maven checks snapshots depends on your configuration choices but if you set it wrong you might build outdated results. Set snapshot update policies on the proxy server that match your requirements through Nexus software (choose to query the remote server every day or update instantly). To handle unpredicted problems Nexus administrators can choose to delete the cached data from a certain repository manually. The problem called dependency hell occurs when different versions of libraries exist in the build system because it leads to unpredictable dependency picking. Solution: The system can regulate available versions through Nexus procurement tools or failing this Maven dependency Management can lock the versions. Use the repository manager to filter the unauthorized third-party components before developers get access to their build resources.
- **Snapshot Overload and Unique Snapshots:** The Maven system helps but needs special handling with SNAPSHOT versions. When enabled by default Nexus saves each published snapshot version as its own time-stamped copy. Before long 100 builds of version 1.0-SNAPSHOT create 100 files in Nexus repository. The purpose is to let builds access specific timestamped snapshots as needed. Having too many snapshots will create data overload without cleaning. To achieve better results you should set up regular cleanup tasks instead of using

Maven deploy plugin options. Wipe out Snapshots or create a special script to keep just the newest n snapshots. Nexus 3 Cleanup Policy enables matching and keeping of the latest X snapshots up to a defined day-range. Nexus creates specific rules with the purpose of managing snapshot versions effectively which make up its main challenge. Nexus 2 featured a Scheduled Task to automate deletion of older snapshots. Take action to control snapshot growth so it does not overload your setup.

- **Build Promotion Complexity:** The plan to promote artifacts is useful yet proving it in practice poses difficulties. Teams regularly encounter difficulties when they need to handle multiple repositories during promotion or file transfers between them. Manual promotion steps increase the chance of human mistakes. Our answer is to create a computer system that handles promotion tasks. Jenkins sends the Release approval via Nexus REST API to initiate artifact movement. When using Nexus OSS switch to Nexus Pro / OSSRH for Central instead of employing the Maven Release Plugin. The hard part is maintaining the artifact's location as it moves through the promotion process. Typically you would move the artifact between repositories without changing its version number. Documenting this process and setting up automation tools makes deployment more efficient. JFrog Artifactory simplifies promotion with features like build info and quality gates. You must create scripts to manually stick pieces together when utilizing Nexus. Companies know this issue exists yet they can fix it with programming commands that update Nexus artifact deployment.
- **Access Token Expirations / Auth failures:** The deployment process of CI jobs often fails because users register wrong passwords or lose access during preparations. This running problem gets solved through strong secrets management integration. Swap hardcoded credentials for Jenkins credentials and consider moving from raw passwords to Nexus 3 API tokens. Change your security credentials regularly and push the updates to your CI platform right away to prevent interruptions.
- **Tool Compatibility and Plugins:** Make sure Jenkins plugins and Maven version run without issues with Nexus API. The Jenkins Nexus uploader plugin normally works with Nexus because it uses stable REST API. Nexus OSS does not offer staging workflow functions so you need to verify your Nexus version supports Maven Release Plugin staging workflows. Nexus Pro offers this feature but you can also get the same result through OSSRH for open source projects if required. Follow the steps outlined to construct your own simpler promotion method if necessary.

Working teams have developed methods to handle these problems in their regular activities. The company detected slower CI builds as their collection of artifacts increased. They set up a retention policy to keep Nexus artifacts for thirty days except those downloaded within seven days. This deleted obsolete artifacts which restored normal build times. After their problem with a faulty release one team learned that Nexus still had their previous build system though it was almost too late because their retention period had expired. Post-mortem: they made sure changed their policy to maintain both latest releases after the update.

Summary of Solutions: Implement Nexus cleaning rules in advance and split release and snapshot operations while automating promotion tasks. Also safeguard credentials and monitor Nexus utilization constantly. Use resources from Sonatype's Knowledge Base when using their Groovy scripts for bulk cleanup or repository transfers. The methods to overcome various Nexus and Maven issues create better artifact processing in CI/CD environments.

IV. Case Study: Jenkins, Maven, and Nexus in a Real-World Project

We will explain our ideas by describing a real-world project that applies Jenkins, Maven, and Nexus together. A financial services team that practices CI/CD develops a Java web application project. Our main target is to construct and introduce the .war file. We transfer the web application archive file (.war) to Tomcat servers in different environments. This includes Development, Quality Assurance, and Production zones with Jenkins as the controlling platform.

Project Overview: Maven helps build the application. At each step in the develop branch when merging into main branch the team applies the Snapshot versioning system. Nexus functions inside our internal server nexus.company.com. The system uses Linux to host Jenkins and each Jenkinsfile source code lives in Git.

Pipeline Implementation: Jenkinsfile defines three core building stages that the project needs. Jenkinsfile covers three main actions: Preparation, Packaging and Deployment.

