Chavi Mittal

23FE10CSE00312

DEVOPS ASSIGNMENT

TASK-1

**Continuous Integration Concepts and Implementation Strategies-**

**1. CI Principles: The Foundation of Modern DevOps-**

1.1 Defining Continuous Integration (CI) Continuous Integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. The key goals of CI are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

1.2 Differentiating CI, CD, and Continuous Deployment While often grouped together, these three terms represent distinct stages in the automation pipeline: Continuous Integration (CI): Focuses on the build and unit testing stages of the software release process. It ensures code from different developers is integrated frequently. Continuous Delivery (CD): An extension of CI where the software is always in a "ready-to-deploy" state. It involves automated testing and environment staging, but the final push to production is a manual decision. Continuous Deployment: This is the most advanced stage where every change that passes all stages of the production pipeline is released to customers automatically, with no human intervention.

1.3 Benefits and Challenges Benefits: Early Bug Detection: By testing every commit, "integration hell" is avoided. Reduced Risk: Smaller, frequent updates are easier to troubleshoot than massive quarterly releases. Developer Productivity: Automation frees developers from manual build tasks. Challenges: Initial Setup Cost: Building a robust pipeline requires significant time and investment in infrastructure. Cultural Resistance: Teams must shift from "owning code" to "owning the pipeline," requiring high discipline in writing tests. Maintenance Overhead: As the project grows, the CI environment (plugins, agents, scripts) requires constant updates.

**2. CI Workflow Components A robust CI pipeline is composed of five critical pillars:**

2.1 Version Control System (VCS) The "Single Source of Truth." Tools like Git (GitHub, GitLab, Bitbucket) allow multiple developers to work on the same codebase. CI begins the moment a "Push" or "Pull Request" is detected in the VCS.

2.2 Build Automation Manual compiling is error-prone. Build automation tools (e.g., Maven for Java, npm for Node.js, Gradle) package the code into executable binaries. In a CI context, the CI server (Jenkins) triggers these tools automatically upon code entry.

2.3 Automated Testing This is the heart of CI. It includes: Unit Tests: Testing individual functions. Integration Tests: Ensuring different modules work together. Static Analysis: Checking code quality and security vulnerabilities (e.g., SonarQube) without executing the code.

2.4 Artifact Repository Once a build is successful, the resulting "Artifact" (a .jar file, a Docker image, etc.) must be stored in a versioned repository. Tools like JFrog Artifactory or Sonatype Nexus ensure that the exact version tested in CI is the one deployed to production.

2.5 Notification Systems Feedback loops are essential. If a build fails, the system must immediately notify the relevant developer via Slack, Microsoft Teams, or Email. This ensures the "broken build" is the team's top priority.

**3. Case Study Analysis**

3.1 Organization A: Global Retailer (Legacy Migration) Context: This organization previously used a "Waterfall" model with manual deployments every six months. The Problem: High failure rates during integration week; developers spent 40% of their time fixing merge conflicts. CI Implementation: They adopted Jenkins and Git. They mandated 80% unit test coverage before code could be merged.

3.2 Organization B: Fintech Startup (Cloud Native) Context: A high-growth startup using microservices. The Problem: Needed to deploy updates 10 times a day to stay competitive while maintaining strict security compliance. CI Implementation: They utilized a "GitOps" approach where CI pipelines automatically triggered security scans and containerized builds via Docker and Kubernetes.

3.3 Traditional vs. CI Approach Comparison Feature Traditional Approach CI-Driven Approach Integration Frequency Monthly/Quarterly Several times per day Build Process Manual scripts / Human intervention Fully Automated Testing End of development cycle (Manual) Continuous (Automated) Bug Fixing Expensive and difficult to trace Cheap and immediate Release Confidence Low (Night-before-deploy stress) High (Tested at every step)

**4. ROI Calculation (Return on Investment)**

To justify CI to management, we use a standard ROI formula based on "Developer Time Saved."

**Assumptions:**

* **Team Size:** 50 Developers.
* **Manual Build/Test Time:** 4 hours per week per developer.
* **Average Developer Salary:** $50/hour.
* **Cost of CI Infrastructure (Tools + Server):** $20,000/year.

**Calculations:**

1. **Current Cost of Manual Work:**

50 developers×4 hours×52 weeks×$50/hr=$520,000/year50 developers×4 hours×52 weeks×$50/hr=$520,000/year

