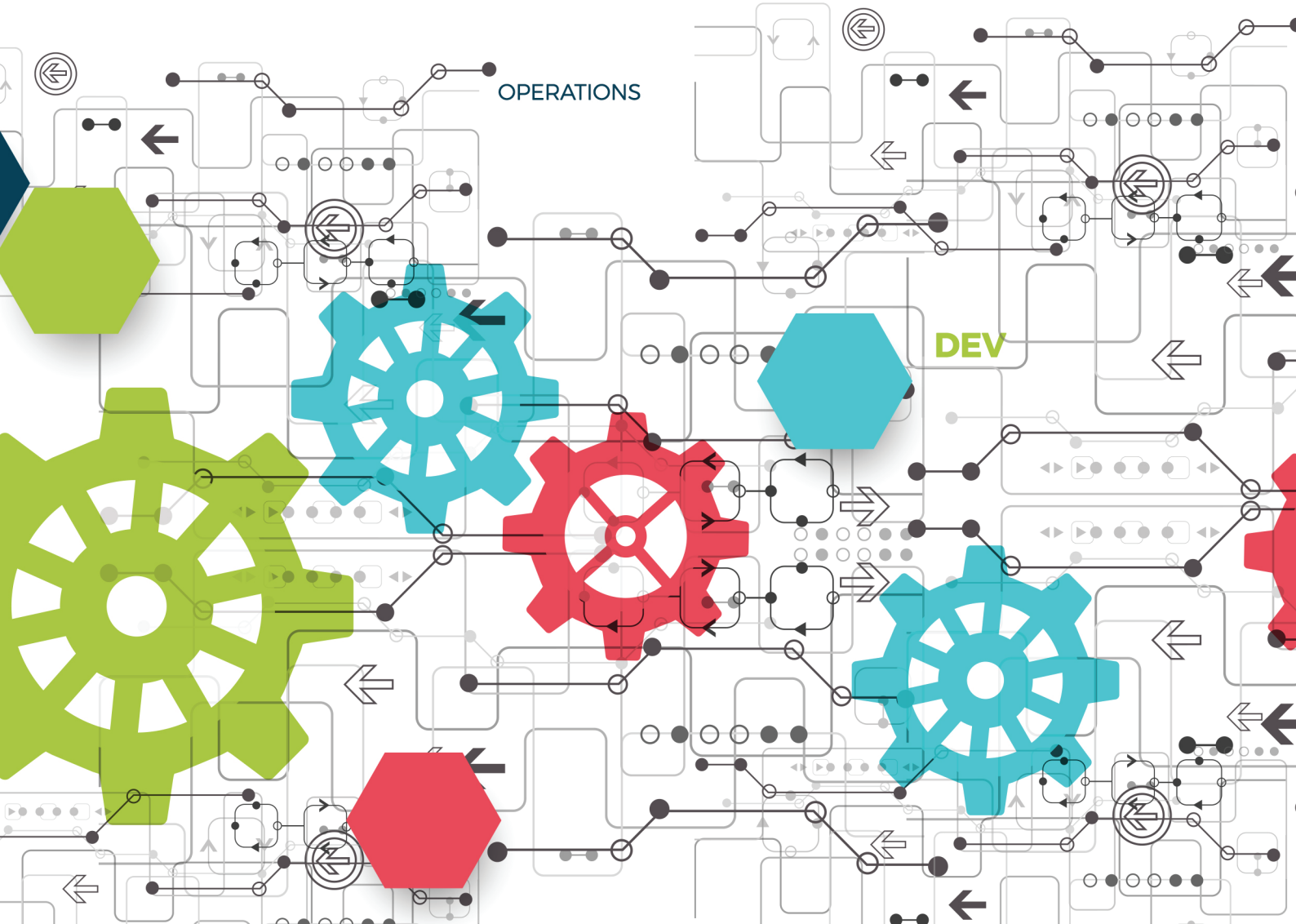
B.Tech Computer Science and Engineering in DevOps

Continuous Integration and Continuous Delivery

MODULE 2

Stages of Continuous Integration and Continuous Delivery



Contents

MODULE 2

Stages of Continuous Integration and Continuous Delivery

Module Objectives

At the end of this module, you will be able to learn about:

Core CI Process

- Explain the concept of VCS
- Identify merging local changes to the integration branch of VCS tool
- Define automated code builds
- Explain static code analysis
- Discuss automated unit testing
- Define code coverage analysis
- State automated packaging of the build artifact

Advanced CI process

- Give an overview of provision and deploy to test environment
- Explain automated functional testing
- Learn how to publish report to the development team

Facilitator Notes:

Explain the module objectives to the participants.

Module Topics

Let us take a quick look at the topics that we will cover in this module:

Core CI Process

- What is VCS ?
- Merging Local Changes to the Integration Branch of VCS Tool
  - How to update fork and create pull request
  - Code review before merging the code
- Automated Code Builds
  - Automated code builds – Key metrics
- Static Code Analysis
  - Static code analysis – Sample snapshot
  - Static code analysis – Sample bug report
  - Static code analysis – OWASP Top 10
  - Static code analysis – SANS Top 25 sample report
- Automated Unit Testing
  - Automated Unit Testing - Junit

- Unit Testing Frameworks - TestNG
- Automated Unit Testing process

- Code Coverage Analysis
  - Code Coverage Methods – Condition Coverage
  - Code Coverage Methods – Line Coverage
  - Publishing Code Coverage reports to Jenkins
- Automated Packaging of the Build Artifact
  - Uploading build artifact to a repository

Advanced CI process

- Provision and deploy to test environment
- Automated Functional Testing
  - Automated Functional Testing - Example
- Publish Report to the Development Team

Google Canary release Case study

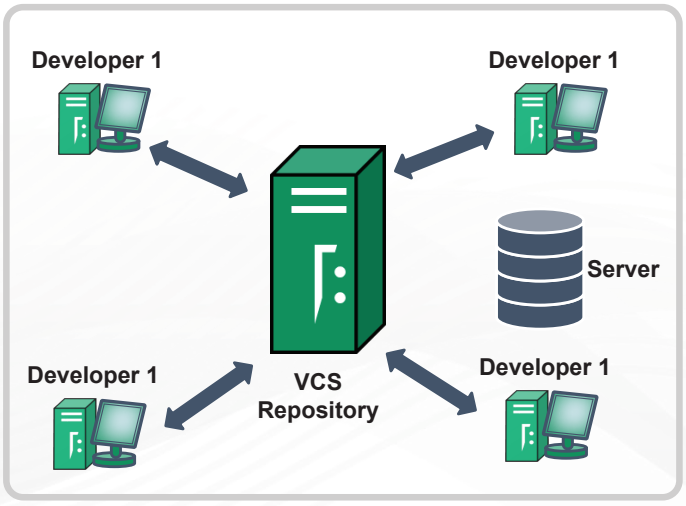
Facilitator Notes:

Inform the participants about the topics that they will be learning in this module.

2.1 What is VCS?

Following are the key features of Version Control System (VCS)

- VCS is easy to use.
- It provides CLI and GUI interface.
- Several developers can collaborate on a single project without losing integrity.
- Developers can continuously add, update, change, and delete new source code to VCS.
- VCS takes care of the version control issues.
- Multiple developers can work on the project without disturbing or waiting for others.



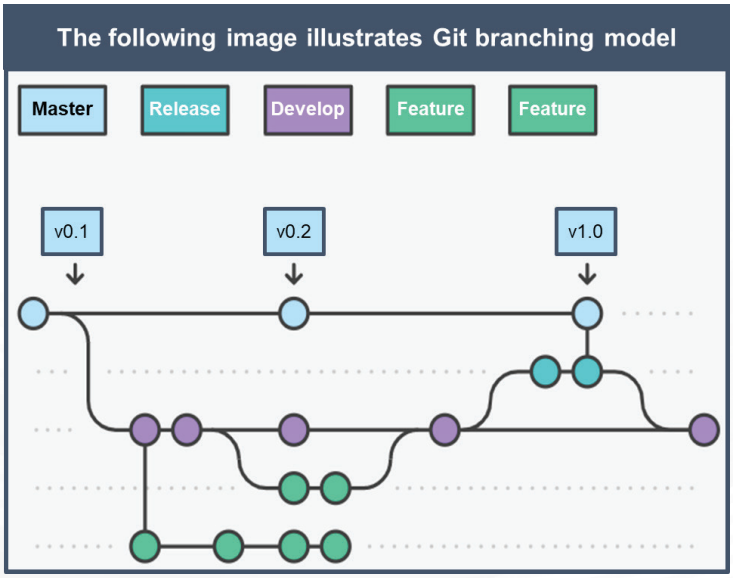
Facilitator Notes:

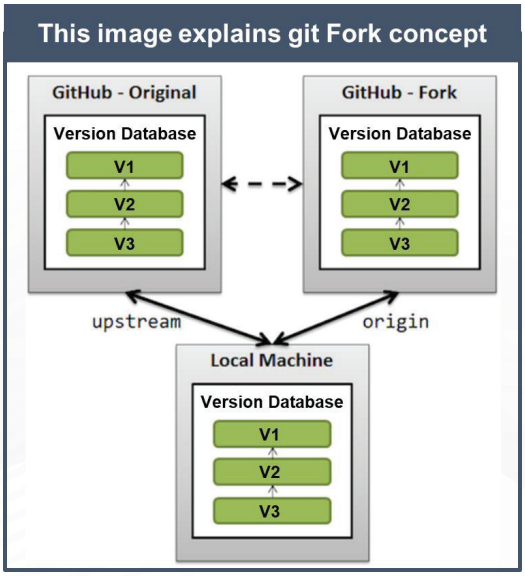
Give a brief to the participants about the VCS features.

The main feature of the VCS is to record the changes to a file over time. Examples of VCS is CVS, Subversion (Centralized VCS) and Git (Distributed).

Version control systems are useful for developers and designers because they allow you to save and store each distinct version of your files. This allows you to roll back to any of the previous versions when you run into a problem in the current version. Thus, no project is completely lost because of errors, forgetfulness, or a crash.

2.1.1 Merging Local Changes to the Integration Branch of VCS Tool





Facilitator Notes:

Explain the participants about the code merging process.

Branching in Git: Branches are used to develop new features isolate from each other. When you create a repository master branch is the “default” branch. You need to use other branches for developing new features and merge them back to the master branch upon completion of that feature.

A *fork* is a copy of the original repository. If you fork a repository all the branches (master, develop, release, feature etc) from main repository will be cloned to your fork copy. Forking the repository allows users to freely experiment their changes without affecting the original project.

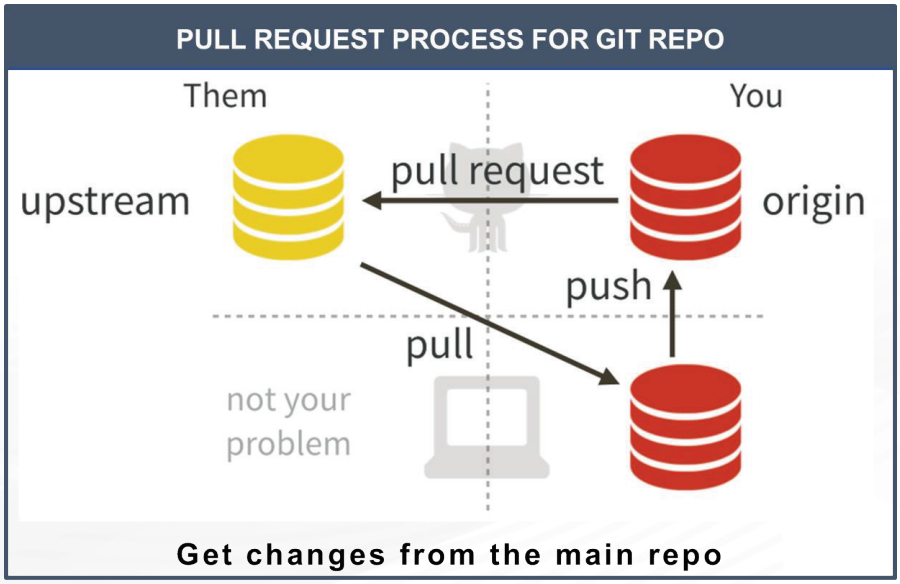
Code from Fork can be pushed to original branch using Pull requests.

One good example of using forks is to propose changes for bug fixes. Rather than logging an a bug for the issue found in the original repository:

- Fork the repository.
- Make the fix in your fork copy.
- Submit a *pull request* to the main repository.
- If the project owner agrees with your fix, they might pull your fix into the original repository.

Please note multiple users working on the same project need to update their Fork everyday in order to update their fork with other developer's changes. It is important to update the Forks before you start your work everyday.

2.1.2 How to Update Fork and Create Pull Request?



Facilitator Notes:

Explain the participants how to create a pull request from their respective forks.

