Plugins & Versioning

In Jenkins, plugins and versioning are crucial for extending functionality and managing updates.

Here's how they work:

Plugins in Jenkins

Jenkins is highly modular, and its core functionalities can be extended via plugins. Plugins allow Jenkins to integrate with a wide variety of tools and platforms like Git, Docker, Kubernetes, and more.

- **Plugin Manager:** Jenkins comes with a Plugin Manager that allows you to search, install, update, and remove plugins.

- **Popular Plugins:**
  - **Git Plugin:** Integrates Jenkins with Git repositories.
  - **Pipeline Plugin:** Defines complex build pipelines using a scripted or declarative DSL.
  - **Docker Plugin:** Builds, runs, and manages Docker containers.
  - **Blue Ocean:** Provides a modern, user-friendly UI for Jenkins pipelines.

- **Installation:** Plugins can be installed via the Jenkins UI or uploaded manually. It's important to keep them updated to the latest versions for security and stability reasons.

Versioning in Jenkins

Versioning in Jenkins primarily refers to tracking changes in jobs, configurations, and code.

- **Job DSL Plugin:** Allows you to version control Jenkins jobs as code. You define jobs and pipelines in a DSL, store it in your version control system, and Jenkins can recreate or update jobs based on this DSL.

- **Pipeline-as-Code (Jenkinsfile):** Jenkins supports versioning of pipeline definitions through Jenkinsfile. This file is placed in your repository and defines the build steps for Jenkins to execute. It's version-controlled alongside your codebase.

- **Jenkins Core Versioning:** Jenkins itself has a core version, and updating Jenkins ensures you get the latest security patches, new features, and performance improvements. You can configure Jenkins to notify you of updates or update manually.

Workspace in Jenkins

The workspace in Jenkins is the directory on the Jenkins server where Jenkins stores the files related to a job.

- **Purpose of Workspace:** When Jenkins triggers a job, it clones the code repository (e.g., Git or SVN) and places it in the workspace. All subsequent build steps operate on this workspace directory.

- **Location:** Workspaces are located under the $JENKINS_HOME/workspace/ directory on the Jenkins server, with each job having its own subdirectory.

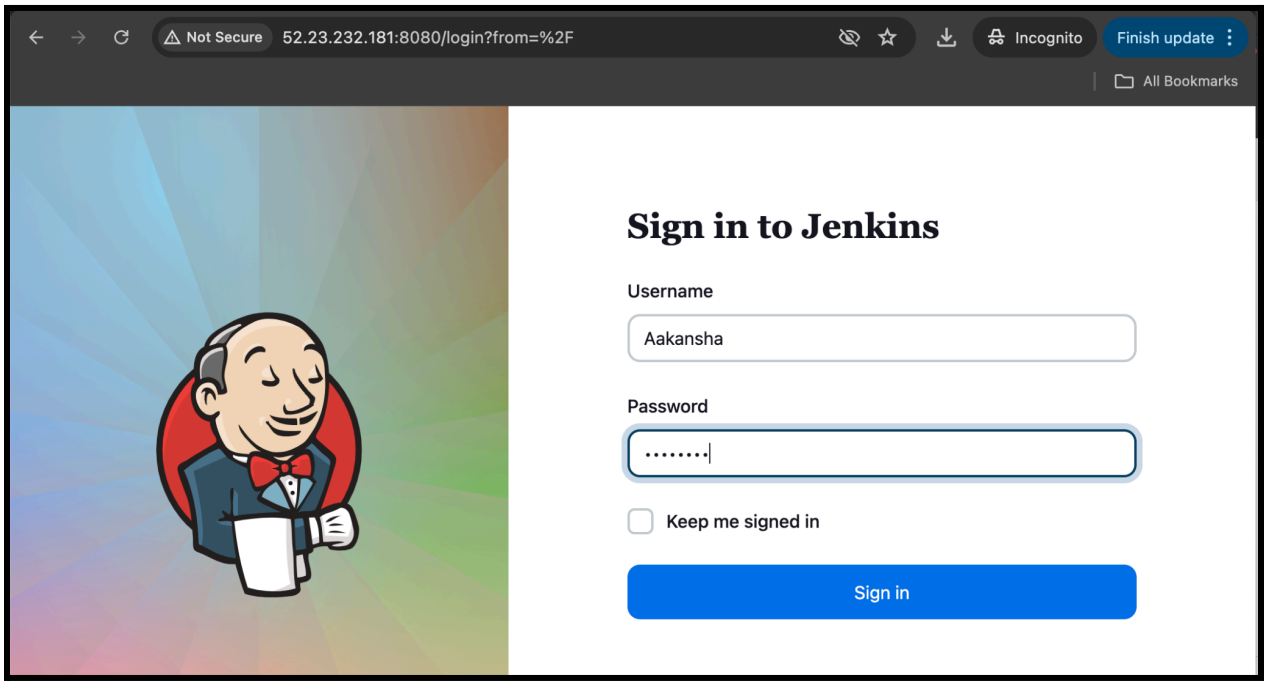
- **Workspace Output:**
  - **Compiled Artifacts:** In build jobs (e.g., Java Maven builds), the workspace contains compiled files like .jar or .war files.
  - **Logs:** Build logs are generated in the workspace and include information about each step in the build.
  - **Reports:** Test reports (JUnit, coverage, etc.) are usually generated in the workspace.

Managing Workspace

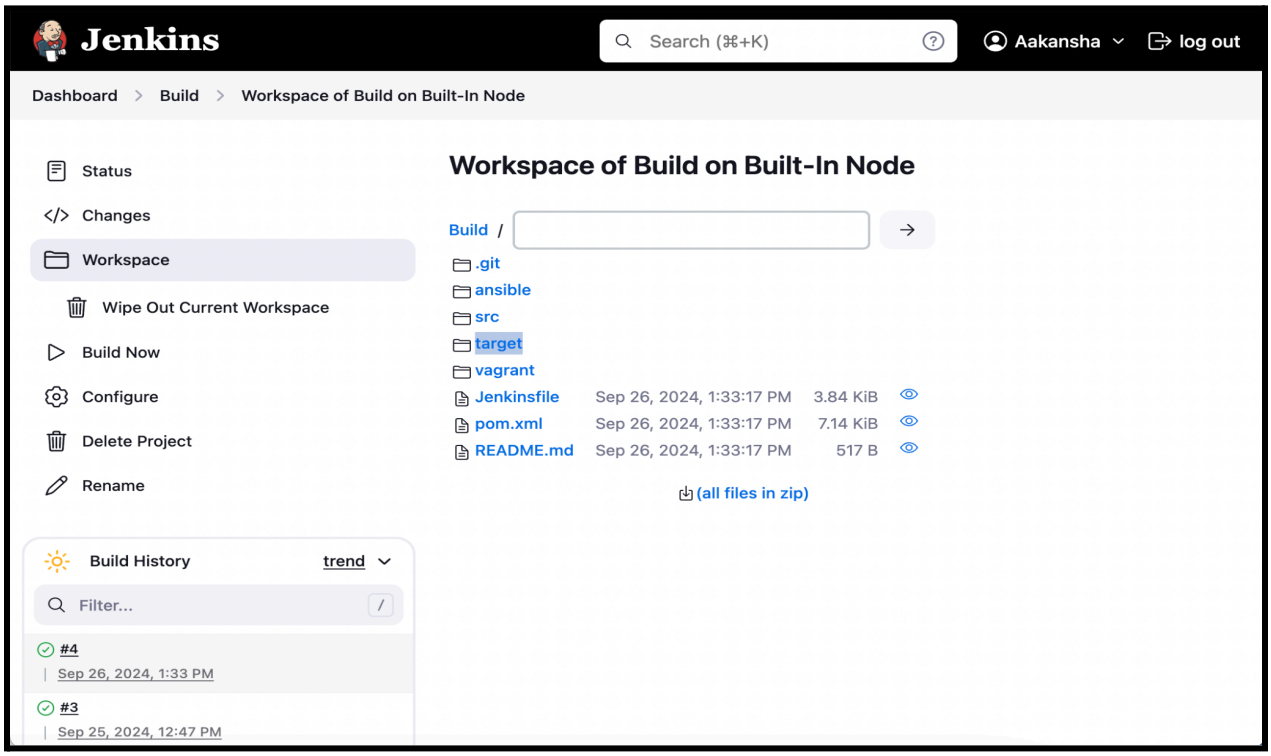
- **Clean Workspace:** Often, after a build, the workspace needs to be cleaned to avoid leftover files from previous builds. Plugins like Workspace Cleanup can help automate this process.

- **Disk Space:** Large workspaces can consume a lot of disk space. It's important to monitor and clean them periodically.