1. **Post-CI Manual Work (Estimated 90% reduction):**

10% of $520,000=$52,000/year10% of $520,000=$52,000/year

1. **Gross Savings:**

$520,000−$52,000=$468,000$520,000−$52,000=$468,000

1. **Net Benefit (Savings - Infrastructure Cost):**

$468,000−$20,000=$448,000$468,000−$20,000=$448,000

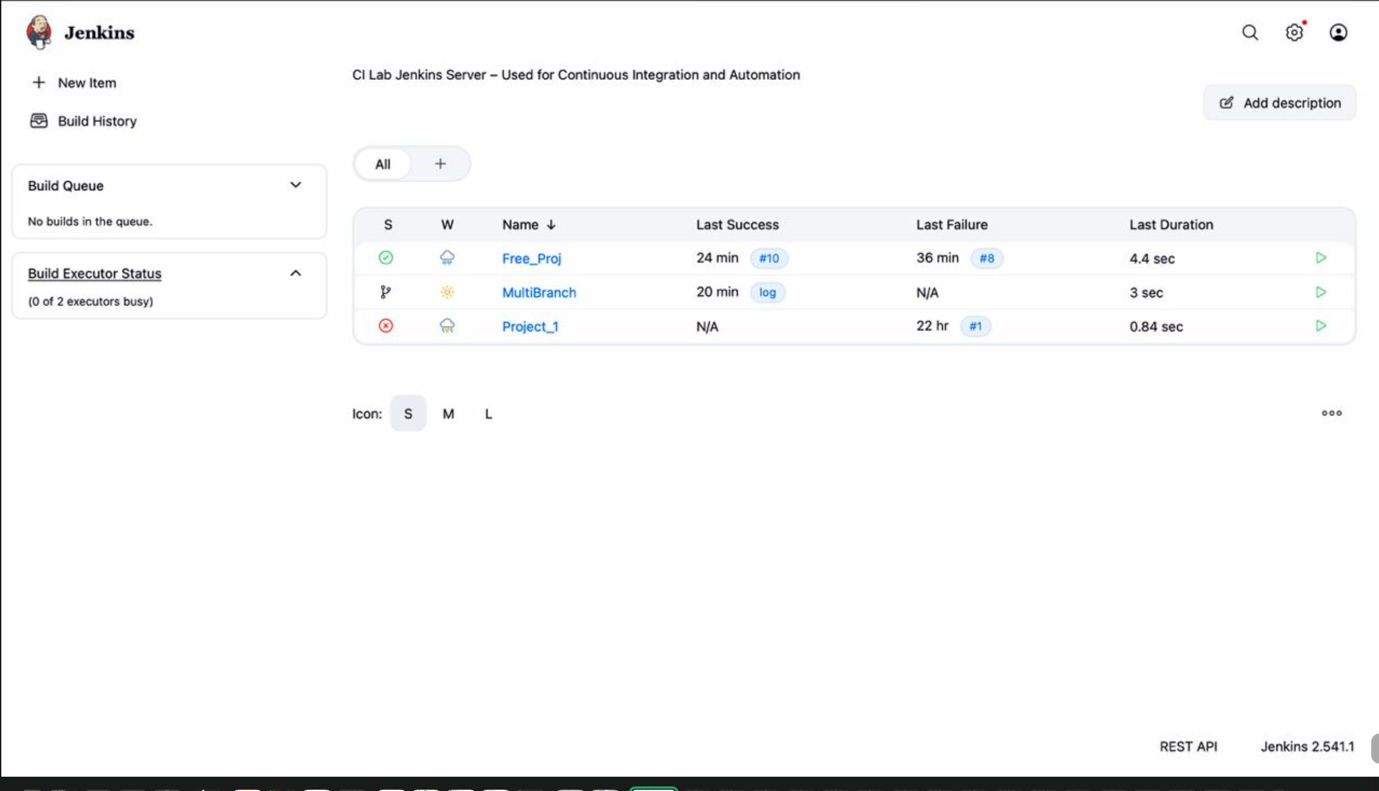
1. **ROI Percentage:**

($448,000/$20,000)×100=2,240%($448,000/$20,000)×100=**2**,**240%**

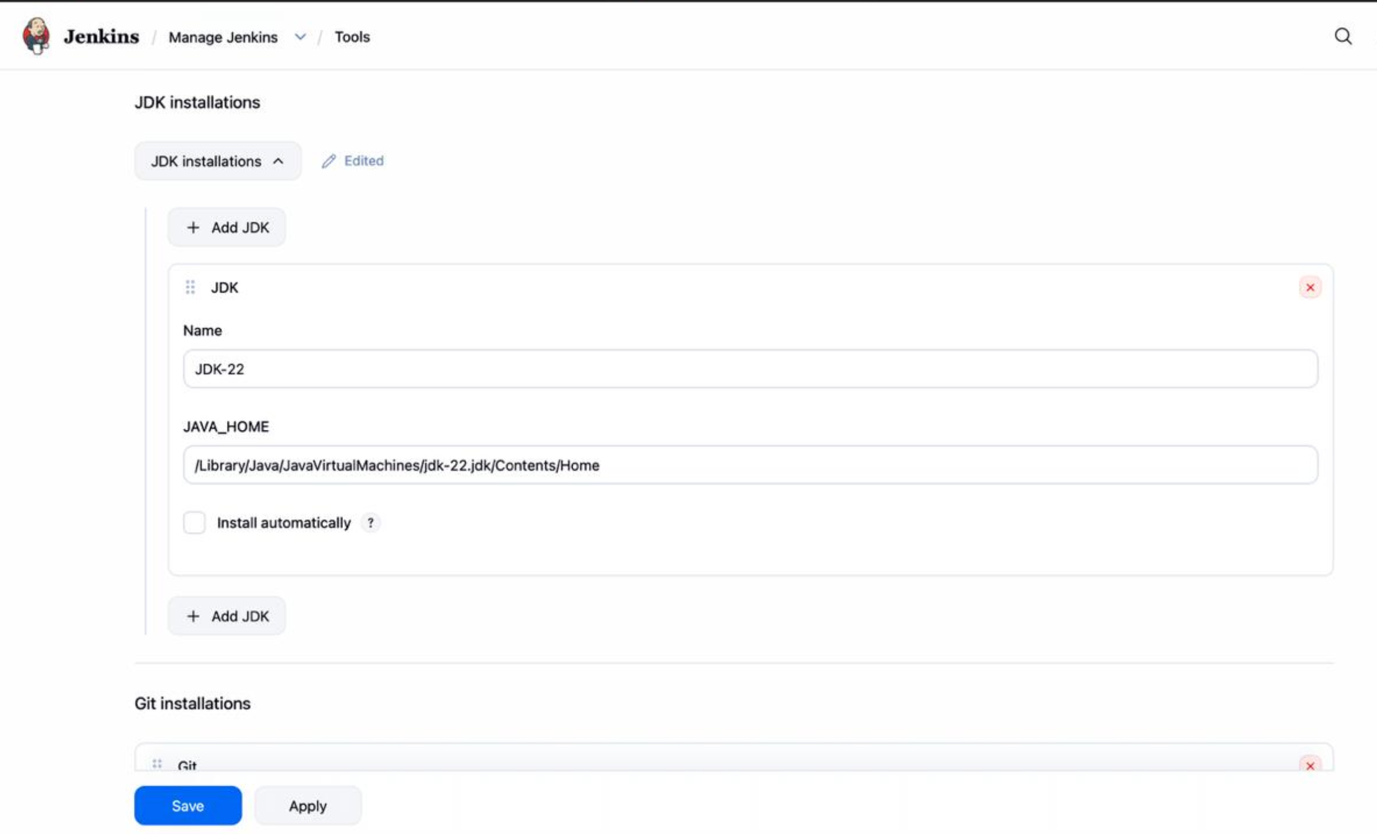
*Conclusion: For every $1 spent on CI infrastructure, the company recovers over $22 in developer productivity.*

