Contents

[Continuous Integration 3](#_Toc488412971)

[Introduction 3](#_Toc488412972)

[The problem 3](#_Toc488412973)

[Solution 3](#_Toc488412974)

[Flow 3](#_Toc488412975)

[a) Source control 4](#_Toc488412976)

[b) Automated builds 4](#_Toc488412977)

[c) Static code quality analysis 4](#_Toc488412978)

[d) Unit testing 5](#_Toc488412979)

[e) Code Coverage 5](#_Toc488412980)

[Implementation 5](#_Toc488412981)

[a) Scenario 1: Java project 5](#_Toc488412982)

[b) Scenario2: Java project (Advanced) 5](#_Toc488412983)

[c) Scenario3: Android project 5](#_Toc488412984)

[Tool Stack 5](#_Toc488412985)

[Best Practice 6](#_Toc488412986)

[Benefits 6](#_Toc488412987)

[Anti-Patterns/Don’t's 6](#_Toc488412988)

[Continuous Deployment 7](#_Toc488412989)

[Introduction 7](#_Toc488412990)

[Continuous Delivery vs Continuous Deployment 7](#_Toc488412991)

[Problem 8](#_Toc488412992)

[Solution 8](#_Toc488412993)

[Flow 8](#_Toc488412994)

[Implementation 9](#_Toc488412995)

[a) Scenario 1: Java project 9](#_Toc488412996)

[b) Scenario2: Java project (Advanced) 9](#_Toc488412997)

[Tool Stack 10](#_Toc488412998)

[Best Practices 10](#_Toc488412999)

[Benefits 10](#_Toc488413000)

[Anti-Patterns/Don’t's 11](#_Toc488413001)

[Continuous Testing 12](#_Toc488413002)

[Introduction 12](#_Toc488413003)

[Problem 12](#_Toc488413004)

[Solution 12](#_Toc488413005)

[Flow 12](#_Toc488413006)

[Implementation 13](#_Toc488413007)

[a) Scenario 1: Java project 13](#_Toc488413008)

[b) Scenario2: Java project (Advanced) 13](#_Toc488413009)

[Tool Stack 14](#_Toc488413010)

[Best Practices 14](#_Toc488413011)

[Benefits 14](#_Toc488413012)

[Anti-Patterns/Don’t's 14](#_Toc488413013)

# Continuous Integration

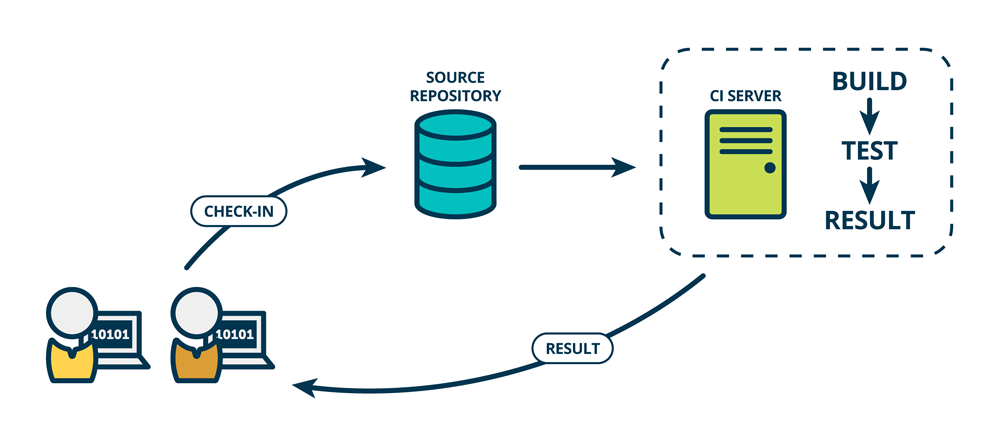
Introduction  
Continuous Integration is a practice that helps developers deliver better software in a more reliable and predictable manner. The developers commit their changes frequently to a shared repository which trigger a build. Unlike traditional practice of daily or weekly commit, frequent builds triggered from CI practice allow teams to detect errors quickly, and locate them more easily. Continuous Integration doesn’t get rid of bugs, but it does make them easier to detect and remove.