Steps to update the Fork:

- Open `Git hub` URL of your project and click on Fork at the top right hand side
- Clone forked repository to your local machine. This becomes your origin

  `git clone –b <branch-name> https://github.com/mgna7/project.git`

- Set your upstream to main repository (We are considering upes as your organization name in this example)

  git remote add upstream `https://github.com/upes/project.git`

- `If you git remote –v` you can see both the origin and upstream in the log
- Everyday you need to run these three commands to update your fork:

  `git fetch upstream` (This command will pull all the latest code from upes git repo)

  `git merge upstream/<branch-name>` (This command will merge your local copy with latest code)

  `git push` (This command will commit the code to your fork)

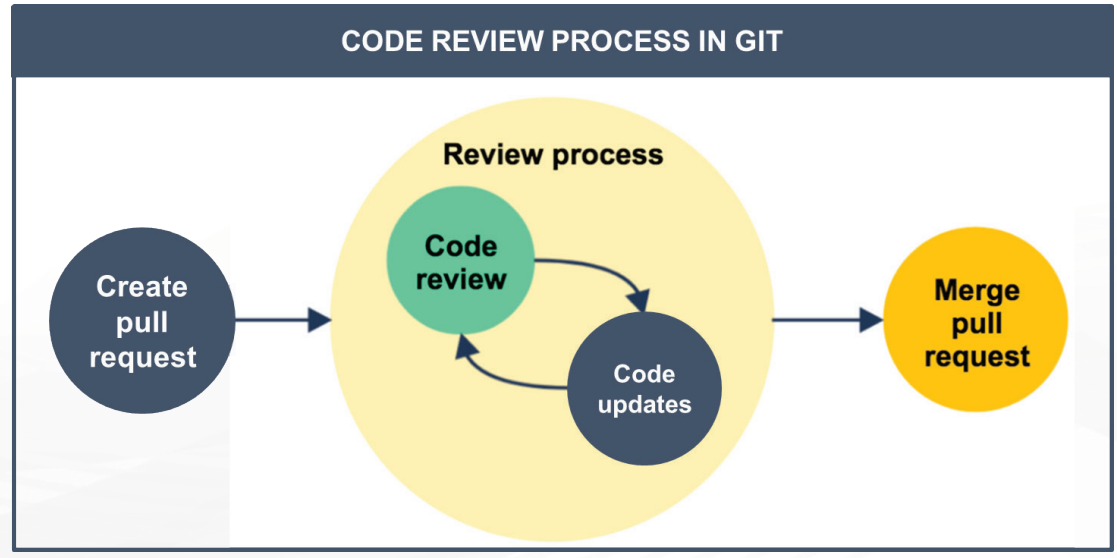
- If there are any code conflicts you need to handle it.

Steps to push your local changes to Fork:

- Please make sure your fork is up-to-date with the main repository. Run step5 from the above before starting your work.
- Make a change to a file
- git add `<filename>` or .
- `git commit –m “Committing to fork”`
- `git push`

6. Raise a pull request from your fork to main repo to push your changes. After code review it will be merged.

2.1.3 Code Review before Merging the Code



Facilitator Notes:

Explain the participants about code review and code merge process.

A code review process framework benefits the organization as follows:

- Collaboration between software development team members
- Identification and elimination of code defects before integration
- Improvement of code quality
- Quick turnaround of development cycle

A code review process framework benefits the organization as follows:

- Developers update the code on Local machine and push it to fork

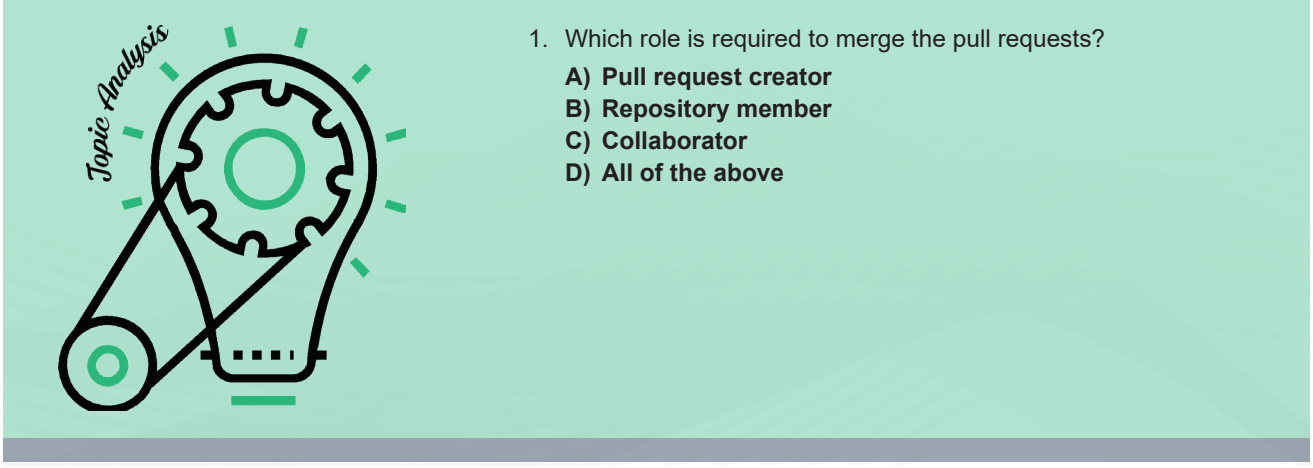
```
git add <filename>
```



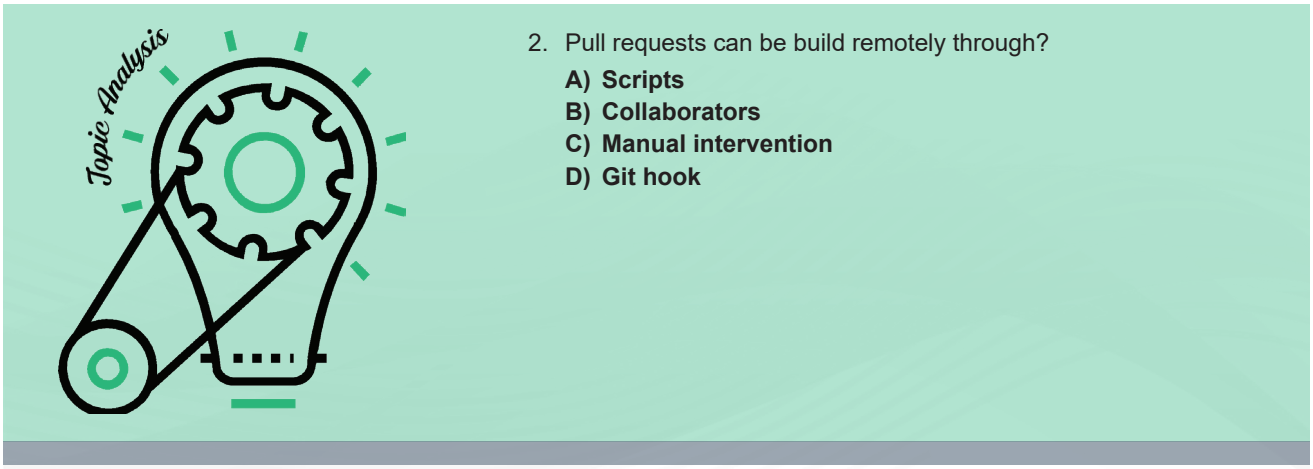
```
git commit –m “Message”

git push
```

- Fork gets updated with the developers commit
- The developer creates a pull request from his fork to the main branch
- Generally in big organization once a pull request is submitted, githook will trigger a build only for that Pull request
- If the build passes, reviewers will review the code and approve it. Reviewers can send back the pull request if code logic is not correct/any other reasons.
- Code is merged to the main branch and integration tests are executed.

What did You Grasp?



1. Which role is required to merge the pull requests?
   A) Pull request creator
   B) Repository member
   C) Collaborator
   D) All of the above



2. Pull requests can be build remotely through?
   A) Scripts
   B) Collaborators
   C) Manual intervention
   D) Git hook

Facilitator Notes:

Answer:

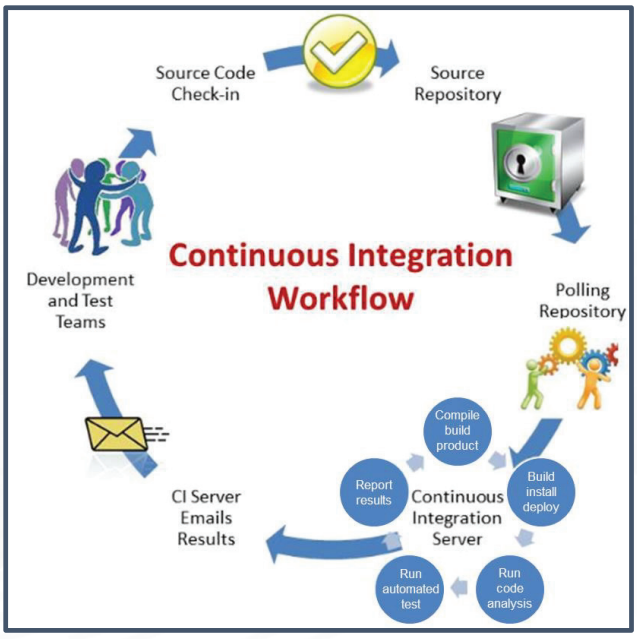
1. C .Collaborator

2. D. Git hooks

Through Git hooks a remote build can be triggered for every pull request.

2.2 Automated Code Builds

Automated Code Build Process	
Step 1	Write and Commit bits of Code
Step 2	Scan the Code for Problems
Step 3	Compile the Code
Step 4	Run Automated Tests
Step 5	Report Any Problems



Facilitator Notes:

Give the participants a brief information about automated code builds.

Automated build is the process of building and deploying the code and the database for a custom application based requirement:

Step 1: Write and Commit bits of Code

Let's say one of our programmers is busy designing your individualized application on his/her own computer. Red light! What happens if the system crashes? Vast amounts of work may be lost forever. So instead, we commit (or upload) the code to a central source code repository. Imagine this repository as a holding zone, a database for our code.

Step 2: Scan the Code for Problems

With the code in the repository, we then scan it for common problems. There are a variety of tools available for this, based on the programming language used, in our projects JSHint, FxCop and Code Duplication Checkers are some of the most commonly used. We like scanners because they can automatically monitor the code before we deploy it.

Step 3: Compile the Code

We compile the code after we've scanned it. In this step we change the programmer's commands into something that works on your desktop, mobile device or web browser. Here we also use scripts to create the database tables, test data and stored procedures.

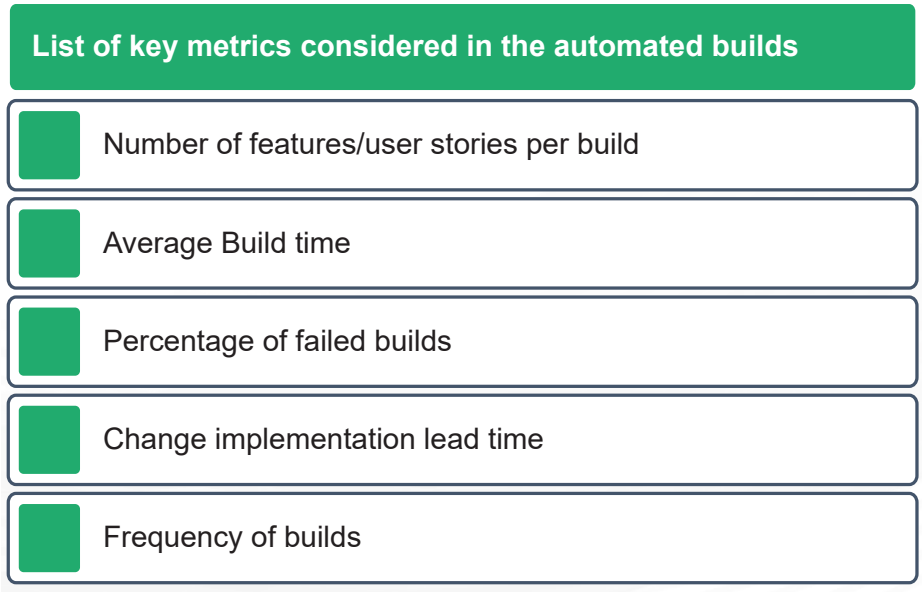
Step 4: Run Automated Tests

Finally, we're ready for the automated tests. For instance, if we're working with a website application, scripts load the browser automatically, land on a specific page, and test how things work on that page. Typically, there are hundreds of automated tests for a given application. The quality of the application is vastly improved because the automated tests run daily.

Step 5: Report Any Problems

When automated tests detect a problem, our team receives immediate notification in order to swiftly repair it. The result is essentially bug free code throughout the development process. When our manual testers discover new issues, we develop a new automated test expressly for that issue. Selenium is our preferred software testing framework for user interface automated tests.

2.2.1 Automated Code Builds – Key Metrics



Facilitator Notes:

Give the participants a brief about automated code build key metrics.