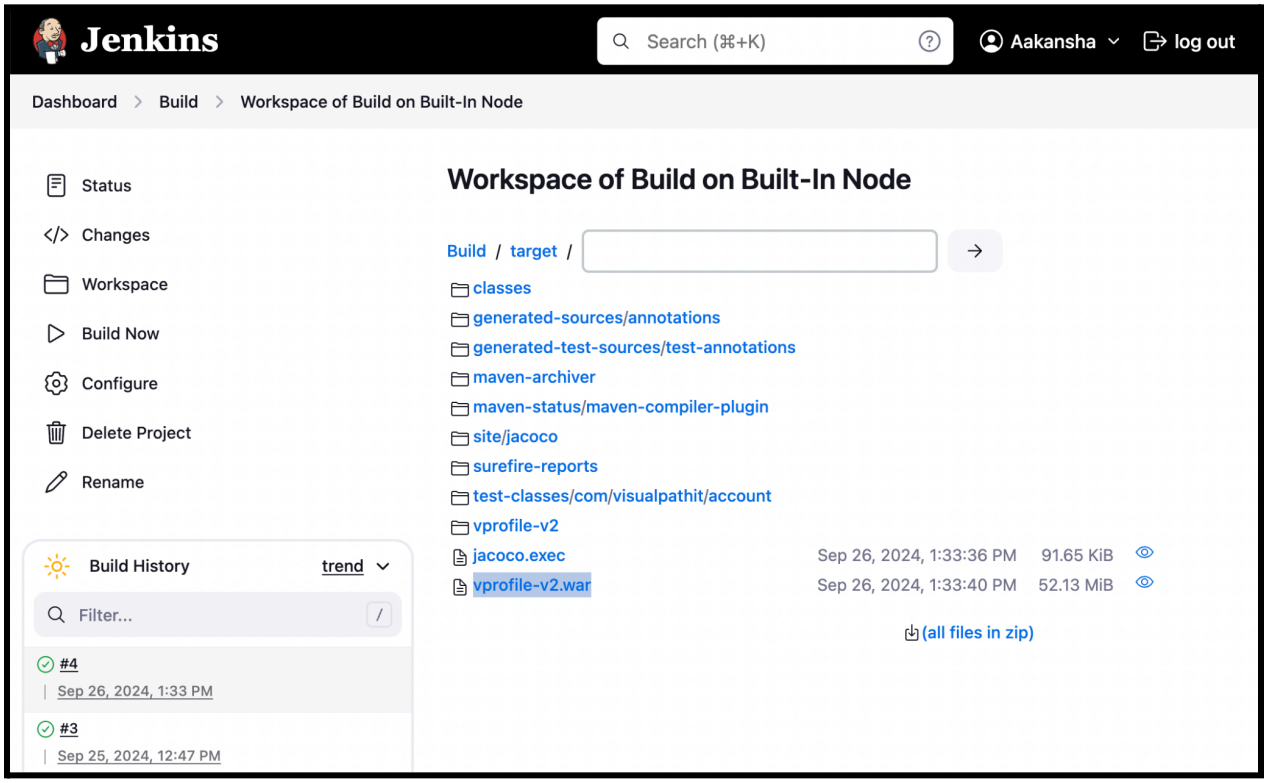
Practice Time



Let’s talk about the workspace and the output being generated in the workspace.

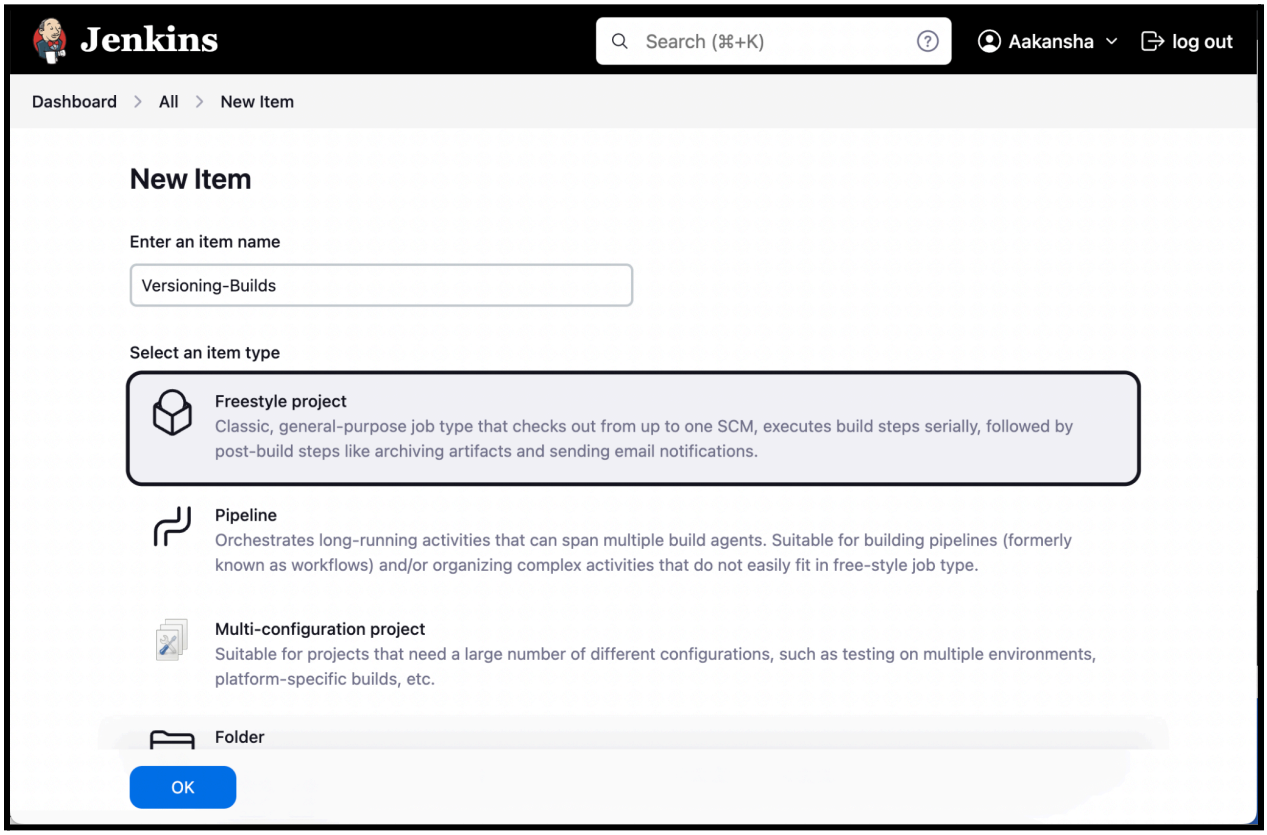


We can see our artifact in the target directory at the workspace.

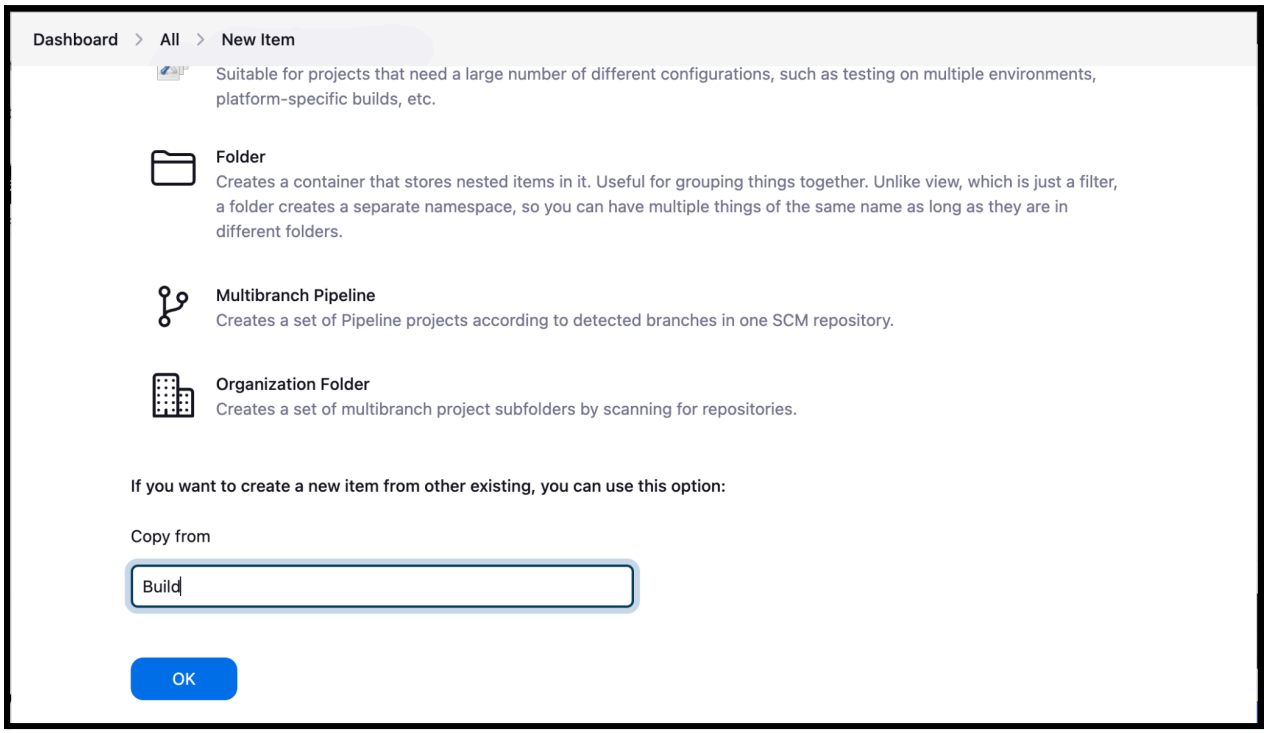


- ➔ Now everytime we will be running the job this artifact will be replaced by the newer artifact and we will be running the job if there will be any code change.
- ➔ Let's say we deploy this artifact and it works fine. Then there is a new code change and we run the build one more time.
- ➔ So now the new artifact will replace the old artifact and we will deploy the new artifact to the server and things break 😣
- ➔ Now we want to rollback to the previous artifact or we want the previous artifact.
- ➔ But we can't get it now as it is overwritten.
- ➔ So to preserve the artifact we can do versioning.
- ➔ Let's start with some simple versioning by executing shell commands and install required plugins for versioning.
- ➔ The target directory is volatile, if someone runs mvn clean everything will be gone from there.

