innovate achieve lead

BITS Pilani

Pilani Campus

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

By Prof A R Rahman

CSIS Group WILP

innovate achieve lead

BITS Pilani
Pilani Campus

Course Name: Introduction to DevOps
Course Code : CSI ZG514/SE ZG514

By Prof A R Rahman
CSIS Group WILP

CS - 7 Continuous Integration and Continuous Delivery

Coverage

Continuous Integration and Continuous Delivery	• Implementing Continuous Integration-Version control, automated build, Test • Prerequisites for Continuous Integration • Continuous Integration Practices • Team responsibilities • Using Continuous Integration Software (Jenkins as an example tool)

Implementing Continuous Integration-Version control automated build, Test

Understanding Continuous Integration

- Continuous integration is a practice where developers regularly merge their code changes into a central repository.
- Each integration is then automatically tested, allowing teams to detect issues early.
- This practice reduces integration problems and allows for quicker feedback on the quality of the code.

Benefits of Continuous Integration

- Early Bug Detection: By testing code frequently, bugs can be identified and fixed early in the development process.

- Improved Collaboration: CI encourages team collaboration, as developers are constantly integrating their work.

- Faster Release Cycles: With automated testing and deployment, teams can release software more frequently.

- Higher Code Quality: Regular testing leads to better code quality and fewer issues in production.

Benefits of Continuous Integration

- Choosing a Version Control System
- Before implementing CI, you need to choose a version control system.
- The most popular options are Git, Subversion (SVN), and Mercurial.
- Git is the most widely used due to its flexibility and powerful branching capabilities.

Benefits of Continuous Integration

- Setting Up a Version Control System
- Install Git: If you choose Git, start by installing it on your machine.
- You can download it from git-scm.com.
- Create a Repository: Initialize a new Git repository in your project directory: bash git init
- Add Files: Add your project files to the repository: bash git add .
- Commit Changes: Commit your changes with a descriptive message: bash git commit -m "Initial commit"

Benefits of Continuous Integration

- Integrating Continuous Integration Tools
- Once your version control system is set up, the next step is to integrate CI tools.
- There are several popular CI tools available, including Jenkins, Travis CI, CircleCI, and GitHub Actions.
- Each tool has its strengths, so choose one that fits your team's needs.

Setting Up Jenkins for CI

Jenkins is a widely used open-source automation server.

Here’s how to set it up for your project:

1. **Install Jenkins**: Download and install Jenkins from jenkins.io.

2. **Create a New Job**: After installation, open Jenkins in your web browser and create a new job.

3. **Configure Source Code Management**: In the job configuration, select Git as your source code management tool and provide the repository URL.

Setting Up Jenkins for CI

4.Set Up Build Triggers: Configure Jenkins to trigger builds on code changes.

- You can set it to poll the repository or use webhooks.

5.Define Build Steps: Specify the build steps, such as running tests or building the application.

- For example, if you are using a Node.js application, you might add: bash npm install npm test

You can also define a Jenkins pipeline using a Jenkinsfile.

```
pipeline {
     agent any
                stages {
                    stage('Build') {
                            steps {
                               sh 'npm install’
                                    }
                          }
                          stage('Test') {
                              steps {
                                     sh 'npm test’
                              }
                          }
                          stage('Deploy') {
                              steps {
                                    sh 'npm run deploy’
                                     }
                              }
                         }
            }
```

Using GitHub Actions for CI

- If your code is hosted on GitHub, GitHub Actions is a great option for CI.
- It allows you to automate workflows directly from your repository.
- Setting Up GitHub Actions

1. **Create a Workflow File**: In your repository, create a directory called .github/workflows and add a YAML file (e.g., ci.yml).

2.**Define the Workflow**: Here’s an example of a simple CI workflow for a Node.js application:
name: CI

Setting Up GitHub Actions

```
name: CI
on:
  push:
      branches:
          - main
jobs:
  build:
    runs-on: ubuntu-latest
      steps:
         - name: Checkout code
           uses: actions/checkout@v2

         - name: Set up Node.js
           uses: actions/setup-node@v2
            with:
            node-version: '14’

          - name: Install dependencies
           run: npm install

          - name: Run tests
           run: npm test
```

Best Practices for Continuous Integration

- Implementing CI is not just about setting up tools; it’s also about following best practices to ensure success.

- 1. Commit Often
- Encourage your team to commit code frequently.
- This practice reduces the chances of large merge conflicts and makes it easier to identify issues.

- 2. Keep Builds Fast
- A slow build process can discourage developers from running tests.
- Optimize your build process to ensure it runs quickly.

Best Practices for Continuous Integration

- 3. Automate Testing
- Automate as many tests as possible.
- This includes unit tests, integration tests, and end-to-end tests.
- The more tests you automate, the more reliable your CI process will be.

- 4. Monitor Build Status
- Set up notifications for build failures.
- This way, your team can quickly address issues as they arise.

- 5. Use Feature Branches
- Encourage the use of feature branches for new development.
- This practice allows developers to work on features without affecting the main codebase.

Troubleshooting Common CI Issues

- Even with the best setup, issues can arise. Here are some common problems and how to address them.

- Build Failures
- If a build fails, check the logs to identify the issue. Common causes include:
- Missing dependencies
- Syntax errors in code
- Failing tests

Troubleshooting Common CI Issues

- Slow Builds
- If your builds are slow, consider the following:
- Optimize your build scripts
- Use caching for dependencies
- Parallelize tests where possible

- Integration Issues
- If you encounter integration issues, ensure that:
- All team members are following the same branching strategy
- Code reviews are conducted before merging changes

Troubleshooting Common CI Issues

- Implementing continuous integration with version control systems is a powerful way to improve your software development process.
- By automating testing and integrating code changes frequently, you can enhance collaboration, reduce bugs, and speed up release cycles.
- Whether you choose Jenkins, GitHub Actions, or another CI tool, following best practices will help you create a robust CI pipeline that benefits your entire team.

Key Benefits of Integration

- Automated Testing and Building: Every commit or merge triggers automated tests and build processes, ensuring that issues are caught early.
- Streamlined Deployment: Automates the deployment process, enabling frequent and reliable releases.
- Improved Collaboration: Facilitates better collaboration among team members by integrating code changes seamlessly into the shared repository.
- Traceability: Provides a clear audit trail of what changes have been made, by whom, and how they impact the software.

Integrating Version Control with CI/CD Pipelines

- 1. Selecting the Right Tools

- Choose a version control system that aligns with your team’s expertise and project requirements.
- Git is the most popular choice for many teams.
- Select CI/CD tools that offer seamless integration with your VCS. Jenkins, GitLab CI/CD, CircleCI, and Travis CI are widely used.

- 2. Setting Up Repository Hooks

- Implement repository hooks in your VCS to trigger CI/CD processes.
- For instance, a post-commit hook in Git can notify the CI/CD tool to start a new build/test cycle.

Integrating Version Control with CI/CD Pipelines

- 3. Branching Strategy

- Adopt a branching strategy that works well with your CI/CD pipeline.
- Strategies like Git Flow or trunk-based development can be effective.
- Configure your CI/CD pipeline to handle different branches differently, e.g., feature branches may trigger a different pipeline than the main branch.

- 4. Automating the Build Process

- Configure the CI/CD pipeline to automatically compile and build the code upon receiving a trigger from the VCS.
- Manage dependencies and ensure that the build environment is consistent and replicable.

Integrating Version Control with CI/CD Pipelines

5. Implementing Automated Testing

Integrate a suite of automated tests (unit tests, integration tests, etc.) in the CI/CD pipeline.
Ensure that tests are run for every commit and that the results are reported back to the team.

6. Deployment Automation

Automate the deployment process to different environments (development, staging, production) based on the pipeline’s rules.
Implement strategies like blue-green deployments or canary releases to minimize deployment risks.

Integrating Version Control with CI/CD Pipelines

- 7. Monitoring and Feedback
- Incorporate monitoring tools in the pipeline to track application performance post-deployment.
- Set up feedback mechanisms to alert developers of pipeline failures or test failures.

Best Practices for Integration

1. Keep the Build Fast

Optimize your build and test processes to complete them as quickly as possible, reducing wait times for feedback.

Break down large monolithic builds into smaller, more manageable parts if necessary.

2. Manage Secrets Securely

Use secret management tools (like HashiCorp Vault, AWS Secrets Manager) to handle credentials and other sensitive data securely in your CI/CD pipelines.

Avoid storing secrets in the source code or version control.

Best Practices for Integration

3. Version Everything

- Version not just the source code, but also the CI/CD pipeline configurations, scripts, and infrastructure as code (IaC).

- This practice ensures that your entire build and deployment process is reproducible.

4. Immutable Artifacts

- Once a build artifact is created, it should remain immutable.
- Any change should trigger a new build.
- Store artifacts in a repository manager like JFrog Artifactory or Sonatype Nexus.

Best Practices for Integration

5. Monitor and Optimize

•Continuously monitor the performance of your CI/CD pipelines.

•Collect metrics and use them to optimize pipeline performance.

6. Documentation and Training

•Document your CI/CD processes and train your team members on best practices.

•Ensure that new team members understand how to interact with the CI/CD system.

Best Practices for Integration

- Integrating version control with CI/CD pipelines is a critical aspect of modern software development, significantly enhancing the speed, efficiency, and reliability of building and deploying software.

- By automating the build, test, and deployment processes, this integration helps in maintaining a high standard of code quality and accelerating the delivery of software products.

- As technologies and practices evolve, continuous adaptation and optimization of the integration between version control and CI/CD pipelines remain essential for DevOps teams.

Streamlining Your Development with Continuous Integration

- Continuous integration (CI) best practices are crucial for faster development cycles, higher quality code, and more frequent, reliable releases.
- Eight continuous integration best practices to optimize your development pipeline.
- By implementing these practices, your team can improve efficiency and achieve key business goals.
- Learn how to maintain a single source repository, automate builds, implement self-testing builds, commit daily, maintain build speed, test in production-like environments, simplify access to executables, and ensure build transparency.

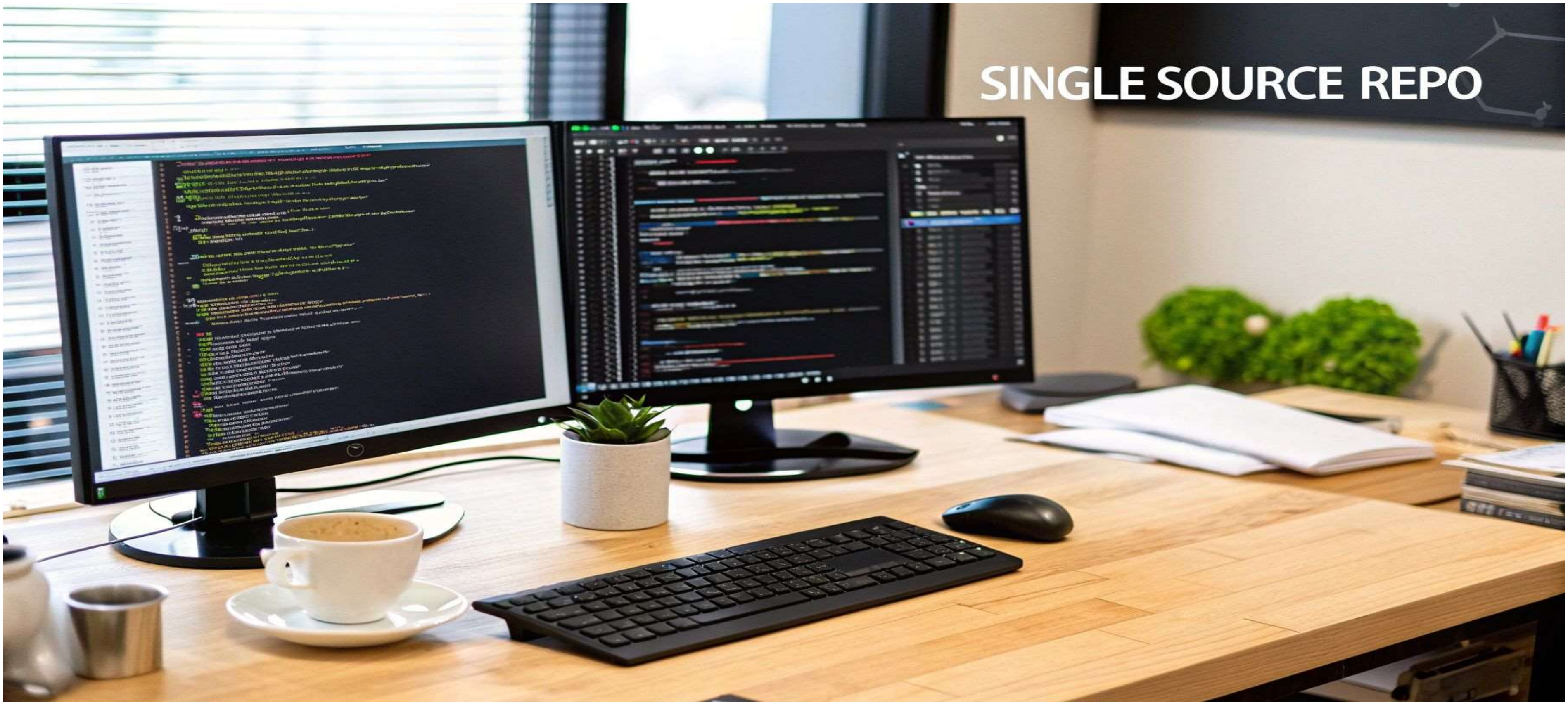
Streamlining Your Development with Continuous Integration

- 1. Maintain a Single Source Repository
- Maintaining a single source repository is a cornerstone of effective continuous integration (CI) and, by extension, successful software development.
- This practice involves consolidating all code, configuration files, scripts, and other related assets into one version control system (VCS).
- This VCS then serves as the definitive source of truth for the entire project.
- This centralized approach ensures all team members, whether they're developers, testers, or operations personnel, work from the same codebase.
- This shared foundation drastically reduces integration conflicts, maintains consistency across the development environment, and fosters seamless collaboration.

Streamlining Your Development with Continuous Integration

- A single source repository, often referred to as a “monorepo” in its purest form, utilizes a centralized VCS like Git, SVN, or Mercurial.
- Branching strategies, such as GitFlow or trunk-based development, are crucial for managing parallel development efforts within the repository.
- This setup offers complete codebase visibility for all team members, fostering transparency and shared understanding.
- Integrated access control and permissions within the VCS further enhance security and control over the codebase.

Streamlining Your Development with Continuous Integration



Streamlining Your Development with Continuous Integration

- This approach deserves its place as a top continuous integration best practice due to several key benefits.

- It provides a single source of truth, simplifying the tracking of changes and versioning.

- This, in turn, facilitates easier code reviews and streamlines collaboration, particularly valuable for distributed teams.

- Moreover, a single source repository enables more reliable build and deployment processes, laying the groundwork for robust CI/CD pipelines.

Features and Benefits:

•Centralized Version Control: Using established systems like Git ensures efficient code management and history tracking.

•Branching Strategies: Implementing strategies like GitFlow allows for organized parallel development and feature isolation.

•Complete Codebase Visibility: All team members have access to the entire project, fostering collaboration and understanding.

•Integrated Access Control: Permissions management within the VCS enhances security and ensures controlled access to the codebase.

Features and Benefits:

- Simplified Tracking and Versioning: Changes are easily tracked and managed, leading to a clear development history.

- Easier Code Reviews and Collaboration: A shared codebase simplifies the review process and encourages collaboration among team members.

- Reliable Build and Deployment Processes: Consistent codebase leads to more dependable and predictable build and deployment pipelines.

Streamlining Your Development with Continuous Integration

- 2. Automate the Build Process
- A cornerstone of continuous integration best practices is automating the build process.

- This involves transitioning from manual execution of build steps to a scripted or declarative approach.

- Instead of developers manually compiling code, running tests, and packaging applications, these tasks are handled by automated systems.