## The problem

How can we enable an early detection of a defect thereby ensuring quality and reduce time to market.

## Solution

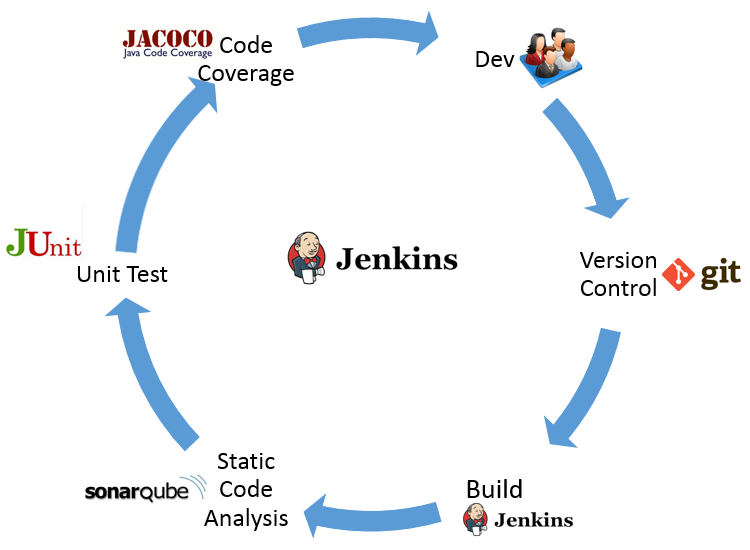
Continuous Integration practice ensures developers to integrate code frequently thereby significantly reducing the back-tracking effort to discover where things went wrong. Longer intervals in integration can make it exponentially difficult to find and fix problems. Such integration problems can make a project off-schedule, or cause it to fail altogether. In Continuous Integration, developers frequently commit to a shared repository thus reduce integration problems allowing rapid software delivery. The Continuous Integration automation enforce consistency and repeatability in the software delivery pipeline, as well as enhanced collaboration among teams.



Continuous Integration

## Flow

The Continuous Integration process maintain a single source repository and every commit done on integration branch trigger the application build. All artifacts required to build the application are stored and managed in the version control system and code is regularly checked in to ensure small increments between builds.

****

Continuous Integration flow

Continuous Integration cycle consists of the following steps:

### Source control

Source control is an integral part of Continuous Integration practice. It is where all source files, configuration, commits and history live and therefore it is where most of the integration happens between multiple developers in a team. All commits for a particular project must be made against a single mainline branch within a repository called the master branch.

### Automated builds

Every commit done on integration branch trigger the application build In Continuous Integration practice. Continuous Integration expects developers to commit frequently as it reduces integration issues and helps to detect and fix bugs.

### Static code quality analysis

Static code analysis, is a method of examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. In Continuous Integration practice, static code analysis is triggered by the CI engine.

### Unit testing

Unit testing is a level of software testing where individual units and components of a software are tested. The purpose of this test is to validate that each unit build in the CI process performs as designed.

### Code Coverage

Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage. Code coverage report ensures that application build in CI practice is tested properly.

## Implementation

### Scenario 1: Java project

A java code checked in to GIT repository trigger a maven build followed by Junit tests. The code quality is inspected and the build artifact (.war) is labelled ready for deployment.

### Scenario2: Java project (Advanced)

A developer checkout java code to his/her private workspace. With the necessary changes made, the code is checked in to SCM system (GIT) that trigger a Jenkins job which dynamically provision a slave. The maven build will run on the slave followed by Junit tests. The code quality is inspected and the build artifact is scanned for vulnerabilities. The artifact is validated against the vulnerability criteria and tagged fit for deployment. The artifact is later checked in to Artifactory and notification sent to stakeholders.