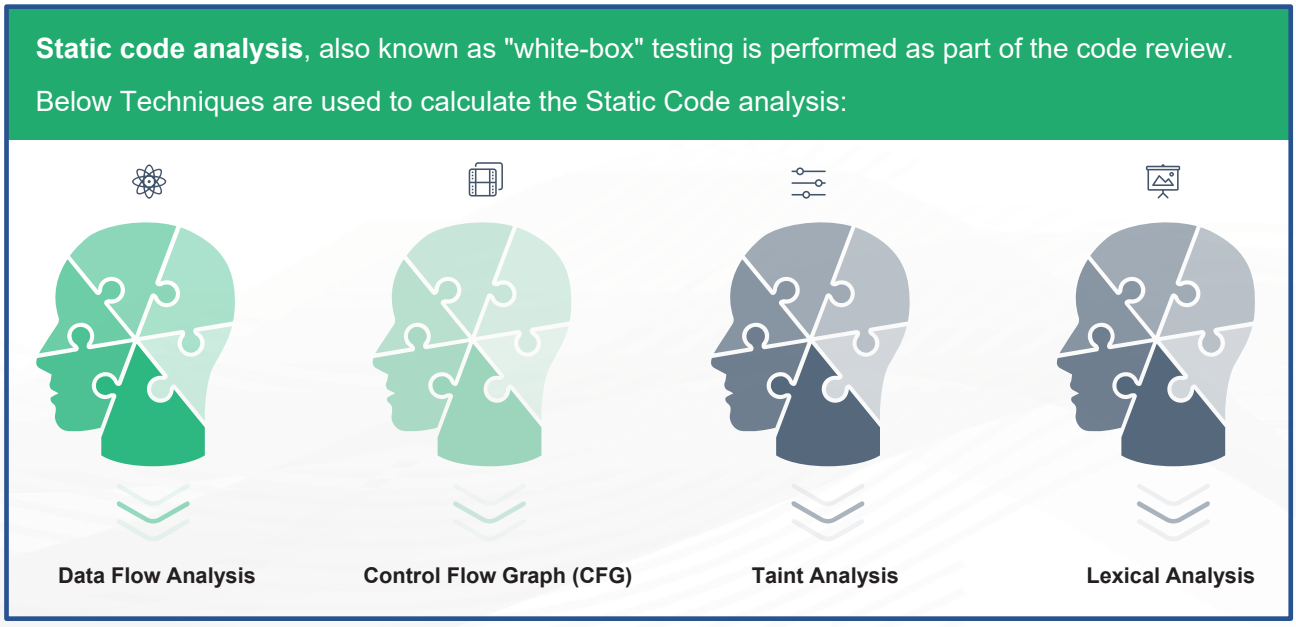
Automated build is the process of scripting and automating the retrieval of code from a repository, compiling it, executing automated functional tests, and publishing it to a shared repository.

Below are the list of key metrics considered in automated builds:

- **Number of features/user stories per build :** This indicates a number of new features being implemented/changes implemented and it directly maps to the business value that is created.
- **Average Build time:** This metric indicates the average time required to perform a build.
- **Percentage of failed builds:** This metric impacts the overall team output due to rework.
- **Change implementation lead time:** Affects the number of releases per given period and overall product roadmap planning.

- **Frequency of builds:** This metric indicated the overall output and it tracks the entire activity of the project.

2.3 Static Code Analysis



Facilitator Notes:

Explain the participants about the static code analysis and different techniques to analyze code.

Static code analysis is carried out at the Implementation phase of an SDL (Security Development Lifecycle). Static Code Analysis, commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as Taint Analysis and Data Flow Analysis.

Data Flow Analysis

- Data flow analysis collects run-time (dynamic) information about data in software code while it is in a static state.
- There are three common terms used in data flow analysis, Control Flow Analysis (the flow of data), basic block (the code), and Control Flow Path (the path the data takes):

○**Basic block:** A sequence of consecutive instructions where control enters at the beginning of a block, control leaves at the end of a block and the block cannot halt or branch out except at its end.

Example PHP basic block:

- 1. $a = 0; 2. $b = 1; 3. 4. if ($a == $b) 5. { # start of block 6. echo “a and b are the same”; 7. } # end of block 8. else 9. { # start of block 10. echo “a and b are different”; 11.} # end of block

- **Control Flow Graph (CFG):** An abstract graph representation of software by use of nodes that represent basic blocks. A node in a graph represents a block; directed edges are used to represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an ‘entry’ block, if a node only has an entry edge, this is known as an ‘exit’ block.

Example Control Flow Graph; ‘node 1’ represents the entry block and ‘node 6’ represents the exit block.

Taint Analysis

- ● Taint Analysis will attempt to identify variables that have been ‘tainted’ with user controllable input and traces them to possible vulnerable functions also known as a ‘sink’. If the tainted variable gets passed to a sink without being sanitized then it is flagged as a vulnerability.

- ● Some programming languages such as Perl and Ruby have Taint Checking built into them and enabled in certain situations such as accepting data via CGI.

Lexical Analysis

- Lexical Analysis will generally convert the source code syntax into the ‘tokens’ of information to abstract the source code and make it easier to manipulate.

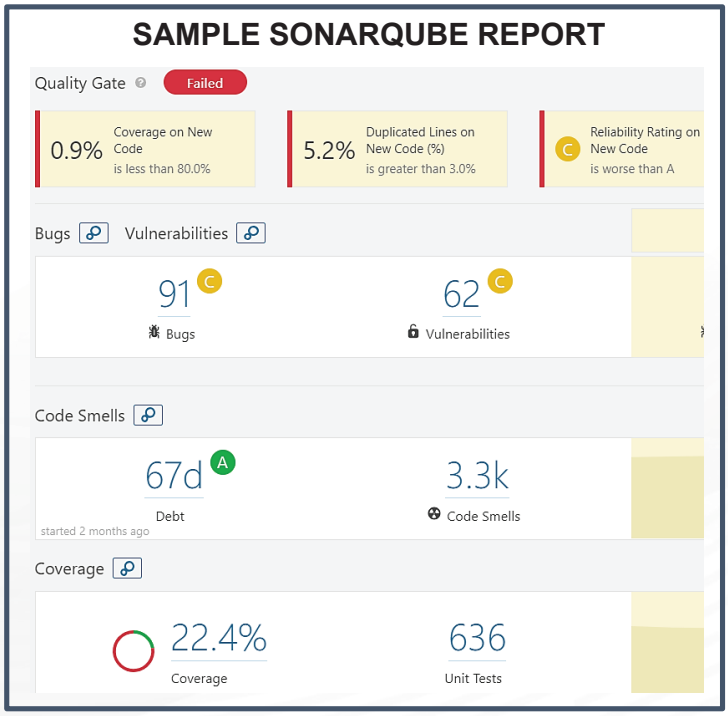
- Pre tokenised PHP source code:

- <?php $name = “Ryan”; ?> Post tokenised PHP source code:

- T_OPEN_TAG T_VARIABLE = T_CONSTANT_ENCAPSED_STRING ; T_CLOSE_TAG

There are many static code analysis tools available. Sonarqube is most widely used tool to do code analysis. Let's discuss about Sonarqube and its sample reports in coming slides.

2.3.1 Static Code Analysis – Sample Snapshot



Facilitator Notes:

Discuss about Static code analysis tool Sonarqube and sample snapshot of static code analysis quality gate report.

What is Sonarqube?

- SonarQube is a free and open source "code quality platform." It gives moment-in-time snapshot of your code quality today, as well as trending of lagging (what's already gone wrong) and leading (what's likely to go wrong in the future) quality indicators. For test coverage (a leading indicator), a score of 50% may not look great.
- SonarQube doesn't just show you what's wrong. It also offers quality-management tools to actively help you put it right: IDE integration, integration for Jenkins, a popular Continuous Integration server, and code-review tools.

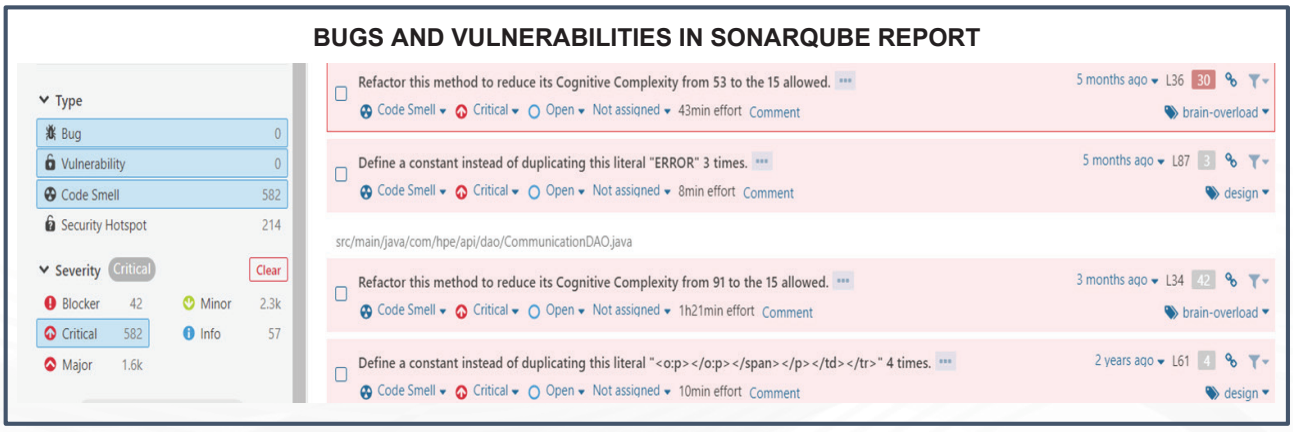
- SonarQube is written in Java, and it started as a way to measure the quality of Java projects, but it's no longer limited to analyzing just Java. There is an ever-growing list of languages you can analyze through SonarQube by adding plugins, many of which are provided by the SonarQube community and offered for free.

The above report is created using Sonarqube. In Sonarqube a quality gate is the best way to enforce a quality policy in your organization. It can answer ONE important question: can we deliver the project to production today or not?

In order to answer this question, you can define a set of Boolean conditions based on measure thresholds against which projects are measured. For example:

- No new blocker issues
- Code coverage on new code greater than 80%
- Etc.

2.3.2 Static Code Analysis – Sample Bugs Report



Facilitator Notes:

Give a brief about the critical issues that is present in the screenshot.

The above screenshot highlights the critical bugs in Sonarqube report. Sonarqube gives recommendation to fix the coding issues.

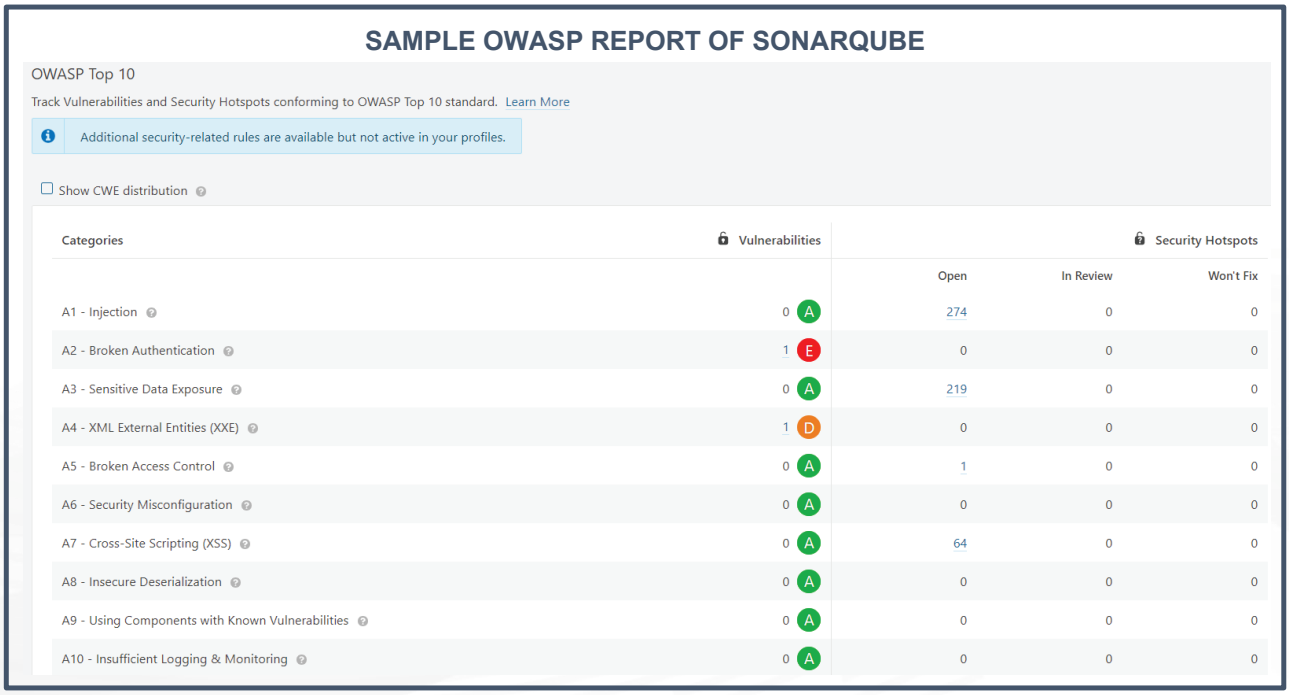
In SonarQube, rules are executed in source code to generate issues. There are four types of rules:

- Code Smell (Maintainability domain)
- Bug (Reliability domain)
- Vulnerability (Security domain)
- Security Hotspot (Security domain)

Security rating is assigned with below alphabets in the report:

A = 0 Vulnerabilities
B = at least 1 Minor Vulnerability
C = at least 1 Major Vulnerability
D = at least 1 Critical Vulnerability
E = at least 1 Blocker Vulnerability

2.3.3 Static code Analysis – OWASP Top 10 Sample Report



SAMPLE OWASP REPORT OF SONARQUBE

OWASP Top 10

Track Vulnerabilities and Security Hotspots conforming to OWASP Top 10 standard. Learn More

Additional security-related rules are available but not active in your profiles.