**Task 2: Jenkins Installation on Local System**

**Jenkins Dashboard**



Global tool config



A screenshot of a computer

Description automatically generated

A screenshot of a chat

Description automatically generated

System config

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

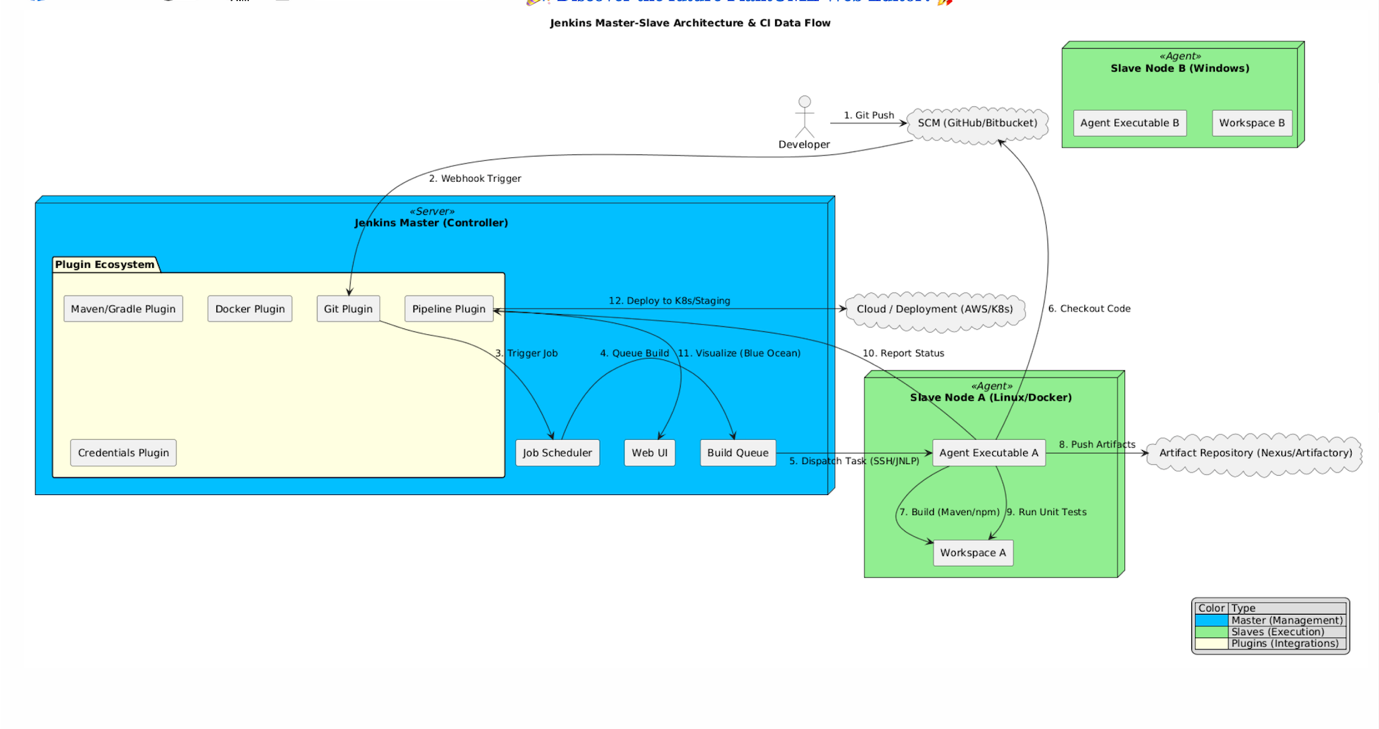
Description automatically generated