- During Build stage Jenkins uses mvn clean verify on the develop branch to start Continuous Integration (CI) testing and compilation. This system trains the automation system to test each new change. Once tests succeed the build becomes a snapshot Jenkins moves to the next step.
- **Package & Publish stage:** Jenkins triggers mvn deploy while on the main release branch for any snapshot updates we want to distribute. Maven is configured (via the pom's distributionManagement and settings.xml on Jenkins) to deploy to Nexus. For example, a successful build of version 1.4.5-SNAPSHOT on develop will deploy app-1.4.5-YYYYMMDD.hhhmmss-1.war to the maven-snapshots repo on Nexus. If it's a release build on main with version 1.4.5

(no SNAPSHOT), Maven deploys app-1.4.5.war to maven-releases on Nexus. Jenkins provides the credentials to Maven through an injected settings.xml with a server ID “nexus” (mapped to a Jenkins credential). After this stage, the artifact is available in Nexus. Jenkins records the build number and version as environment variables for the next stage.

- **Deploy stage:** It has a deployment job that allows customization of settings. To serve the development team faster results the Dev setup automatically deploys every snapshot to production. Develop branch updates send reaction by scheduling a Tomcat deployment on the Dev environment right after the snapshot upload to Nexus. We fetch the artifact through a script to perform its deployment from Nexus. In this scenario the pipeline runs a shell operations using these commands.

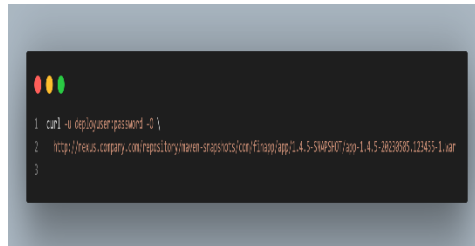


Figure 7: Nexus Artifact Download using cURL

(the exact URL is constructed; The system builds LATEST snapshots when snapshot capability is active Nexus). People need to download the artifact and apply an scp or Docker copy method to move it onto their Dev Tomcat container and reboot the service. The system pushes new development changes to Dev quickly and continuously.

The team applies these updates to both environments but at minimal frequency through official product deliveries. Jenkins records version 1.4.5 as a release candidate after the team merges into main and performs the Jenkins build. The QA team receives notifications while a QA pipeline job starts processing to obtain app-1.4.5. The release pipeline retrieves war from releases repo and deploys it to QA Tomcat. They test the application. If the team identifies an issue in version 1.4.5 The development team enhances code to fix the issue then increases the version number to 1.4.6 and restarts the process. Eventually, version 1.4.6 passes QA. They deploy the Prod pipeline when VERSION=1.4.6 is set as the parameter. The pipeline fetches app-1.4.6. Our system downloads Nexus-stored war files version 1.4.5 and sends them to all production server endpoints. The team relies entirely on pre-built application packages during deployment since they maintain their trust in those packages.

Rollback Scenario: For production deployment of version 1.4.6 the team needs to prepare a return plan because unexpected problems may happen. The team returns to operating version 1.4.5 because it worked well before. Thanks to Nexus, this is straightforward – the 1.4.5 artifact is still in the releases repository. The ops engineer can go to Nexus GUI, locate app-1.4.5.war (which is still present because releases are not deleted), and click “Copy to Clipboard” for the download URL. They activate Jenkins rollback job automatically or run a special script to restore the latest production version from Nexus. The service restarts version 1.4.5 in less than a minute. This task took far less time than rebuilding it from the source material. This technique keeps running the same program code without introducing errors between releases. The team points out successful rapid rollback service restores in its post-mortem proceedings because having Nexus creates advantages. Our team successfully restored the previous stable version without manual searching because of Nexus technology.

Benefits Observed: During a few months this system delivered various positive results for the team.

- **Consistency:** The implementation of a single WAR file across different environments eliminated deployment anomalies which triggered discrepancies between QA and Prod. Nexus functioned as the reference point for deployed systems.
- **Faster Deployments:** It takes less time to deploy via file copy compared to executing a complete Maven build. Deployments utilizing Nexus-based methods decreased in duration from approximately fifteen minutes to build and deploy the artifact down to less than two minutes for artifact deployment. The deployment time benefited from improved speed which resulted in shorter lead times.
- **Secure Storage:** Before Nexus implementation developers used to manually store .war files on a fileshare network drive. Users would formerly save .war files on a network drive before migration to manual operation. The manual storage process was both error-sensitive and unregulated. Nexus provides a system that enables access control through RBAC for all artifacts. The CI service and release managers gained complete control to delete items from Nexus while other users retained only read access to the system. others have read. The system blocks unauthorized modification or unintentional removal of data. Users made mistakes during manual file storage because files were not versioned which led to the risk of overwrites. The versioned directories in Nexus eliminate directory-related uncertainties.

- **Traceability:** The company integrated with their change management process through Nexus API. All production deployments require Nexus artifact checksums and URLs to be documented in change requests. Auditors verify deployment contents through later audits. Nexus checksums can serve as a reference for binary comparison within the system. Before formal artifact management systems were in place this level of trackability was unattainable.
- **Cleanup and Maintenance:** The team implemented scheduled Nexus Cleanup Policy for snapshots which deletes old snapshots beyond thirty days that have zero download activity. Snapshots automatically get deleted if they exceed 30 days of existence without anyone downloading them. This keeps storage usage manageable [2][3]. The research revealed that developers required snapshots only within one week at most hence rebuilding from Git was an effective alternative if needed older data. Release data stays on Nexus because they maintain a single monthly release schedule. Regular backups of Nexus repositories occurred offsite although Nexus developed into an essential point of truth.