- This ensures builds are reproducible, consistent, and can be triggered automatically whenever code changes are committed, or on a scheduled basis.

- This consistency is crucial for continuous integration, enabling faster feedback cycles and quicker identification of integration issues.

Streamlining Your Development with Continuous Integration

Streamlining Your Development with Continuous Integration

- Automating your build process offers several key features such as script-based or declarative build configurations, providing flexibility in how you define your build pipeline.
- Robust dependency management and resolution mechanisms ensure that all required libraries and components are correctly fetched and integrated.
- The process culminates in the automated generation of artifacts, which can be anything from compiled binaries and deployment packages to container images.
- Build environment consistency is also paramount, and automation helps ensure that builds behave predictably across different environments.
- Finally, many build tools support parallel execution capabilities, significantly reducing build times and improving resource utilization.

Streamlining Your Development with Continuous Integration

- This practice deserves its place on the list of continuous integration best practices because it significantly improves efficiency and reliability.

- Manual builds are prone to human error and inconsistencies, whereas automation eliminates these risks, leading to more stable and predictable deployments.

- By implementing these continuous integration best practices, particularly automating your build process, organizations in IN, regardless of size or industry, can significantly improve their software development lifecycle, accelerating time to market and delivering higher-quality software.

Streamlining Your Development with Continuous Integration

- 3. Make Builds Self-Testing
- A cornerstone of effective continuous integration (CI) is the practice of making builds self-testing.
- This means integrating automated tests directly into your build process.
- Whenever new code is committed and a build is triggered, a comprehensive suite of tests automatically runs, verifying code functionality, integration points, and checking for regressions.
- This continuous validation of code quality prevents defects from propagating to later stages of development or, worse, into production.
- It allows developers to catch issues early, when they are easier and less expensive to fix.
- This is crucial for maintaining a rapid and reliable development lifecycle, a key element of successful continuous integration best practices.

Streamlining Your Development with Continuous Integration

Streamlining Your Development with Continuous Integration

- Self-testing builds incorporate various levels of testing, ensuring comprehensive coverage. These levels typically include:

- Unit Tests: These test individual components or functions in isolation to ensure they work correctly.

- Integration Tests: These verify the interaction between different modules and services, ensuring they function correctly together.

- System Tests: These test the entire system as a whole, simulating real-world scenarios.

- UI Tests: These focus on the user interface, ensuring that the application behaves as expected from the user’s perspective.

Streamlining Your Development with Continuous Integration

- The build process generates reports and metrics based on the test results, providing valuable insights into code quality and potential issues.
- Notifications are automatically sent if the build fails due to failing tests, alerting the development team to take immediate action.
- Code coverage analysis is often included to measure how much of the codebase is actually being tested.

Streamlining Your Development with Continuous Integration

- 4. Everyone Commits to the Mainline Every Day

- This continuous integration best practice, often referred to as trunk-based development, emphasizes integrating code changes into the main branch (typically ‘main’ or ‘trunk’) at least once per day.

- This seemingly simple act has profound implications for development workflows and drastically improves the overall software delivery process.

- It’s a cornerstone of effective continuous integration and a key differentiator between high-performing development teams and those struggling with integration headaches.

Streamlining Your Development with Continuous Integration

- How it Works:

- Instead of developers working in isolation on long-lived feature branches, they commit small, incremental changes directly to the mainline.

- This encourages frequent merging and early detection of integration conflicts.

- For features that are not yet ready for release, techniques like feature toggles (also known as feature flags) are used to hide the incomplete functionality from end-users.

- For larger, more disruptive changes, the branch by abstraction pattern can be employed to incrementally refactor and integrate the new functionality while maintaining a working mainline.

Streamlining Your Development with Continuous Integration

- 5. Keep the Build Fast

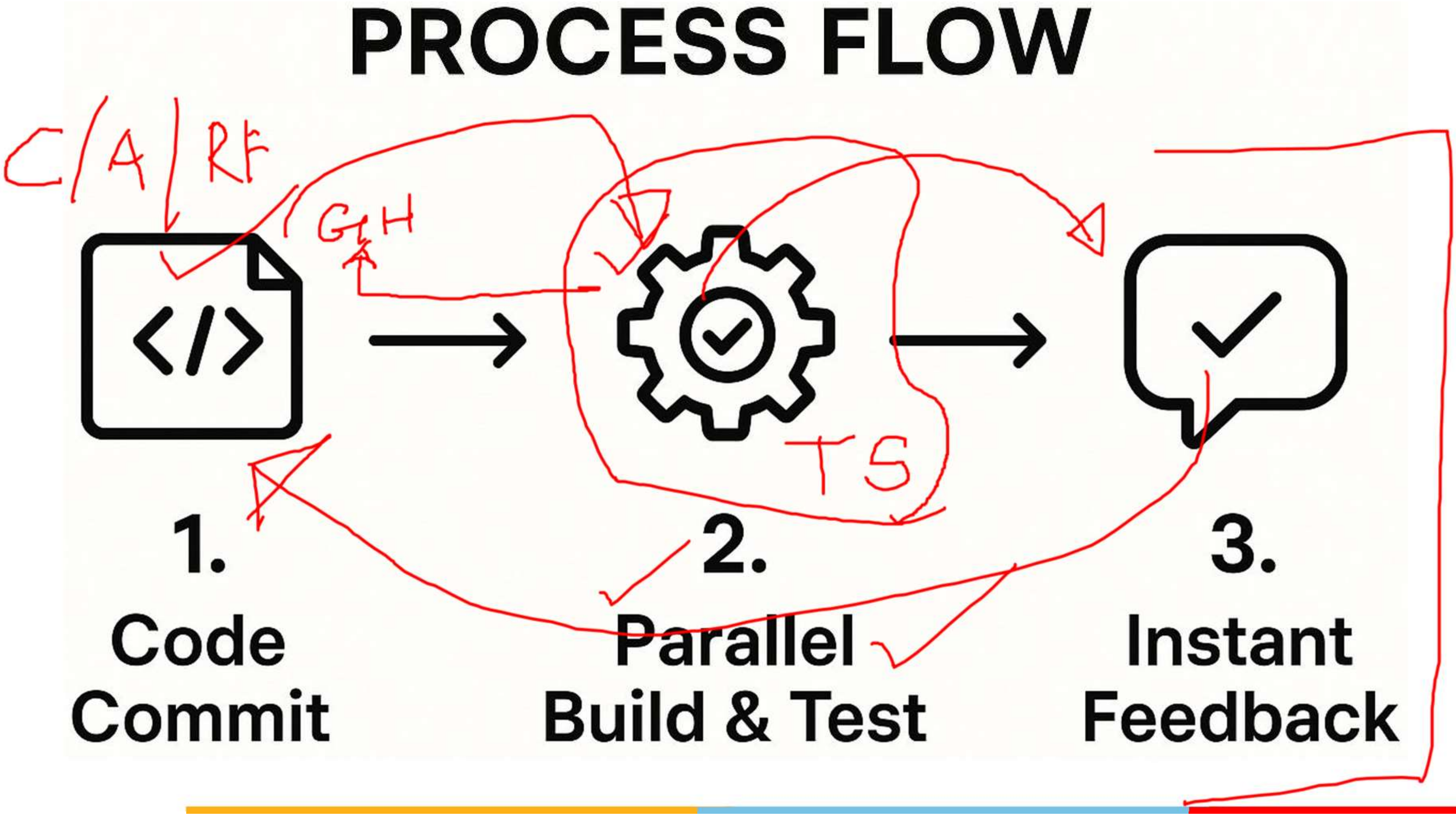
- Maintaining fast builds is a cornerstone of effective continuous integration (CI), making it a critical best practice.

- Quick builds provide the rapid feedback developers need to catch and correct issues early, minimizing disruptions and maintaining a productive flow state.

- A sluggish build process, on the other hand, discourages frequent integration, leading to larger, more complex integrations and slowing down the entire development cycle.

- This is crucial for any team practicing continuous integration, from startups in India to established enterprise IT departments.

Streamlining Your Development with Continuous Integration



Streamlining Your Development with Continuous Integration

- The process starts with “Code Changes” and proceeds through “Build Trigger,” “Build Execution,” and “Test Execution,” finally culminating in “Deployment.”

- Each step highlights potential bottlenecks and corresponding optimization strategies, such as caching, parallelization, and incremental builds, underscoring the importance of streamlining each stage for optimal speed and efficiency.

- As the visualization clearly shows, optimizing each stage contributes to faster overall build times, which in turn reduces delays and ensures faster feedback loops.

Streamlining Your Development with Continuous Integration

- Fast builds are achieved through a combination of build time optimization techniques, such as incremental builds that process only changed components, build parallelization and distributed execution across multiple machines, test suite optimization and selective test execution, and multi-stage build pipelines with quick feedback loops.

- Features like these are what enable organizations to achieve truly rapid feedback and development cycles.

Streamlining Your Development with Continuous Integration

- This practice deserves its place in the list of continuous integration best practices because it directly impacts developer productivity and the overall efficiency of the development process.

- The benefits of fast builds are numerous: faster feedback to developers, encouragement of more frequent commits and integrations, reduced context switching for developers waiting for builds, increased team velocity and productivity, and a greater number of iterations and improvements achievable each day.

- However, achieving and maintaining fast builds can also present challenges. It can require a significant engineering investment, potentially leading to trade-offs between build speed and the thoroughness of testing.

- Ongoing maintenance is necessary as the codebase grows, and additional infrastructure costs might be incurred to support distributed builds or caching mechanisms.

Streamlining Your Development with Continuous Integration

- Despite these potential drawbacks, the advantages often outweigh the costs, especially for growing companies and enterprises.

- Examples of successful implementations highlight the value of investing in build speed.

- Google’s build system, for example, processes changes in under 10 minutes despite billions of lines of code, a testament to the power of dedicated build optimization.

- Facebook’s Buck build system, designed for their massive monorepo, provides another example of speed optimization at scale.

- Shopify, too, realized significant gains, reducing their build times from 40 minutes to under 10 minutes.

Streamlining Your Development with Continuous Integration

- 6. Test in a Production-like Environment
- Testing in a production-like environment is a crucial continuous integration best practice that significantly reduces the risk of deployment failures and improves the reliability of software releases.

- This approach involves creating a testing environment that closely mirrors the actual production environment, replicating infrastructure, data patterns, network configurations, system dependencies, and even security policies.

- By doing so, you can validate that your application behaves consistently when deployed and catch environment-specific issues early in the development cycle, long before they impact your users.
- This is particularly important for continuous integration as it allows for rapid and reliable deployments.

Streamlining Your Development with Continuous Integration

- How it Works:

- The core principle is to minimize the differences between testing and production.

- This means using similar hardware, software versions, network topologies, and data sets.

- The closer the resemblance, the more accurate your testing results will be and the higher your confidence in a successful deployment.

- This can be achieved through various techniques like Infrastructure-as-Code, containerization, and data masking.

Streamlining Your Development with Continuous Integration

- Features of a Production-like Environment:

- Infrastructure-as-Code (IaC): Tools like Terraform, CloudFormation, or Ansible allow you to define your infrastructure in code, ensuring consistent environment provisioning across testing, staging, and production.

- Containerization: Docker and Kubernetes enable packaging applications and their dependencies into portable containers, guaranteeing consistent execution across different environments.

- Production Data Sampling or Simulation: Using sanitized or simulated production data provides realistic test scenarios and helps uncover data-related bugs.

Streamlining Your Development with Continuous Integration

- Features of a Production-like Environment:

- Network Condition Replication: Simulate network latency, bandwidth limitations, and even network failures to test application resilience.

- Similar Scaling and Performance Characteristics: The test environment should have similar scaling capabilities to production to identify potential performance bottlenecks under load.

- Realistic Security Configurations: Implementing security policies and access controls mirroring production helps uncover vulnerabilities early on.

Streamlining Your Development with Continuous Integration

- Why This Approach Matters:
- This best practice directly addresses the pervasive “works on my machine” problem by eliminating the discrepancies between development, testing, and production environments.
- It also allows teams to validate non-functional requirements like performance, security, and scalability much earlier in the development lifecycle.
- This reduces firefighting during deployments and fosters a culture of proactive problem-solving

Streamlining Your Development with Continuous Integration

- This best practice is highly relevant for various stakeholders including:
- Startups and early-stage companies looking to build robust and scalable systems.
- Enterprise IT departments managing complex applications and infrastructure.
- Cloud architects and developers responsible for designing and deploying cloud-native applications.
- DevOps and infrastructure teams tasked with automating and optimizing the software delivery pipeline.
- Business decision-makers and CTOs concerned with application reliability and minimizing downtime.

Streamlining Your Development with Continuous Integration

- By embracing the “Test in a Production-like Environment” best practice, organizations in the IN region and globally can significantly improve their continuous integration processes, enhance software quality, and increase confidence in their releases.

- This approach fosters a proactive and preventative approach to software development, reducing the risk of costly production failures and empowering teams to deliver high-quality software at speed.

Streamlining Your Development with Continuous Integration

- 7. Make it Easy for Anyone to Get the Latest Executable

- A crucial continuous integration best practice is ensuring easy access to the latest executable for all stakeholders.

- This involves setting up a robust and accessible artifact repository where build artifacts are stored, versioned, and easily retrievable.

- Streamlining access to executables facilitates smoother testing, simplified demonstrations, and consistent deployments, ultimately contributing to a more efficient and reliable CI/CD pipeline.

- This practice is especially important for implementing continuous integration best practices effectively.

Streamlining Your Development with Continuous Integration

- 7. Make it Easy for Anyone to Get the Latest Executable

- A crucial continuous integration best practice is ensuring easy access to the latest executable for all stakeholders.

- This involves setting up a robust and accessible artifact repository where build artifacts are stored, versioned, and easily retrievable.

- Streamlining access to executables facilitates smoother testing, simplified demonstrations, and consistent deployments, ultimately contributing to a more efficient and reliable CI/CD pipeline.

- This practice is especially important for implementing continuous integration best practices effectively.

Streamlining Your Development with Continuous Integration

- How it Works:

- An artifact repository acts as a central hub for all your build outputs.

- When your CI pipeline completes a build, it publishes the resulting executable (along with other artifacts like libraries or documentation) to this repository.

- The repository assigns a unique version number to the build and stores metadata associated with it, such as the commit ID, branch information, and test results.

- Team members can then easily access and download the specific version they need through a user-friendly interface, ensuring everyone works with consistent and verified binaries.

Streamlining Your Development with Continuous Integration

- Examples of Successful Implementation:

- Large organizations like Microsoft and Google maintain extensive internal artifact repositories that serve thousands of developers, enabling them to easily share and access the latest builds of various software components.

- Spotify's sophisticated release system, heavily reliant on artifact management, allows developers to deploy to production with confidence, leveraging automated versioning and deployment pipelines tied into their artifact repositories.

Streamlining Your Development with Continuous Integration

- When and Why to Use This Approach:
- Implementing an artifact repository should be a priority from the early stages of any software project.
- It becomes increasingly crucial as the project grows, the team expands, and the number of builds increases.
- This approach is particularly valuable in the following scenarios:

Streamlining Your Development with Continuous Integration

- Distributed Teams: Ensures consistent access to the latest executables regardless of location.
- Microservice Architectures: Simplifies management and deployment of numerous interconnected services.
- Frequent Releases: Facilitates rapid iteration and deployment of new features and bug fixes.
- Compliance Requirements: Provides a clear audit trail of build artifacts and their associated metadata.

Streamlining Your Development with Continuous Integration

- 8. Ensure Transparent Build Process with Visible Results

- A crucial best practice in continuous integration (CI) is ensuring a transparent build process with readily visible results.

- This transparency, a cornerstone of effective CI/CD pipelines, empowers teams to proactively monitor build health, swiftly identify and resolve issues, and foster a shared sense of responsibility for code quality.