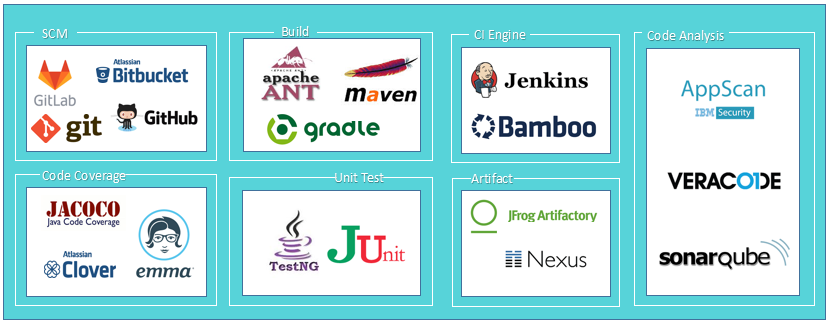
### Scenario3: Android project

A new change checked in to GitHub repository trigger an ANT build that build the project based on the build.xml file configuration. Jenkins manage AVDs (Android Virtual Devices) and execute JUnit tests and calculate code coverage with Emma tool. The test results are later published and email notification is sent to the stakeholders.

Prerequisite:

1. Jenkins CI server installed on your build server
2. Android SDK
3. Android Emulator Plugin for Jenkins.

## Tool Stack

****

## Best Practice

1. **Code Repository:** All artifacts required to build the project should be placed in the repository with minimum branching. All changes should be integrated to the trunk/mainline at regular intervals and the code has to be managed by strict version control policies.
2. **Build automation:** A single command should have the capability of building the system including integration and post build deployment.
3. **Self-Testing Builds:** Once the code is built, all tests should run automatically to validate that the software behaves as the developers expect it to.
4. **Quick builds:** To reduce the number of conflicting changes, developers should commit to the baseline frequently.
5. **Code quality:** Create comprehensive test suite at the unit level and functional level together with the code.

## Benefits

1. Integration issues can be easily detected as any change is built over the current state of the code base.
2. Continuous Integration help to notice trends in build success, failure, overall quality, and other pertinent information.
3. Faster builds as builds are triggered for every commit.
4. Ensure the delivery of a stable build.
5. Integrating code more frequently leads to reduced risk levels on any project.

## Anti-Patterns/Don’t's

1. Don’t keep the changes to a feature branch for longer period.
2. Don't check-in on a broken build.
3. Don’t check in untested code.

