

# Advanced Continuous Integration



# Objective

---

- Automate building and testing code in a CI environment.
- Implement artifact management strategies for storing and deploying builds.
- Configure notifications and reporting for better visibility in CI pipelines.
- Understand the role of different environments (Beta, Gamma, Prod, etc.) in CI/CD.
- Apply CI optimization techniques for efficiency and scalability.





**Explain the importance  
of automated build and  
testing in CI pipelines**

# Importance of automated build & testing

---

## Importance of Automated Build:

1. Early Issue Detection
2. Efficiency and Speed
3. Consistency

## Importance of Automated Testing:

1. Quality Assurance
2. Efficiency and Speed
3. Comprehensive Coverage
4. Reduced Manual Effort
5. Improved Collaboration



# Pop Quiz

---

Q. What is the primary purpose of automated testing in CI/CD pipelines??

**A**

To reduce manual intervention  
and improve quality

**B**

To increase manual testing efforts

# Pop Quiz

---

Q. What is the primary purpose of automated testing in CI/CD pipelines??

**A**

To reduce manual intervention  
and improve quality

**B**

To increase manual testing efforts



**Demonstrate a sample  
CI pipeline running  
automated tests for a  
project**

# Let's do it

---

Here's a simplified example of a CI pipeline that runs automated tests for a project using Jenkins, a popular CI tool. This example assumes a Java project using Maven for build management and JUnit for unit testing.

Sample CI Pipeline:

1. Trigger
2. Build stage
  - Checkout Code
  - Build Project
3. Test stage
  - Run Unit Tests
  - Run Integration Tests





# Let's do it

---

## 4. Deployment Stage

- Deploy to Staging

## 5. Post-Deployment Stage

- Run End-to-End Tests

Example Jenkins file:

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        sh 'mvn clean compile'
      }
    }
    stage('Unit Tests') {
      steps {
        sh 'mvn test'
      }
    }
    stage('Integration Tests') {
      steps {
        sh 'mvn verify'
      }
    }
    stage('Deploy to Staging') {
      steps {
        sh 'mvn deploy'
      }
    }
    stage('End-to-End Tests') {
      steps {
        sh './run-e2e-tests.sh'
      }
    }
  }
}
```





# **Introducing test types used in CI**

# Test types in CI

---

In Continuous Integration (CI), several types of tests are used to ensure software quality and reliability. Here's a brief overview of the key test types:

## 1. Unit Tests:

- Purpose: Validate individual components or units of code.
- Scope: Focus on specific functions or methods within the codebase.
- Benefits: Fast execution, low cost, and easy to maintain.



# Test types in CI

## 2. Integration Tests:

- Purpose: Ensure that different components or modules work together seamlessly.
- Scope: Test interactions between multiple units or services.
- Benefits: Identify integration issues early, though they are more complex and slower than unit tests.

## 3. End-to-End Tests:

- Purpose: Simulate real user workflows to test the entire application.
- Scope: Cover complete user journeys from start to finish.
- Benefits: Validate overall system functionality, though they can be expensive and time-consuming.

# Pop Quiz

---

Q. What type of test simulates real user interactions to validate the entire application workflow?

**A**

Performance Test

**B**

End to end tests

# Pop Quiz

---

Q. What type of test simulates real user interactions to validate the entire application workflow?

**A**

Performance Test

**B**

End to end tests





# **The concept of artifacts in CI/CD**

# Artifacts in CI/CD

---

- In the context of Continuous Integration/Continuous Deployment (CI/CD), an artifact refers to a deployable package or file that is produced during the build process.
- This package typically contains the compiled code, libraries, and other necessary components required to deploy the software to a production environment.
- Artifacts are crucial in CI/CD pipelines as they are the tangible outputs that are tested, validated, and eventually deployed to users.
- Artifacts play a central role in streamlining the software delivery process by providing a standardized and reliable way to package and deploy software changes.





# **The Popular Artifact Repositories**

# Artifact Repositories

Here are some popular artifact repositories:

## 1. Sonatype Nexus Repository:

- Purpose: Primarily used for storing and managing binary artifacts like Maven and npm packages.
- Features: Offers open-source and commercial versions, supports proxy repositories, and provides features like versioning and access control.

# Artifact Repositories

---

## 2. JFrog Artifactory:

- Purpose: Designed to manage a wide range of package formats, including binaries and container images.
- Features: Supports multi-format repositories, fine-grained access control, and extensive lifecycle management.

## 3. Docker Hub:

- Purpose: Specialized for storing and managing Docker container images.
- Features: Provides a centralized location for Docker images, allowing easy sharing and deployment across environments.



**Demonstrating  
publishing an artifact as  
part of a CI pipeline**

# Let's do it

---

Publishing an artifact as part of a CI pipeline involves the following steps:

1. Identify Artifact Path: Determine the directory or files you want to publish as artifacts.
2. Use YAML to Publish Artifacts: In Azure Pipelines, use the 'PublishBuildArtifacts' or 'PublishPipelineArtifact' task in your YAML file.
3. Example YAML Snippet