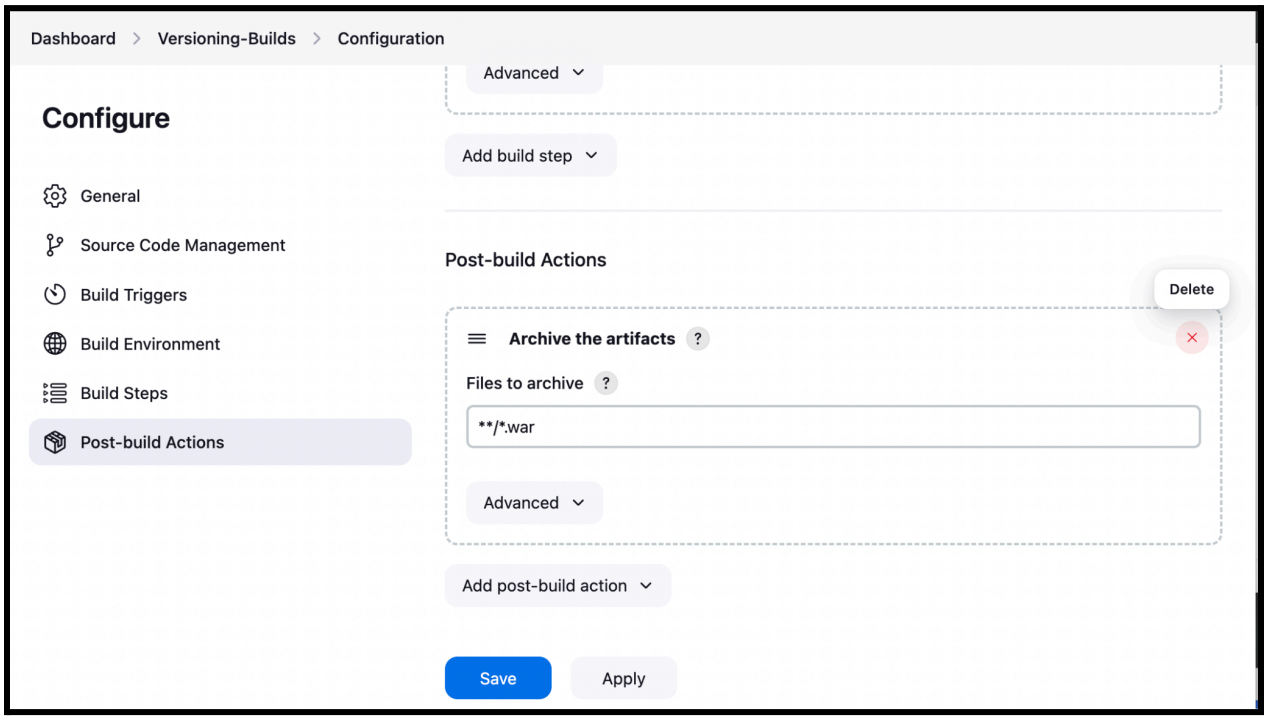
Now let’s create another job for versioning.



Below this copy everything from the previous job build and click ok.

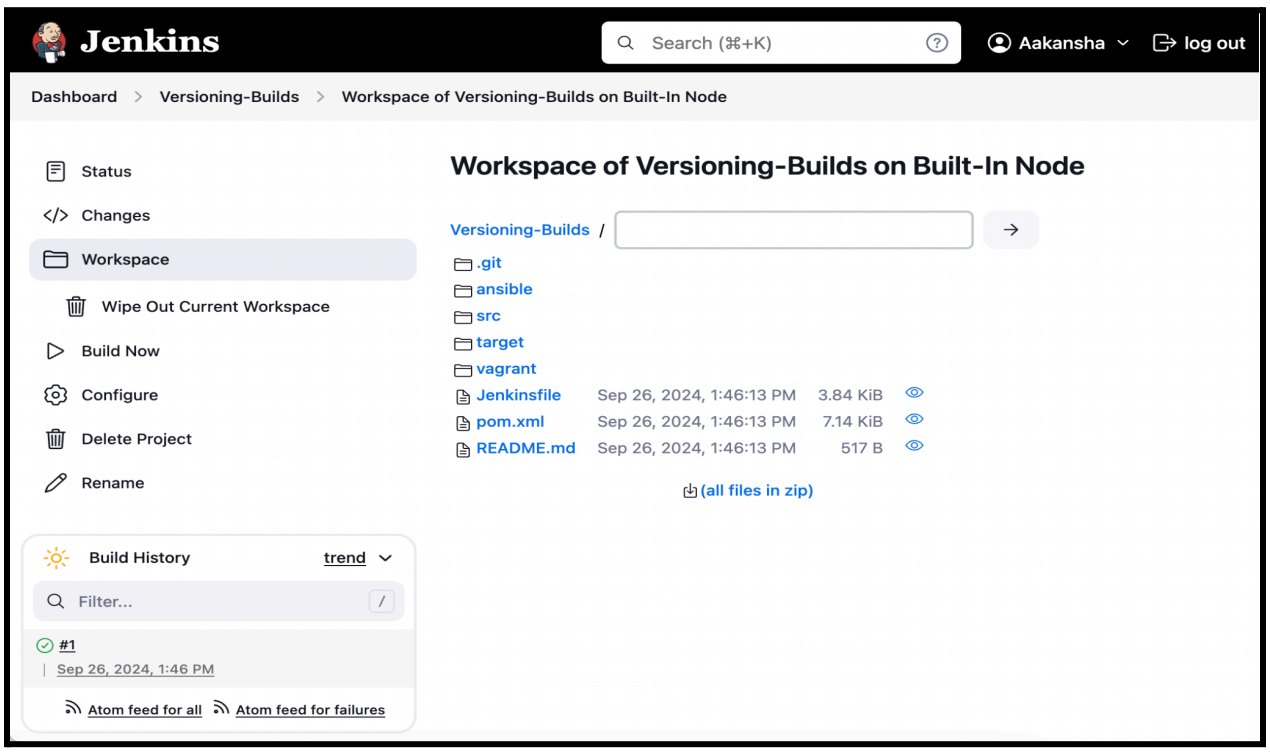


Go down and remove the archive of the artifact because it will start storing this artifact into different- different directories with different- different build numbers which will consume the disk space.

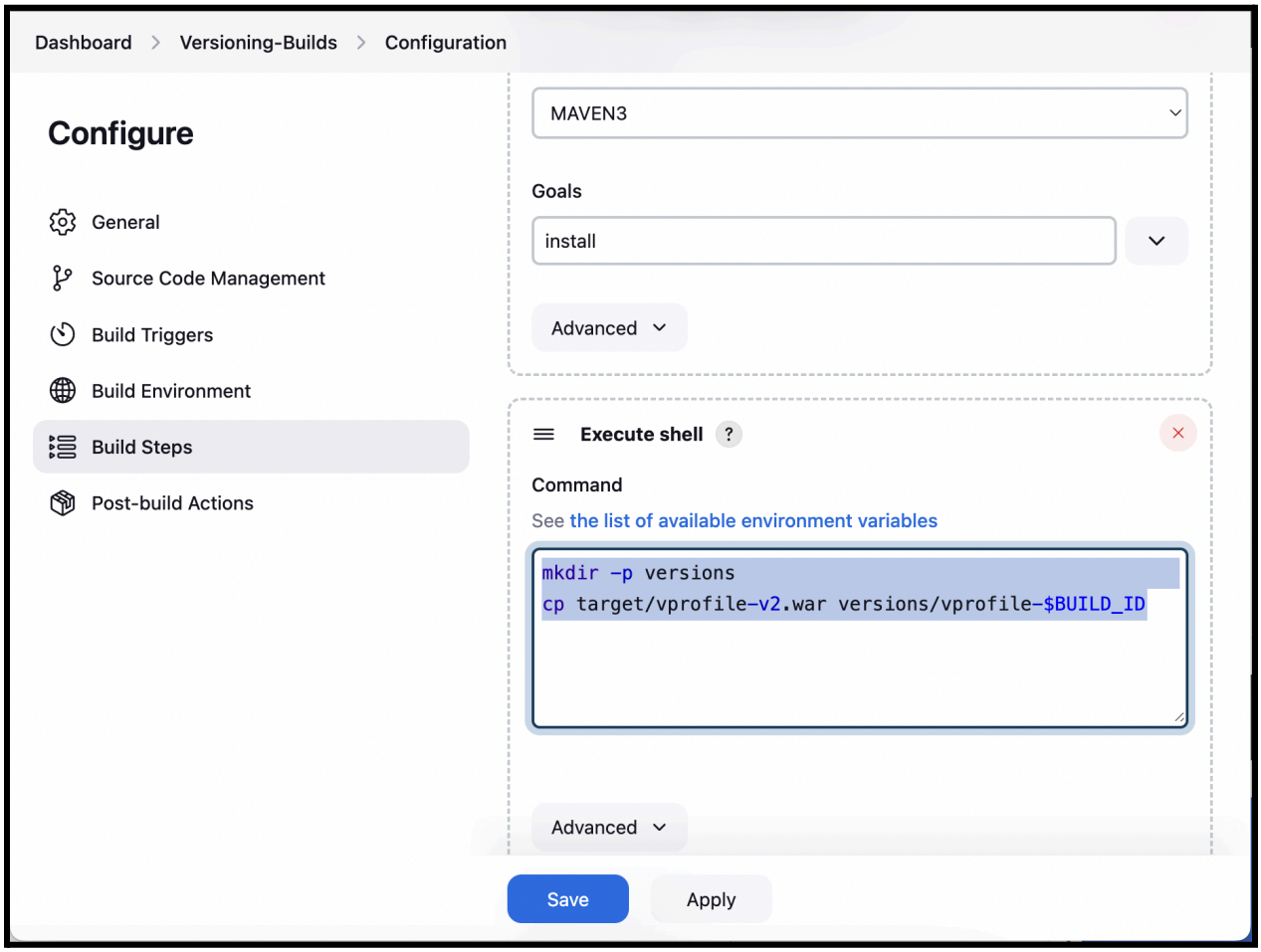


Save and run this job

Now once the job runs successfully we must be having the artifact in our workspace.



Configure the job and come down to the build section. Add build step and select execute shell.