Show CWE distribution

Categories	Vulnerabilities	Security Hotspots		
		Open	In Review	Won't Fix
A1 - Injection	0 A	274	0	0
A2 - Broken Authentication	1 E	0	0	0
A3 - Sensitive Data Exposure	0 A	219	0	0
A4 - XML External Entities (XXE)	1 D	0	0	0
A5 - Broken Access Control	0 A	1	0	0
A6 - Security Misconfiguration	0 A	0	0	0
A7 - Cross-Site Scripting (XSS)	0 A	64	0	0
A8 - Insecure Deserialization	0 A	0	0	0
A9 - Using Components with Known Vulnerabilities	0 A	0	0	0
A10 - Insufficient Logging & Monitoring	0 A	0	0	0

Facilitator Notes:

Explain the participants about the static code analysis – OWASP Top 10 report.

The following is the list of OWASP Top 10 Security Risks, and offers best practices and solutions to prevent or remediate them.

1. Injection

- Injection flaws, such as LDAP injection, SQL injection, and CRLF injection, occur when an attacker starts sending untrusted data to an interpreter that is executed as a command without proper authorization.

Solution: * Application security testing can easily detect injection flaws. Developers must use parameterized queries in the code prevent injection flaws.

2. Broken Authentication and Session Management

- Incorrectly configured session and user authentication can allow attackers to compromise the passwords, session tokens or keys, or get all the details of users' accounts to assume their identities.

Solution: * Multi-factor authentication (FIDO) or dedicated apps, can reduce the risk of compromised accounts.

3. Sensitive Data Exposure

- APIs and Applications that don't properly protect sensitive data such as usernames and passwords, financial data, or health information, could allow attackers to access sensitive information to steal identities or commit fraud.

Solution: * Encryption of data at rest and in transit can help you comply with data protection regulations.

4. XML External Entity

- Poorly configured XML processors expose external entity references within XML documents. There is a change of using external entities by attackers to execute the code remotely, and to disclose SMB file shares and internal files.

Solution: * Static application security testing (SAST) can discover this issue by inspecting configuration and dependencies.

5. Broken Access Control

- Improperly configured or missing restrictions on authenticated users allow them to access unauthorized functionality or data, such as viewing sensitive documents, accessing other users' accounts, and modifying access and data rights.

Solution: * Penetration testing can detect non-functional access controls; whereas other testing methods can only detect where access controls are missing.

6. Security Misconfiguration

- This risk refers to improper implementation of controls that is intended to keep application data safe, such as information leakage, misconfiguration of security headers, and not patching or upgrading systems, components, and frameworks.

Solution: * Dynamic application security testing (DAST) can detect misconfigurations, such as leaky APIs.

7. Cross-Site Scripting

- Cross-site scripting (XSS) is a flaw that can enable an attacker to inject client-side scripts into the application, for example, to redirect users to malicious websites.

Solution: * Developer training complements security testing to help programmers prevent cross-site scripting with best coding best practices, such as input validation and encoding data.

8. Insecure deserialization

- Insecure deserialization flaw can enable an attacker to execute attacker's code in the application remotely, delete serialized or tamper objects, conduct injection attacks, and elevate privileges.

Solution: * Application security tools can detect deserialization flaws, but penetration testing is required frequently to validate the problem.

9. Using Components With Known Vulnerabilities

- Developers frequently can not find which open source and third-party components are in their applications, making it difficult to update components when new vulnerabilities when they are discovered. Attackers can exploit an insecure component to steal sensitive data or take over the server.

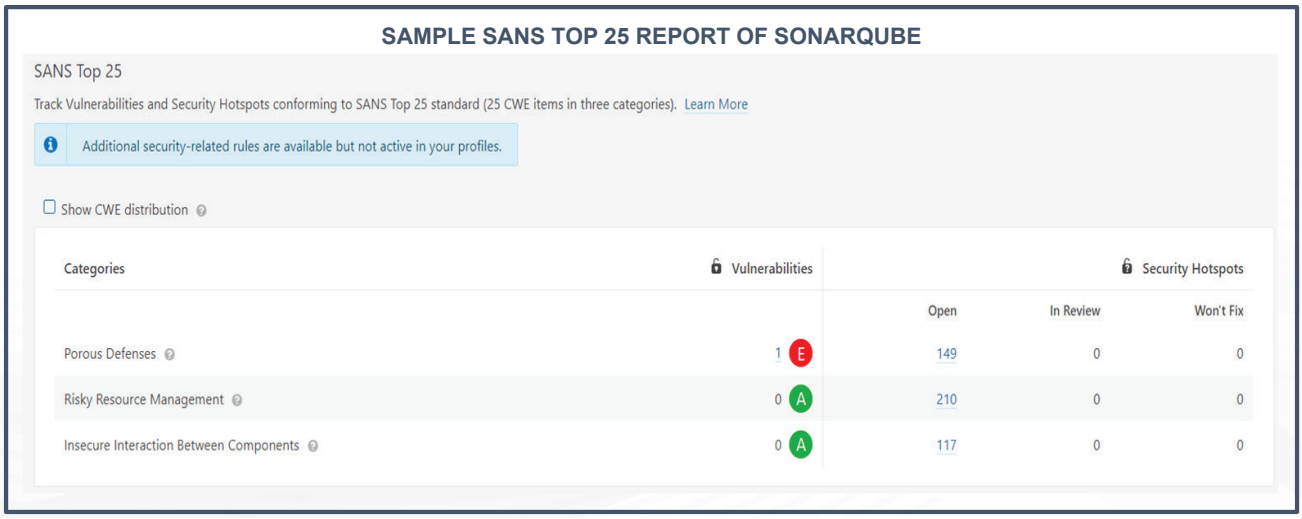
Solution: * Software composition analysis with static analysis can identify insecure versions of components.

10. Insufficient Logging and Monitoring

- The time to detect a breach is frequently measured in weeks or months. ineffective integration and Insufficient logging and with security incident response systems allow attackers to pivot to other systems and maintain persistent threats.

Solution: * Think like an attacker and use pen testing to find out if you are using efficient security monitoring; examine your logs after pen testing.

2.3.4 Static Code Analysis – SANS Top 25 Sample Report



Facilitator Notes:

Explain the participants about the Static code analysis – SANS Top 10 report.

SANS TOP 25 highlights the most Dangerous Top 25 Software Errors

Insecure Interaction Between Components:

CWE ID Name

CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

CWE-434 Unrestricted Upload of File with Dangerous Type

CWE-352 Cross-Site Request Forgery (CSRF)

CWE-601 URL Redirection to Untrusted Site (‘Open Redirect’)

Risky Resource Management:

The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.

CWE ID	Name
CWE-120	Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)
CWE-22	Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
CWE-494	Download of Code Without Integrity Check
CWE-829	Inclusion of Functionality from Untrusted Control Sphere
CWE-676	Use of Potentially Dangerous Function
CWE-131	Incorrect Calculation of Buffer Size
CWE-134	Uncontrolled Format String
CWE-190	Integer Overflow or Wraparound

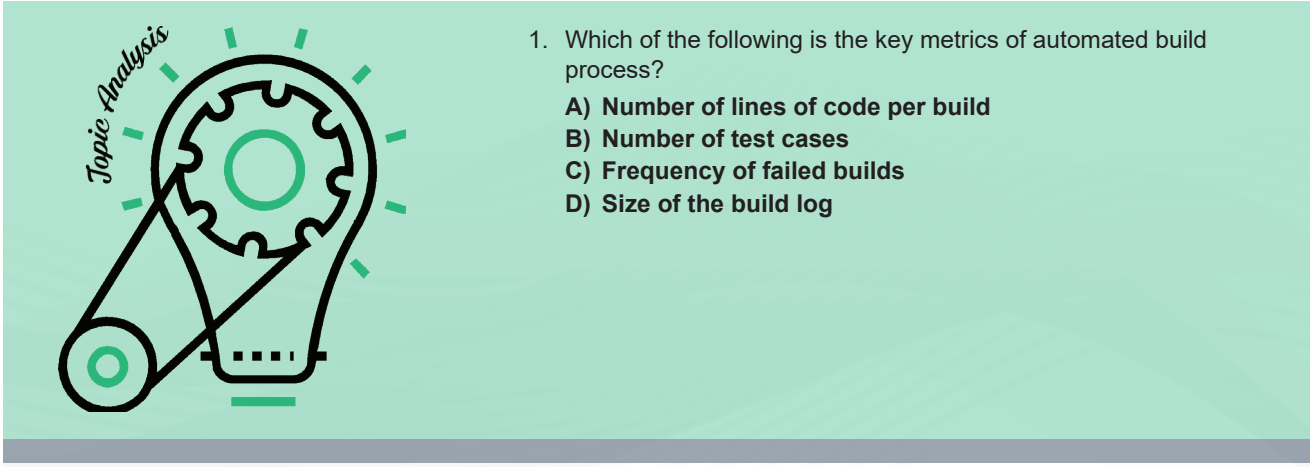
Porous Defenses:

The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.

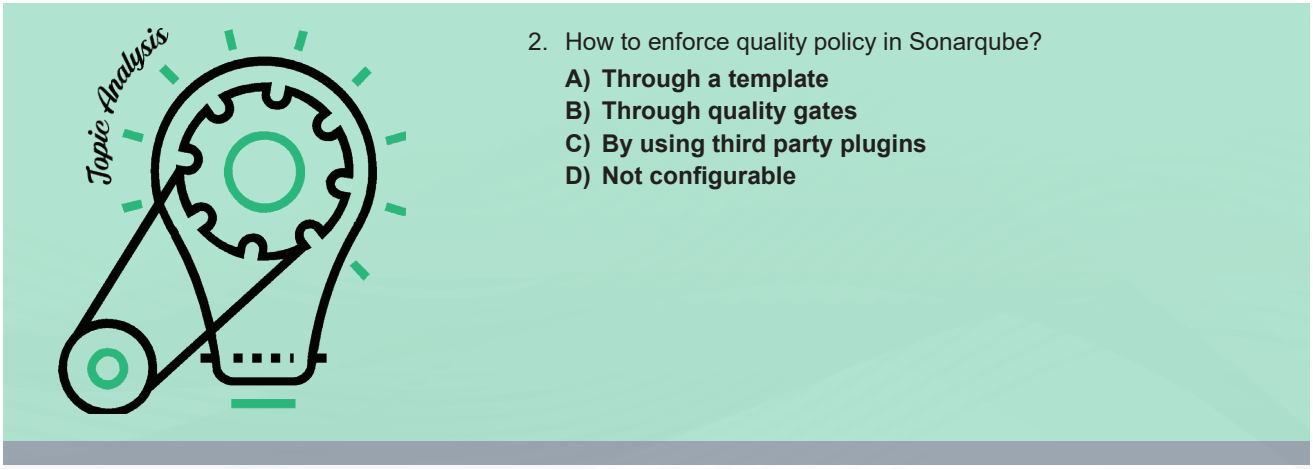
CWE ID	Name
CWE-306	Missing Authentication for Critical Function
CWE-862	Missing Authorization
CWE-798	Use of Hard-coded Credentials
CWE-311	Missing Encryption of Sensitive Data

CWE-807	Reliance on Untrusted Inputs in a Security Decision
CWE-250	Execution with Unnecessary Privileges
CWE-863	Incorrect Authorization
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-307	Improper Restriction of Excessive Authentication Attempts
CWE-759	Use of a One-Way Hash without a Salt

What did You Grasp?