```
- task: PublishBuildArtifacts@1
  displayName: 'Publish build artifact'
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: 'drop'
```

1. Publish Across Stages: If needed, you can publish artifacts in one stage and then load them in another using stage dependencies.

# Take A 5-Minute Break!



- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes





# **The importance of notifications in CI pipelines**

# Notifications in CI pipelines

---

Notifications in Continuous Integration (CI) pipelines are crucial for maintaining efficiency and reliability.

Here's why they are important:

- Immediate Feedback:
- Proactive Issue Resolution
- Collaboration and Accountability
- Customization and Flexibility





# **Introducing different notification channels**

# Notifications Channels

---

Here's a brief introduction to different notification channels:

## 1. Email:

- Purpose: Send notifications via email to individuals or groups.
- Benefits: Wide reach, easy to set up, and suitable for asynchronous communication.

## 2. Slack:

- Purpose: Deliver notifications to specific Slack channels for real-time team updates.
- Benefits: Enhances team collaboration, allows for immediate feedback, and integrates well with CI/CD tools.

# Notifications Channels

---

## 3. Teams:

- Purpose: Similar to Slack, sends notifications to Microsoft Teams channels.
- Benefits: Integrates with Microsoft ecosystem, supports collaboration, and provides real-time updates.

## 4. Webhooks:

- Purpose: Send notifications to custom endpoints or services for further processing.
- Benefits: Highly customizable, allows integration with custom applications or services, and supports automation workflows.



**Demonstrating  
configuring a pipeline to  
send notifications on  
build success/failure**

# Let's do it

---

Here's a brief demonstration of configuring a pipeline to send notifications on build success or failure using Jenkins as an example:

## 1. Install Notification Plugins:

- Install plugins like Slack, HipChat, or Email Extension in Jenkins to enable notifications.

## 2. Define Notification Steps:

- In your Jenkinsfile, define functions for sending notifications on build start, success, or failure.

# Let's do it

---

## 3. Example Notification Functions:

```
def notifyStarted() {
    slackSend (color: '#FFFF00', message: "STARTED: Job '${env.JOB_NAME}'
[${env.BUILD_NUMBER}]' (${env.BUILD_URL})")
}

def notifySuccessful() {
    slackSend (color: '#00FF00', message: "SUCCESSFUL: Job
 '${env.JOB_NAME}' [${env.BUILD_NUMBER}]' (${env.BUILD_URL})")
}

def notifyFailed() {
    slackSend (color: '#FF0000', message: "FAILED: Job '${env.JOB_NAME}'
[${env.BUILD_NUMBER}]' (${env.BUILD_URL})")
}
```

# Let's do it

---

## 4. Integrate Notifications into Pipeline:

- Call these functions at appropriate stages in your pipeline.