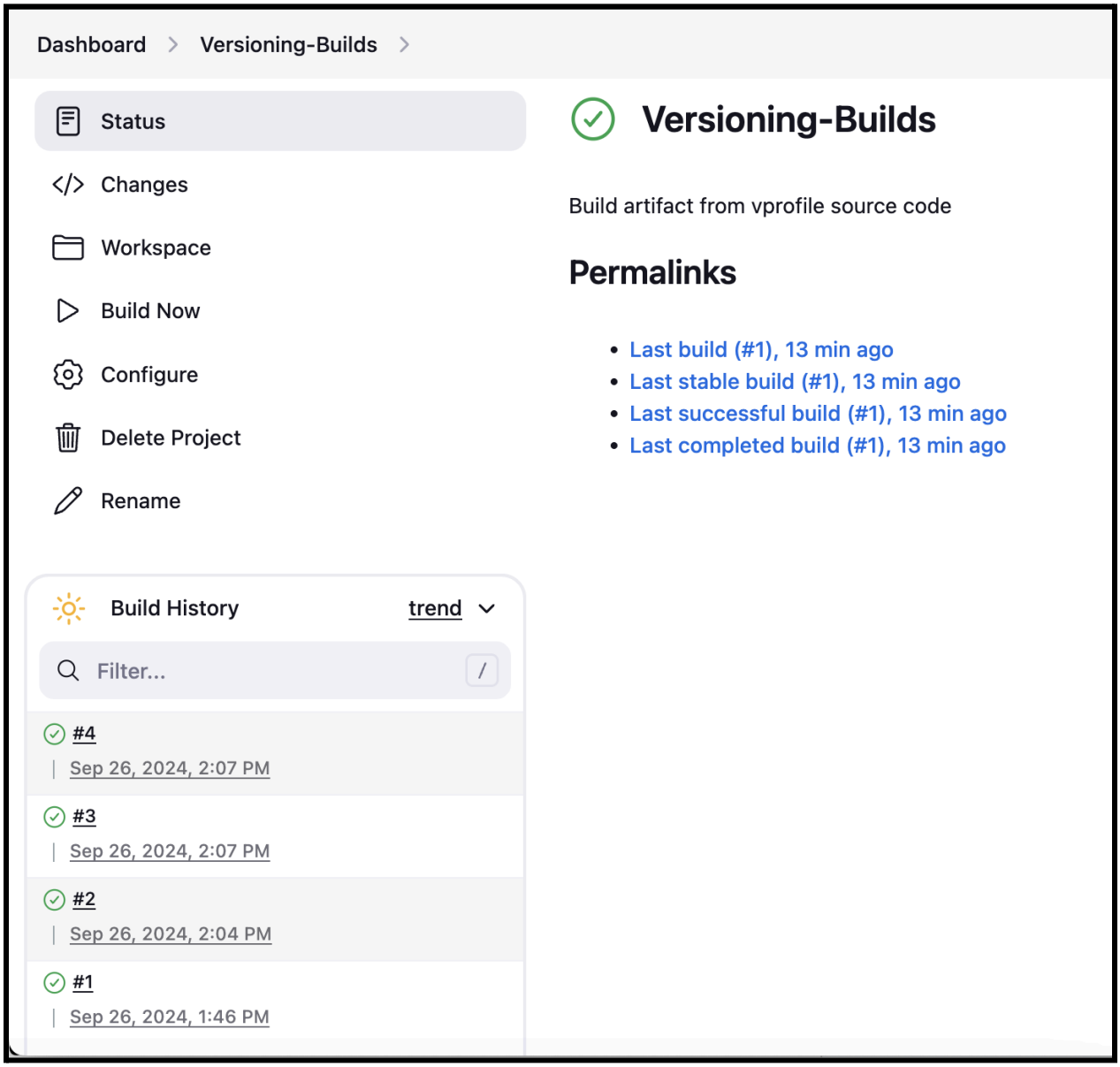
➔ **These commands create a directory to store different versions of the build artifact and copy the `.war` file into it, with a name that includes the build ID for easy identification.**

Commands Explanation

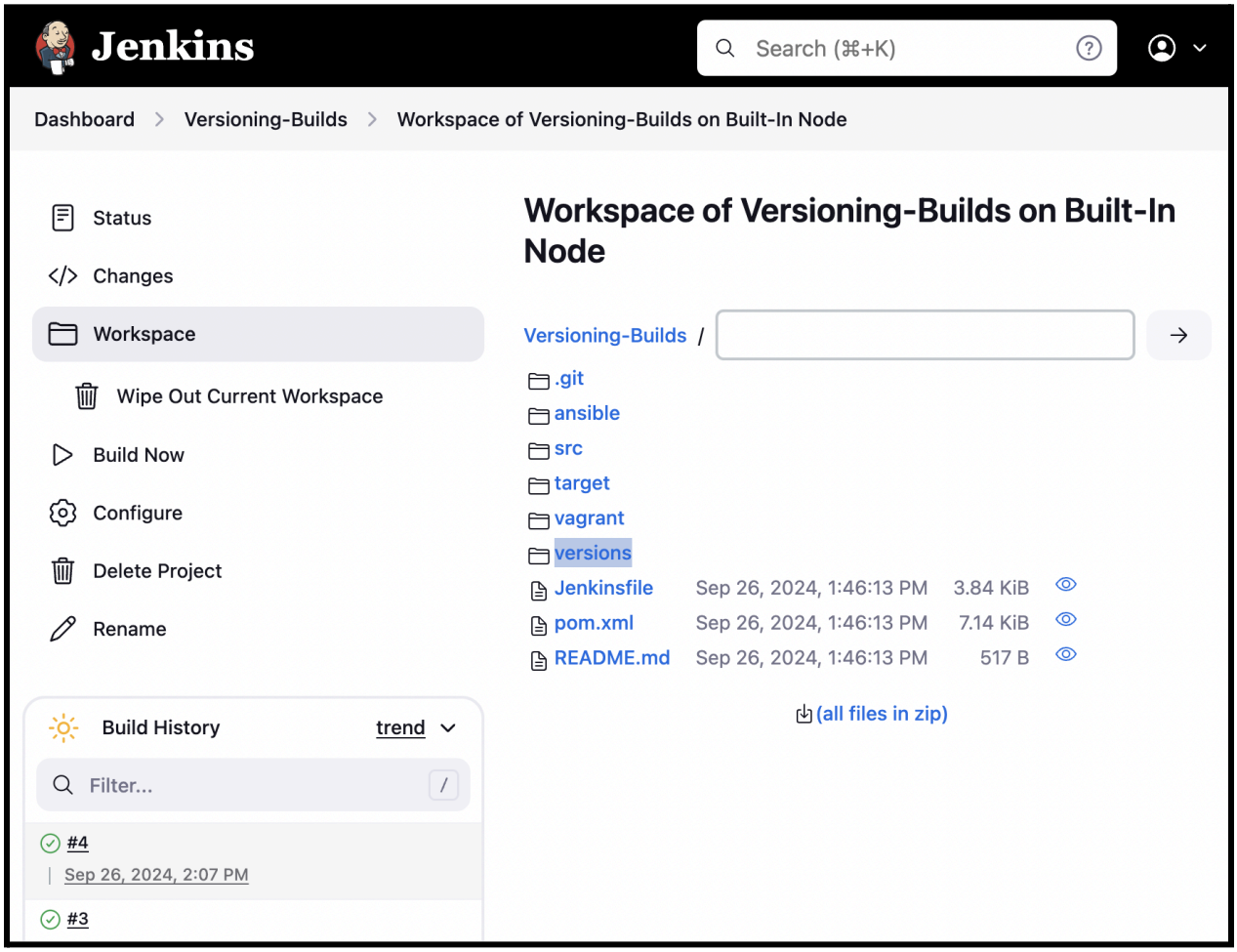
1. **mkdir -p versions:**
    - This command creates a directory named versions.
    - The -p option ensures that the directory is created only if it doesn't already exist. It also creates any necessary parent directories.
2. **cp target/vprofile-v2.war versions/vprofile-$BUILD_ID:**
    - This copies the file vprofile-v2.war from the target/ directory (usually where the build artifacts are generated, like after a Maven build) into the versions/ directory.
    - The destination file is renamed to vprofile-$BUILD_ID, where $BUILD_ID is an environment variable in Jenkins that holds the unique identifier of the current build. This ensures each build artifact is stored with a version number tied to the specific build, allowing for versioning and easier tracking of different build outputs.

❖ **Check the link for Jenkins Set Environment Variables**
https://wiki.jenkins.io/display/JENKINS/Building+a+software+project

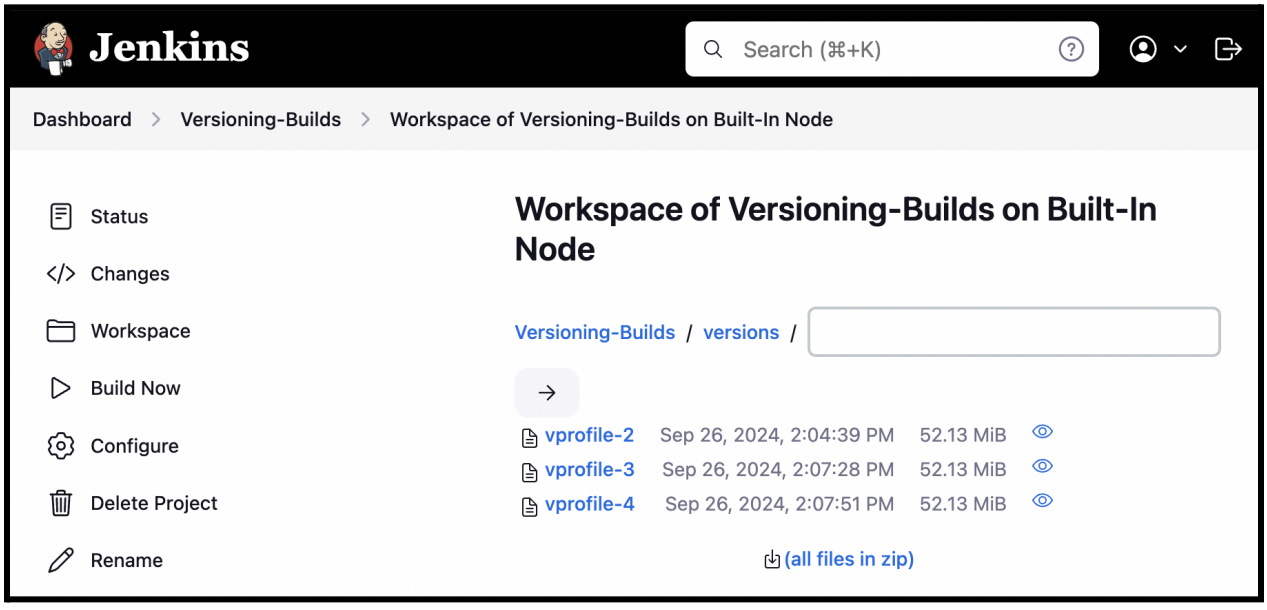
Now run the build a few times.



Now let's check the workspace. We can see the versions directory.



We can see the different versions of the artifact.