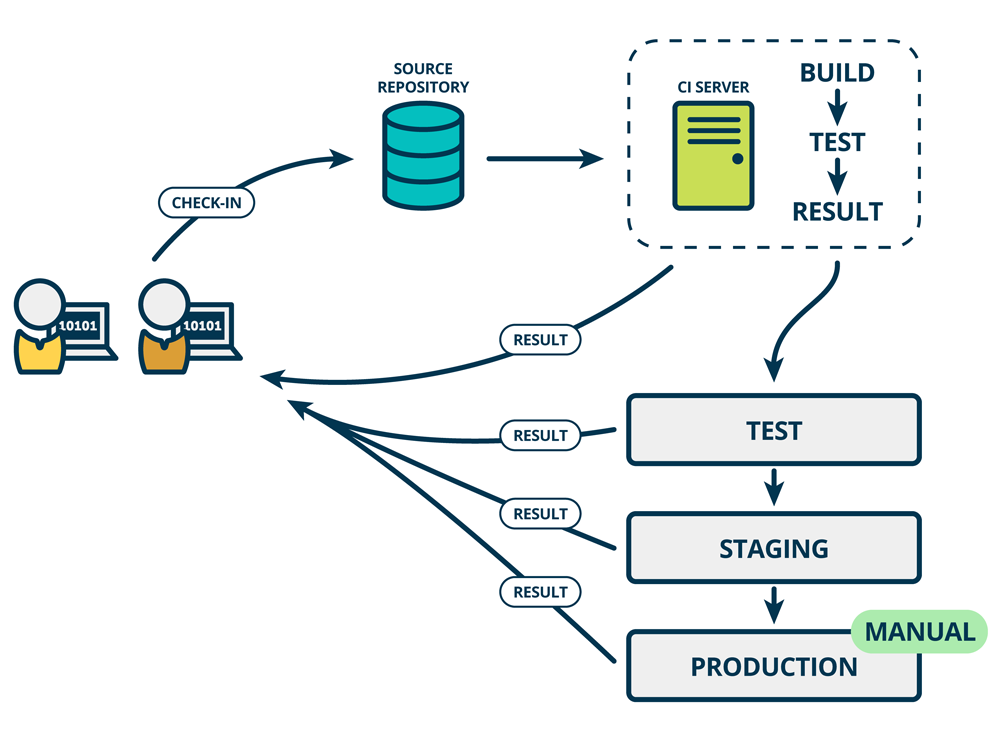
# Continuous Deployment

## Introduction

Continuous Deployment is a practice of deploying every change to production environment rapidly and safely after the execution of a set of automated tests. The application is first deployed to lower environment and later progressed to production passing though the quality gates. Continuous Deployment ensure that the product is always release ready and any change for the application is automatically pushed to production.

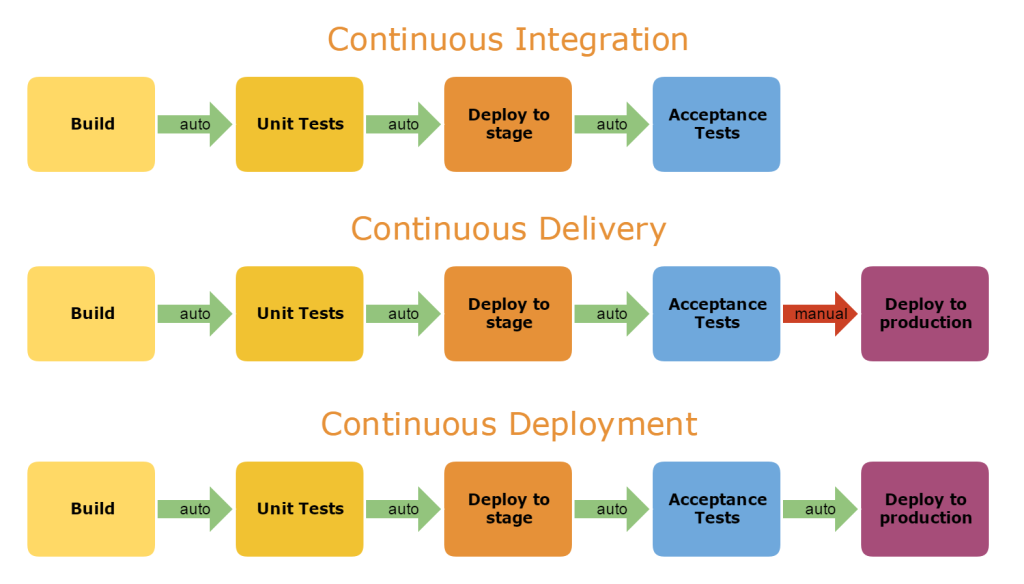
## Continuous Delivery vs Continuous Deployment

Continuous Delivery is a series of practices designed to ensure that code can be rapidly and safely deployed to production by delivering every change to a production-like environment. The practice also ensures that applications and services function as expected through rigorous automated testing and can be deployed to production with a push of a button.



Continuous Delivery

Continuous Deployment is the next step of Continuous Delivery. Every change that passes the quality gate is deployed to production automatically.