- By making build status, test results, and other key metrics easily accessible to all stakeholders, organizations can significantly improve their development workflows and deliver higher-quality software faster.

Streamlining Your Development with Continuous Integration

- 8. Ensure Transparent Build Process with Visible Results

- This practice is especially important for startups and enterprise IT departments in the IN region looking to scale their development efforts and maintain a competitive edge.

- For cloud architects, developers, DevOps, and infrastructure teams, transparency offers valuable insights into the health and efficiency of the CI/CD pipeline.

- Even business decision-makers and CTOs can benefit from the clear overview provided by transparent build processes, enabling them to make informed decisions based on real-time data.

Streamlining Your Development with Continuous Integration

- How it Works:
- Transparency in CI/CD involves implementing tools and practices that make various aspects of the build process readily observable. This includes:

- Real-time build status dashboards: Dashboards offer an at-a-glance view of the current state of builds, highlighting successes, failures, and ongoing processes.

- Comprehensive build logs and artifacts: Detailed logs provide a record of each step in the build process, making it easier to pinpoint the cause of failures. Access to build artifacts allows for thorough examination of the generated outputs.

- Automated notifications for build events: Notifications through email, chat platforms, or dedicated monitoring tools alert teams to critical events like build failures or successful deployments.

Streamlining Your Development with Continuous Integration

- How it Works:

- Historical build performance metrics: Tracking metrics like build time, success rate, and test coverage over time provides valuable insights for process optimization.

- Test coverage and quality reports: Visibility into test results, including code coverage and the details of failed tests, helps ensure code quality and identify areas for improvement.

- Integration with communication platforms: Connecting the CI/CD pipeline with platforms like Slack or Microsoft Teams facilitates seamless communication and collaboration around build events.

Streamlining Your Development with Continuous Integration

- Examples of Successful Implementation:
- Imagine a team in Bengaluru using a large wall-mounted screen displaying the build status of their various projects, much like Atlassian's wallboards.
- This allows everyone, from developers to product managers, to instantly understand the current state of development.
- A startup in Hyderabad could implement a system similar to Amazon's internal deployment dashboards, providing all engineering teams with real-time visibility into the deployment pipeline, fostering a culture of shared responsibility.
- Similarly, a development team in Mumbai could leverage tools akin to Spotify's squad health monitoring and build transparency tools to gain a deep understanding of their team's performance and identify bottlenecks in the CI/CD process.

Prerequisites for Continuous Integration

- Check-In Regularly – The most important practice for continuous integration to work properly is frequent check-ins to trunk or mainline of the source code repository.

- The check-in of code should happen at least a couple of times a day.

- Checking in regularly brings lots of other benefits.

- It makes changes smaller and thus less likely to break the build.

Prerequisites for Continuous Integration

- It means the recent most version of the software to revert to is known when a mistake is made in any subsequent build.

- It also helps to be more disciplined about refactoring code and to stick to small changes that preserve behavior.

- It helps to ensure that changes altering a lot of files are less likely to conflict with other peoples work.

- It allows the developers to be more explorative, trying out ideas and discarding them by reverting back to the last committed version.

Prerequisites for Continuous Integration

- Create a Comprehensive Automated Test Suite – If you don't have a comprehensive suite of automated tests, a passing build only means that the application could be compiled and assembled.

- While for some teams this is a big step, its essential to have some level of automated testing to provide confidence that your application is actually working.

- Normally, there are 3 types of tests conducted in Continuous Integration namely unit tests, component tests, and acceptance tests.

Prerequisites for Continuous Integration

- Unit tests are written to test the behavior of small pieces of your application in isolation.
- They can usually be run without starting the whole application.
- They do not hit the database (if your application has one), the filesystem, or the network.
- They don't require your application to be running in a production-like environment.
- Unit tests should run very fast your whole suite, even for a large application, should be able to run in under ten minutes.

Prerequisites for Continuous Integration

- Component tests test the behavior of several components of your application.
- Like unit tests, they don't always require starting the whole application.
- However, they may hit the database, the filesystem, or other systems (which may be stubbed out).
- Component tests typically take longer to run.

Prerequisites for Continuous Integration

- Keep the Build and Test Process Short – If it takes too long to build the code and run the unit tests, you will run into the following problems.

- People will stop doing a full build and will run the tests before they check-in.

- You will start to get more failing builds.

- The Continuous Integration process will take so long that multiple commits would have taken place by the time you can run the build again, so you won't know which check-in broke the build.

- People will check-in less often because they have to sit around for ages waiting for the software to build and the tests to run.

Prerequisites for Continuous Integration

- Don't Check-In on a Broken Build − The biggest blunder of continuous integration is checking in on a broken build. If the build breaks, the developers responsible are waiting to fix it.

- They identify the cause of the breakage as soon as possible and fix it.

- If we adopt this strategy, we will always be in the best position to work out what caused the breakage and fix it immediately.

- If one of our colleagues has made a check-in and has as a result broken the build, then to have the best chance of fixing it, they will need a clear run at the problem.

- When this rule is broken, it inevitably takes much longer for the build to be fixed.
- People get used to seeing the build broken, and very quickly you get into a situation where the build stays broken all of the time.

Prerequisites for Continuous Integration

- Always Run All Commit Tests Locally Before Committing – Always ensure that the tests designed for the application are run first on a local machine before running them on the CI server.

- This is to ensure the right test cases are written and if there is any failure in the CI process, it is because of the failed test results.

- Take Responsibility for All Breakages that Result from Your Changes – If you commit a change and all the tests you wrote pass, but others break, the build is still broken.

- Usually this means that you have introduced a regression bug into the application.
- It is your responsibility because you made the change to fix all tests that are not passing as a result of your changes.
- In the context of CI this seems obvious, but actually it is not a common practice in many projects.

Continuous integration best practices

- Continuous integration (CI) helps dev teams be more productive and improve overall code quality.
- However, implementing CI is just one step to achieving faster deployments.
- To get the most out of your CI system, it’s important to incorporate best practices for continuous integration into your workflow.

Continuous integration best practices

- Commit early, commit often
- It's much easier to fix small problems than big problems, as a general rule.
- One of the biggest advantages of continuous integration is that code is integrated into a shared repository against other changes happening at the same time.
- If a development team commits code changes early and often, bugs are easier to identify because there is less code to sort through.
- By testing in small batches, code quality is improved and teams can iterate more effectively.

Continuous integration best practices

- Read the documentation (and then read it again)

- Continuous integration systems make documentation widely available, and this documentation can be very helpful long after you've implemented CI into your workflow.

- At GitLab, we have thorough CI/CD documentation that is updated frequently to reflect the latest processes.

- In can be helpful to reference the documentation in READMEs or in other accessible formats.

- Encourage team members to read the documentation first, bookmark links, create FAQs, and incorporate these resources into onboarding for new team members.

Continuous integration best practices

- Optimize pipeline stages

- CI pipelines contain jobs and stages: Jobs are the activities that happen within a particular stage, and once all jobs pass, code moves to the next stage.

- To get the most out of your CI pipelines, optimize stages so that failures are easy to identify and fix.

- Stages are an easy way to organize similar jobs, but there may be a few jobs in your pipeline that could safely run in an earlier stage without negatively impacting your project if they fail.

- Consider running these jobs in an earlier stage to speed up CI pipelines.

Continuous integration best practices

- Make builds fast and simple

- Nothing slows down a pipeline like complexity.
- Focus on keeping builds fast, and the best way to do that is by keeping things as simple as possible.
- Every minute taken off build times is a minute saved for each developer every time they commit.
- Since CI demands frequent commits, this time can add up.

- Martin Fowler discusses a guideline of the ten-minute build that most modern projects can achieve.

- Since continuous integration demands frequent commits, saving time on commit builds can give developers a lot of time back.

Continuous integration best practices

- Use failures to improve processes
- Improvement is a process. When teams change their response to failures, it creates a cultural shift for continuous improvement.
- Instead of asking *who* caused the failure, ask *what* caused the failure.
- This means shifting from a blaming culture to a learning culture.

- If teams are doing frequent commits, it becomes much easier to identify problems and solve them.
- If there are patterns in failed builds, look at the underlying causes.

- Are there non-code errors that are causing builds unnecessarily? Maybe incorporate an allow_failure parameter.
- Look for ways to continually improve, make failures blameless, and look for causes (not culprits).

Continuous integration best practices

- Test environment should mirror production

- In continuous integration, every commit triggers a build. These builds then run tests to identify if something will be broken by the code changes you introduce.

- The test pyramid is a way for developers to think of how to balance testing.

- End-to end testing is mostly used as a safeguard, with unit testing being used most often to identify errors.

- One important thing to keep in mind with testing is the environment.

- When the testing and production environments match, it means that developers can rely on the results and deploy with confidence.

Continuous integration best practices

- Test environment should mirror production
- Review Apps put the new code into a production-like live environment to visualize code changes.
- This feature helps developers assess the impact of changes.
- Continuous integration helps developers deploy faster and get feedback sooner.
- Ultimately, the best continuous integration system is the one you actually use.
- Find the right CI for your needs and then incorporate these best practices to make the most of your new CI workflow.

.

DevOps Team: Roles and Responsibilities

- DevOps roles and responsibilities revolve around **bridging the gap between development and operations to foster a collaborative**, agile, and efficient environment.

- Teams typically comprise **developers, testers, automation experts, quality assurance, security professionals, and release managers, each contributing to streamlining the CI/CD pipeline, automating repetitive tasks**, and ensuring robust infrastructure management through practices like Infrastructure as Code (IaC).

- Leadership, often in the form of a DevOps evangelist, is essential for driving cultural change, breaking down silos, and ensuring that security and compliance are integrated from the outset.

- Overall, DevOps emphasizes continuous improvement, rapid iteration, and cross-functional teamwork to deliver high-quality software that meets customer-centric business objectives.

DevOps Team: Roles and Responsibilities

Role	Primary Responsibilities
DevOps Engineer	Implements automation (CI/CD, IaC), integrates tools, and ensures systems are secure and scalable.
Release Manager	Orchestrates the software release lifecycle – planning releases, managing deployments, and monitoring post-release feedback.
Site Reliability Engineer (SRE)	Ensures system reliability in production, automating ops tasks and monitoring system health (often overlaps with DevOps).
Platform Engineer	Builds and maintains internal developer platforms to streamline workflows and reduce cognitive load for dev teams.
DevOps Evangelist	Champions DevOps culture and practices across the org, aligning teams and driving the DevOps adoption and mindset.

DevOps Team: Roles and Responsibilities

- How is a DevOps Team Structured?

- A DevOps team is structured as a cross-functional unit that brings together members from development, IT operations, quality assurance, and security. Rather than silos, all these roles work in unison on a shared goal: fast, continuous delivery of software.

- Each role has a clear place: for example, developers and testers build and validate code, while ops engineers and SREs manage deployment and reliability.

- A DevOps Evangelist or lead coordinates across the group to drive DevOps principles.

- The exact team composition can vary by organization size, but the core idea is to include all the skill sets needed to take an idea from code to production within a single collaborative team.

DevOps Team: Roles and Responsibilities

- Senior DevOps: As the team’s core responsibility would be on the person who owns the DevOps team, a senior person from the organization would be an ideal person to lead the team.
- DevOps Evangelist: This will ensure that the responsibilities of DevOps processes are assigned to the right people.
- Software developer/tester: Builds and tests code throughout the application lifecycle.
- Automation engineer/automation expert: Focuses on automating manual, repetitive tasks like deployments, tests, and infrastructure.
- Quality assurance professional: This person ensures that the software meets quality standards before deployment. They design test plans, run automated and manual tests, and validate CI/CD output.

DevOps Team: Roles and Responsibilities

- Security Engineer: Embeds security into every development phase by implementing DevSecOps practices, scanning code for vulnerabilities, managing secrets, and enforcing compliance policies.

- Release manager: Oversees the entire release lifecycle, from planning, scheduling, and building CI/CD pipelines to ensuring smooth deployment into production.

- Platform Engineer: This position focuses on building and maintaining internal developer platforms (IDPs) to streamline software delivery and reduce developers’ cognitive load.

- Site Reliability Engineer (SRE): Combines software and systems engineering to build and run large-scale, distributed, and fault-tolerant systems.

DevOps Team: Roles and Responsibilities

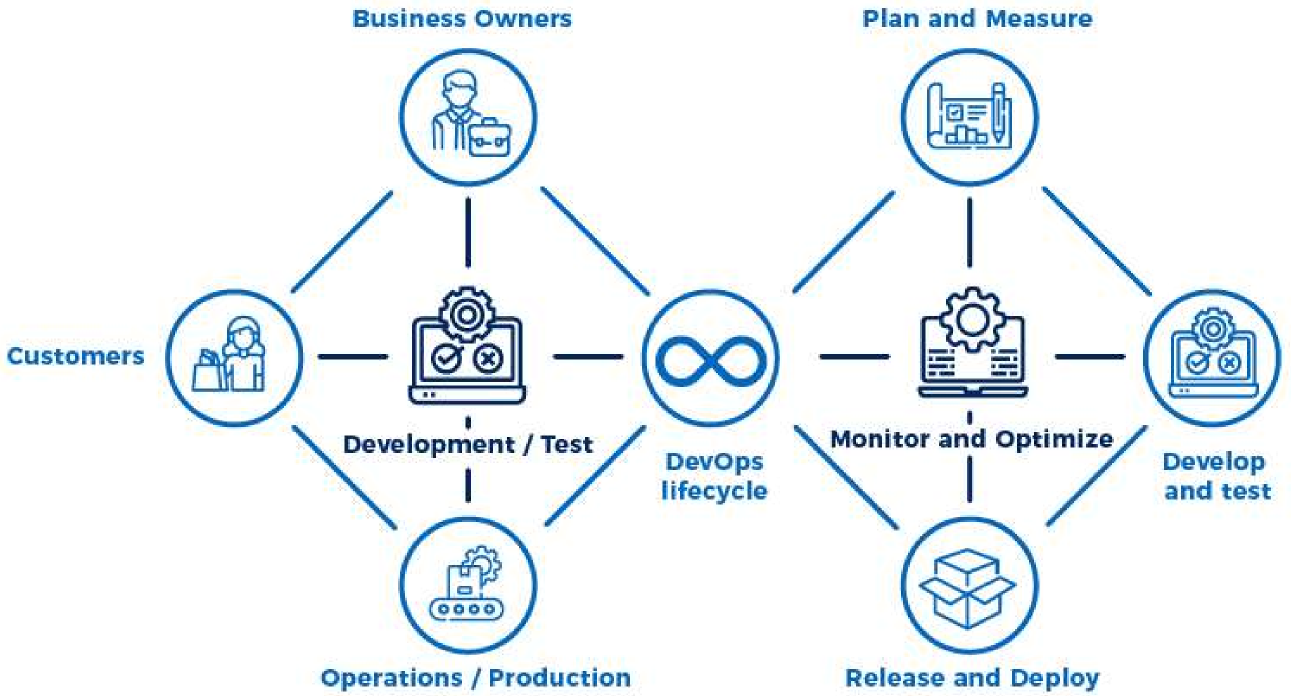
AI/ML Ops Engineer: Specializes in integrating machine learning models into the CI/CD pipeline, ensuring seamless deployment and monitoring of AI applications.

AIOps Engineer: Implements AI-driven solutions in IT operations, Leveraging machine learning for proactive monitoring, anomaly detection, and automated incident response to improve system reliability and efficiency.

Prompt Engineer: Focuses on designing and refining prompts for AI/ML models (especially large language models) to ensure accurate, relevant outputs

DevSecOps Engineer: Integrates security practices into the DevOps process, emphasizing a “shift-left” approach to identify and address security issues early in the development lifecycle.

What are the key responsibilities of a DevOps team?