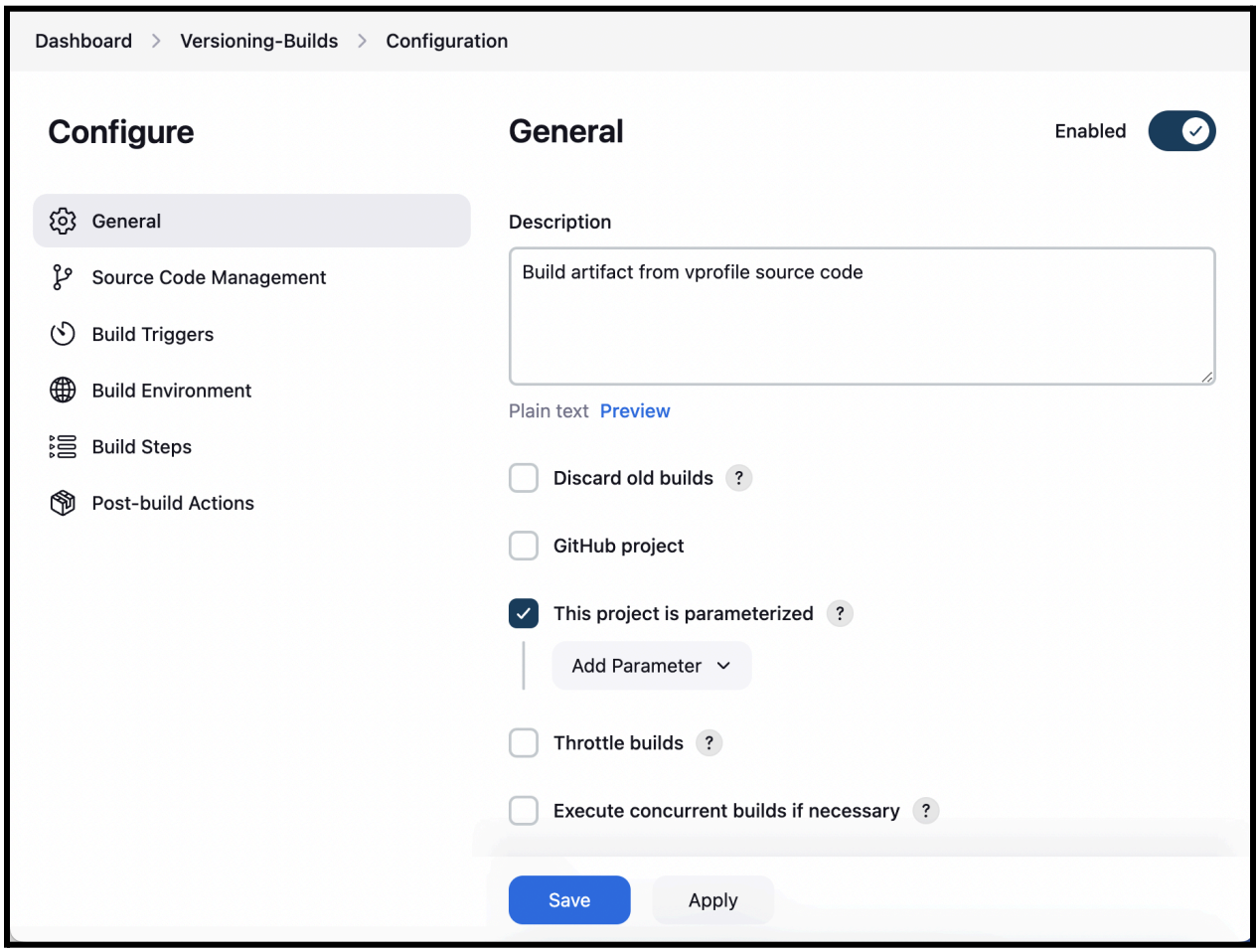


Can we use some other versioning format?

What if I want to specify my own, like a user argument.

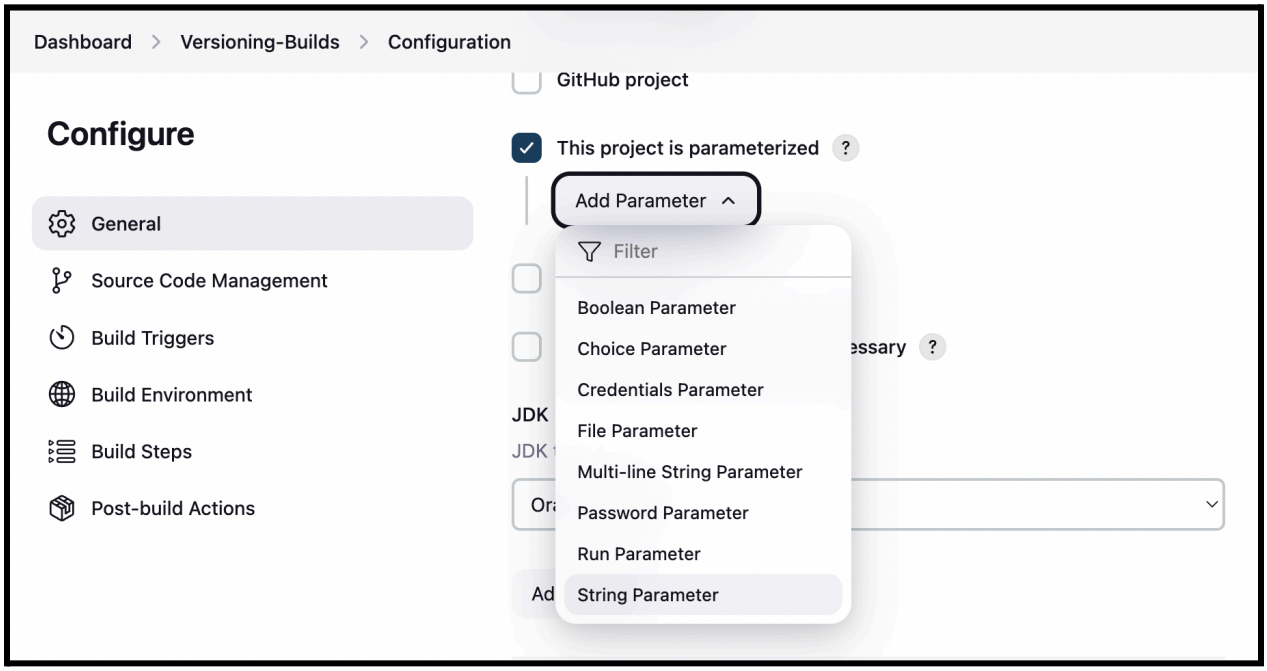
❖ Let's Try!

Go to configure.. Jenkins job has one option here in the general: This project is parameterised.

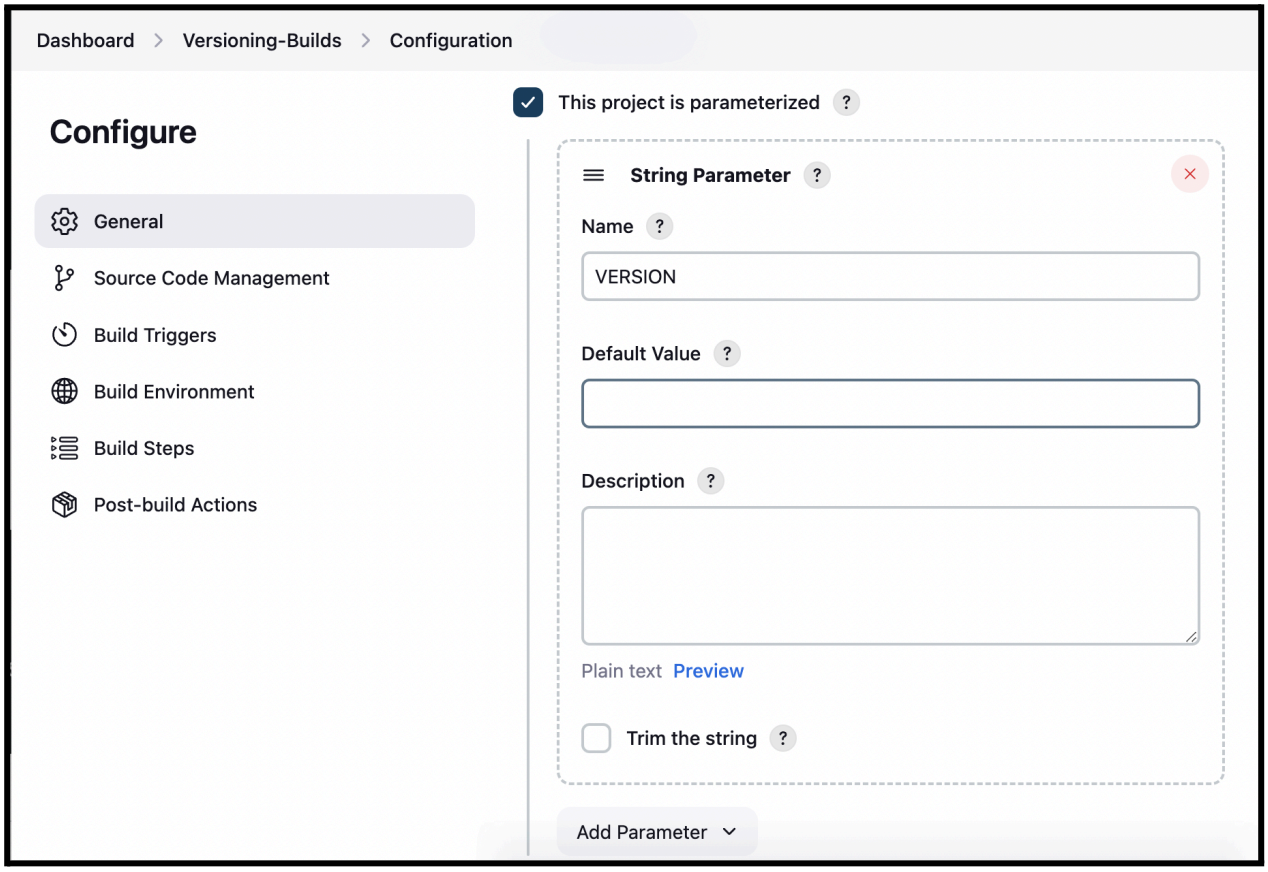


➔ This will be taking the user input.

Add parameters. Let’s take the string parameter.