User management

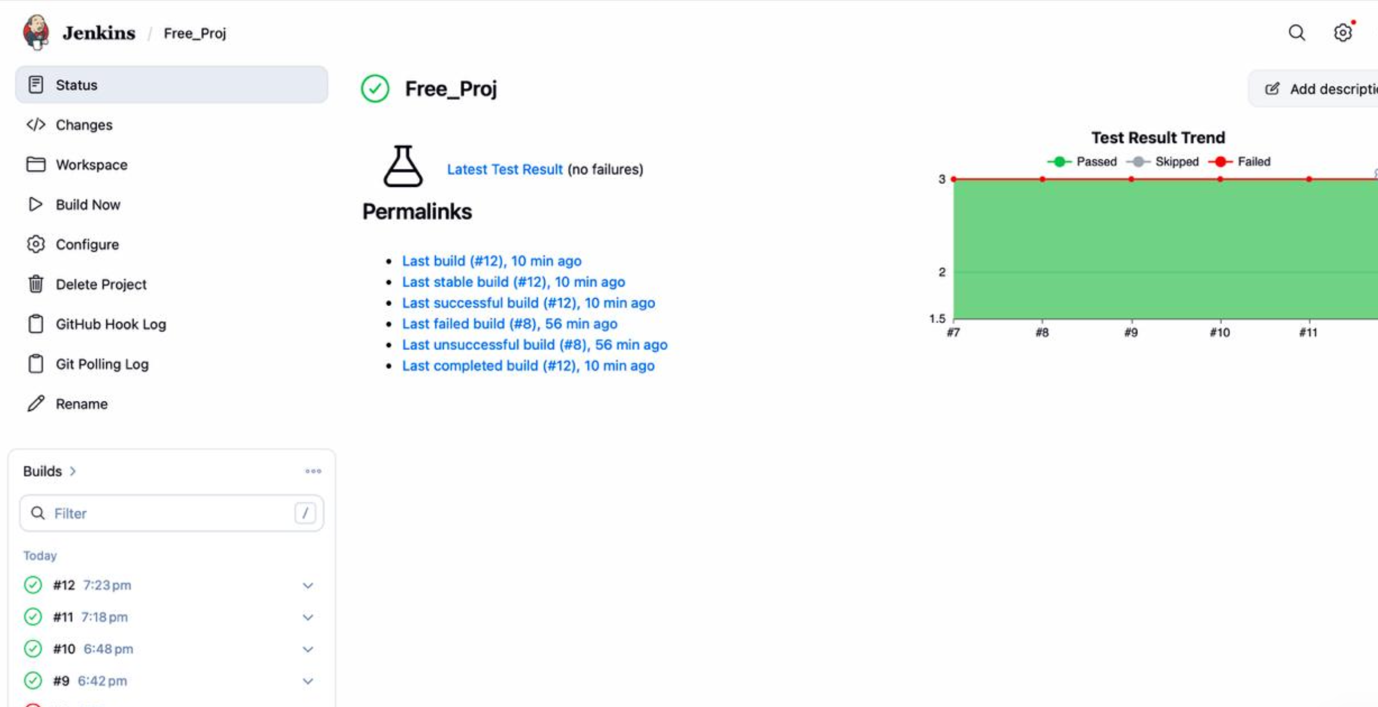
A screenshot of a computer

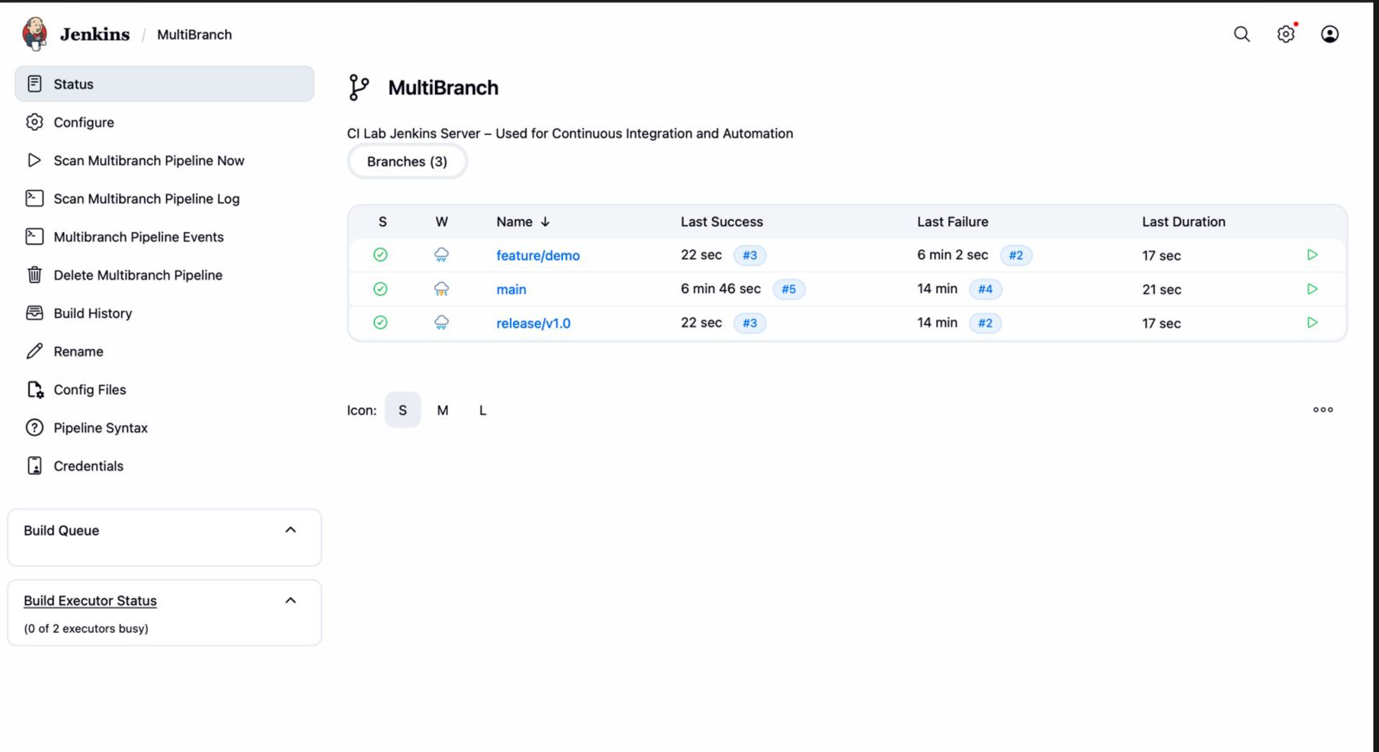
Description automatically generated

**Task 3: Jenkins Architecture Diagram**

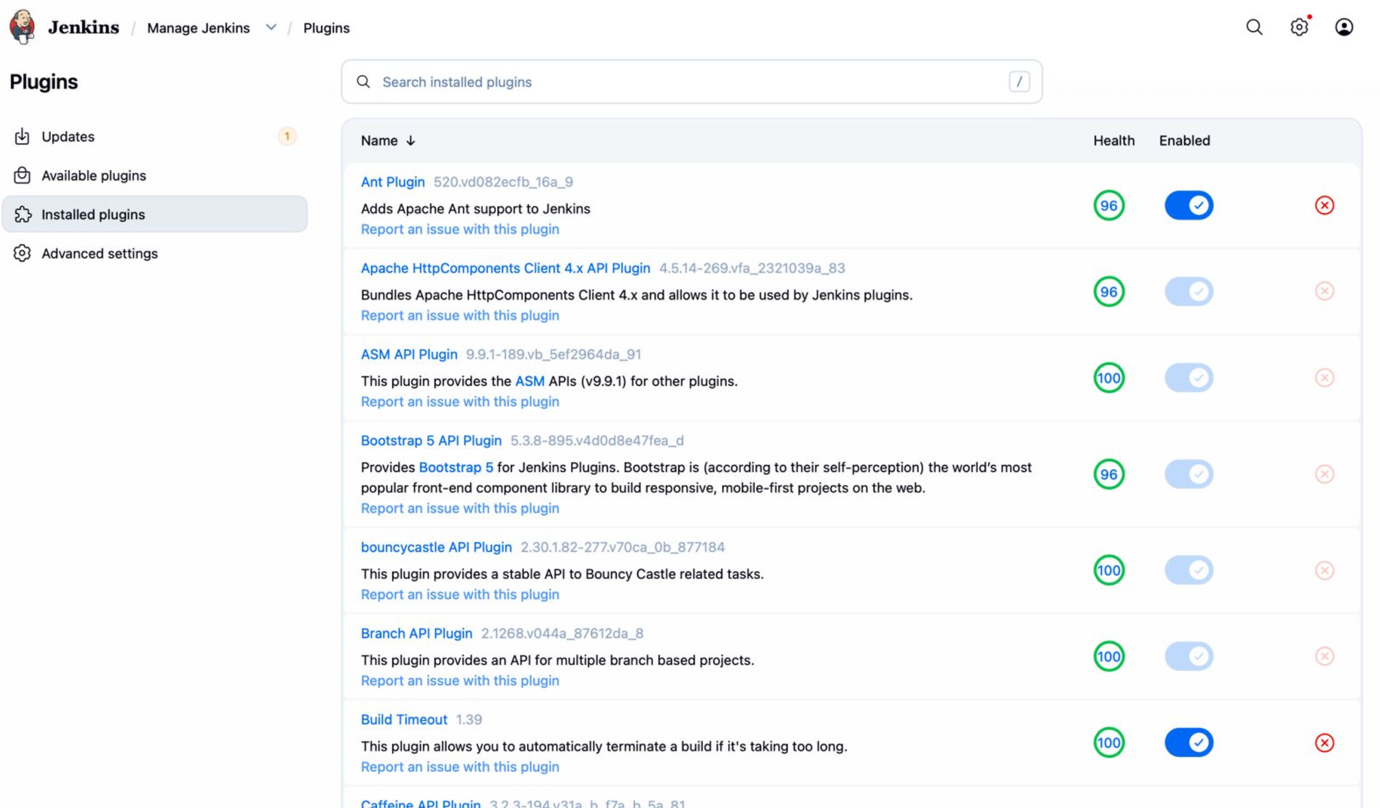


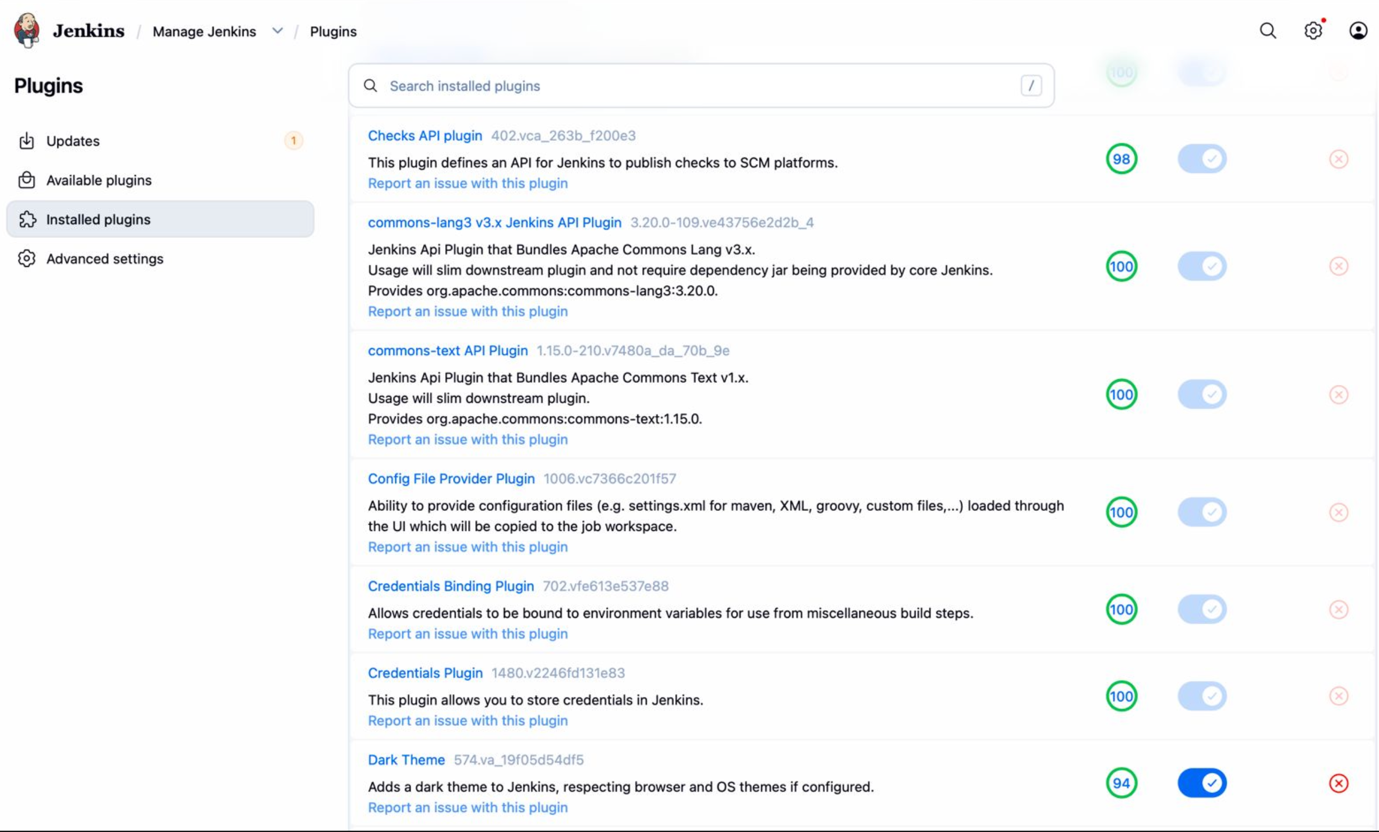
**Task 4: Create Different Types of Jenkins Jobs**