Value Stream. Delivery Pipeline- optimizing the value stream

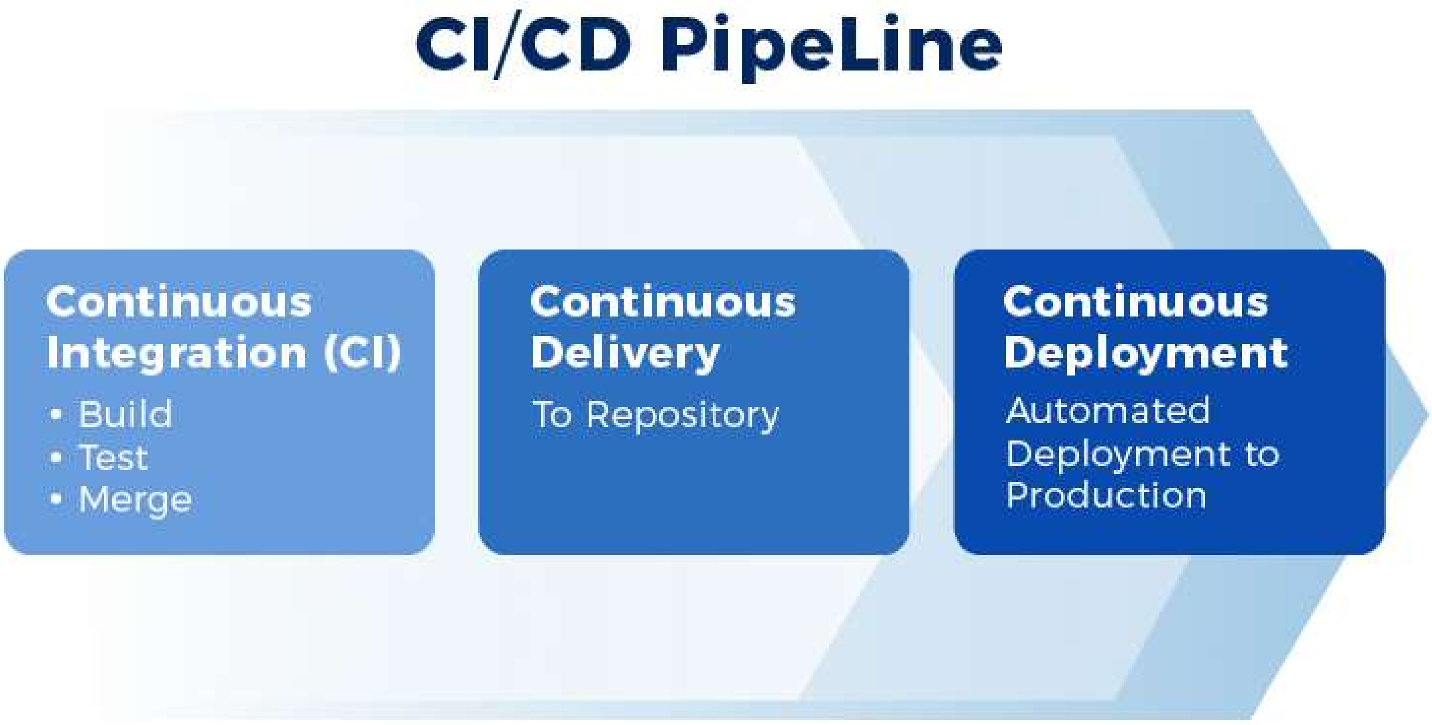
DevOps Responsibilities: CI/CD Pipelines

- Continuous Integration and Continuous Deployment (CI/CD) sits at the heart of DevOps.
- This pipeline comprises integrated processes to automate build, test, and deployment.
- serverless CI/CD pipelines have become prevalent, allowing teams to build and deploy applications without managing the underlying infrastructure
- In the Build phase, a compilation of the application takes place using a version control system.
- Here, the build is validated against organizational compliance requirements.

DevOps Responsibilities: CI/CD Pipelines

- In the test phase, the code is tested, and the Release phase delivers the application to the repository.
- In the deployment phase, the application is deployed to the required platforms.
- Serverless CI/CD platforms like GitHub Actions, AWS CodeCatalyst, and Harness are becoming standard, enabling teams to scale deployments without provisioning agents or runners manually.

DevOps Responsibilities: CI/CD Pipelines



CI/CD PipeLine

Continuous Integration (CI)

- Build
- Test
- Merge

Continuous Delivery

To Repository

Continuous Deployment

Automated Deployment to Production

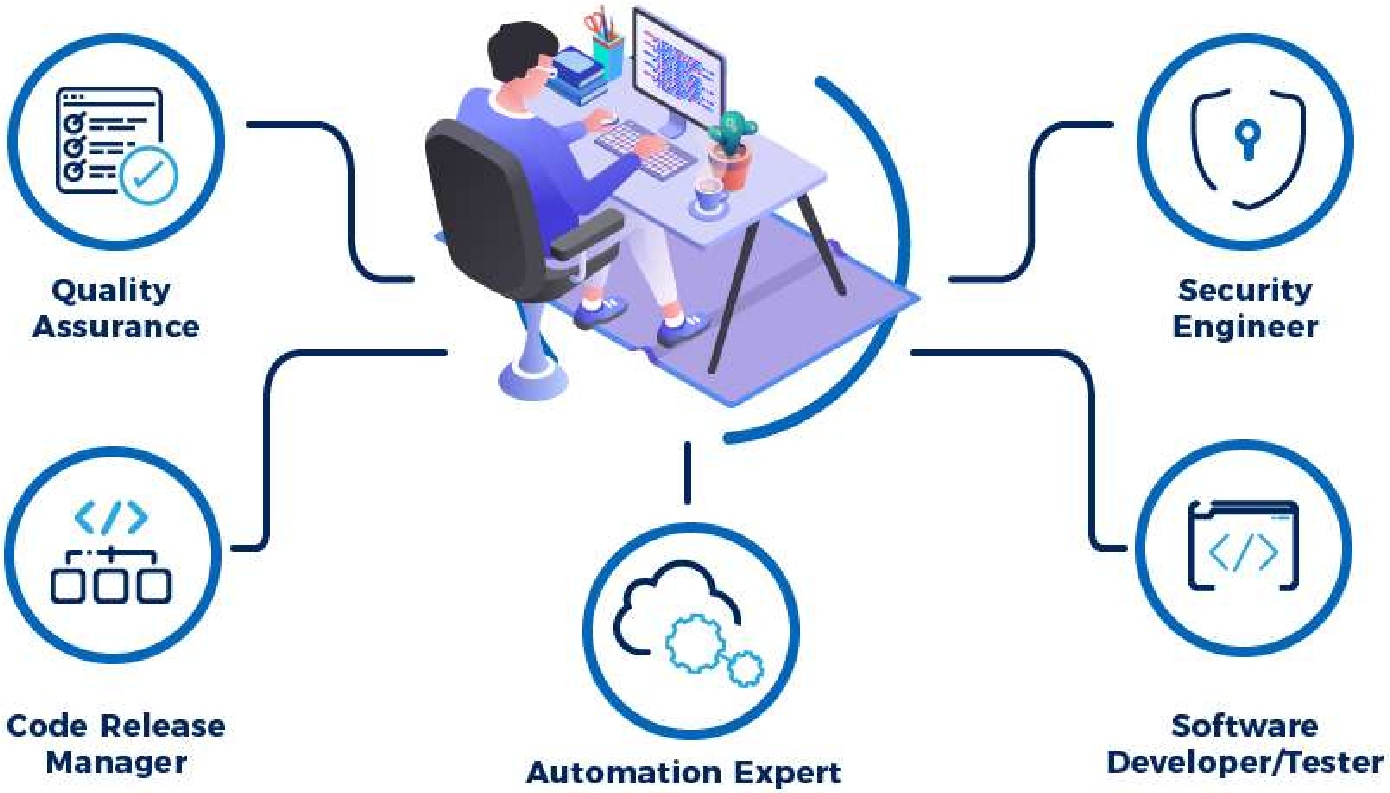
DevOps Responsibilities: CI/CD Pipelines

- Continuous Delivery takes the applications and delivers them to selected infrastructures.

- Testing moves to the left part of the CI/CD pipeline, wherein code is automatically tested before being delivered to production. It improves collaboration and quality.

- CI/CD has a huge impact on software development; that’s why we’ll see this trend in the Future of DevOps.

What are the Roles in a DevOps Team?



Best Practices for a Successful DevOps Team

- **Choose the right talent**: After acquiring the right talent, organize your teams across customer value streams. Start with smaller teams and scale up.

- **Allow each team to choose its tools**: Processes autonomously while maintaining a shared tool strategy and centralized visibility and monitoring.

- **Seamless communication**: Across the organization cannot be ignored.

- Using the right chat tools and communication tools is recommended.

Best Practices for a Successful DevOps Team

•Alert escalation and incident management tools: Play a handy role in helping members receive timely alerts and keep themselves updated with what’s happening across the infrastructure.

•They can integrate monitoring tools and share a common workflow.

•Deal with members individually: Regular pep talks, motivations, and inspirations would boost members’ morale, which would significantly impact the system’s overall productivity.

•New DevOps metrics gaining adoption in 2025 include Mean Time to Restore (MTTR), Deployment Frequency, Change Failure Rate, and Developer Experience Index (DXI).

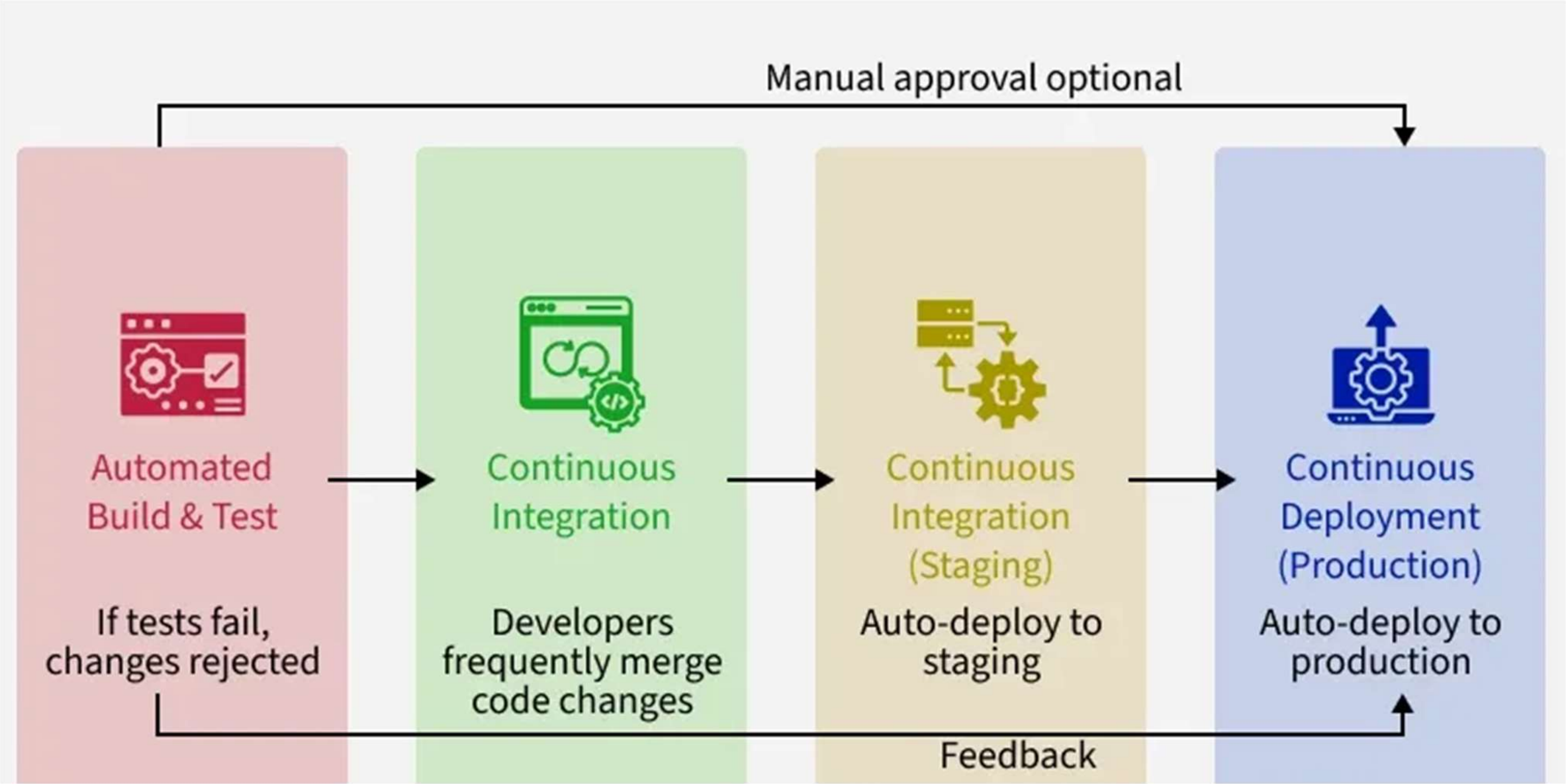
Using Continuous Integration Software (Jenkins)

- Jenkins is an open-source automation server that enables developers to reliably build, test, and deploy applications.

- It supports continuous integration and continuous delivery (CI/CD) workflows that allow teams to frequently deliver high-quality software.

- Jenkins provides an automation engine with an extensive plugin ecosystem that offers integrations for practically any DevOps toolchain. Its capabilities include:
- Automates the software delivery pipeline (build → test → deploy).
- Integrates seamlessly with version control systems like Git.
- Runs automated tests on each code change to ensure quality.
- Supports CI/CD workflows for faster and more reliable releases.
- Offers 1,800+ plugins to connect with almost any DevOps toolchain.
- Scales easily to handle distributed builds across multiple agents.

Using Continuous Integration Software (Jenkins)


Manual approval optional
Automated Build & Test
If tests fail, changes rejected
Continuous Integration
Developers frequently merge code changes
Continuous Integration (Staging)
Auto-deploy to staging
Continuous Deployment (Production)
Auto-deploy to production
Feedback

Using Continuous Integration Software (Jenkins)

- First, an automated build compiles the code and runs tests immediately.
- If the tests fail, the changes are rejected. This catches errors early.
- Next is continuous integration.
- Developers frequently merge their code changes into the main branch.
- With each merge, builds and tests are run to ensure nothing breaks.
- Then comes continuous delivery.
- The changes are deployed to testing/staging servers.
- This is like a dress rehearsal before the actual release.
- Finally, in continuous deployment, the changes are deployed directly to production with little manual intervention.
- This enables quick delivery of features to users.

Using Continuous Integration Software (Jenkins)

- Key components and concepts
- Jobs: The basic building blocks in Jenkins, used to run tasks like tests, builds, and deployments.
- They can be triggered automatically and configured with inputs, environments, and reports.
- Pipelines: Chain multiple jobs into an end-to-end workflow, providing visibility and control over the entire delivery process from code to production.
- Agents: Distributed servers or environments where Jenkins executes jobs. They help scale capacity but require managing distributed resources.
- Plugins: With 1,600+ options, plugins extend Jenkins to integrate with tools like Git, Docker, Kubernetes, and more—adding features for notifications, dashboards, analysis, and security.

Understanding Jenkins Pipelines

- Jenkins Pipelines bring all these stages together, creating an automated workflow that delivers software quickly, securely, and reliably.
- Code: Developers write source code and manage it in Git repositories, which track all changes. Webhooks notify Jenkins on new commits, triggering build pipelines automatically.
- Build: Jenkins fetches the latest code and uses build tools like Maven, Gradle, or MSBuild via plugins to compile, package (JARs, WARs, containers), and run unit tests and code checks.
- Test: Jenkins coordinates different test types (UI, performance, security, compatibility) using frameworks like Selenium, providing detailed reports, logs, and metrics.