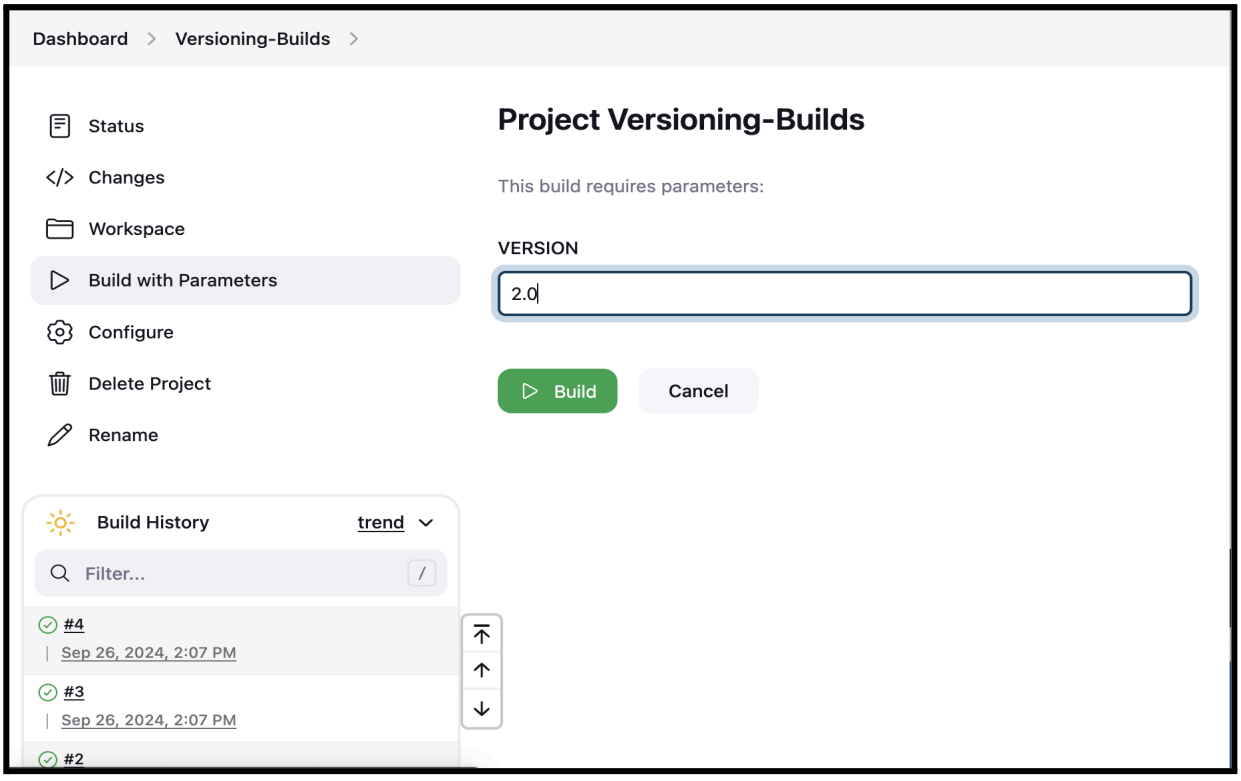


Now when the user enters the value it’s going to store it in this variable and we can use it in our job.

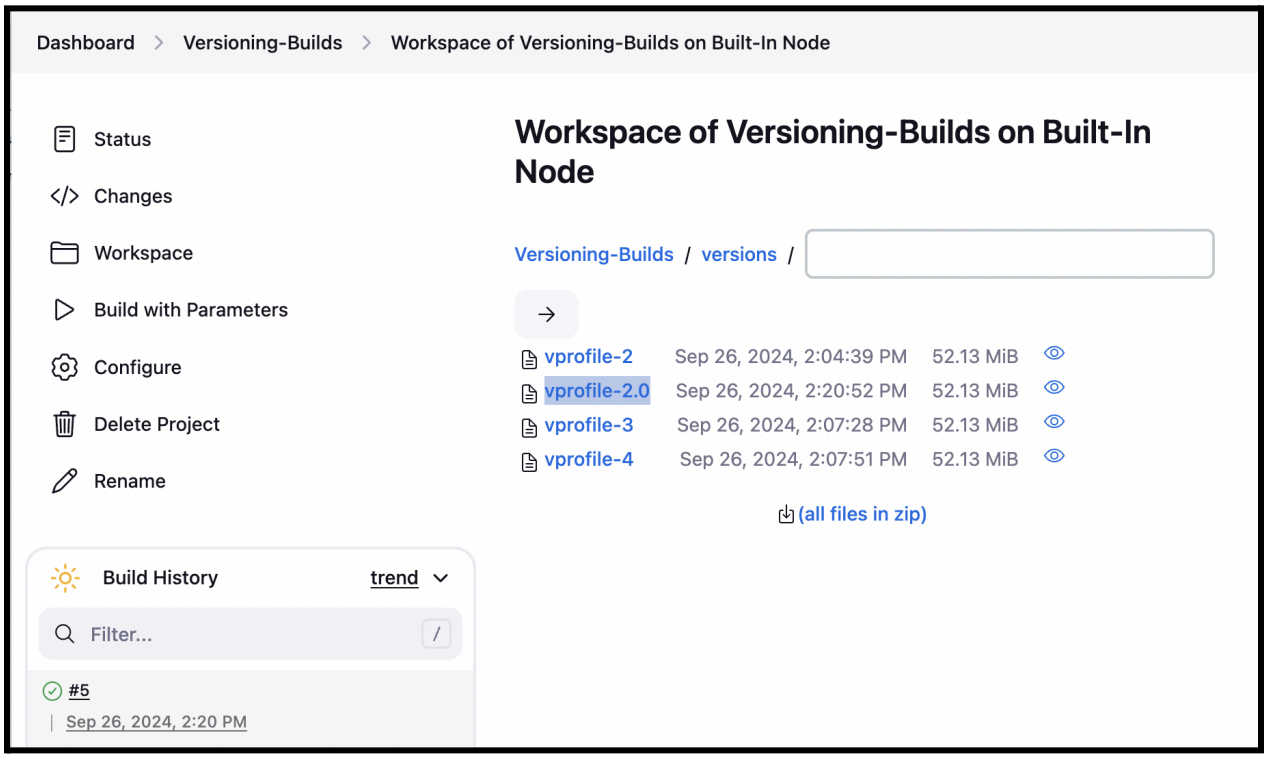


Let’s use our variable and save it.

Now let’s test it.



Now let's see the output.



Here it is 👋

Question Time

Why is it important to avoid making Jenkins jobs interactive, and how does it impact the automation process in a CI/CD pipeline?

Avoiding interactive jobs in Jenkins is generally recommended for several key reasons:

1. Automation and Unattended Execution

- Jenkins jobs are meant to run in an automated, continuous integration/continuous delivery (CI/CD) environment without manual intervention. Interactive jobs that require user input (like prompting for confirmation or asking for credentials) break the automation flow, requiring someone to monitor and respond, which defeats the purpose of CI/CD.

2. Consistency and Reliability

- Non-interactive jobs ensure consistent, repeatable builds and deployments. If user interaction is needed, different responses or mistakes during interaction can cause inconsistent results. Removing human input ensures that jobs run the same way every time.

3. Scalability

- CI/CD environments often need to scale, running many jobs in parallel across multiple nodes or in distributed systems. Interactive jobs can block the execution of other jobs, limiting scalability, especially when jobs need to be manually handled.

4. Pipeline Automation

- Jenkins pipelines are designed to be triggered automatically by events (like commits to a repository) or scheduled to run at specific times. Interactive jobs stop the pipeline and delay the entire process, making it inefficient for continuous integration.

5. Error Handling and Monitoring

- Non-interactive jobs are easier to monitor, log, and manage errors. Interactive prompts make it harder to track automated failures and troubleshoot issues, especially when running jobs remotely or at scale.

6. Integration with Other Tools

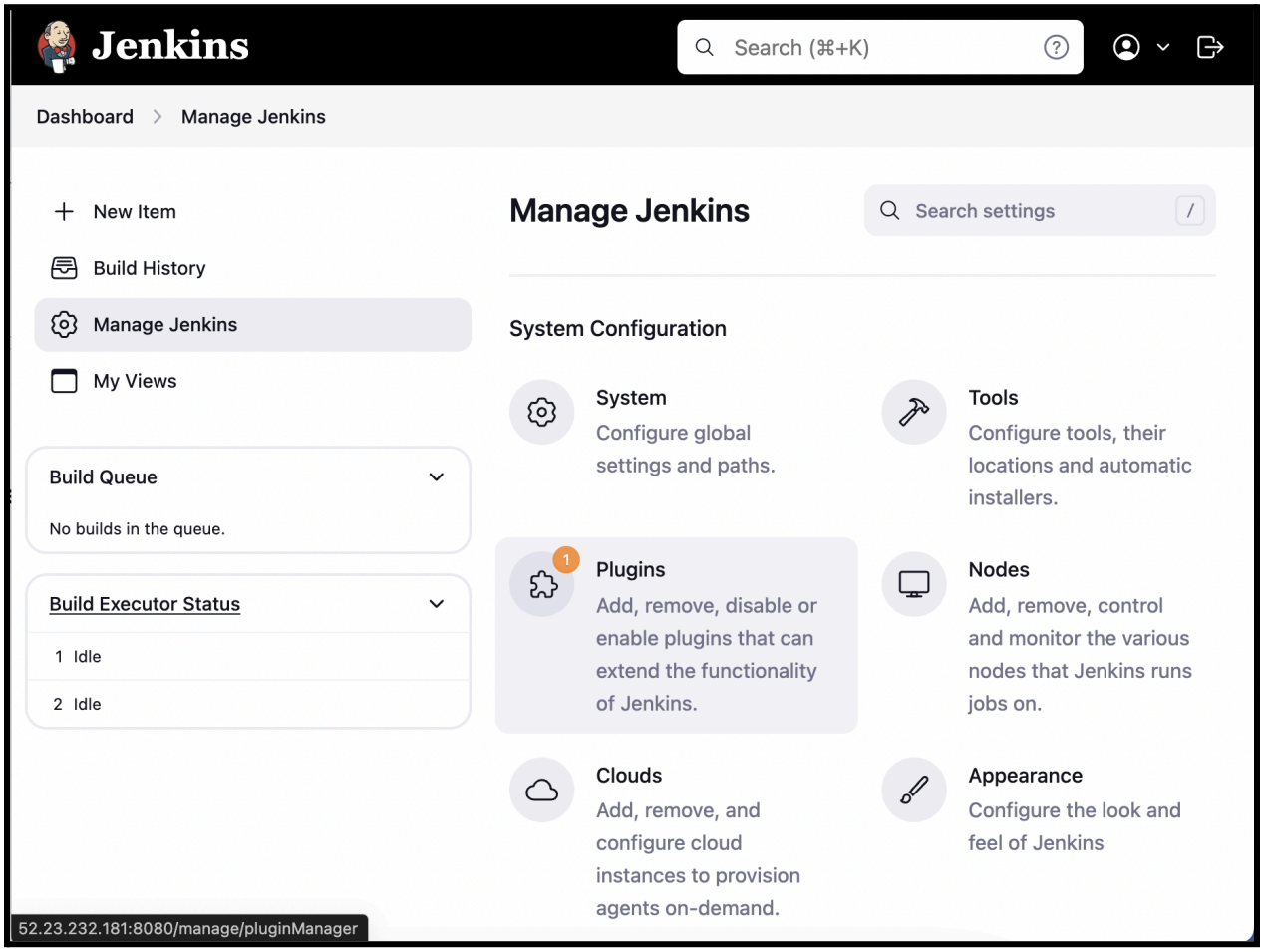
- Interactive jobs hinder integration with external tools and systems, such as version control, artifact repositories, or deployment services, which expect automated, non-blocking job execution.