1. Which of the following is the key metrics of automated build process?
   - **A) Number of lines of code per build**
   - **B) Number of test cases**
   - **C) Frequency of failed builds**
   - **D) Size of the build log**



2. How to enforce quality policy in Sonarqube?
   - **A) Through a template**
   - **B) Through quality gates**
   - **C) By using third party plugins**
   - **D) Not configurable**

Facilitator Notes:

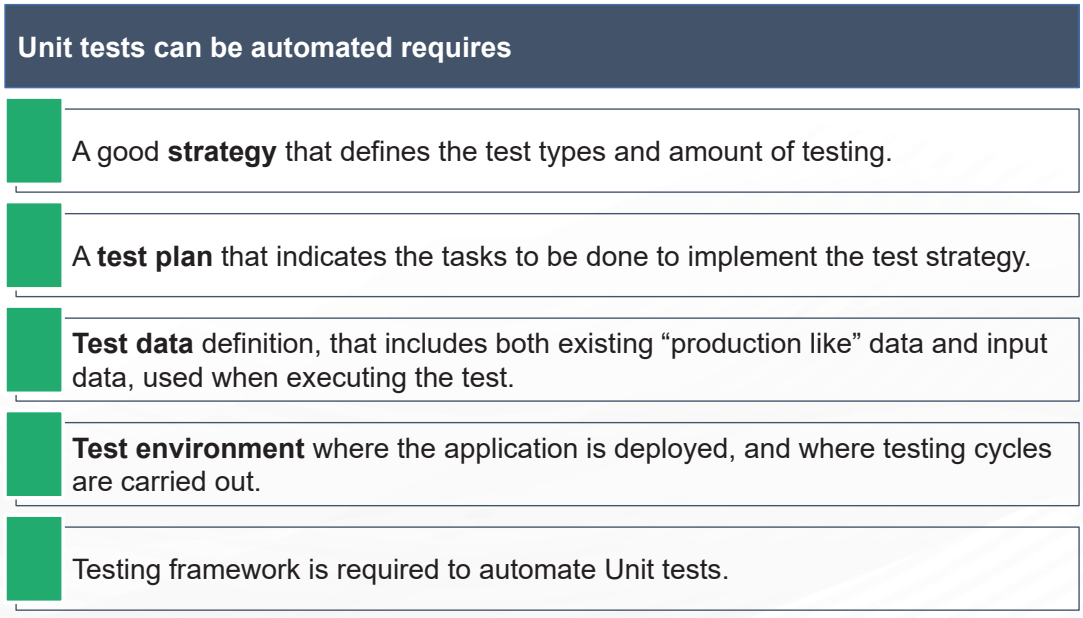
Answer: 3. Frequency of failed builds.

If the build failure ratio is high, then the quality of code is not up to the mark.

2. Through quality gates

Quality gates are the best way to enforce quality profiles in Sonarqube. There are some Plugins which can fail the build based on Quality gates score.

2.4 Automated Unit Testing



Facilitator Notes:

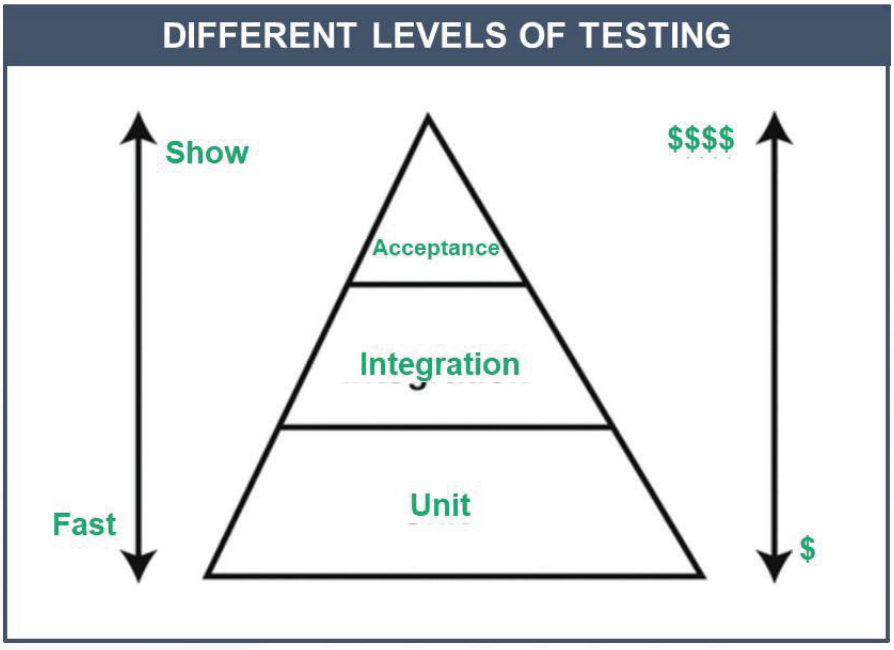
Give a brief about continuous delivery.

Many organizations create test cases in excel sheet and few in word documents. Some organizations use HP ALM to document the test cases.

Every test case have similar attributes:

- Title
- Description
- Priority
- Assumptions and/or pre-conditions
- A set of test steps
- Test data to be used to test the functionality
- Expected result
- Notes etc

2.4.1 Unit Testing



Facilitator Notes:

Discuss about Unit tests and its benefits in detail.

Levels of Testing:

As you start practicing TDD (Test Driven Development), you will write different levels of tests. Your application should be composed of tests in each of the following levels. Each of these levels focuses on a different aspect of code and provides different feedback. Let's look at them one by one.

Unit testing: Here you test individual software components to verify if the individual unit does the right thing in isolation.

Integration testing: Here you test multiple units together to verify if they work correctly as a unit.

Acceptance testing: Here you test the full system to verify if it works as per user expectations. It is often referred to as functional testing.

Benefits of Unit Testing:

unit testing is no longer a post-development exercise. It is as equally important as writing production code and must be done up front. Benefits are Unit testing are:

- Determines specifications
- Provides early error detection
- Supports Maintenance
- Improves Design
- Better product documentation

To run Unit tests when building the code we need a Framework (Example: Junit). Junit framework will execute all the Unit tests one by one while executing your code.

To run Junit tests the first step it to add dependency information in the pom.xml

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
```



```
<version>4.12</version>
<scope>test</scope>
</dependency>
```

Let’s look at the below class and Junit test:

- Below class will concatenate both the strings:

```
public class MyJUnit {

public String concatenate(String one, String two

)

      {

       return one + two;

      }

}
```

2. Junit testcase for this class:

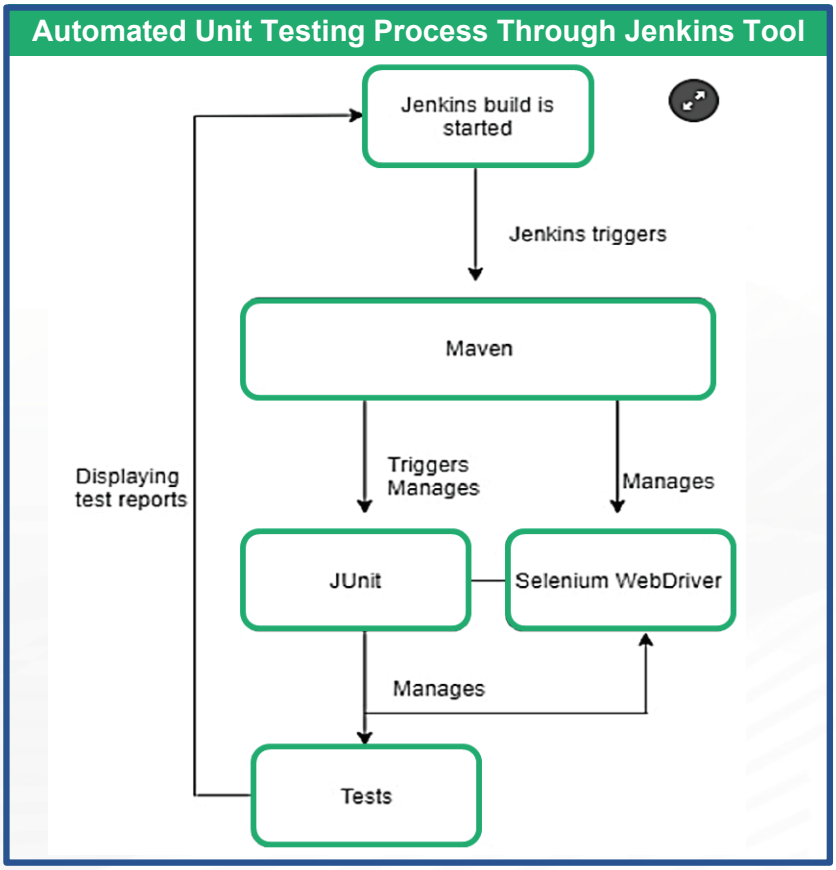
```
import org.junit.Test;

import static org.junit.Assert.*;

public class MyJUnitTest {

 @Test

 public void testConcatenate() {

 MyJUnit myJUnit = new MyJUnit();

 String result = myjUnit.concatenate(“one”, “two”);
```

```
 assertEquals(“onetwo”, result);

 }

}
```

This is considered as positive testcase. In Negative testcase we can test assertNotEquals(“twoone”,result)

The above unit test is an ordinary class, with one method - testConcatenate(). This method is annotated with the @Test annotation. This will signal the unit test runner, that this is a unit test, that should be executed. If methods are not annotated with @Test will not be executed by the test runner.

2.4.3 Automated Unit Testing



Facilitator Notes:

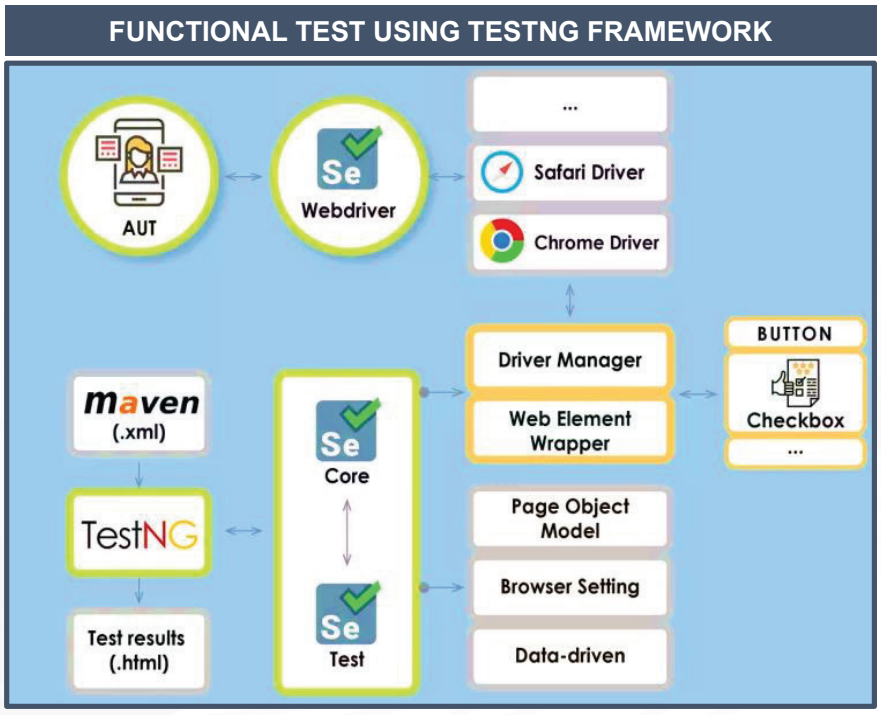
Give a brief about automated unit testing.

Refer the above picture. Logical flow of Automated unit testing through Jenkins is shown.

Automated Unit tests are executed in the following flow:

- New code is checked-in to SCM (Example - Git)
- Git hooks send message to build tool (Example - Jenkins)
- Jenkins will execute the build and run Unit tests through a framework (Junit/TestNG)
- If all the tests are passed, build is passed on to next stage and reports are deployed to the dashboard
- If any Unit test fails build will fail and developers will be notified through standard communication channels

2.4.2 Functional test example using - TestNG



Facilitator Notes:

Give a brief about sample functional test example using TestNG framework.

Above picture represents a sample functional test using TestNG framework. Let us see more details about TestNG framework:

- TestNG is most popularly used with Selenium for UI testing.
- TestNG is similar to Junit testing framework, but it is more powerful than Junit, especially in testing integrated classes. TestNG inherits all of the benefits that JUnit is offering.
- TestNG eliminates most of the limitations of the older framework and gives developers the ability to write more flexible and powerful tests. Some of the highlight features are: easy annotations, grouping, sequencing and parameterizing.
- TestNG eliminates most of the limitations present in the old framework, it allows developers to write more powerful and flexible test cases with the help of easy annotations, sequencing, grouping and parameterizing.

In the above screenshot we are testing an application login feature using Firefox browser.

Let's discuss about the methods used in the below test case:

- **@BeforeMethod** : Launch Firefox browser and open the Base URL
- **@Test** : Enter Username & Password to Login, Print console message and Log out
- **@AfterMethod** : Close Firefox browser

```
package automationFramework;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;
```