Tools and Configuration Recap: When first implementing Nexus Artifact Uploader Jenkins served the team but they eventually selected plain mvn deploy for its simplicity. Maven's settings.xml on Jenkins contained:



```
1 <servers>
2   <server><id>nexus</id><username>jenkins</username><password>{API_TOKEN}</password></server>
3 </servers>
4 <mirrors>
5   <mirror>
6     <id>internal-nexus</id>
7     <url>http://nexus.company.com/repository/maven-public/</url>
8     <mirrorOf>*</mirrorOf>
9   </mirror>
10 </mirrors>
```

Figure 8: Maven settings.xml Configuration for Nexus Authentication and Mirror Setup

Internal-nexus was implemented to validate that all dependencies received proper internal processing. The implementation blocked the build agents from accessing external Maven Central and made their builds immune to dependencies when Maven Central suffered an outage since Nexus maintained dependency cache.

On Nexus, they created a Role called ci-deployment that allowed nx-repository-view-maven-*-upload on their Maven hosted repos. User Jenkins received this particular role assignment. The security measure added restricted outsider access to hosted repository metadata while requiring authentication for viewing purposes. The Maven Central proxy repository maintained open access for anonymous readers because it contains publicly accessible artifacts.

Lessons Learned: The original deployment error resulted in both snapshot and release deployments targeting the same repository. The POM received incorrect setup by a new team member who selected both snapshot and release repositories. Snapshot artifacts appeared in the release repository creating confusion about the deployment (snapshot artifacts in the release repo). To address this issue they established different code depositories while providing proper training to their team members. They also added a Nexus script to quarantine any *-SNAPSHOT version that somehow appears in the releases repo, just as a safety net.

Another lesson: The execution of Maven tasks required configuration through Jenkins options which allowed them to prevent oversize file transfers on their sluggish link when the WAR file exceeded 50MB across different data centers that hosted Jenkins VM and Nexus. To address the issue they positioned Nexus alongside the same data center for improved network speeds and preset Maven to conduct extra failover attempts.

The implementation of Jenkins + Maven + Nexus together proved to enhance the CI/CD pipeline operation of an actual project. The configuration offered developers quick feedback cycles and automatic deployment capabilities together with artifact versioning and easy rollback systems for safer releases along with auditing capabilities. The project required disciplined version control and configuration work yet users experienced noticeable improvements in both speed and stability through their release process.

- artifactory#:~:text=From%20my%20opinion%2C%20I%20left,any%20comments%20on%20that%20point
- [6]. Rodolfo Costa, “Deploying Java artifacts on a Nexus Repository with Maven.” Medium, Jan 2021.referred from: <https://rodolfofombc.medium.com/deploying-java-artifacts-on-a-nexus-repository-with-maven-939a80406acf#:~:text=%3Csettings%3E%20%3Cservers%3E%20%3Cserver%3E%20%3Cid%3Emy,password%3C%2Fpassword%3E%20%3C%2Fserver%3E%20%3C%2Fservers%3E%20%3C%2Fsettings%3E%20%3CdistributionManagement%3E%20%3Crepository%3E%20%3Cid%3Enexus%3C%2Fid%3E%20%3Cname%3ERelases%3C%2Fname%3E%20%3Curl%3Ehttp%3A%2F%2Flocalhost%3A8081%2Frepository%2Fmaven,snapshots%3C%2Furl>
- [7]. Sonatype Help, “Maven Repositories – default configuration.” Sep 2023 Referred from: <https://help.sonatype.com/en/maven-repositories.html#:~:text=%3CdistributionManagement%3E%20%3Crepository%3E%20%3Cid%3Enexus%3C%2Fid%3E%20%3Cname%3ERelases%3C%2Fname%3E%20%3Curl%3Ehttp%3A%2F%2Flocalhost%3A8081%2Frepository%2Fmaven,snapshots%3C%2Furl>
- [8]. Dmitriy Akulov (Sonatype resource), “Publishing Artifacts to Sonatype Nexus using Jenkins Pipelines.” June 12, 2020.Referred from: <https://www.sonatype.com/blog/workflow-automation-publishing-artifacts-to-sonatype-nexus-using-jenkins-pipelines#:~:text=stage%28,script>
- [9]. Amr Abdelrazik, “Sonatype Nexus 3 launches into Mesosphere DC/OS.” d2iq.com blog, Apr 2017.Referred from: <https://d2iq.com/blog/sonatype-nexus-3-launches-mesosphere-dcos#:~:text=%2A%20Efficiency%20,qualityapplications%20and%20less%20unplanned%20work>
- [10]. Krish Sivanathan, Packagecloud, “Repository Showdown: Artifactory vs. Nexus vs. ProGet.” blog.packagecloud.io, Apr 2022.Referred from: <https://blog.packagecloud.io/repository-showdown-artifactory-vs-nexus-vs-proget/#:~:text=Every%20software%20development%20team%20needs,will%20work%20best%20for%20you>