By keeping jobs non-interactive, Jenkins can operate continuously and autonomously, leading to more reliable, scalable, and efficient pipelines.

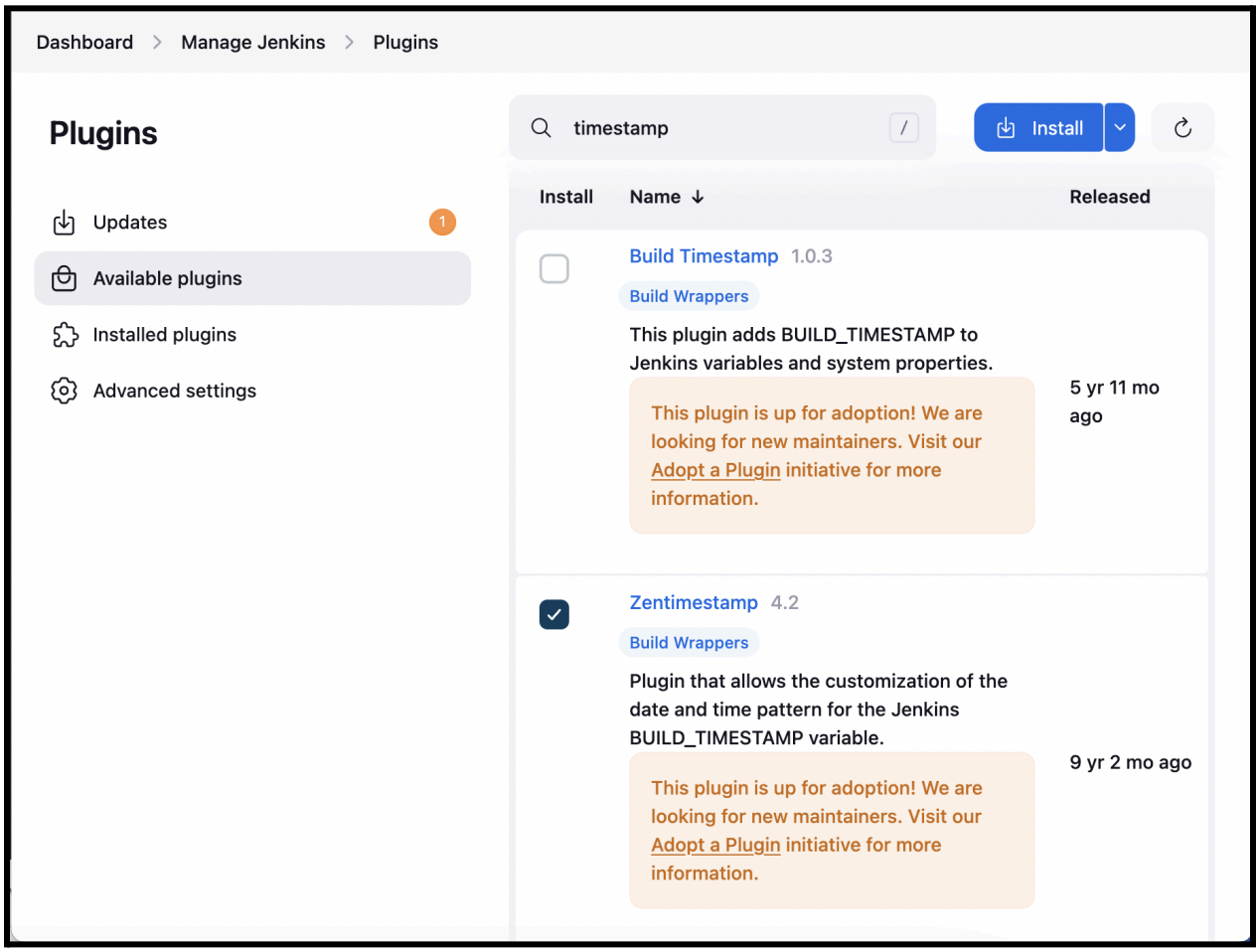
Application of plugins

Now let’s move to a better way of versioning using plugins.

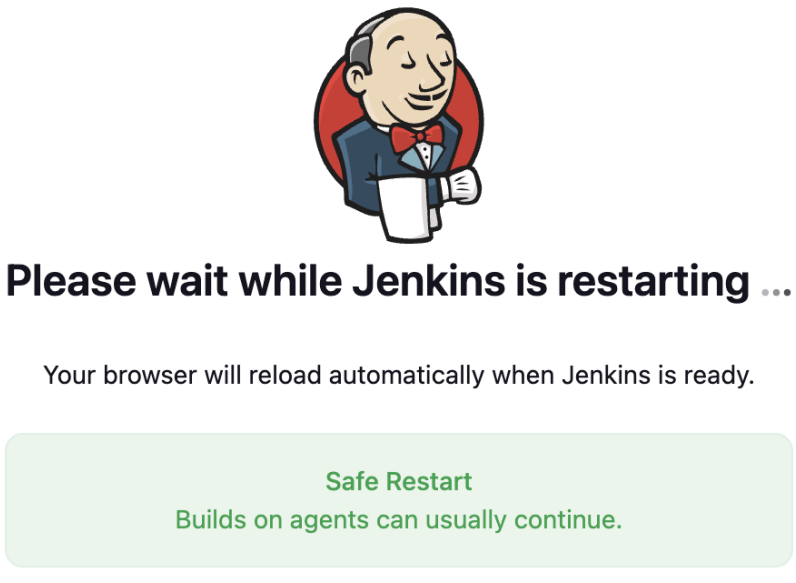
Go to Dashboard > Manage Jenkins > Plugins



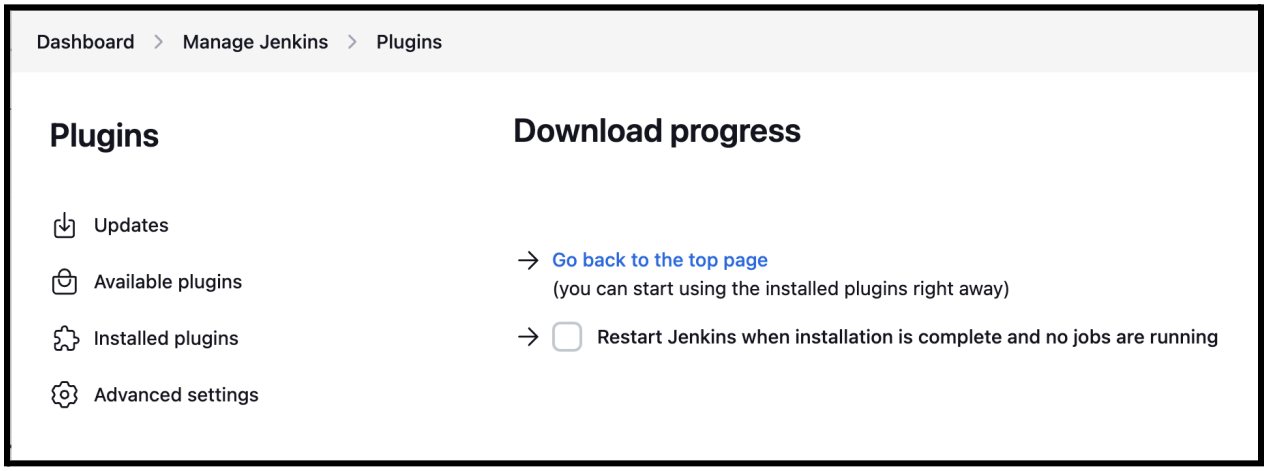
Let’s look for a plugin that gives us a timestamp.