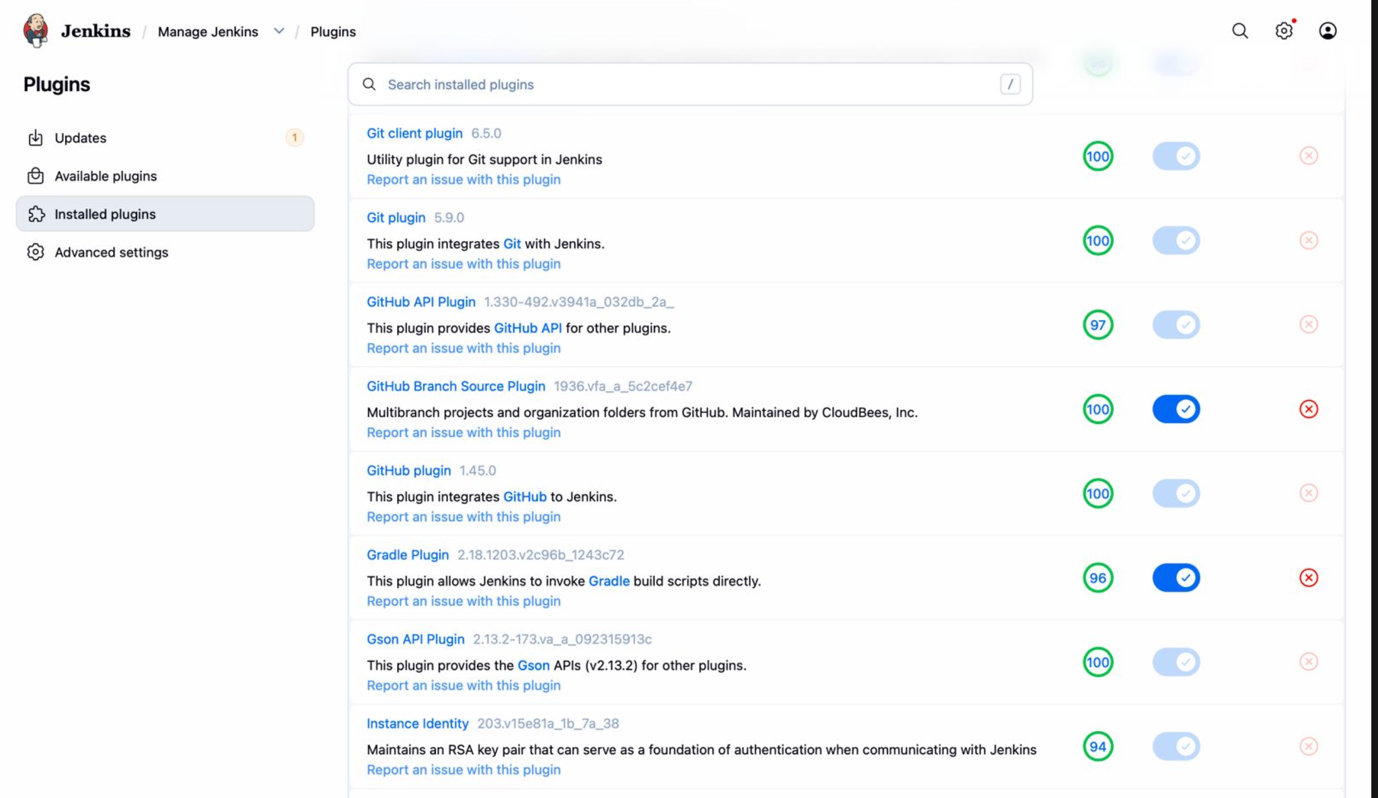


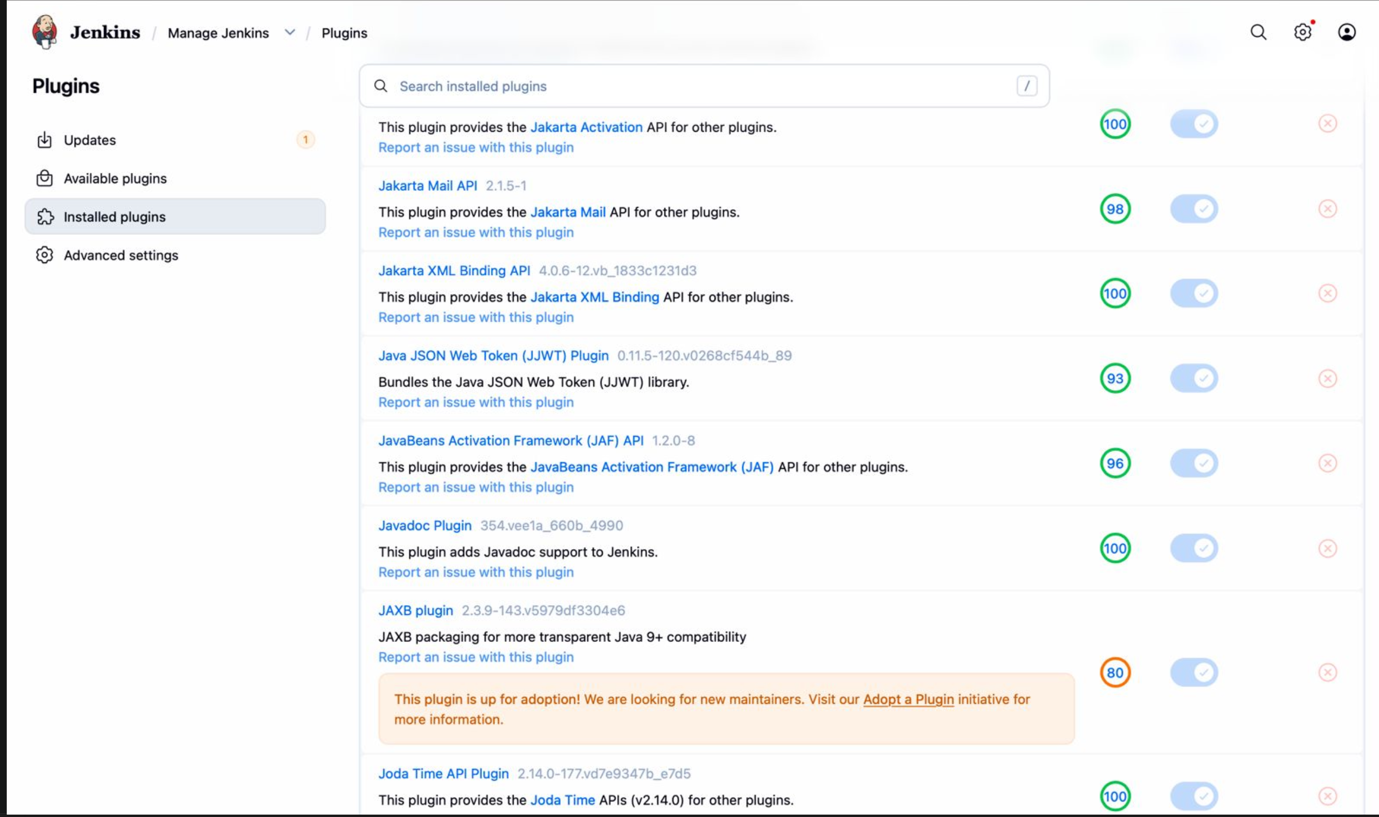


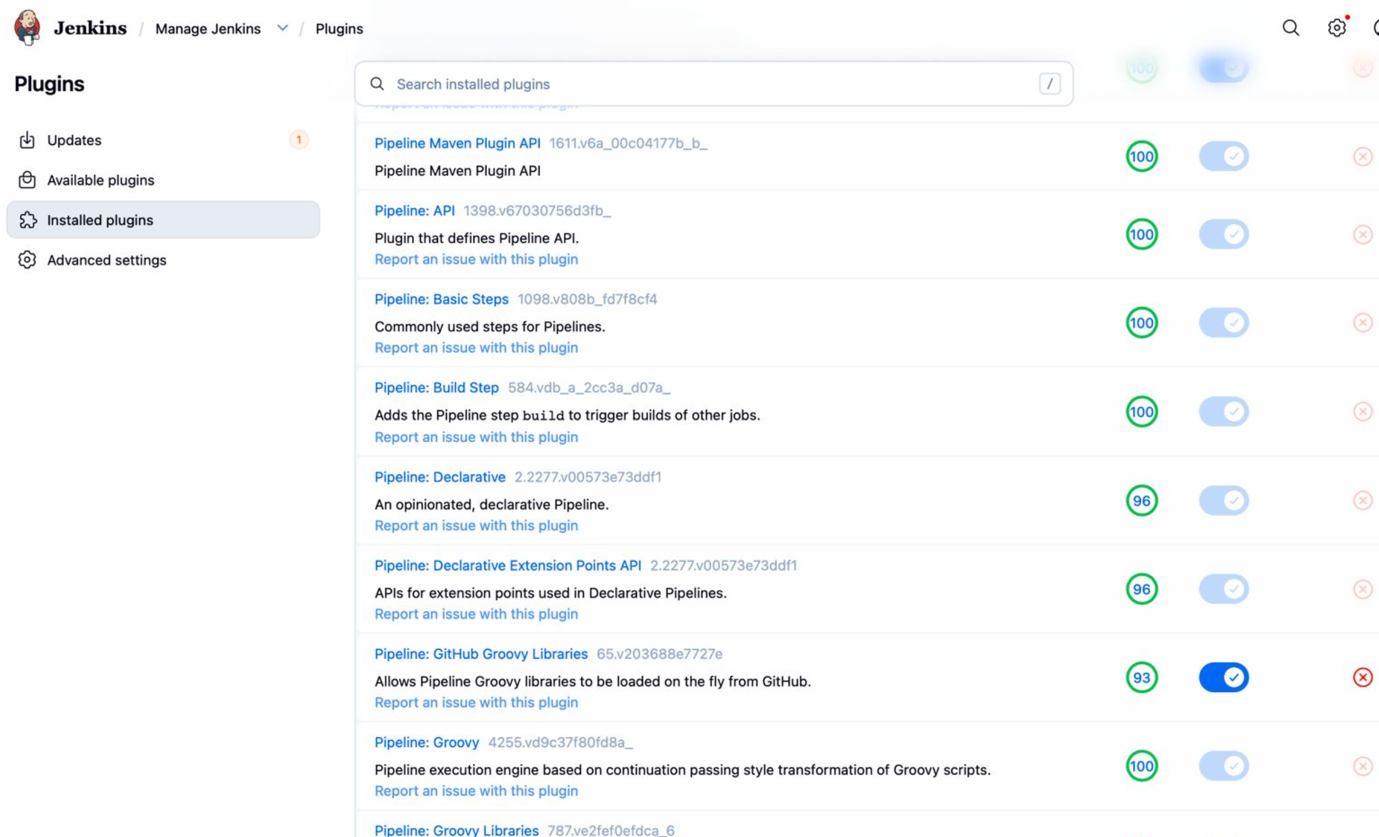
**Task 5: Plugin Management**