Continuous Integration vs Delivery vs Deployment

## Problem

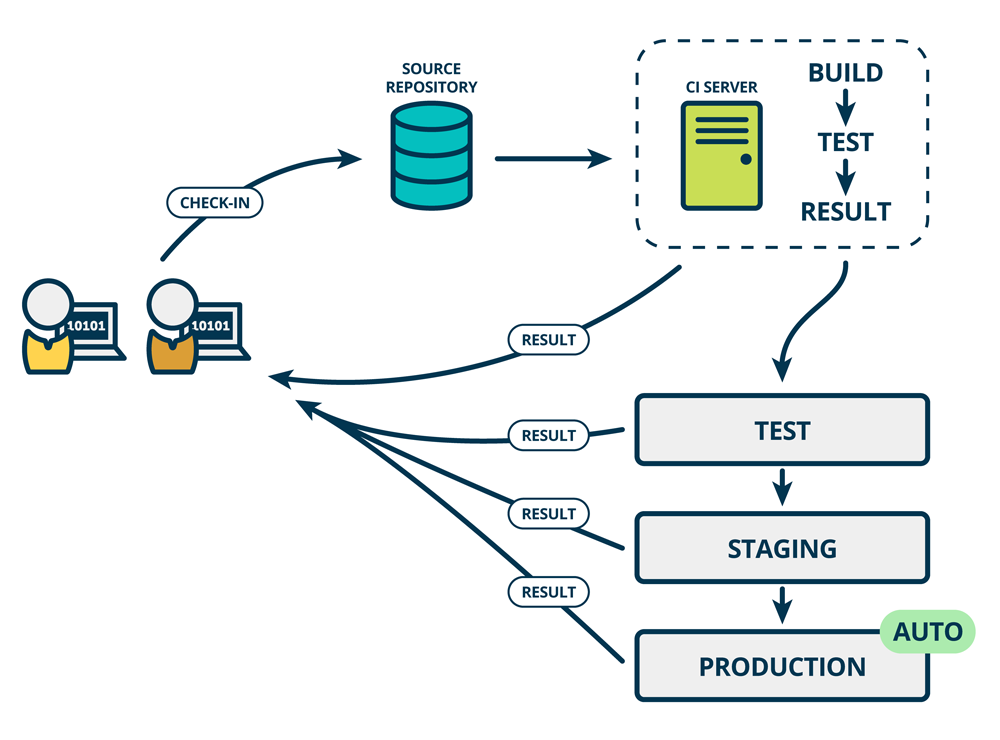
How can I ensure that the artifact build by Continuous Integration is deployed to production automatically.

## Solution

Continuous Deployment expands upon Continuous Delivery ensuring that the artifact build with respect to a change is tested and deployed automatically to production. The change is first pushed to a non-production testing or staging environment and gets validated against the security gates. With all the gates cleared, the change is deployed automatically to production.

## Flow

Continuous Deployment ensures reliable, low-risk releases and makes it possible to continuously adapt software in line with user feedback, shifts in the market and changes to business strategy. With Continuous Deployment, revisions are deployed to a production environment automatically without explicit approval from a developer, making the entire software release process automated.



Continuous Deployment cycle consists of the following steps:

1. The application build through Continuous Integration process is rigorously tested by running automated tests and deployed to lowest environment in the build pipeline.
2. The application deployed in the lowest environments is tested further and slowly progressed along the build pipeline based on the test result.
3. Constant feedback is collected from every environment and any failure is fixed and the application is finally deployed to production automatically.

## Implementation

### Scenario 1: Java project

A java code checked in to GIT repository trigger a maven build followed by Junit tests. The code quality is inspected and the build artifact (.war) is deployed along the build pipeline till pre-prod (production like) environment. The deployment is followed by acceptance test until the artifact is production ready. The artifact is then deployed to production automatically.

### Scenario2: Java project (Advanced)

The developer checkout java code to his/her private workspace. With the necessary changes made, the code is checked in to SCM system (GIT) that trigger a Jenkins job which dynamically provision a slave. The maven build will run on the slave followed by Junit tests. The code quality is inspected followed by Docker image creation for the application and the image is pushed to Docker registry. The image is then deployed to a container orchestrated by kubernetes followed by functional tests. The image deployments progresses thought different stages of the build pipeline along with the tests and quality gate validation and finally deployed to production environment. The notification is sent to the stakeholders and the deployment dashboard is updated.