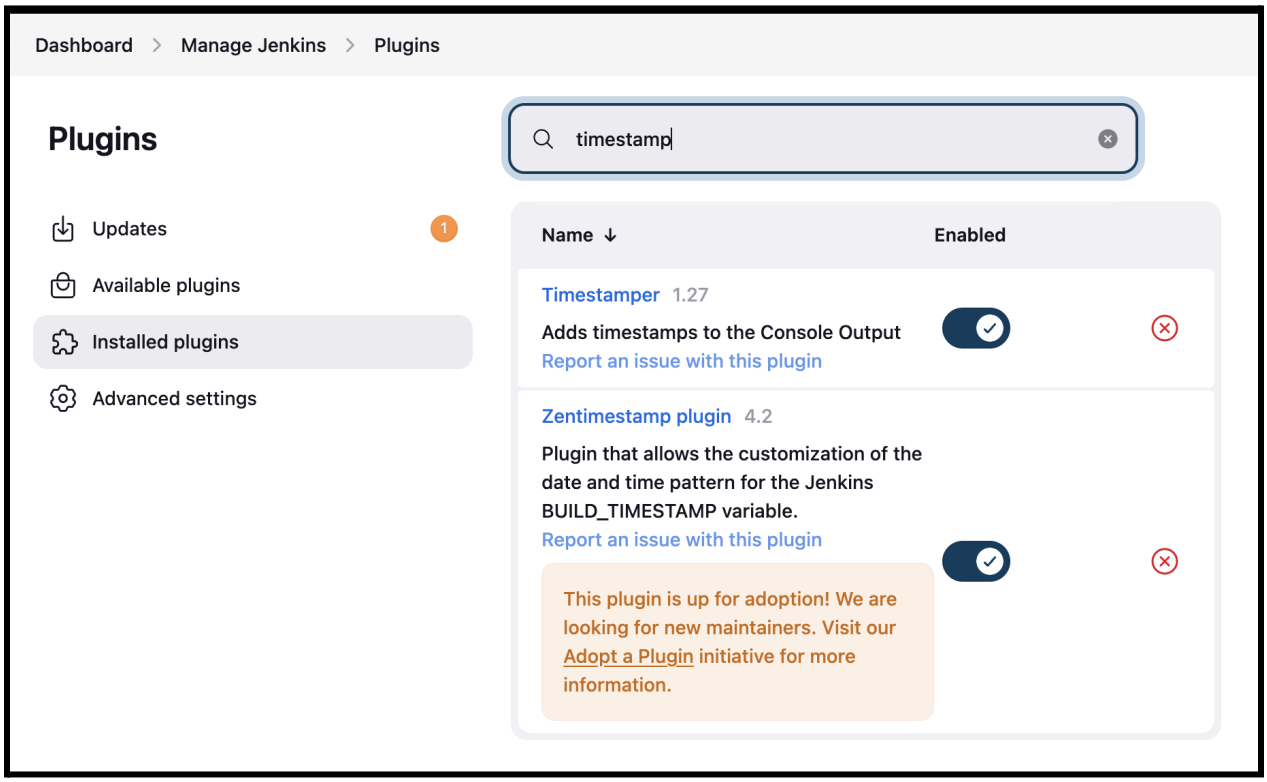
Click on install and wait for some time.



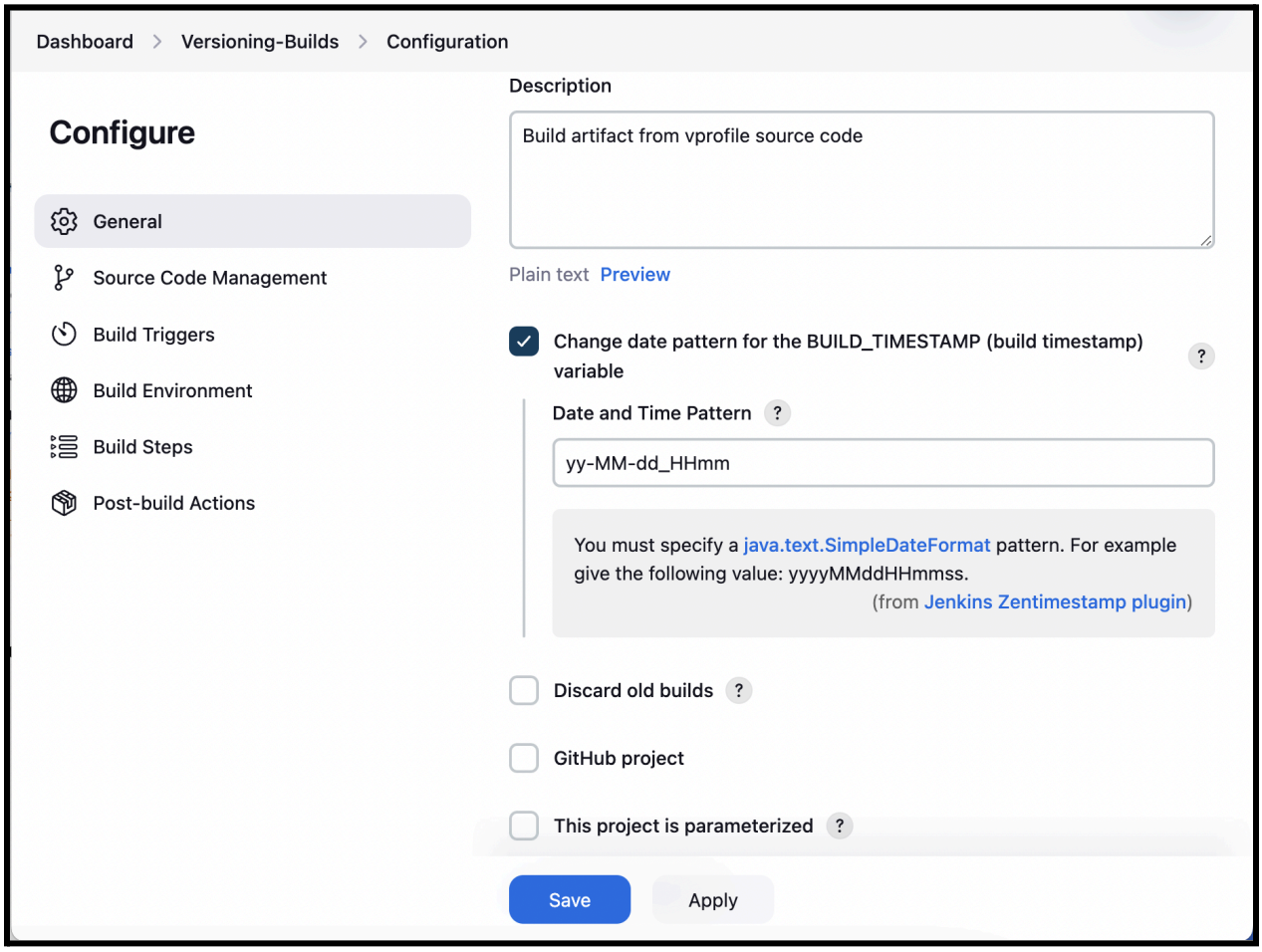
The download is in progress.



Check the installed plugins.

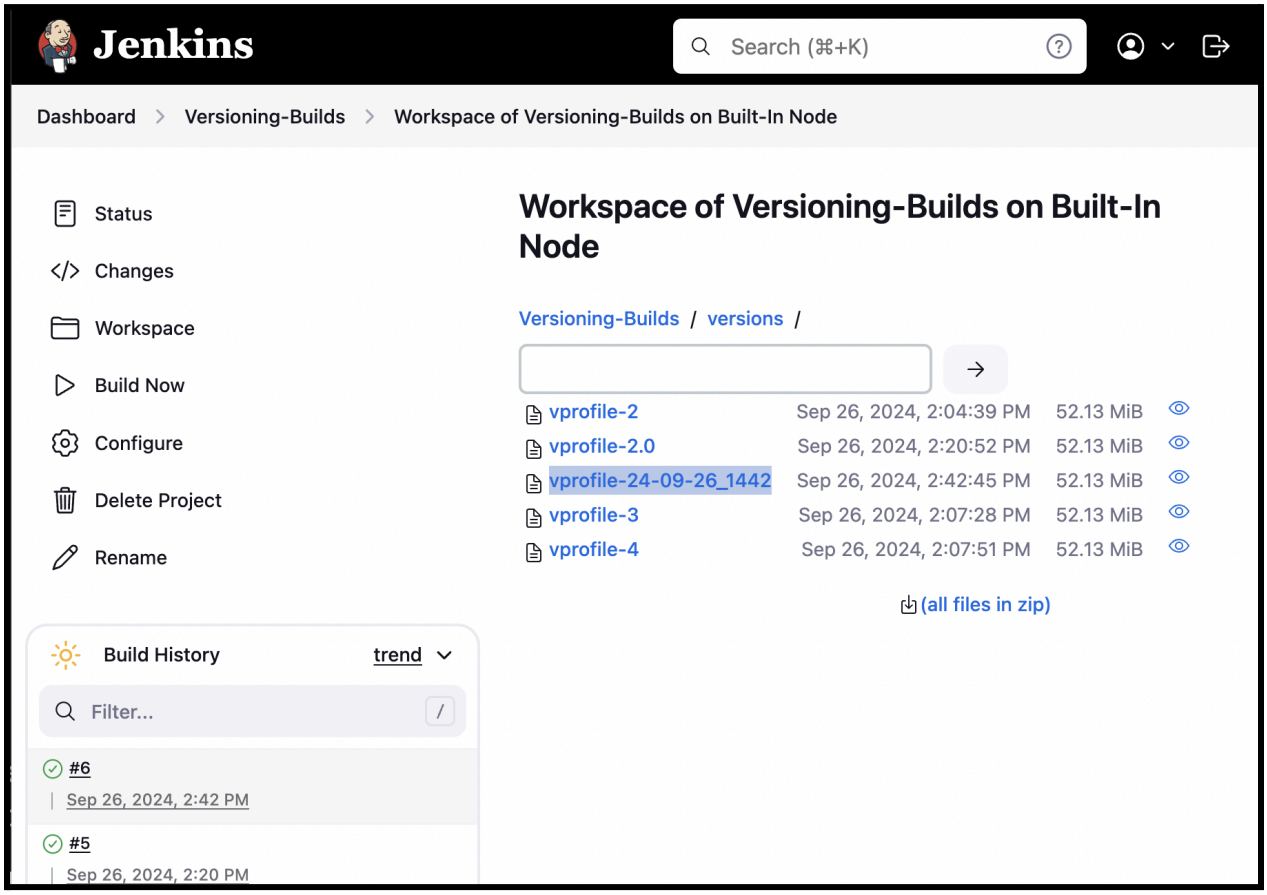


Go back to the versioning job and click configure.



Add the variable $BUILD_TIMESTAMP. Save and build now.

Let’s check the workspace now.



So it’s looking better now. We can give more types of format.

As Jenkins is primarily a continuous integration server and not meant for long-term storage, it's important to transfer these artifacts to a more suitable location to avoid consuming excessive disk space. Congratulations on reaching this stage, and keep up the great work as you move forward!