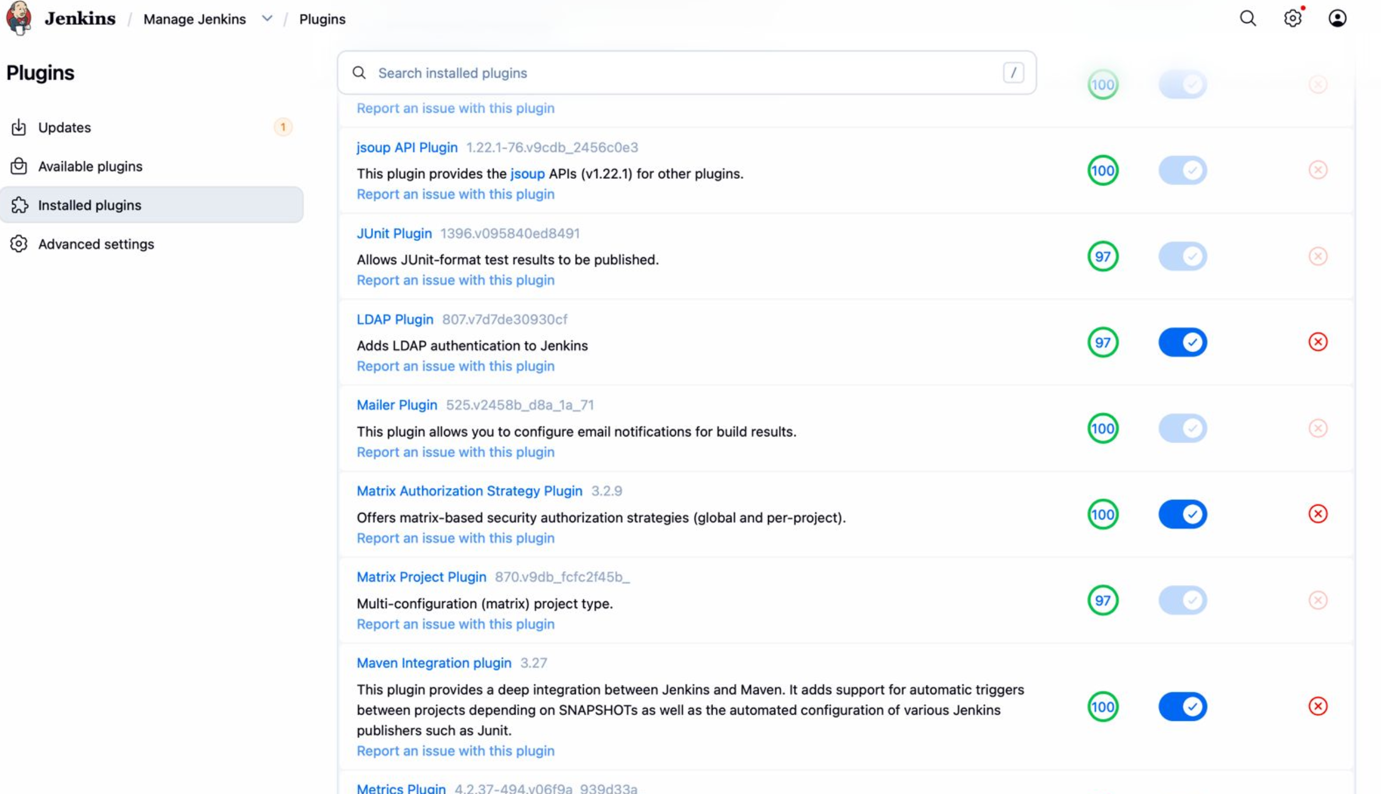








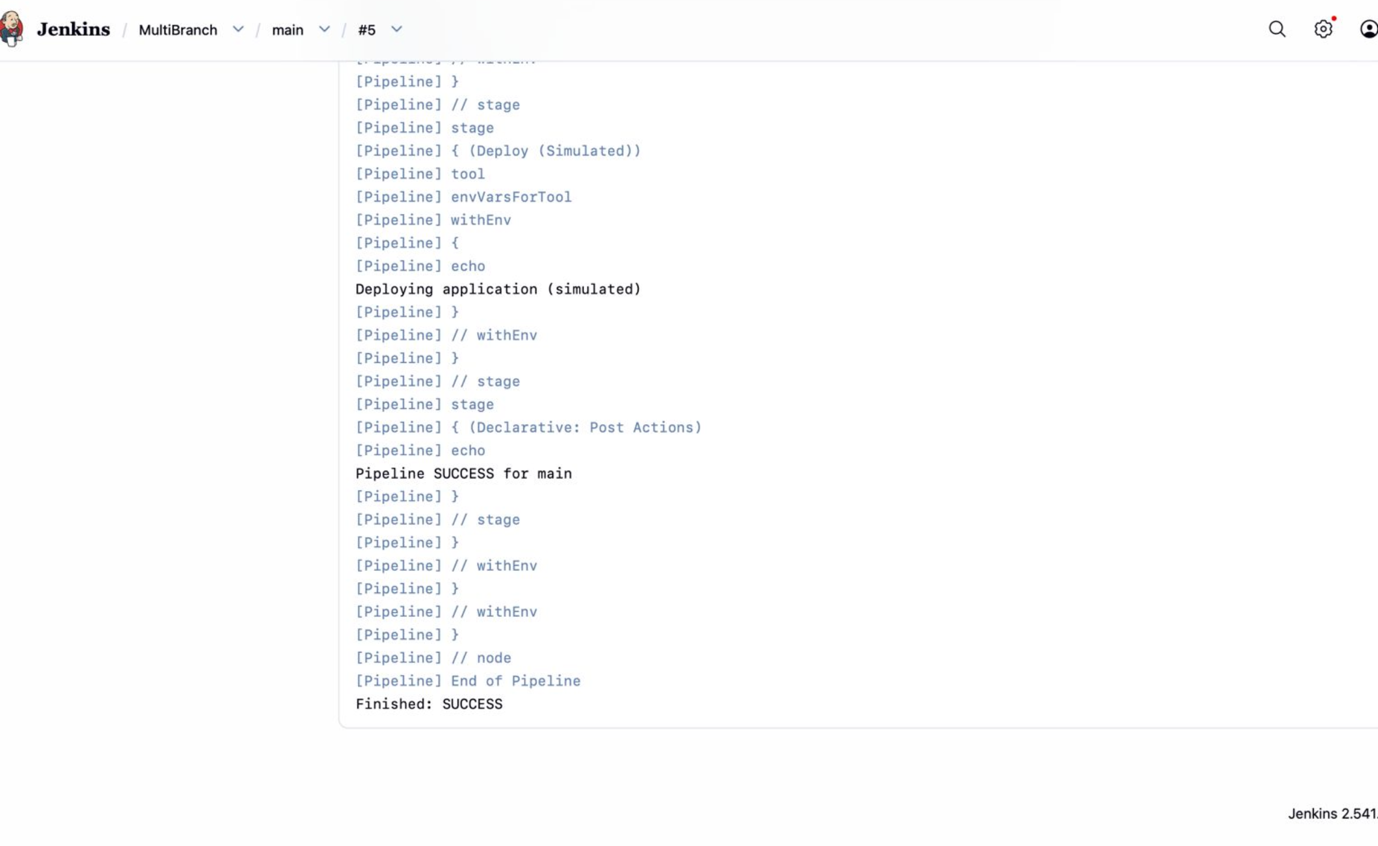




Multibranch Configuration

A screenshot of a computer