Understanding Jenkins Pipelines

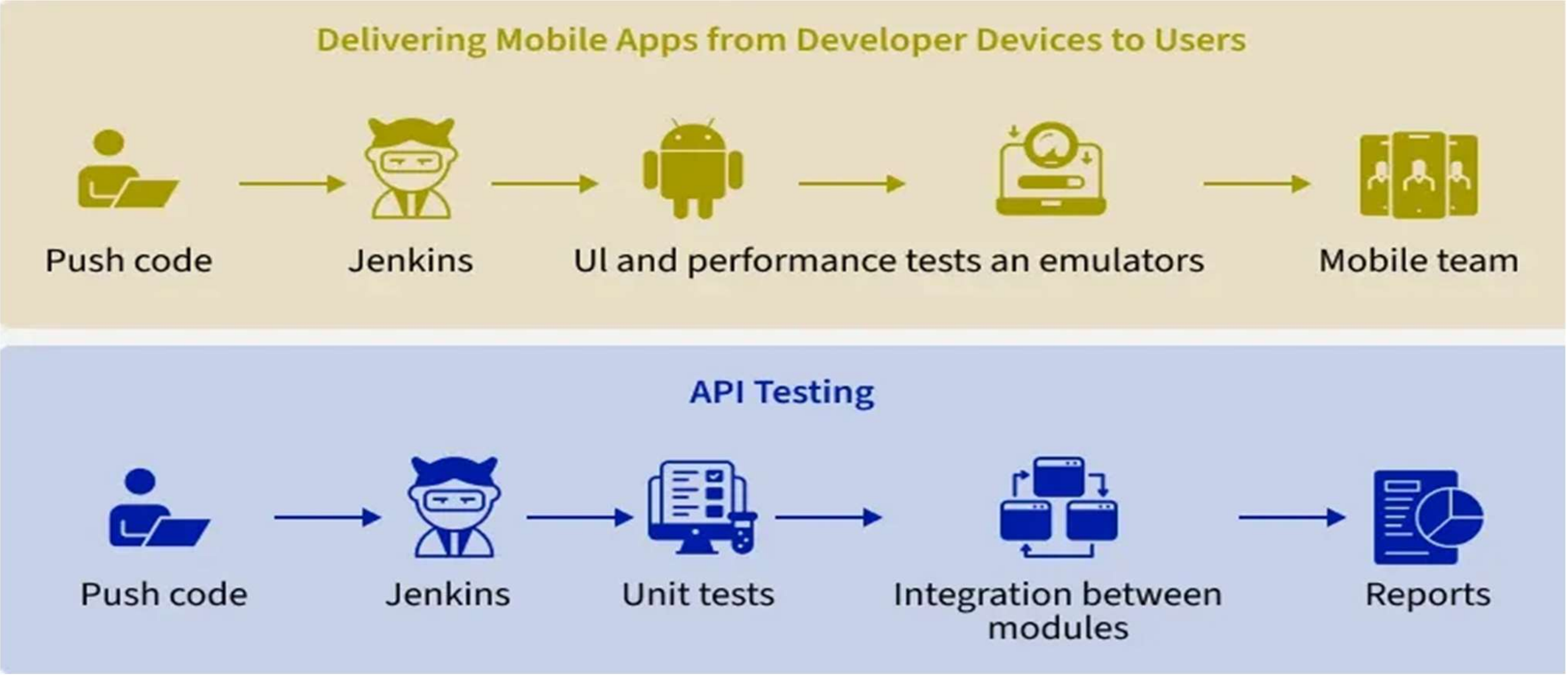
- Signing/Security: Builds undergo code signing, vulnerability scans, and manual approvals to enforce governance and security policies.

- Deploy: Jenkins deploys validated code to testing, staging, or production using containers or cloud platforms like AWS, Azure, Kubernetes, and Docker.

- Inform: Teams get automated email updates and dashboards showing pipeline status, logs, and reports.

Pipeline examples for web apps, mobile apps, and API testing

Jenkins pipelines automate continuously releasing high-quality app updates by coordinating infrastructure. They enable engineer teams to deliver frequently.


Delivering Mobile Apps from Developer Devices to Users
Push code
Jenkins
UI and performance tests an emulators
Mobile team
API Testing
Push code
Jenkins
Unit tests
Integration between modules
Reports

Pipeline examples for web apps, mobile apps, and API testing

- First, deploying web apps to Kubernetes.
- When developers push code, Jenkins starts the pipeline.
- It builds a Docker container image with the code and sends it to a registry.
- Jenkins then updates Kubernetes config files with the new image details and tells Kubernetes to deploy the web app using this image.

Pipeline examples for web apps, mobile apps, and API testing

- Now delivering mobile apps from developer devices to users.
- The pipeline starts after code is added to the main codebase.
- It compiles Android and iOS app versions.
- Then it runs UI and performance tests on these builds in emulators.
- If tests pass, Jenkins submits the build for publishing in the Play Store or App Store and tells the mobile team.

Pipeline examples for web apps, mobile apps, and API testing

- Finally, API testing.
- Jenkins kicks off this pipeline when developers merge code.
- It runs unit tests on individual modules.
- Integration tests check modules work together correctly.
- Load tests hit the API server to measure performance.
- Jenkins shows reports on test results, code coverage, and issues.
- Based on results, it promotes the API to higher environments or warns developers of broken builds.

Creating Jenkins Jobs

- First, you'll want to log into your Jenkins dashboard.
- This is like entering a workshop full of tools to help build your software.
- Once inside, click "New Item" to start a new project.
- Give your job a nice name - maybe after your favorite movie character or snack food.

Creating Jenkins Jobs

- Then picking your job type.

- There are a few options here that do different things:

- Freestyle project - Lets you run custom commands and scripts. Like following a recipe step-by-step.

- Pipeline - For stacking tasks together into an automated workflow.

- Kind of like an assembly line!

- Multibranch Pipeline - When you have code in multiple branches, and want to build from each. Like building several models of a toy from different molds.

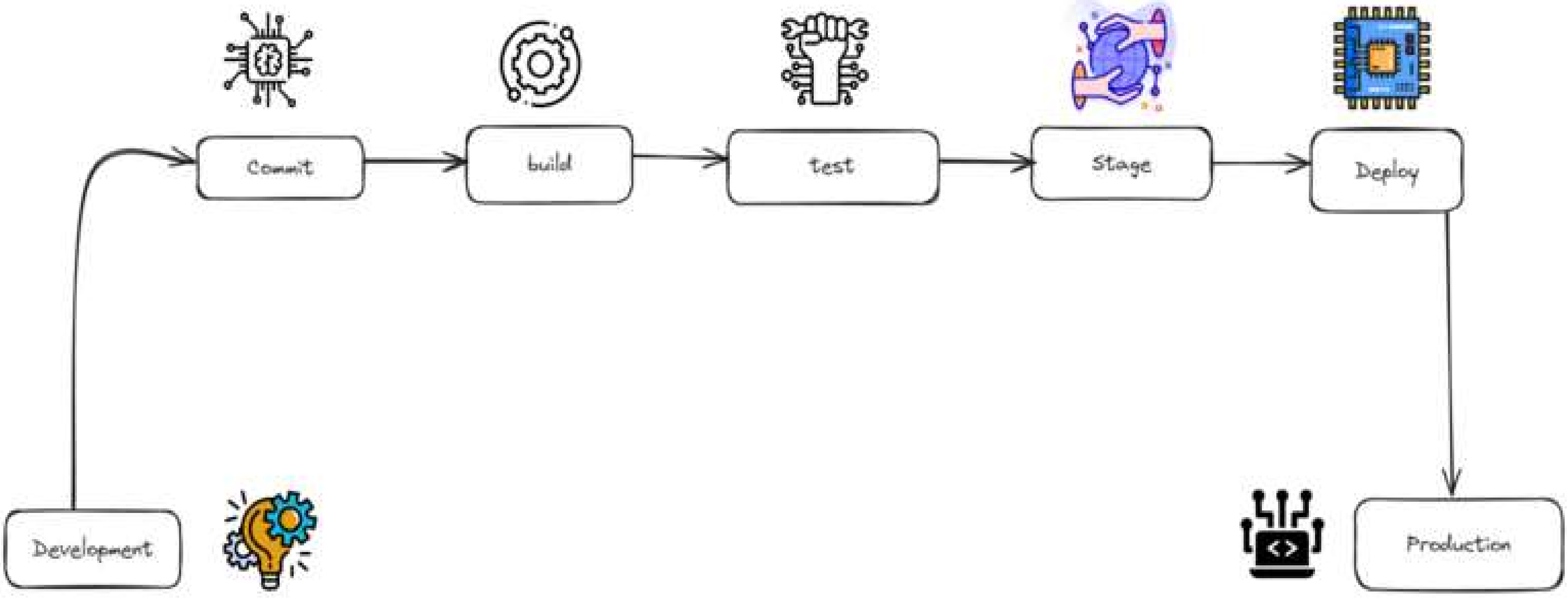
Creating Jenkins Jobs

- There are a few more types too, but these cover most use cases.
- Next, configure your job's settings.
- Here you can pick and choose what you want it to do - things like:
- Fetch code from version control
- Build and compile the code
- Run automated tests
- Deploy it somewhere after a successful build

Creating Jenkins Jobs

- The options are endless.
- Set up your job just as you need it.
- Finally, save your job and click "Build Now" to test it out.
- Watch your job execute each step and voila - you've successfully automated the build process!
- Now whenever new code lands, your job will wake up and do its work automatically.
- No more manual building or testing.
- With Jenkins, you can set up an assembly line for your software projects!

Creating Jenkins Jobs



CI-CD Pipeline in Jenkins

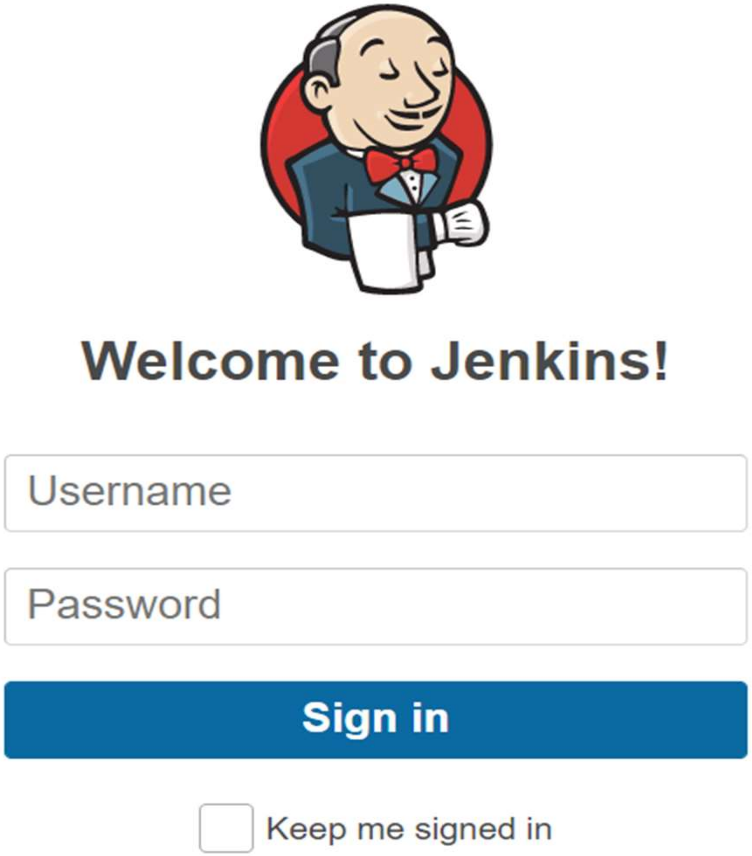
A CI/CD pipeline in Jenkins automates the process of building, testing, and deploying code. It helps developers integrate changes and deploy them quickly and reliably. Jenkins makes this automation easy and efficient.

Create CI-CD Pipeline in Jenkins

Here are the steps to create CI-CD Pipeline:

Step 1: Log In to Jenkins

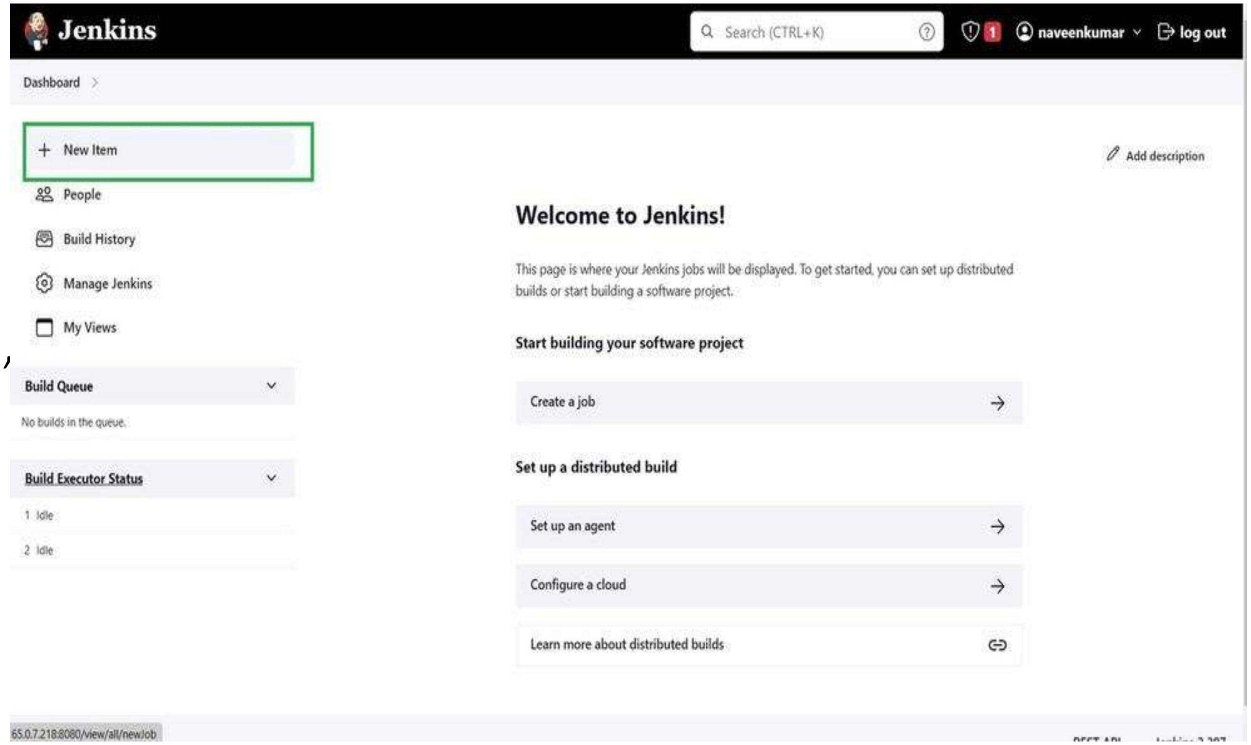
Log in to your Jenkins account. This is where all your Jenkins jobs and pipelines will be configured and monitored.



CI-CD Pipeline in Jenkins

Step 2: Redirected to the Jenkins Dashboard

When you logged in, you will be redirected to the Jenkins console or dashboard. This is where you can manage all your Jenkins jobs, pipelines, and settings.



No builds in the queue.

Welcome to Jenkins!

This page is where your Jenkins jobs will be displayed. To get started, you can set up distributed builds or start building a software project.