```
import org.testng.annotations.Test;

import org.testng.annotations.BeforeMethod;

import org.testng.annotations.AfterMethod;

public class TestNG {

     public WebDriver driver;

 @Test

 public void main() {

     // Find the element that’s ID attribute is ‘account’(My Account)

 driver.findElement(By.id(“account”)).click();

 // Find the element that’s ID attribute is ‘log’ (Username)

 // Enter Username on the element found by above desc.

 driver.findElement(By.id(“log”)).sendKeys(“testuser_1”);

 // Find the element that’s ID attribute is ‘pwd’ (Password)

 // Enter Password on the element found by the above desc.

 driver.findElement(By.id(“pwd”)).sendKeys(“Test@123”);

 // Now submit the form. WebDriver will find the form for us from the element

 driver.findElement(By.id(“login”)).click();

 // Print a Log In message to the screen

  System.out.println(“ Login Successfully, now it is the time to Log Off
buddy.”);

 // Find the element that’s ID attribute is ‘account_logout’ (Log Out)
```



```
 driver.findElement(By.id(“account_logout”));

 }

 @BeforeMethod

 public void beforeMethod() {

       // Create a new instance of the Firefox driver

 driver = new FirefoxDriver();

 //Put a Implicit wait, this means that any search for elements on the page
could take the time the implicit wait is set for before throwing exception

 driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

 //Launch the Online Store Website

 driver.get(“http://www.onlinestore.toolsqa.com”);

 }

 @AfterMethod

 public void afterMethod() {

       // Close the driver

 driver.quit();