Description automatically generated



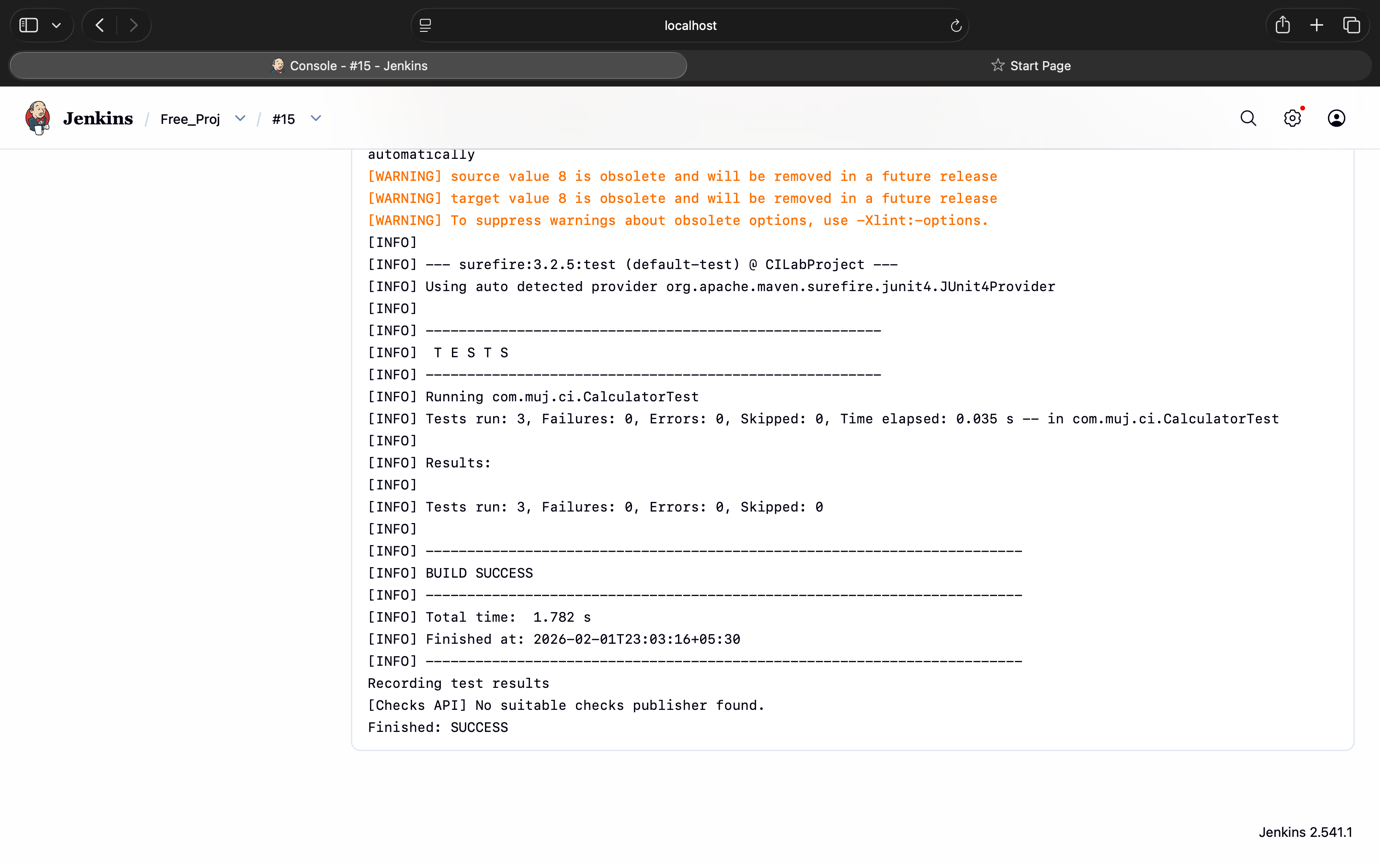
**Freestyle Project Config-**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