## Tool Stack



## Best Practices

1. **Version control:** Version control every artifact to be deployed and the process defined for deployment. This process ensures reliability as there is just a single source of truth.
2. **Single build:** The same artifact need to be deployed to all the targets of the pipeline. There should not be separate builds specific to a stage. This onetime compiling eliminates the risk of untracked differences due to various deployment environments and third-party libraries.
3. **Consistent deployment process:** An inconsistent deployment process can become a source of configuration drift across environments. The same set of steps must be repeated from start to finish for all environments.
4. **Deployment pipeline:** Create a deployment pipeline starting with the lowest critical environment till production with entry and exit gates for validating the progression through various stages.
5. **Deploy into a copy of production:** Create a production-like or pre-production environment, identical to production, to validate changes before pushing them to production. This will eliminate mismatches and last minute surprises.
6. **Smoke-Test your deployments:** A non-exhaustive software test that doesn’t bother with finer details but ascertains that the most crucial functions of a program work, will give a confidence that application is decently stable.

## Benefits

1. **Deliver software with fewer bugs and lower risk:** Frequent and smaller change releases help to catch errors much earlier in the development process. Automated testing at every stage of development ensure that failed code is not passed to the next stage. And it’s easier to roll back smaller changes when needed.
2. **Frequently releases:** Releasing new features early and often enables effective feedback, giving the ability to iterate and learn from your customers.
3. **Quick response:** Market conditions change constantly. Continuous Deployment enables to understand and adapt to market needs faster.
4. **Life is saner:** Continuous Deployment means the responsibility for software delivery is distributed much more widely, and this shared responsibility and collaboration make life better.
5. **Smooth deployment:** Make deployments frictionless without compromising security.
6. **Improves efficiency:** Integrate teams and processes with a unified pipeline thereby improving efficiency.
7. **Better visibly:** Provide a holistic view on the deployment status across all applications and environments.

## Anti-Patterns/Don’t's

1. Don’t just focus on the tools. Understand the current process and pain points.
2. There is no single solution for all scenarios. Pick the right tool and define the process with a holistic picture.
3. Avoid building monolithic applications leading to long development, test and deploy cycles.

# Continuous Testing

## Introduction

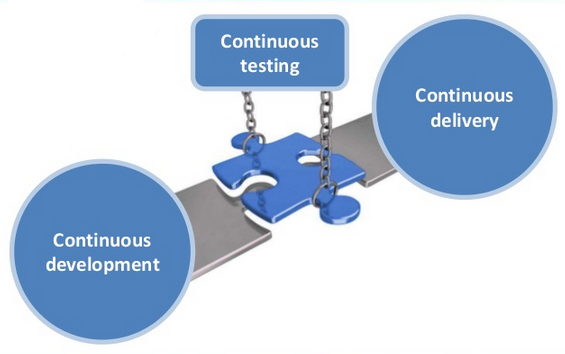
Continuous Testing is the process of executing automated tests as part of the software delivery pipeline in order to obtain feedback on the business risks associated with a software release candidate as rapidly as possible. It evolves and extends test automation to address the increased complexity and pace of modern application development and delivery. Continuous Testing is seamlessly integrated into the software delivery pipeline and execute the right set of tests at the right stage of the delivery pipeline.

## Problem

How can testing be integrated with build process and track issues early in the SDLC process.

## Solution

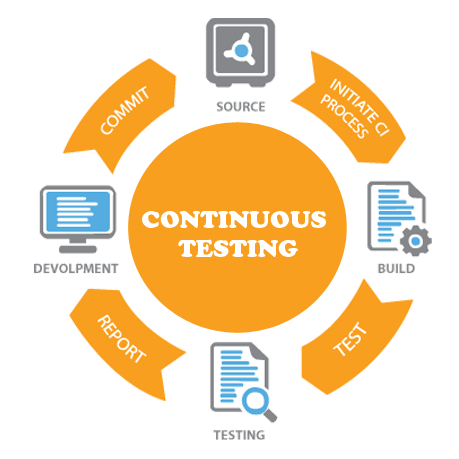
Continuous Testing automate the testing process and integrate with build process as early as possible. It can automatically execute tests after Continuous Development and ensure that issues are almost identified immediately. Continuous Testing enables continuous improvement by providing fast and continuous feedback regarding the level of business risk in the latest build. This information can then be used to determine if the software is ready to progress through the delivery pipeline at any given time.​



Continuous Testing, when implemented diligently, allows flawless continuity in the delivery cycle. Integrating a fully automated Continuous Testing process into the SDLC is the most effective solution for a successful DevOps process.