Start building your software project

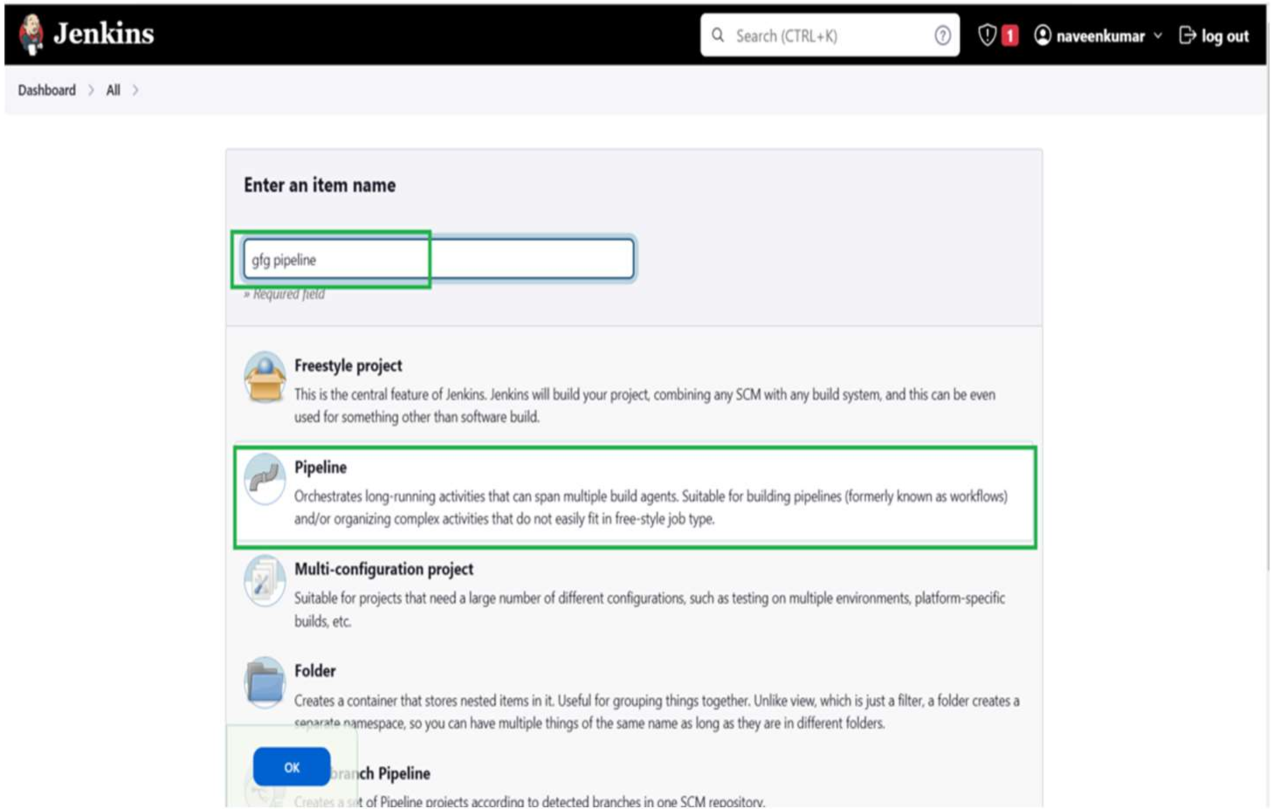
Set up a distributed build

Step 3: Create a New Project

On the Jenkins dashboard, click on the "New Item" option to create a new project select the option available in the Dashboard.

CI-CD Pipeline in Jenkins

Step 4: Configure the Project Type
On the next screen, list of options will be visible, along with a field to name the pipeline. Add a suitable name and select the "Pipeline" option to proceed.



Enter an item name

gfg pipeline

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

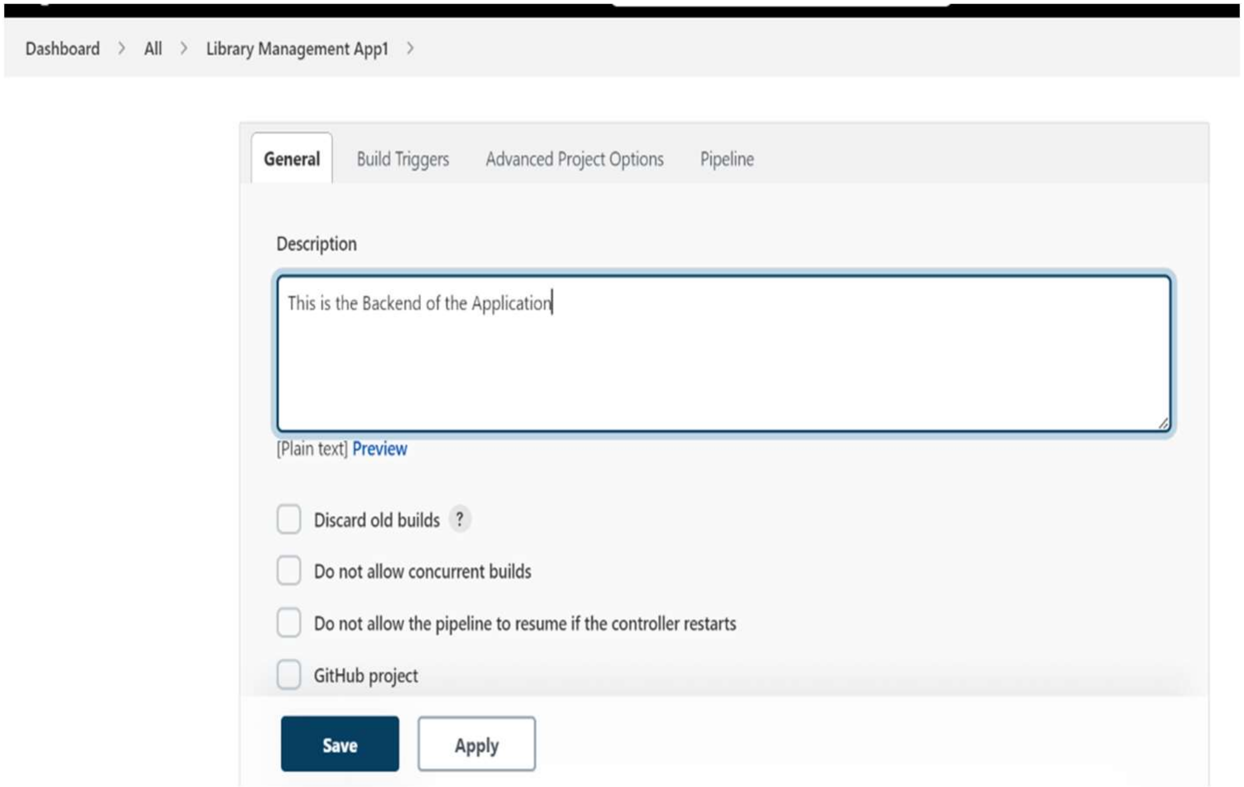
ranch Pipeline
Creates a set of Pipeline projects according to detected branches in one SCM repository.

CI-CD Pipeline in Jenkins

Step 5: Configure the General Section

The pipeline configuration page will open. In the General section:

- Provide a description of the pipeline, describing the purpose or what it will do.
- Establish the connection to the repository where Jenkins can access the project to trigger the pipeline.



Description

This is the Backend of the Application

□ Discard old builds

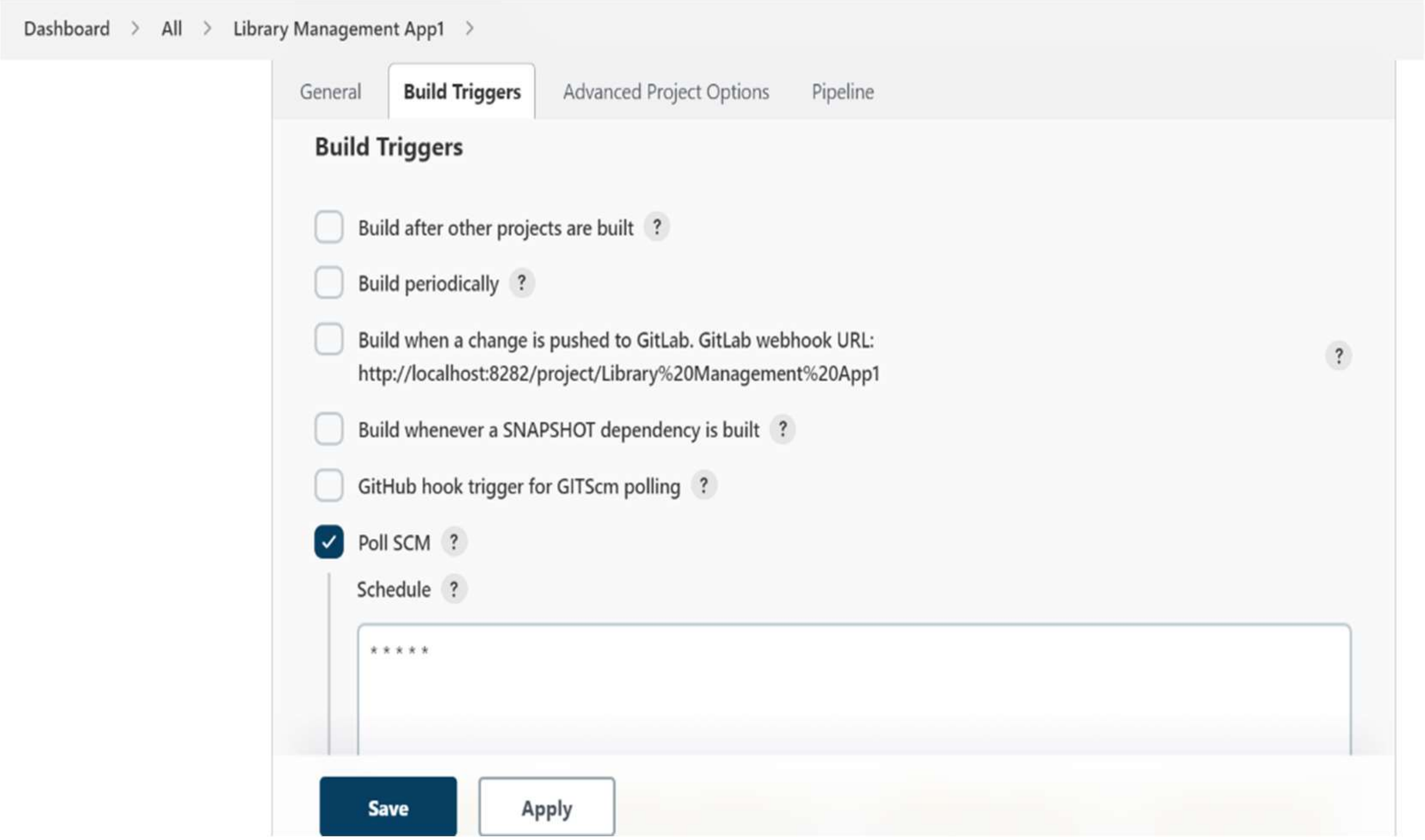
□ Do not allow concurrent builds

□ Do not allow the pipeline to resume if the controller restarts

□ GitHub project

CI-CD Pipeline in Jenkins

Step 6: Set Build Triggers
In the Build Triggers section:
We need to specify the branch and repository and give the credentials too. And add additional behaviors if required so far.



Build Triggers

□ Build after other projects are built ?

□ Build periodically ?

□ Build when a change is pushed to GitLab. GitLab webhook URL: http://localhost:8282/project/Library%20Management%20App1 ?

□ Build whenever a SNAPSHOT dependency is built ?

□ GitHub hook trigger for GITScm polling ?

☑ Poll SCM ?

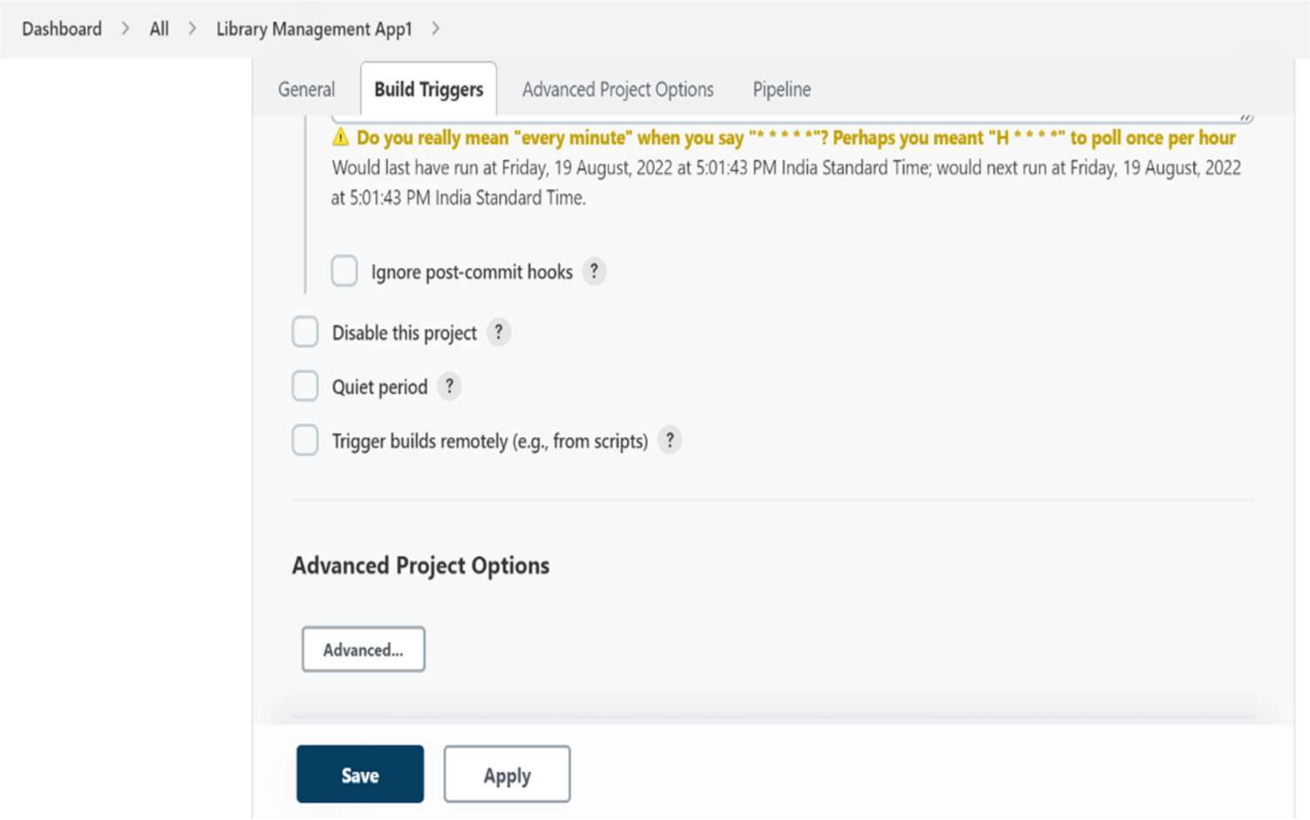
Schedule ?

```
* * * * *
```

CI-CD Pipeline in Jenkins

Step 7: Advanced Project Options
In the Advanced Project Options section:

- This section is primarily used for advanced configurations and special pipeline settings.
- For simpler projects, no configuration is required in this section, so you can leave it as is or adjust it based on your project's needs.



⚠ Do you really mean "every minute" when you say "* * * * *"? Perhaps you meant "H * * * *" to poll once per hour
Would last have run at Friday, 19 August, 2022 at 5:01:43 PM India Standard Time; would next run at Friday, 19 August, 2022 at 5:01:43 PM India Standard Time.

□ Ignore post-commit hooks ?

□ Disable this project ?

□ Quiet period ?

□ Trigger builds remotely (e.g., from scripts) ?