**Installation Guide**

**Prerequisites**

Ensure the following software is installed on your system:

* Java JDK 11 or higher
* Git
* Maven 3.8+
* Jenkins LTS (2.x)
* Web browser (Chrome / Firefox)

**Installation Steps**

**Step 1: Clone the Repository**

git clone https://github.com/chavi15/CI\_LAB.git

cd CI\_LAB

**Step 2: Install Jenkins**

* Download Jenkins LTS from the official website
* Start Jenkins using:
* java -jar jenkins.war
* Access Jenkins at:  
  http://localhost:8080

**Step 3: Initial Jenkins Setup**

* Unlock Jenkins using the initial admin password
* Install **Suggested Plugins**
* Create an admin user

**Step 4: Install Required Plugins**

Navigate to:

Manage Jenkins → Plugins

Install:

* Pipeline
* Git
* GitHub Branch Source
* Multibranch Pipeline
* Maven Integration
* JUnit
* Email Extension (Mailer)

**Step 5: Configure Build Tools**

Go to:

Manage Jenkins → Tools

Configure:

* **Maven**
  + Name: Maven\_3
  + Enable “Install automatically”

Save the configuration.

**Step 6: Create Multibranch Pipeline Job**

* Jenkins Dashboard → New Item
* Select **Multibranch Pipeline**
* Configure GitHub repository URL
* Enable branch discovery
* Save and run **Scan Multibranch Pipeline**

**User Manual**

**Project Overview**

This project demonstrates a **CI pipeline using Jenkins Multibranch Pipeline**, Maven, and GitHub.

**How the Pipeline Works**

| **Branch Type** | **Pipeline Behavior** |
| --- | --- |
| main | Build → Test → Package → Deploy (simulated) |
| feature/\* | Build → Test |
| release/\* | Build → Test → Package |

**Using the Pipeline**

1. Push code changes to GitHub
2. Jenkins automatically detects changes
3. Pipeline executes based on branch type
4. Results can be viewed in:
   * Console Output
   * Stage View
   * Multibranch Dashboard

**Viewing Build Results**

* Open Jenkins Dashboard
* Select the Multibranch Pipeline
* Click on a branch
* View:
  + Build logs
  + Stage execution
  + Test results

**3️Troubleshooting Guide**

**Error: mvn: command not found**

**Cause:** Maven not configured in Jenkins pipeline.

**Solution:**

* Ensure Maven is configured in:
* Manage Jenkins → Tools
* Ensure Jenkinsfile includes:
* tools {
* maven 'Maven\_3'
* }

**Branch build failing but main succeeds**

**Cause:** Jenkinsfile not updated in feature/release branches.

**Solution:**

git checkout feature/demo

git merge main

git push origin feature/demo

Repeat for release/\* branches.

**Jenkins not detecting branches**

**Cause:** Branch indexing not run.

**Solution:**

* Open Multibranch Pipeline
* Click **Scan Multibranch Pipeline Now**

**Git push rejected (fetch first)**

**Cause:** Remote branch ahead of local.

**Solution:**

git pull --rebase origin main

git push origin main

**Tests not visible in Jenkins**

**Cause:** JUnit plugin missing or test reports not generated.

**Solution:**

* Install **JUnit Plugin**
* Ensure tests run via mvn test

**Report-**

**Continuous Integration**

Continuous Integration (CI) is a software development practice where developers frequently integrate their code into a shared repository. Each integration is automatically verified through builds and tests to detect issues early in the development cycle. CI helps reduce integration problems, improves code quality, and ensures faster feedback to developers. In this project, CI is implemented using Jenkins, where every code push to the GitHub repository automatically triggers a build and test process.

**Continuous Integration and Continuous Deployment**

Continuous Integration focuses on automating the process of building and testing code whenever changes are made. Continuous Deployment, on the other hand, automatically deploys the application to production after successful testing. This project primarily demonstrates Continuous Integration, while deployment is simulated to avoid real production changes. This approach allows safe validation of the CI workflow without introducing deployment risks.

**Jenkins Architecture**

Jenkins follows a master–agent architecture. The Jenkins Controller is responsible for managing jobs, scheduling builds, handling plugins, and maintaining build history. Jenkins Agents are responsible for executing the actual build tasks such as compiling code and running tests. This architecture enables scalability and efficient distribution of workloads across multiple machines.

**Multibranch Pipeline Concept**

A Multibranch Pipeline automatically detects branches in a source code repository and creates separate pipelines for each branch. This eliminates the need to manually configure individual jobs for every branch. In this project, Jenkins automatically detects the main branch, feature branches, and release branches, allowing different CI behaviors to be applied depending on the branch type.

**Plugin Ecosystem in Jenkins**

Jenkins uses plugins to extend its functionality. Plugins enable source code management integration, pipeline execution, build automation, testing, and visualization. In this project, plugins were used to integrate GitHub for source control, Maven for building the application, JUnit for testing, and pipeline visualization for monitoring build stages. These plugins work together to provide a complete CI environment.

**CI Pipeline Workflow**

The CI pipeline begins when a developer pushes code to the GitHub repository. Jenkins detects the change and checks out the latest source code. The application is then built using Maven, followed by execution of unit tests. Test results are collected and displayed in Jenkins. The pipeline provides immediate feedback about the success or failure of the build, ensuring quick detection of errors.

**Branch-Based Pipeline Strategy**

Different pipeline strategies are implemented based on the branch type. Feature branches focus on building and testing new functionality. The main branch represents stable code and includes additional steps such as packaging and simulated deployment. Release branches prepare production-ready builds by performing complete validation. This strategy reflects real-world CI practices used in software development.

**Challenges Faced During Implementation**

Several challenges were encountered during implementation. One common issue was Maven not being detected by Jenkins, which was resolved by configuring Maven in Jenkins and referencing it explicitly in the Jenkinsfile. Another issue involved branch builds failing due to outdated Jenkinsfile versions, which was fixed by merging the main branch into feature and release branches. Git synchronization issues were resolved using proper pull and rebase commands.

**Overall Analysis**

The implemented CI system demonstrates how Jenkins can be used to automate software builds and testing efficiently. The multibranch pipeline ensures consistent behavior across different branches while allowing flexibility in build strategies. Automation reduces manual effort, improves reliability, and increases development speed. The system closely follows industry-standard CI practices.

**Conclusion**

This project successfully demonstrates the implementation of Continuous Integration using Jenkins, GitHub, and Maven. By automating builds, tests, and branch-specific workflows, the project highlights the importance of CI in modern software development. The solution provides a reliable, scalable, and maintainable CI pipeline suitable for real-world applications.