## Flow

Continuous Testing provides instant insight on whether a release candidate is too risky to proceed through the delivery pipeline. Continuous Testing expects testing to be embedded within the development process, not tacked on at the end.



1. **Follow CI:** Continuous Testing triggers with the completion of Continuous Integration which begin with testing performance and functionality of the code developed using tools like Selenium. Tests can include functional, performance, UI, API and more along with regression tests to ensure a comprehensive test coverage.
2. **Suite creation:** Test suite is created and stored in a version controlled system along with the code.
3. **Test Initiation:** Sanity test is performed on the application deployed to the stage/test environment.
4. **Automated NFR Testing (Load and Performance Testing):** Perform Load test, Performance test, Security test and capture its results.
5. **Verification Approval:** Auto approve/reject the release based on test results. The application is progressed based on the test results.
6. **Test Closure:** Archive the test results.

## Implementation

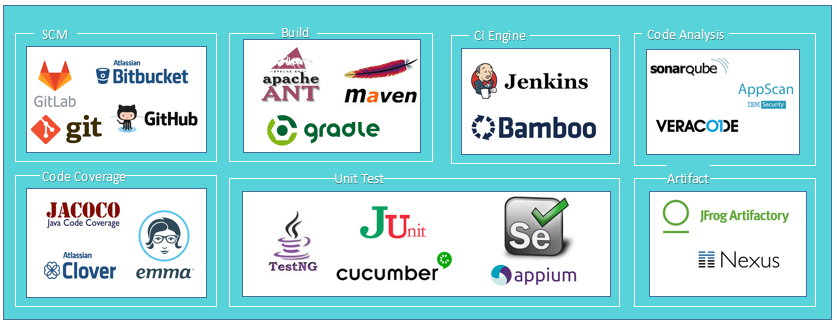
### Scenario 1: Java project

Once the deployment is completed, the test suite is triggered by CI engine and executed on the application deployed. The test result is published and the build is progressed to later environments based on the test result.

### Scenario2: Java project (Advanced)

The test plan is created and checked in to a version control system. Once the deployment is completed, test suite is triggered by CI engine on the application deployed. The test result is published and the result is validated against the gate parameters. When the gate is passed, the build is progressed to later stages of delivery pipeline. Additional post-live monitoring can further enable Continuous Testing.

## Tool Stack



## Best Practices

1. **Lean testing approach:** Follow a lean approach reducing unnecessary testing artifacts, such as extensive test plans and test cases. Also collaborate developers and testes to ensure effective unit testing
2. **Continuous review:** Continuously review and optimize the test suite to eliminate redundancy and maximize business risk coverage.
3. **Versioning:** Upload and version all test suite and test plan on the SCM system storing source code.
4. **Collaborate With Business:** Build a close relationship between QA and Business Analysts. Remove any ambiguity in user stories and include acceptance criteria.
5. **Incremental tests:** Tests should be incremental and repeatable

## Benefits

1. **Faster releases:** The process of Continuous Testing ensures that cleaner code is released to end users in a timely manner.
2. **Reduce risk:** Identifying the problem is much easier when Continuous Testing is performed and the issue is caught immediately so problems can be corrected.
3. **Maintain consistency:** Continuous Testing makes it easier to achieve consistency by providing a carefully configured environment and maintaining the same correct configuration for all relevant tests.
4. **Improves speed to market:** Organizations can test potentially releasable software very early in the lifecycle allowing the client to understand the quality of development thereby enabling faster release.
5. **Improves quality:** Continuous Testing helps improve the quality of the application while verifying each component at the development phase through more automated tests.

## Anti-Patterns/Don’t's

1. Don’t create monolithic test suites as they are difficult to maintain and increment.
2. Don't automate from day one as requirements and design take time to stabilize.