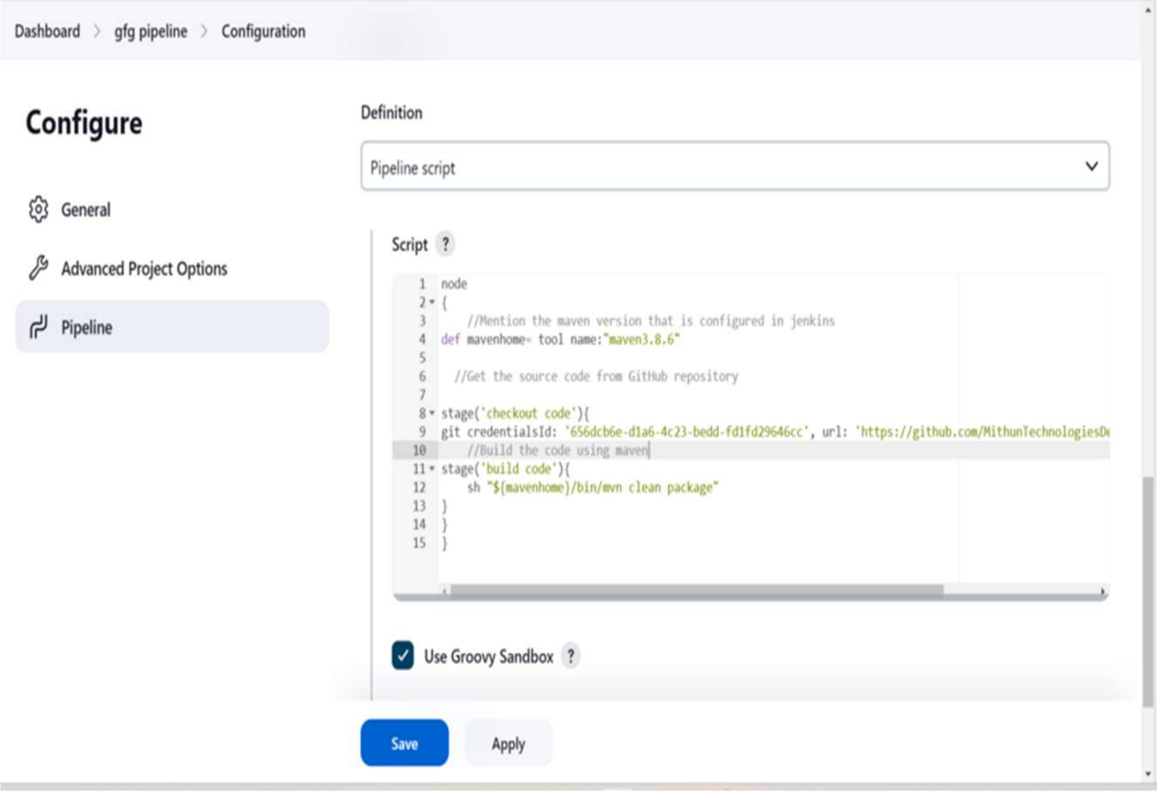
Advanced Project Options

CI-CD Pipeline in Jenkins

Step 8: Configure the Pipeline Section

This is the most critical part of the configuration process. In the Pipeline section:

- Define the Jenkinsfile that Jenkins will use to build and deploy your application. The Jenkinsfile contains the pipeline script with all the stages and steps for the CI/CD process.
- Specify where Jenkins should retrieve the pipeline script (either from the repository or from a file stored locally).
- Provide any required credentials or paths to the Jenkinsfile.



Definition: Pipeline script

Script ?

```
node
{
    //Mention the maven version that is configured in jenkins
def mavenhome= tool name:"maven3.8.6"

  //Get the source code from GitHub repository

stage('checkout code'){
git credentialsId: '656dcb6e-d1a6-4c23-bedd-fd1fd29646cc', url: 'https://github.com/MithunTechnologiesD
    //Build the code using maven
stage('build code'){
    sh "${mavenhome}/bin/mvn clean package"
}
}
}
```

☑ Use Groovy Sandbox ?

Sample Jenkins Pipeline Script

```
node {
 // Define the Maven tool used for the build process
def mavenHome = tool name: 'Maven 3.8.6'
// Configure build properties (e.g., log rotation and SCM polling)
properties([
 buildDiscarder(logRotator(artifactDaysToKeepStr: ‘’,
artifactNumToKeepStr: '5', daysToKeepStr: '', numToKeepStr: '5')),
 pipelineTriggers([pollSCM('* * * * *')])    ])
// Checkout code from the Git repository
stage('Checkout Code') {
            git branch: 'development', credentialsId: 'github-credentials-id’,
url: 'https://github.com/your-username/your-repository.git’
 }
```

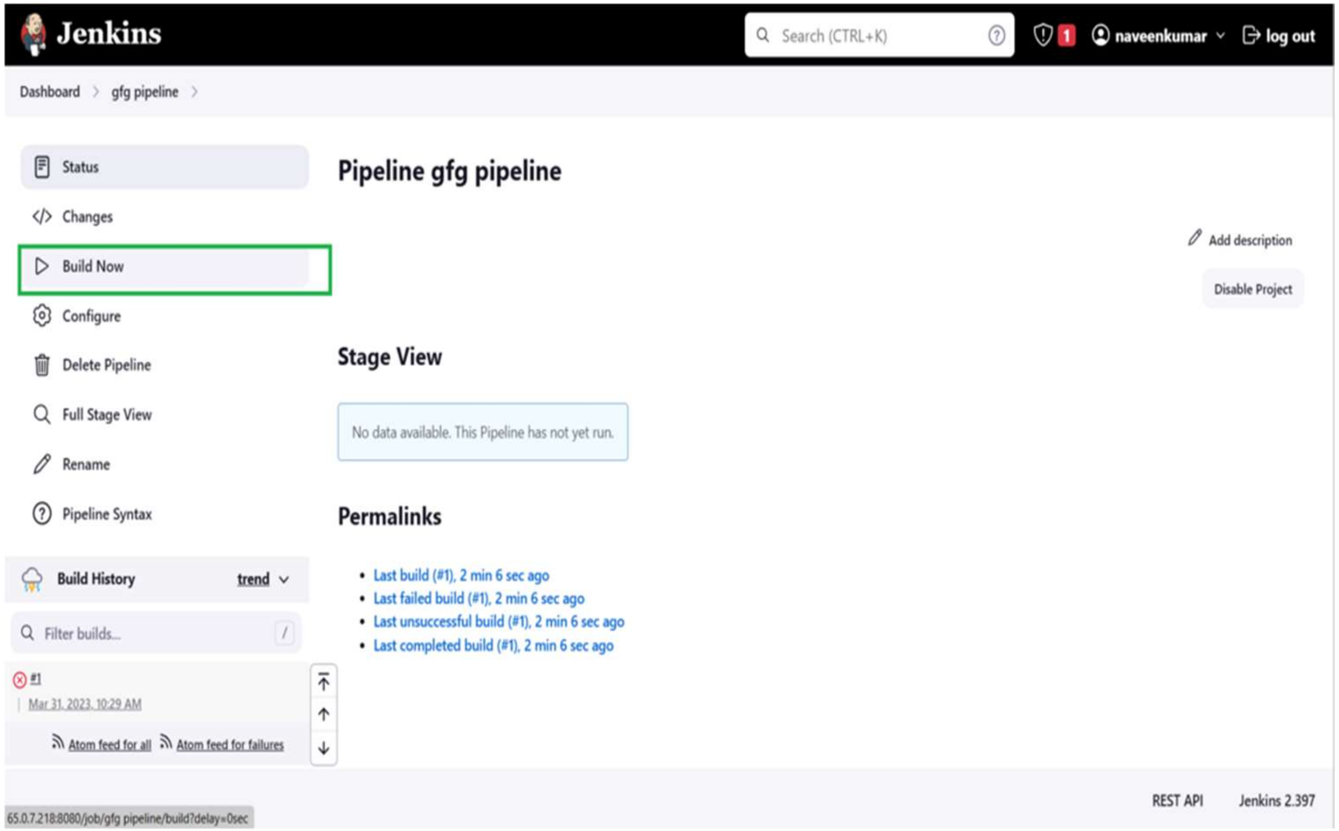
Sample Jenkins Pipeline Script

```
// Build the project using Maven
                stage('Build') {
                sh "${mavenHome}/bin/mvn clean package"
}
// Execute code quality analysis with SonarQube
                stage('Execute SonarQube Analysis') {
                sh "${mavenHome}/bin/mvn clean sonar:sonar"
}
 // Upload the build artifact to Nexus repository
                stage('Upload Build Artifact') {
                sh "${mavenHome}/bin/mvn clean deploy"
 }
 // Deploy the application to Tomcat using SCP
                stage('Deploy to Tomcat') {
                sshagent(['your-ssh-credentials-id']) {
          sh "scp -o StrictHostKeyChecking=no target/maven-web-application.war
                ec2-user@your-server:/opt/apache-tomcat-9.0.64/webapps/"
                                }
                }
}
```

CI-CD Pipeline in Jenkins

Step 9: Save the Pipeline and Run It

After writing the pipeline is done click on save it will be directly redirected to the Dashboard of the project there we can use, the "Build Now" option to run the pipeline and check if it is successful or not, by using stage view or console output.



Pipeline gfg pipeline

Stage View

No data available. This Pipeline has not yet run.

Permalinks

- Last build (#1), 2 min 6 sec ago
- Last failed build (#1), 2 min 6 sec ago
- Last unsuccessful build (#1), 2 min 6 sec ago
- Last completed build (#1), 2 min 6 sec ago

65.0.7.218:8080/job/gfg pipeline/build?delay=0sec

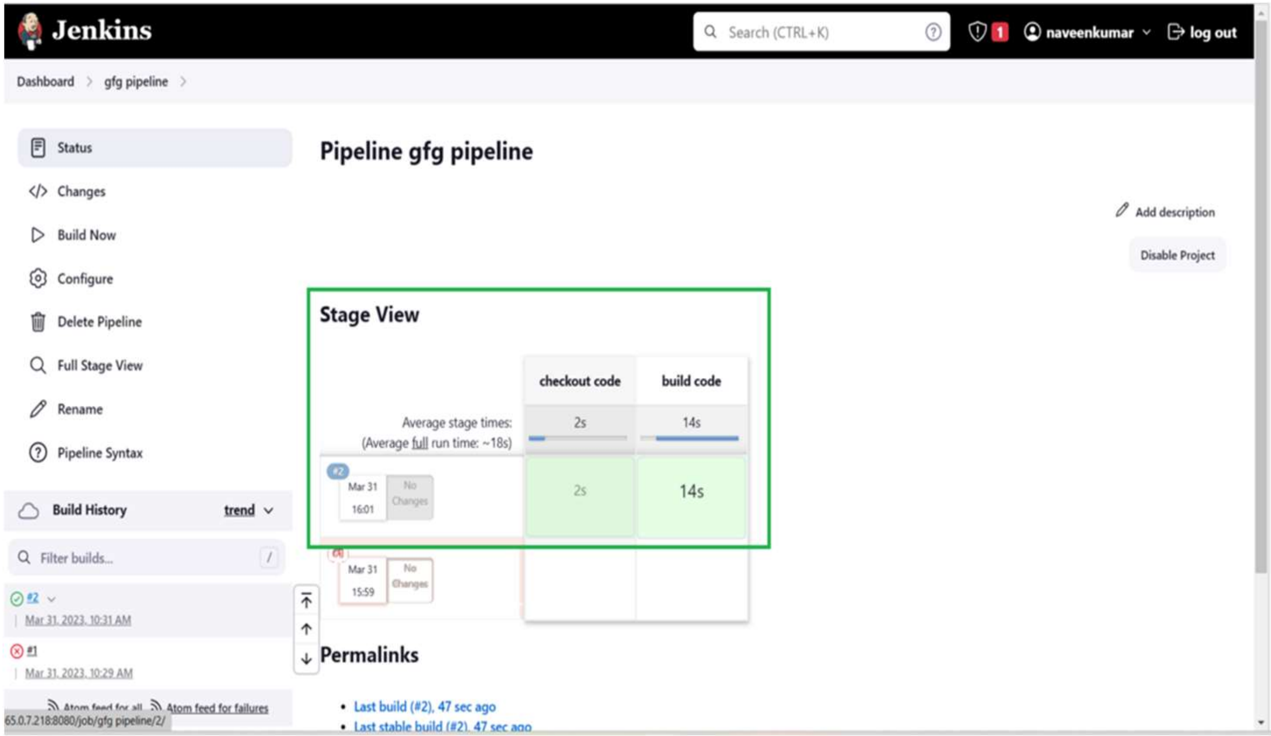
CI-CD Pipeline in Jenkins

Step 10: Monitor the Pipeline Execution

Once the pipeline is triggered, you can monitor its progress:

- Go to the Jenkins dashboard and click on your project.
- Under Build History, click on the build you just triggered.
- You can monitor the pipeline using the Stage View or Console Output.

Stage View: Provides a visual representation of the pipeline stages and their status (success, failure, etc.).



Pipeline gfg pipeline

Stage View

	checkout code	build code
Average stage times: (Average full run time: ~18s)	2s	14s
#2 Mar 31 16:01 No Changes	2s	14s
#1 Mar 31 15:59 No Changes		

Permalinks

- Last build (#2), 47 sec ago
- Last stable build (#2), 47 sec ago

Console Output: Displays the detailed logs of each stage. This is useful for debugging any issues that may occur during the pipeline execution.

CI-CD Pipeline in Jenkins

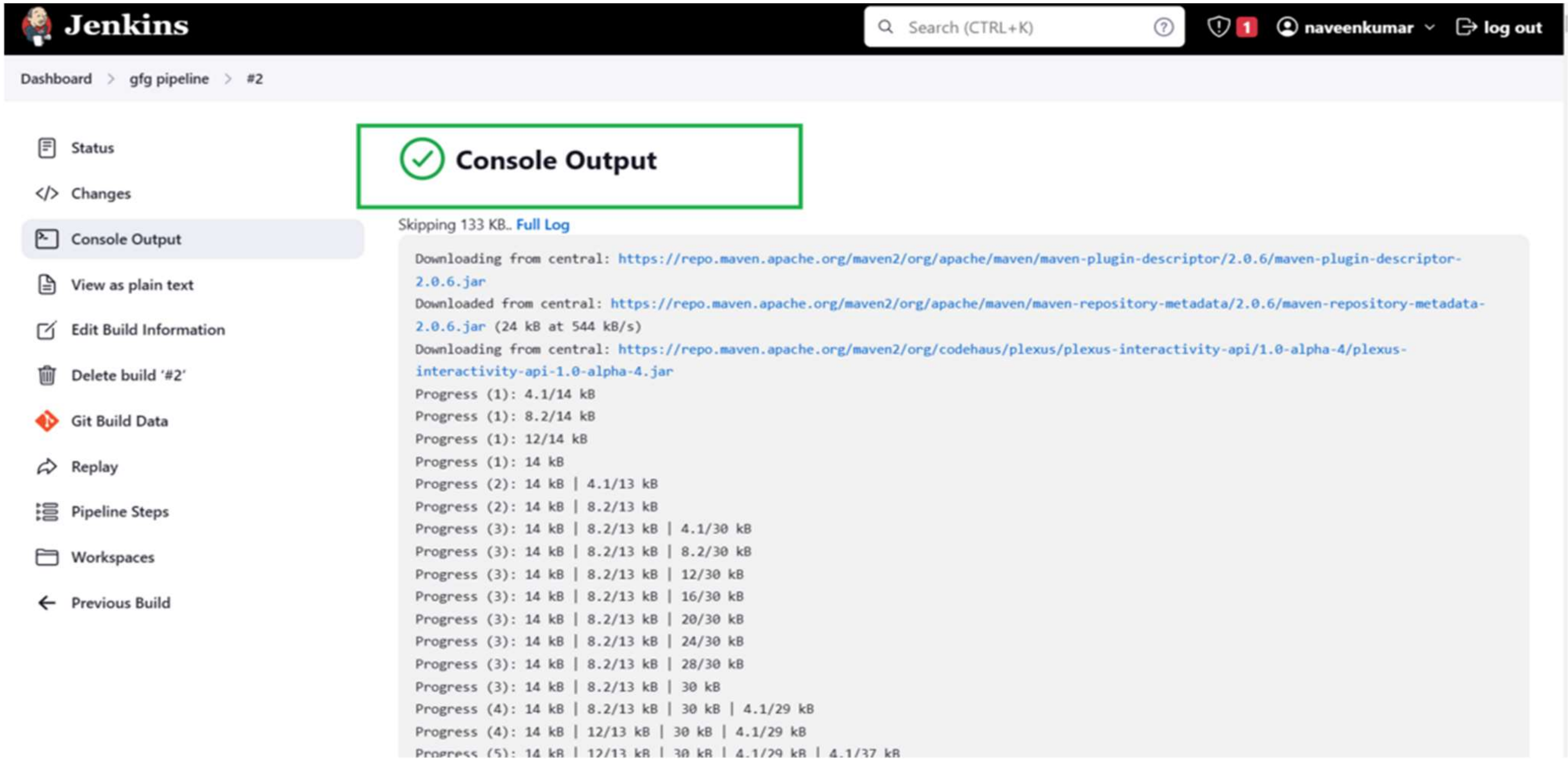
Step 11: Review the Build Status

At the bottom of the Console Output or Stage View, you will see the status of the pipeline:

- **Success**: If all stages are completed without errors, you will see "Finished: Success.“

- **Failure**: If any stage fails, you will see error messages in the logs, which can help in debugging the issue.