 }

}
```


2.5 Code Coverage Analysis

The following are the key details of code coverage analysis

- The process of discovering the code that is not exercised by the tests.
- This information will be used to improve the test suite by adding test cases or modifying the existing ones to increase coverage.
- Code Coverage Analysis can find the quality of your unit testing.
- It enables developers to quickly and easily improve the quality of their unit tests.
- An application with higher code coverage has chances of lower bugs.
- Generally code coverage is one of the exit criteria for each milestone.

Facilitator Notes:

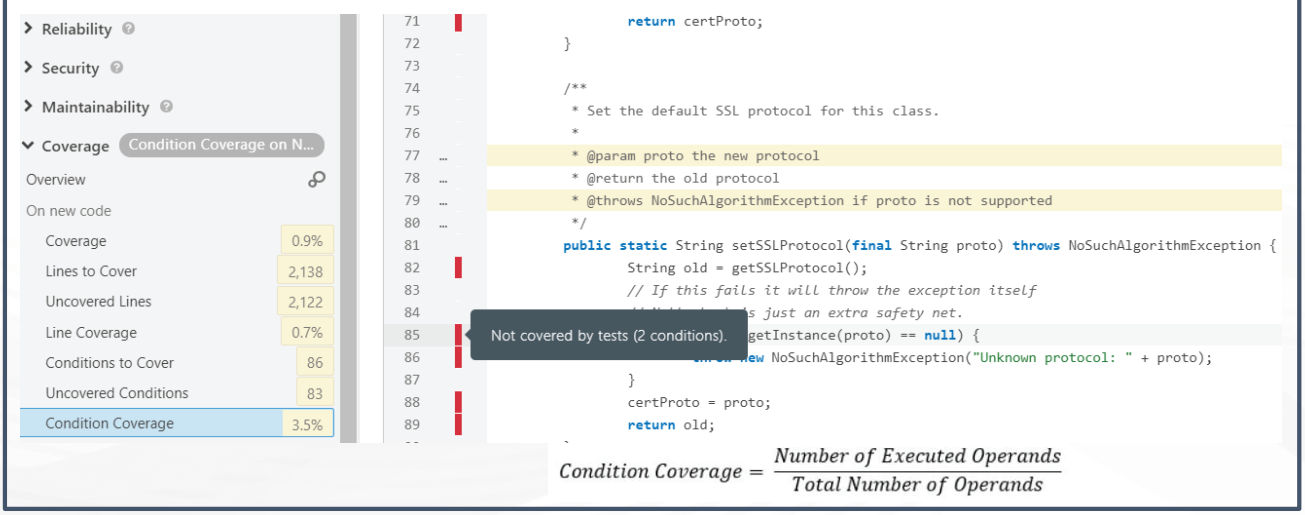
Give a brief about code coverage analysis.

Code coverage is a mechanism to find how many lines/arcs/blocks/ of your code is executed while running the automated tests. Code coverage is collected by specialized tools by adding tracing calls and run a full set of automated tests against the product. A good code coverage tool will not only give the percentage of the code that is covered, but also will show which lines of code were covered during a particular test.

There are multiple code coverage types. We will now discuss some sample report of condition and line coverage in the next slide.

2.5.1 Code Coverage Methods – Condition Coverage

SAMPLE CONDITIONAL COVERAGE REPORT FROM THE SONARQUBE



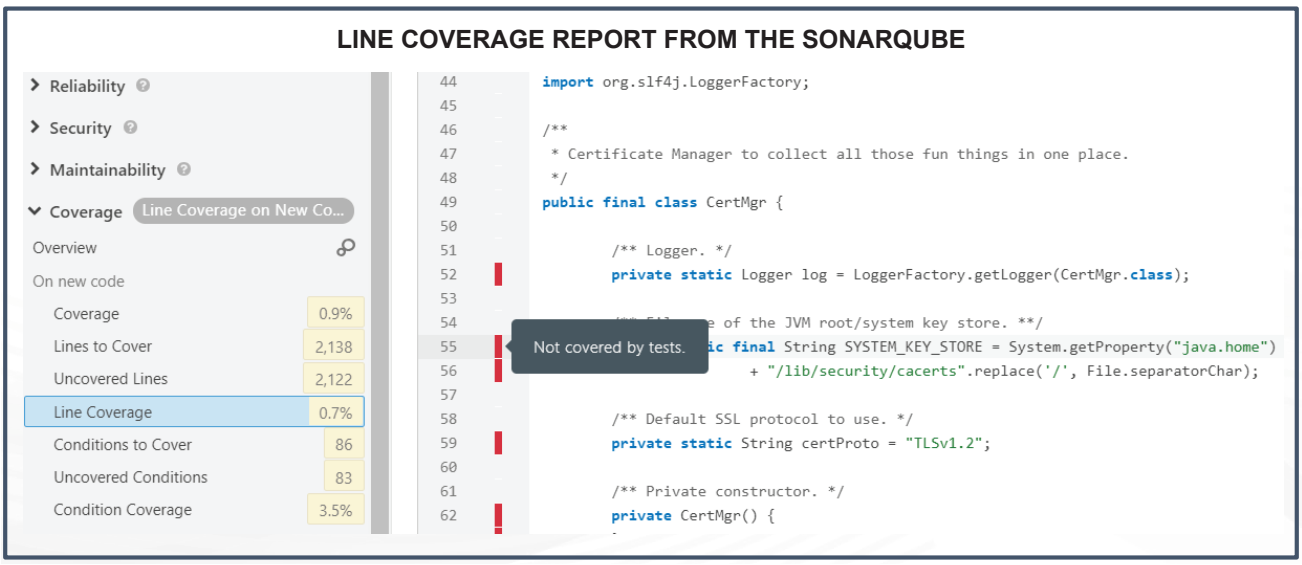
Facilitator Notes:

Explain the participants about the condition coverage in code coverage methods.

The above screenshot shows sample Condition coverage report from the Sonarqube. The conditions which are not covered by the tests are highlighted as shown in the above picture

Conditional coverage will reveal how the subexpressions or variables in the conditional statement are evaluated. In this type of coverage, expressions with only logical operands are considered. For example, if an expression has Boolean operations like OR, AND, XOR, which are total possibilities. Condition coverage does not give a guarantee about full decision coverage.

2.5.2 Code Coverage Methods – Line Coverage



Facilitator Notes:

Explain the participants about the line coverage in code coverage methods.

The above screenshot shows sample Line coverage report from the Sonarqube. The lines which are not covered by the tests are highlighted as shown in the above picture

Line coverage is also known as statement coverage. The formula used to calculate statement coverage is:

Statement Coverage=(Number of statements exercised/Total number of statements)*100

Studies have shown that black-box testing can actually achieve only 60% to 75% statement coverage, but still this leaves around 25% to 40% of the statements untested.

To illustrate the principles of code coverage lets take an which is not specific to any programming language. We have numbered the code lines just to illustrate the statement coverage example, however this may not always be correct.

READ A

READ A

IF A>B

PRINT “A is greater than B”

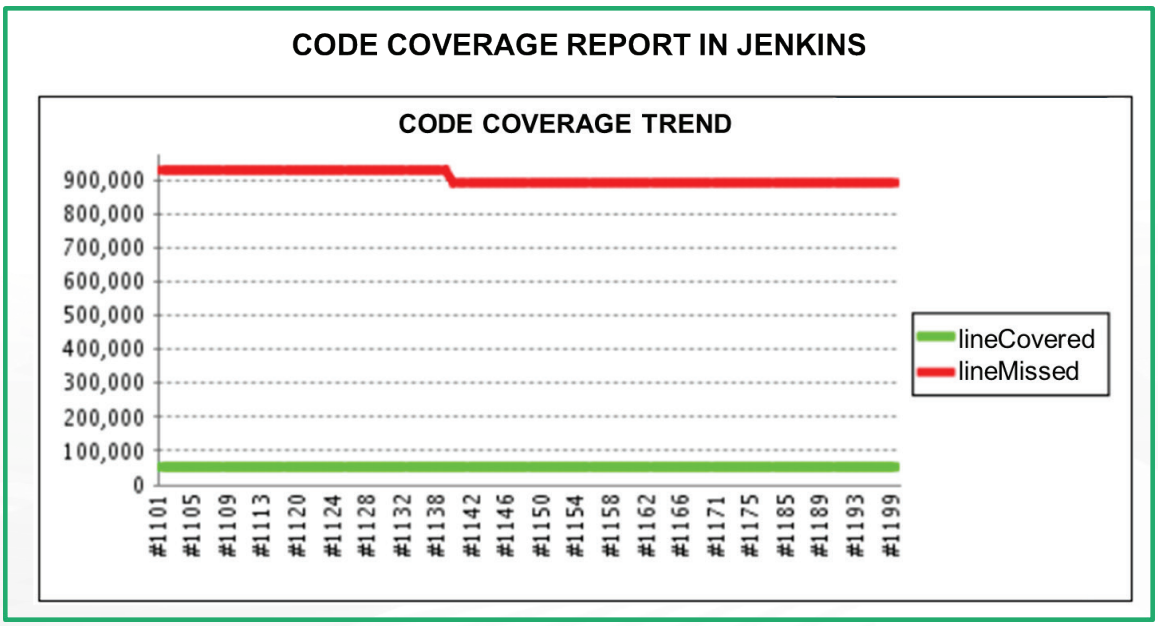
ENDIF

Let us see how can we achieve 100% code coverage for this example code, we can have 100% coverage by just one test set in which variable A is always greater than variable B.

TEST SET 1: A=10, B=5

A statement may be spread over several lines or in a single line. A statement can contain more than one statement. Some code coverage tools execute statements together in a block and consider them as a single statement.

2.5.3 Publishing Code Coverage Reports to Jenkins



Facilitator Notes:

Explain the participants how to publish code coverage report in Jenkins.

The picture in above slide shows the Code coverage report in Jenkins.

To enable code coverage in Jenkins:

- Go to Post build actions section.
- Select the Record Emma Coverage report.
- In the text box enter pattern like **/coverage.xml.
- If you leave it blank, Jenkins will search the entire workspace for Coverage.xml generated during the build.
- Jenkins will publish the coverage report as shown in the above picture.

What did You Grasp?

1. Fill in the blank.

   In code coverage analysis, conditional coverage can be calculated using……..:

   A) Number of execute operands/Total number of operands
   B) Number of conditions checked/Total number of conditions
   C) Number of lines covered/Total number of conditions
   D) Number for lines not covered/Total number of conditions

2. State whether below statement is True or False:

   Unit tests can be executed to calculate code coverage.

   A) True
   B) False

Facilitator Notes:

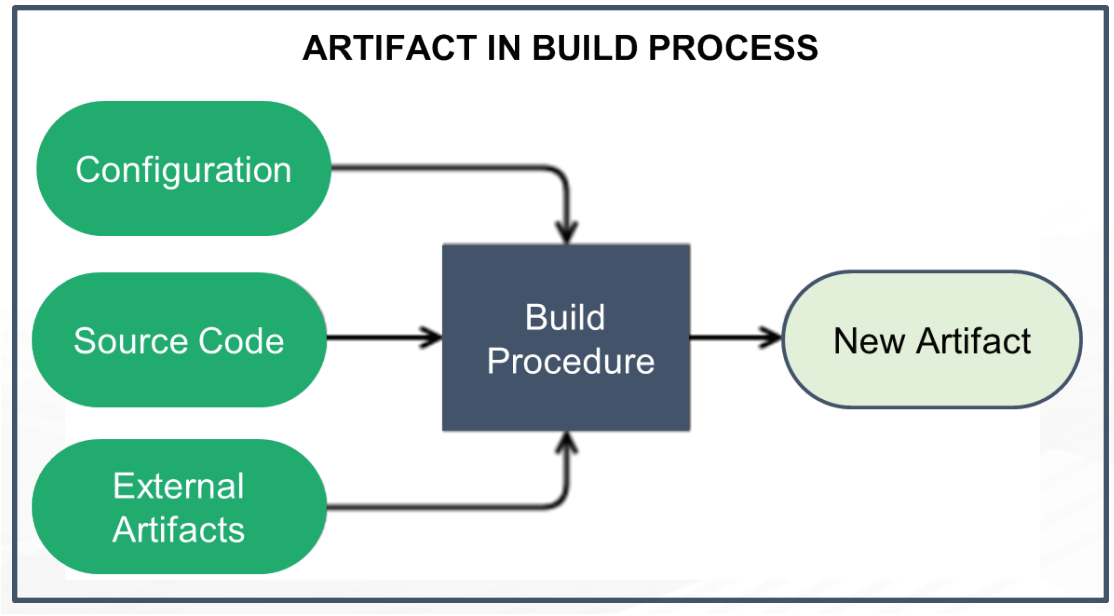
Answer: 1. Number of execute operands/Total number of operands

Conditional coverage can be calculated using the above predefined formula by any code analysis tools.

1. True

Unit tests are executed by code coverage tools through plugins like Jaccoco, emma to calculate the code coverage of the project.

2.6 Automated Packaging of the Build Artifact



Facilitator Notes:

Explain the participants about the automated packaging of the build artifact.

Artifact is automatically generated as part of a build process. Artifacts could be:

- The source code
- The compiled application
- A deployable package
- Documentation
- Examples are jar, war, tar, etc.

The output of a build job is persisted into an artifact and it can be used later to run your application. The build process automatically packages the artifact as per your project configuration.

Example 1: In a Java project you can package all your class files and dependencies into .jar file which is called an artifact.

Example 2: A build artifact file generated by your project, for example .apk which can be uploaded to Googleplay.

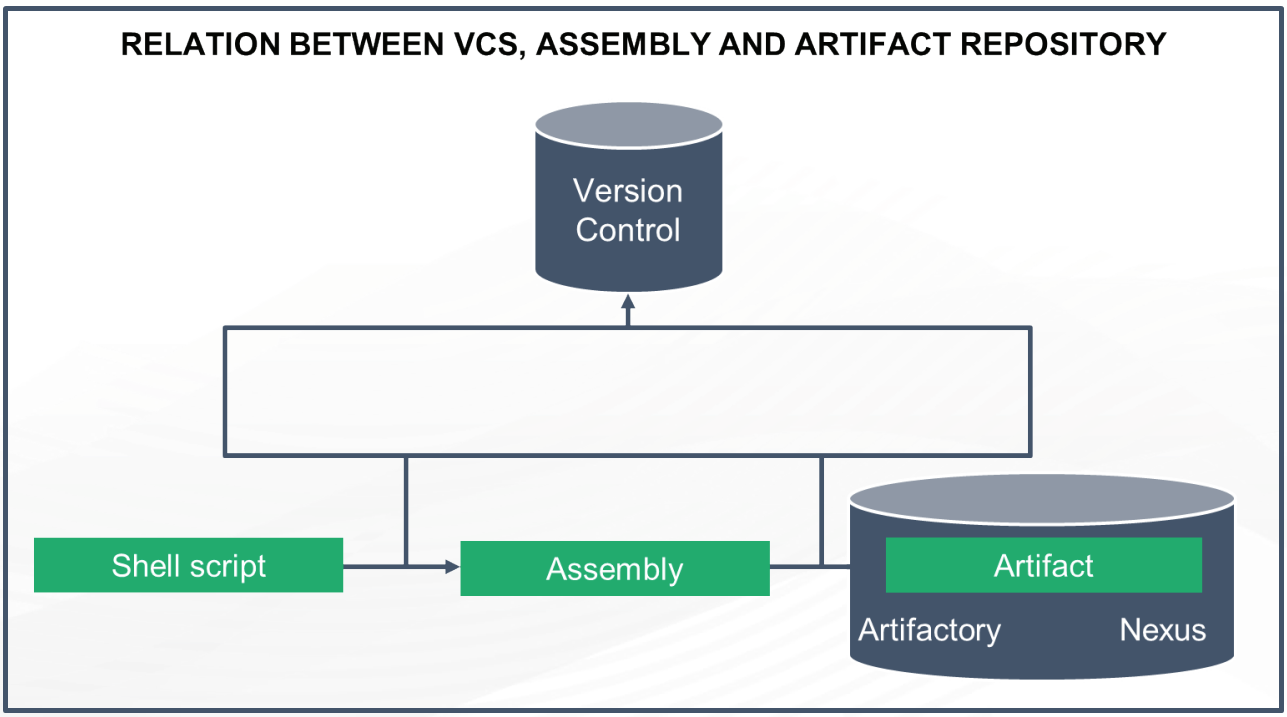
As per naming convention of Maven a build artifact can be identified using:

groupID: It is a unique identifier for your project

artifactID: It is the actual name of your artifact with out version

version: For distribution purpose you can choose a version for your artifact

2.6.1 Uploading Build Artifact to a Repository



Facilitator Notes:

Explain the process of uploading the artifact to a repository.

The above picture displays the relation between VCS, assembly and artifact repository.

- Build tools automatically generate the artifact and it can be shared across teams.
- Most of the organizations publish their artifacts to a repository like artifactory, nexus, etc either through a build process or a script.

In maven you need to configure the artifact type (Jar,war,zip etc), repository server configuration in your pom.xml and run **mvn deploy** command to deploy your artifacts to a Repository

In Gradle you need to define what to publish and where to publish in your configuration file and run **gradle publish** to deploy the artifacts to a repository.

What did You Grasp?

1. Fill in the blank.
   Artifact attributes can be specified in……..
   A) Global settings.xml
   B) Pom.xml
   C) Build tool
   D) External script

2. Which one of the following is not an artifact?
   A) Jar
   B) War
   C) Tar
   D) iso

Facilitator Notes:

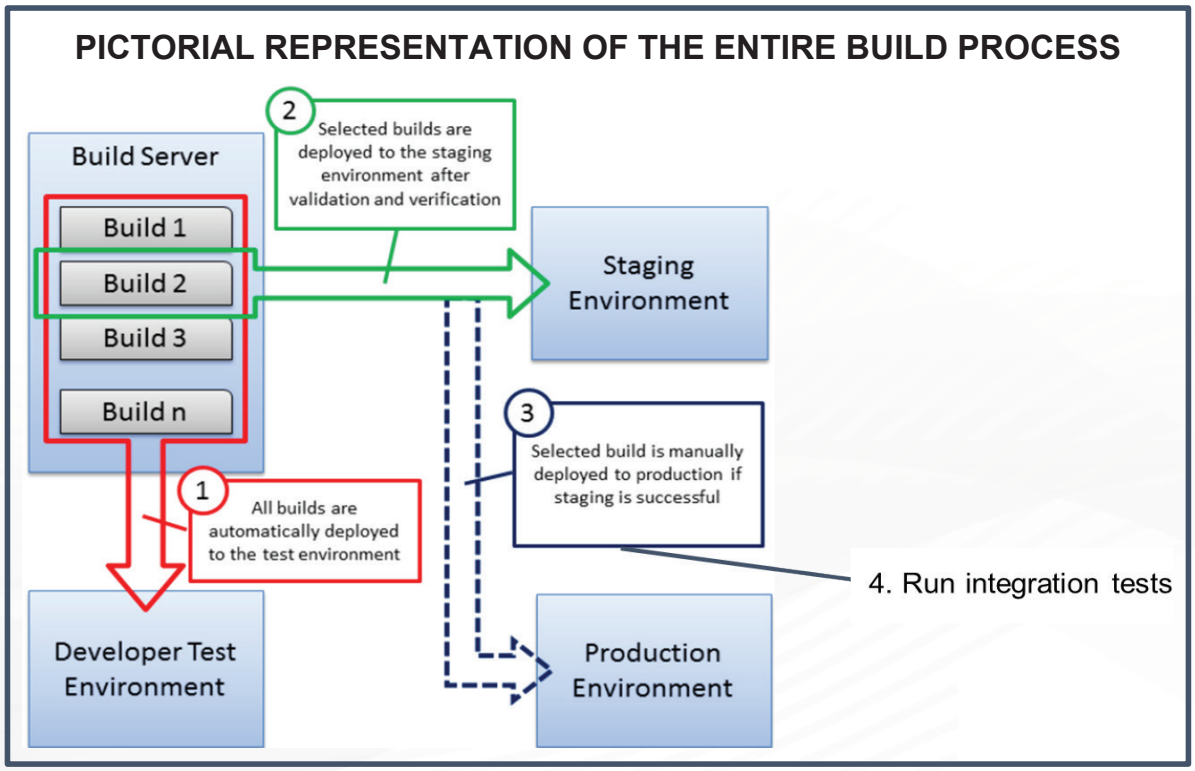
Answer:

1. B. Pom.xml

Maven artifact attributes - **GroupID**, **ArtifactID**, **Version, Classifier** and **Type** should be specified in pom.xml.

2. D. iso

2.7 Provision and Deploy to Test Environment



Facilitator Notes:

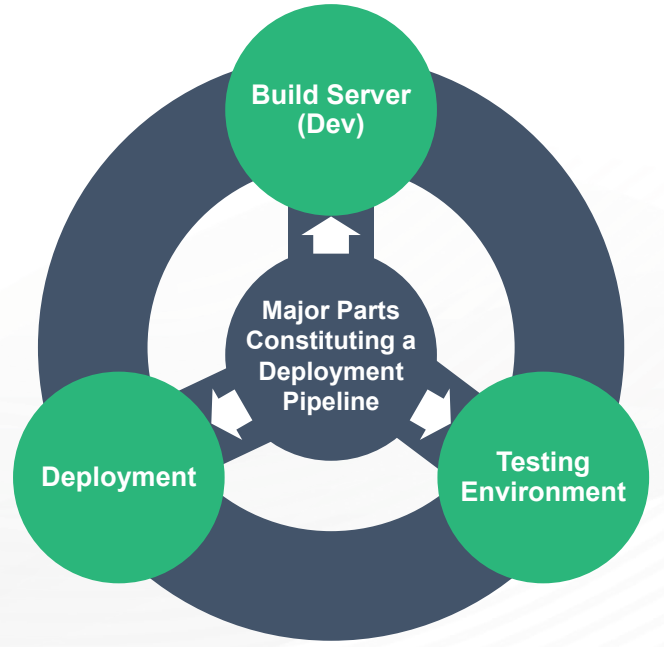
Give the participants an overview of the provisioning and deployment to test environment

The above image shows a pictorial representation of the entire build process.

By default, all the builds are deployed to the testing environment.

- Developer check-in the code to an SCM
- SCM will trigger a remote build through git hooks or polling
- Build tool will build the application and deploy it to test environment