```
try {  
    notifyStarted()  
    // Build steps here  
    notifySuccessful()  
} catch (e) {  
    currentBuild.result = "FAILED"  
    notifyFailed()  
    throw e  
}
```

- This setup allows you to receive notifications on build events, ensuring timely feedback and action.



# **The purpose of different environments in a software release cycle**



# Environments in a software release cycle

---

Here's a brief explanation of the purpose of different environments in a software release cycle:

## 1. Test Suite (Automated Tests):

- Purpose: Run automated tests to validate software functionality and catch bugs early.
- Benefits: Ensures quality and reliability before moving to further environments.

## 2. OneBox (Isolated Dev Testing):

- Purpose: Provides developers with an isolated environment for testing and debugging their code changes.
- Benefits: Allows for safe experimentation without affecting other environments.

# Environments in a software release cycle

---

Here's a brief explanation of the purpose of different environments in a software release cycle:

## 1. Test Suite (Automated Tests):

- Purpose: Run automated tests to validate software functionality and catch bugs early.
- Benefits: Ensures quality and reliability before moving to further environments.

## 2. OneBox (Isolated Dev Testing):

- Purpose: Provides developers with an isolated environment for testing and debugging their code changes.
- Benefits: Allows for safe experimentation without affecting other environments.

# Environments in a software release cycle

---

## 3. Beta/Gamma (Staging Environments):

- Purpose: Simulate production conditions to test software with real-world scenarios.
- Benefits: Identifies issues before deployment to production, ensuring a smoother release.

## 4. Production (Live Deployment):

- Purpose: The final environment where software is deployed for end-users.
- Benefits: Ensures that only thoroughly tested and validated software reaches users, maintaining reliability and user satisfaction.



**How CI/CD pipelines  
promote builds across  
these environments**

# Let's Discuss

---

CI/CD pipelines promote builds across different environments by automating the process of building, testing, and deploying software. Here's how they facilitate this across various environments:

## 1. Test Suite (Automated Tests):

- Role in CI/CD: Pipelines automate these tests, ensuring that code changes are validated before moving to further environments.

## 2. OneBox (Isolated Dev Testing):

- Role in CI/CD: Pipelines can deploy builds to OneBox environments for initial testing, allowing developers to validate changes in isolation.

# Let's Discuss

---

## 3. Beta/Gamma (Staging Environments):

- Role in CI/CD: Pipelines deploy builds to staging environments for thorough testing, ensuring that software works as expected before production.

## 4. Production (Live Deployment):

- Role in CI/CD: Pipelines automate the deployment of validated builds to production, ensuring that only thoroughly tested software reaches users.



**Discussing common  
bottlenecks in CI and  
how to address them**

# Let's Discuss

---

Here's a brief discussion on common bottlenecks in Continuous Integration (CI) pipelines and how to address them:

## 1. Reducing Build Times with Caching:

- Bottleneck: Long build times due to repeated compilation of unchanged code.
- Solution: Implement caching mechanisms to store compiled artifacts, reducing the need for redundant compilation.



# Let's Discuss

---

## 2. Parallelizing Test Execution:

- Bottleneck: Slow test execution due to sequential testing.
- Solution: Use parallel testing strategies to run multiple tests simultaneously, significantly speeding up the testing phase.

## 3. Using Lightweight Docker Containers:

- Bottleneck: Heavy resource usage by large Docker images.
- Solution: Optimize Docker images to be lightweight, reducing startup times and resource consumption, which improves overall pipeline efficiency.



**Explain best practices  
for scalable CI/CD**

# Best Practices

---

Here are some best practices for scalable CI/CD pipelines:

## 1. Keeping Pipelines Modular:

- Purpose: Break down pipelines into smaller, reusable components.
- Benefits: Enhances maintainability, reduces duplication, and allows for easier updates across multiple projects.

# Best Practices

---

## 2. Implementing Automated Rollbacks:

- Purpose: Automatically revert to previous stable versions if issues arise.
- Benefits: Minimizes downtime and quickly recovers from deployment failures, ensuring system reliability.

## 3. Regularly Reviewing and Optimizing Build Steps:

- Purpose: Continuously monitor and refine pipeline stages to eliminate bottlenecks.
- Benefits: Improves efficiency, reduces build times, and ensures pipelines remain aligned with evolving project needs.



# Time for Case Study

# Important

---

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks



SKILLS

!