CI-CD Pipeline in Jenkins

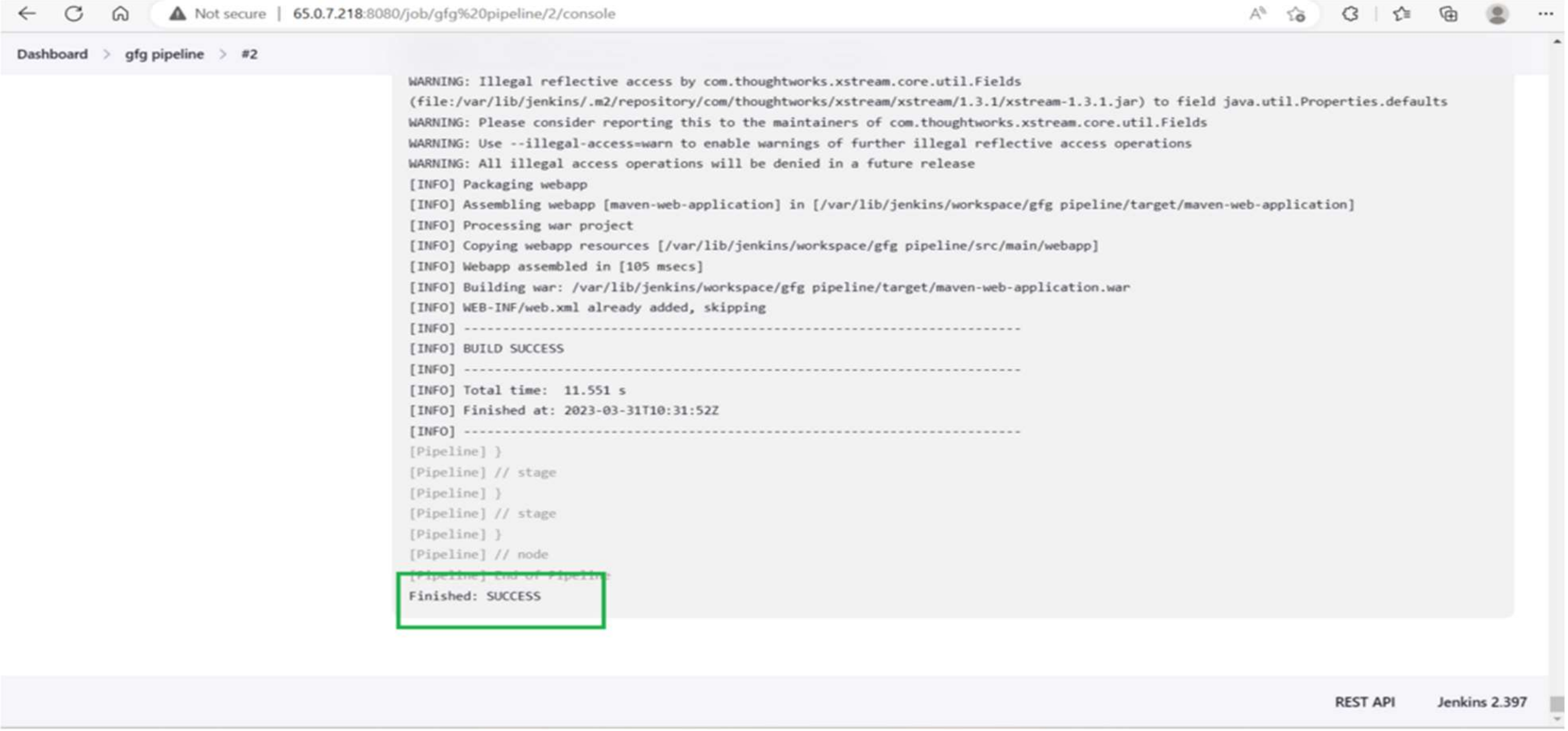


Console Output

Skipping 133 KB.. Full Log

```
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-plugin-descriptor/2.0.6/maven-plugin-descriptor-
2.0.6.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-repository-metadata/2.0.6/maven-repository-metadata-
2.0.6.jar (24 kB at 544 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-interactivity-api/1.0-alpha-4/plexus-
interactivity-api-1.0-alpha-4.jar
Progress (1): 4.1/14 kB
Progress (1): 8.2/14 kB
Progress (1): 12/14 kB
Progress (1): 14 kB
Progress (2): 14 kB | 4.1/13 kB
Progress (2): 14 kB | 8.2/13 kB
Progress (3): 14 kB | 8.2/13 kB | 4.1/30 kB
Progress (3): 14 kB | 8.2/13 kB | 8.2/30 kB
Progress (3): 14 kB | 8.2/13 kB | 12/30 kB
Progress (3): 14 kB | 8.2/13 kB | 16/30 kB
Progress (3): 14 kB | 8.2/13 kB | 20/30 kB
Progress (3): 14 kB | 8.2/13 kB | 24/30 kB
Progress (3): 14 kB | 8.2/13 kB | 28/30 kB
Progress (3): 14 kB | 8.2/13 kB | 30 kB
Progress (4): 14 kB | 8.2/13 kB | 30 kB | 4.1/29 kB
Progress (4): 14 kB | 12/13 kB | 30 kB | 4.1/29 kB
Progress (5): 14 kB | 12/13 kB | 30 kB | 4.1/29 kB | 4.1/37 kB
```

CI-CD Pipeline in Jenkins


Not secure | 65.0.7.218:8080/job/gfg%20pipeline/2/console
Dashboard > gfg pipeline > #2
REST API
Jenkins 2.397

```
WARNING: Illegal reflective access by com.thoughtworks.xstream.core.util.Fields
(file:/var/lib/jenkins/.m2/repository/com/thoughtworks/xstream/xstream/1.3.1/xstream-1.3.1.jar) to field java.util.Properties.defaults
WARNING: Please consider reporting this to the maintainers of com.thoughtworks.xstream.core.util.Fields
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
[INFO] Packaging webapp
[INFO] Assembling webapp [maven-web-application] in [/var/lib/jenkins/workspace/gfg pipeline/target/maven-web-application]
[INFO] Processing war project
[INFO] Copying webapp resources [/var/lib/jenkins/workspace/gfg pipeline/src/main/webapp]
[INFO] Webapp assembled in [105 msecs]
[INFO] Building war: /var/lib/jenkins/workspace/gfg pipeline/target/maven-web-application.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  11.551 s
[INFO] Finished at: 2023-03-31T10:31:52Z
[INFO] ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Here we successfully created a Jenkins CI/CD pipeline to automate the process of building, testing, and deploying a Java web application .war file.

Stages in Jenkins pipelines

```
pipeline {
agent any
stages {
stage('Checkout') {
steps {
git 'https://github.com/org/repo'
}
}
stage('Build') {
steps {
// compile code
// save build artifacts to S3 bucket
}
}
stage('Test') {
steps {
// download artifacts from S3
// load test data into Postgres DB
// run integration tests
// continue only if all tests pass
}
}
stage('Deploy') {
steps {
// download artifacts from S3
// deploy to Kubernetes cluster
}
}
}
}
```

Explanation of each of the 4 stages in the pipeline:

- **Get latest code:** Uses the git step to check out code from a Git repository.

- Provides the URL to the repo as a parameter.

- Clones the latest commit to the local workspace.

- Ensures all subsequent steps have access to the freshest source code.

- **Build code**: Compiles the application source code into executables/packages.
- Saves the build artifacts like JARs, containers, etc., to cloud storage.
- Uses S3 or an equivalent object store for storing the outputs.
- Makes the build outputs available for later stages.

Explanation of each of the 4 stages in the pipeline:

- Test code: Fetches the build artifacts from the storage location.
- Loads test data and schemas into a Postgres database.
- Executes automated integration tests against the application.
- Continues only if all test cases pass as expected.
- Gates deployment on successfully passing tests.

Explanation of each of the 4 stages in the pipeline:

- Deploy code: Downloads the vetted build artifacts for deployment.

- Deploys the application to a Kubernetes cluster.

- Pushes the containers to the target runtime environment.

- Makes the built and tested application available to users.

- This pipeline has discrete steps for each major phase of the CD process - repo checkout, build, test, and deploy.

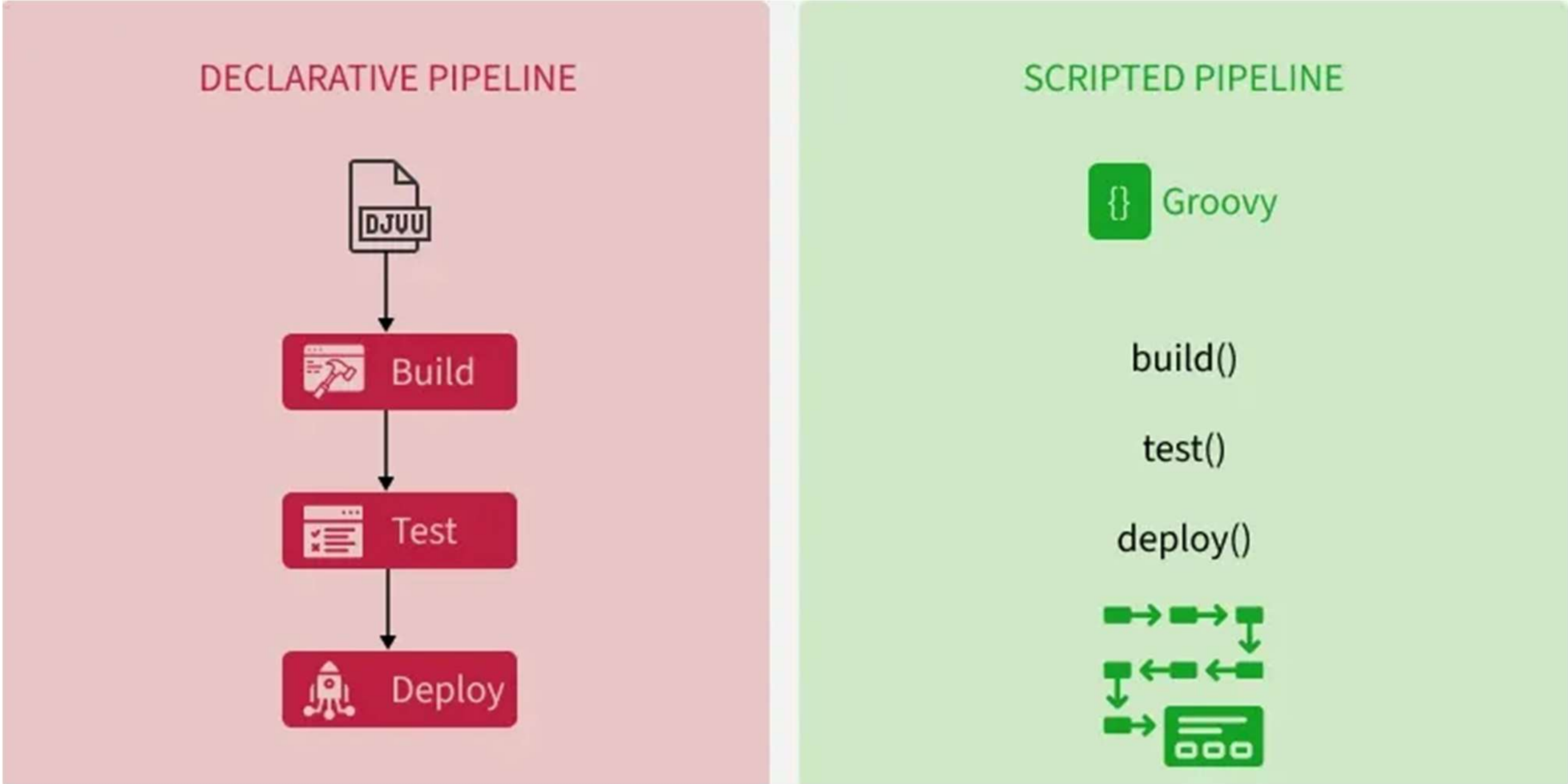
- Each stage focuses on one concern and uses standard tools.

- This modular design enables extensibility and maintainability.

Declarative Pipeline Vs Scripted Pipeline

- Using Jenkins for continuous integration and delivery, the concept of pipelines seemed confusing and complex.
- There were two primary approaches for defining and creating pipelines in Jenkins - declarative pipelines and scripted pipelines.
- Both approaches could accomplish the core goal of automating software workflows, but in markedly different ways.

Declarative Pipeline Vs Scripted Pipeline



Declarative Pipeline Vs Scripted Pipeline

- Declarative pipelines take a more structured, easy-to-visualize approach.

- The Jenkinsfile format allows you to lay out pipeline stages, such as build, test, and deploy, in a linear, sequential order.

- Each stage has a clean block defining what steps should occur within that stage.

- This maps very cleanly to the progression of taking code from version control, through build, validation, and release processes.

- We can look at the Jenkinsfile like a flowchart and understand the logical order of events.

Declarative Pipeline Vs Scripted Pipeline

- Scripted pipelines offer much more flexibility and customization capability, at the cost of increased complexity.
- Steps are defined procedurally using Groovy code encapsulated within methods like build(), test(), etc.
- The logic flow is harder for me to follow, as I have to trace through the method calls to see the overall sequence.
- The highly customizable nature of scripted pipelines enables much more sophisticated orchestration, but requires more Groovy programming expertise.
- Declarative pipelines are best for simple, linear CI/CD workflows, while scripted pipelines offer greater flexibility and customization for complex scenarios.

Declarative Pipeline Code Syntax

```
pipeline {
        agent any
        stages {
        stage('Build') {
        steps {
        // build steps
        }
}
stage('Test') {
        steps {
        // test steps
        }
}
stage('Deploy') {
        steps {
        // deploy steps
             }
        }
    }
}
```

Scripted Pipeline Code Syntax

```
node {
        stage('Build') {
        // build steps
}

stage('Test') {
        // test steps
}

stage('Deploy') {
        // deploy steps
        }
}
```

Best practices of Jenkins

- Using Jenkins pipelines can really help software teams work better.
- Jenkins lets you automate a lot of manual work.
- But many teams just starting with Jenkins struggle.
- They don't always use best practices.
- Then their pipelines become messy, break often, and have security holes.
- Teams end up wasting time babysitting jobs or fixing issues.

Best practices of Jenkins

- Based on research and talks with engineers, how to use Jenkins right.
- We see ways to structure code for reliability and security.
- How to define infrastructure as code so it's reproducible.
- Plugins to use, scaling your setup, and baking security in from the start.
- Real examples from companies like Square, Twilio, and Adobe.
- These teams use Jenkins well to improve their development.

Best practices of Jenkins

- Jenkins is powerful but you need to be careful.
- Governance is important - you need a balance between giving developers freedom while maintaining control centrally.
- Mature teams strike this balance.
- The goal is to help teams use Jenkins to enhance their workflows, not hinder them.
- With the right practices, Jenkins can help deliver software faster and more reliably.

QUESTIONS AND DISCUSSION
THANK YOU