- Various integration tests are executed in the test environment.

2.7.1 Deploying application to test environment



Facilitator Notes:

Give the participants an overview of the continuous deployment process.

- In test environment the build is subjected to multiple test suites.
- The build is tested with various parameters.
- It is the duty of these test suites to catch the bugs.

- Testing result decides whether to release the application to production or not based on number of bugs.

Deploying the application to staging involves these steps:

- Deploying the application
- Testing the deployment
- Activating the release for all users

2.8 Automated Functional Testing

Following are the key details of functional tests

- **Functional testing** is an **automated testing** that tests how applications function with the rest of the system.
- Traditionally, **functional testing** is implemented by a team of testers, independent of the developers.
- Functional testing will be done based on the requirements and business scenarios.
- Functional testing will ensure that new bug fixes did not break existing functionality.
- Functional testing plan include – testing purpose, test construction, test automation, execution and checking the results.

Facilitator Notes:

Give the participants an overview of functional tests.

Functional testing has many categories and these can be used based on the scenario.

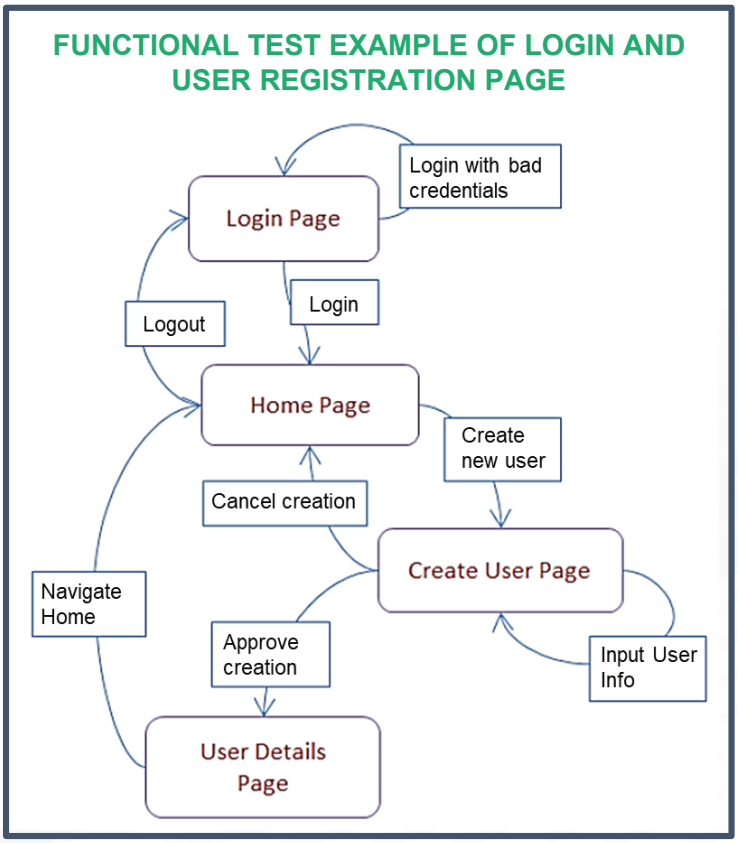
- Unit testing
- Sanity testing
- Smoke testing
- Regression tests
- Integration tests

- Beta/Usability testing

There are various steps involved in functional testing:

- Determine the functionality of the product that is to be tested
- Created input data for the functionality as per the requirement specification
- Executed the test cases that are prepared
- Review the output to find whether the functionality is as per the expectation or not

2.8.1 Automated Functional Testing - Example



Facilitator Notes:

Give the participants an overview of the example functional testing in the above screenshot.

For example take an online learning portal which is shown above where the student registers/

logs in with his user account and password. On the home page, there are two pages Create user and Login page. If a user creates his account it should be approved by the admin. If user has already registered he can enter a username and password, and two buttons: Login and Cancel. Successful login takes the user to the learning portal home page and cancel will cancel the login.

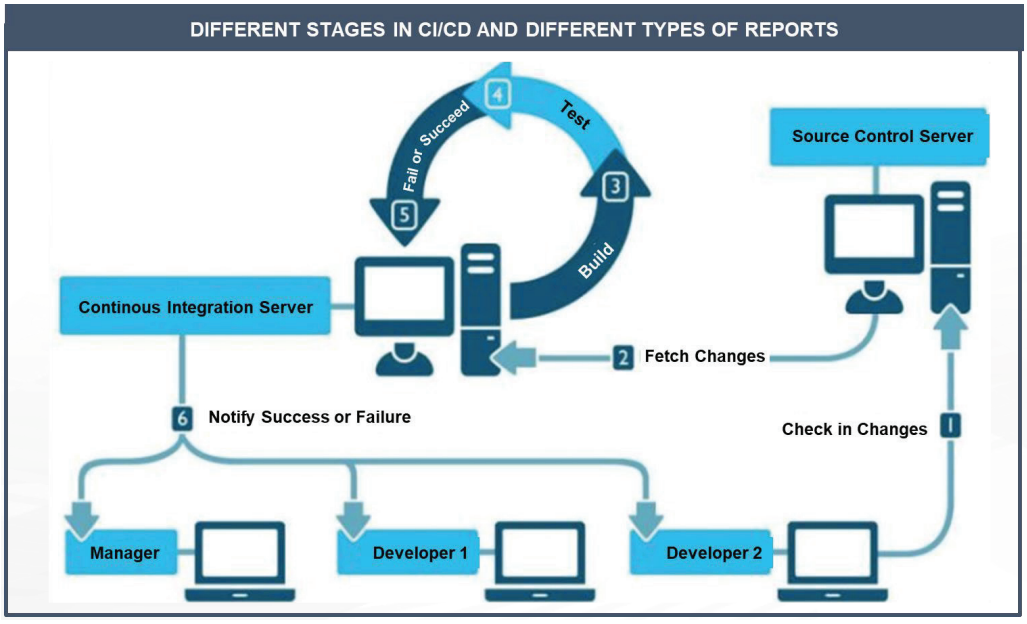
Specifications are as shown below:

#1) The user id field accepts a minimum of 7 characters, a maximum of 12 characters, letters(a-z, A-z), numbers(0-9), special characters (only underscore, hyphen allowed) and it cannot be left blank. User id must begin with a character or a number and special characters are not allowed.

#2) Password field accepts a minimum of 8 characters, a maximum of 12 characters, letters (a-z, A-Z), numbers (0-9), special characters (all) and cannot be blank.

Based on these specifications, functional tests can be executed and check if the portal is working as per the specifications. If users try to login using bad credentials, it is an example of negative testing.

2.9 Publishing Report to Development Team



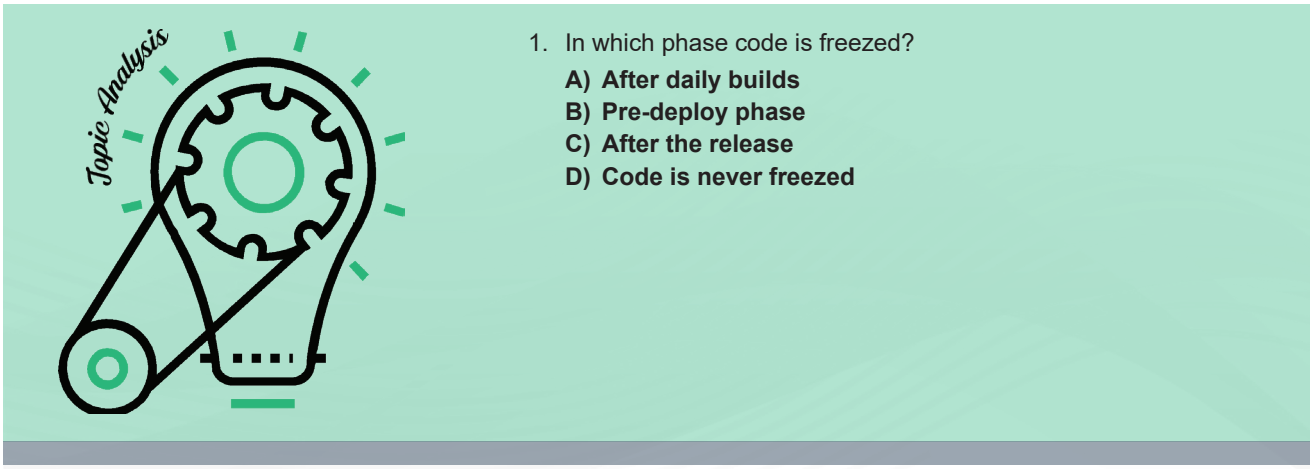
Facilitator Notes:

Give the participants an overview about the reports published to development team.

Many reports are created as part of CI/CD process. Developers are interested in below reports:

- Unit test reports
- Static code analysis reports
- Build status reports (Success/Failures)
- Integration test reports
- Deployment reports

What did You Grasp?



2. All builds are automatically deployed to:
   A) Staging environment
   B) Production environment
   C) Testing environment
   D) UAT environment

Facilitator Notes:

Answer:

1. B. Pre-Deploy phase

In Pre-deploy phase code is freezed and developers can not check-in the code. Only critical bug fixes can be committed.

2. C. Testing environment

By default, all the builds are published to the developer's environment (Test environment). Only qualified builds go to Staging and then UAT/Production.

Group Discussion

Facilitator Notes:

Form different groups of students. Each group should talk about the topics that is covered so far.

Notes to participants:

We have so far discussed about concept of VCS, merging local changes to the VCS tool, automated code builds, static code analysis, automated unit testing, code coverage analysis, automated packaging of the build artifact, Overview of provision and deploy to test environment, automated functional testing and publishing report to the development team.

In next slide let us discuss about Google Canary release strategy which is being used to release new features through Continuous delivery.

2.10 Google's Continuous Delivery – Case Study

- Secret to Google's engineering release is Canary release.
- A Canary release is releasing features only to some users.
- If the features are not successful then they are quickly rolled back or fixed/removed.
- Canary release is tested by close to 50000 employees.
- The advantages of Google's canary release approach are:
    - stability – there's always a release to roll back to
    - feedback – real world users give real world reactions

Facilitator Notes:

DIscuss about the Google's Continuous delivery case study with the participants.

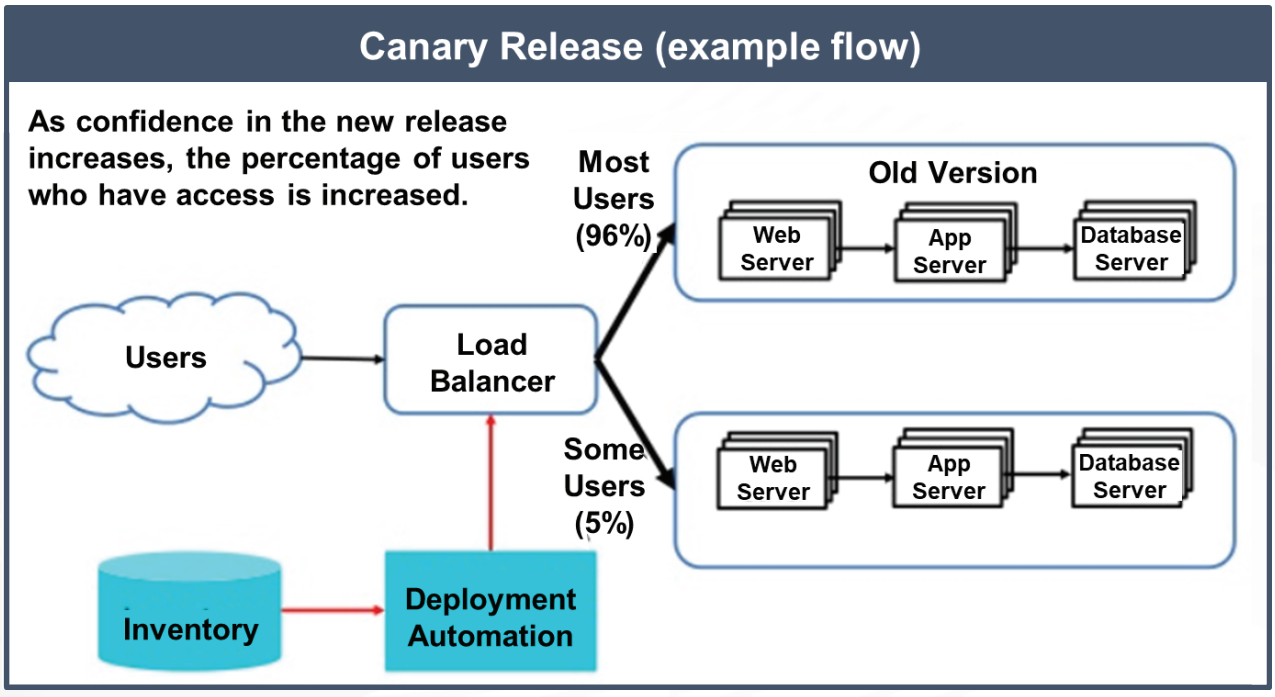
In previous slides, we have talked a lot about Continuous Integration / Continuous Delivery. Let's go through Google's continuous delivery mechanism to understand how they are releasing new features successfully.

Google uses Canary release methodology to release new features.

- From Monday throughout the week, Google gradually releases new features internally to its own users. Google monitors hundreds of different metrics on stability and scalability. Google's employees act as canaries – immediately provide details about any issues that come up while testing. If any of the core metrics are impacted significantly, then the release is rolled back entirely. However, critical issues are fixed and the release goes to the entire Google employee base.
- On the coming Monday, Google will take the canary release that is rolled out to Google employees externally. Google will start delivering the new release to more users. Google will continue to monitor the new feature for stability and scalability – external users are considered as the true "staging" environment. Google will then do a sentiment analysis for keywords like "Gmail Sucks" on twitter, etc. If there are spikes, Google will stop or roll back the new release.

2.11 Google's Continuous Delivery – Case Study (Contd.)

The following image explains Canary release model in detail.



Facilitator Notes:

Explain the participants about the Google's Canary release – Case study.

The above picture explains Canary release model in detail.

- Deployment models like canary can prevent major production outages
- Canary release helps the team to pinpoint deployment failures in the process and correct them
- Pulling together a basic failure response plan

In a nutshell, we learnt

1.2 a Core CI Process

- Merging local changes to the integration branch of VCS tool
- Automated code builds
- Static code analysis
- Automated unit testing
- Code coverage analysis
- Automated packaging of the build artifact

1.2 b Advanced CI Process

- Provision and deploy to test environment
- Automated functional testing
- Publish report to the development team

Facilitator Notes:

Share the module summary with the audience.

Ask the participants if they have any questions. They can ask their queries by raising their hands.

