Topic -2 Principles of software delivery

The various principles of software delivery include:

1. Create a repeatable, reliable process for releasing the software.
2. Automate all the possible phases or almost everything.
3. Keep everything in Version control.
4. With Tough and difficult Testing phases, do it more frequently and bring the pain involved continuously forward.
5. Build and ensure quality.
6. Stating as "Done"- demands is to Complete the delivery process by adding value to the users.
7. Ensure collaboration and everybody is responsible for the delivery process
8. Ensure continuous improvement.

1. Create a repeatable, reliable process for releasing the software.

The release of a software after several verification and testings should be easy, because you have tested every single part of the release process hundreds of times before. The properties of repeatability and reliability is derived from two principles.

a. Automate almost everything.
b. Keep everything related to the built, deploy, test and release of the application under version control.

The software deploy includes three things.

a) Provisioning and managing the environment, which include the hardware configuration, software, infrastructure and other external services in which the application needs to be run.
b) Install, the correct version of the application into the environment.
c) Configure the application (deployment settings) to be launched, which includes all the data, or any state, it requires for its implementation.

The entire deployment of the application is implemented in a fully automated process from the version control, with its necessary script and states, kept in version control or database. Hardware cannot be kept in version control, but definitely the cheap virtualization technology and the tools like puppet could make the provisioning process fully automated.

2. Automate all the possible phases or almost everything.

Though, it is impossible to automate everything, but all most near to everything can be automated. The acceptance testing, the database upgrades and downgrades can be automated. the network and firewall configuration can be automated. But the exploratory testing done by experienced testers, that involves demonstration of the working software to representatives of user community, cannot be performed by the computers. This is done for approval and performances compliance, needs definite human intervention. By all means, automation is a prerequisite for the deployment pipeline. Because it is through the automation that the end users are guaranteed to get delivered with what is to be delivered at a push of a button.

3. Keep everything in Version control.

All the various stages that include build, deploy, test and release of the application should be kept in version storage. This includes the requirement documents that include technical documentation, automated test cases, database creation, Upgradation, Downgradation, the Libraries, and tool chains. It also includes the Test script, network configuration script, deployment script, Initialization script, application stack configuration script, and so on. All these documents needs to be in the version control. All such details require to be stored under a single identifier, such as a build number or a version control change set number. This enables anybody to build the application deployed into any environment. And gets to know which version is used from the version control.

4. With Tough and difficult Testing phases, do it more frequently and bring the pain involved continuously forward.

In the deployment process, the testing is a very painful process. If it is done just before the release or towards the end, it disastrous and will create problems, it is required to do it continuously from the beginning of the project. Every time, when changes are made, and it passes all the automated tests, the aim should be to release it every time. The release should be made in a production like environment. So that in the end of deployment, the software has gone through a series of strenuous test and we are confident to deploy and release the product in reliable and repeatable way with success.

5. Build and ensure quality.

The technique which is described to build quality is continuous Integration with comprehensive automated testing and automated deployment, which is designed to catch the defects as early as in the delivery process. Whenever the defect is detected, the next step is to fix them. The delivery team must be disciplined about fixing the defects as soon as they are found. So as to ensure quality delivery that satisfies the end customers.

6. **Stating as "Done"- demands is to Complete the delivery process by adding value to the users.**

Any feature can be told as "done" or completed only when it's delivering value to the users which is the major motivation behind the practice of continuous deployment with continuous integration. But normally the term "done" means 'released into production'. Actually the functionality can be told as "done" once it is being successfully, showcased or demonstrated to, and tried by the representatives of the user community in a production like environment. There is no a case like 80% done, because either it is done or they are not.

7. **Ensure collaboration and everybody is responsible for the delivery process**

All the people in the organization, should be aligned with the goals and should work together to help each other to meet the goals. Normally, in many projects, the reality is developers pass their work to the testers, the testers pass their work over the operation steam and the release time. Whenever something goes wrong, people spend time blaming each other. Contrary to this, we need a system which can be seen by everybody in the team at a glance can see the status of the application, how it works, it's various test it has passed, the state of the environment in which it is deployed. Under a collaborative situation with greater collaboration, all members need to be involved in the software delivery in order to release valuable software faster, at the same time, more reliable.

8. **Ensure continuous improvement.**

All the applications in continuous integration model evolve continuously accompanied by a number of releases. The entire team needs to regularly gather together to retrospect the delivery process, which means they should reflect on what has gone wrong, what is working, and discuss the ideas to how to improve things. This is known as continuous improvement for reliable, Continuous delivery of software products to the end users which adds value to the users.

Topic -3 Configuration Management

Configuration management is also known as *version control*. Configuration management refers to the process by which all the *relevant artifacts of a project* and the relationships between them are stored. The stored version is uniquely identified by the *build number* and can be updated and modified. In the configuration management strategy, the records stored identify the *evolution of*

system, where the details of applications are recorded to *manage and identify the changes* that happen in a project along its development life cycle.

A good *configuration management strategy* should identify the below listed aspects:

a. The ability to exactly ***reproduce any of the environments*** that include the operating system- its patch level, network configuration, the software stack used and the applications deployed.

b. The ability ***to integrate or make any incremental change*** to the individual items and deploy the change to any or all of the configurations in the environment.

c. The ability **to *reproduce the change***, occurred in an environment, and trace it back, to see the change, who made it and when it was made.

d. The ability to ***satisfy all the compliance regulations***, subject to end user satisfaction.

e. The *ability* ***to update or pass information*** regarding the changes to all team members.

The configuration management is **addressed in three areas**.

i. **Manage the applications**: Manage everything by **storing** the details with the process of *build, deploy, test and release* in the version control storage and manage the **dependencies.**

ii. **Environment configuration:** The environment configuration management refers to *managing the software, hardware and infrastructure related with the applications* like OS, application servers, DBs and any external services.

iii. **Using version control**: The version control system/*source control* is used to keep multiple versions of a file in the repository. The first popular version control system is a unique tool called *Source Code Control System (SCCS).* This became *Revision Control System (RCS)* and later *Concurrent Versions System (CVS).* Different open source and proprietary version control systems are available. The *Subversion, Mercurial and Git are* open source systems*.* The main aim of a version control system is:

- It retains and *provides access to every version* of the file stored in it.
- This also has the provision to *store metadata* that includes information that describes stored data.

- Also, this allows the teams which are distributed across the space and time to *collaborate among each other*.

A version control stores all the artifacts related to the creation of a software which includes the source code, the test cases, the database scripts, build script, deployment script, documentation, libraries and application configuration files, the compiler and collection of tools etc. This also stores information0020required to recreate the testing, production environment, in which the application runs in the *project history*. The testers use it to store the test scripts and procedures the project manager stores. The release plans, progress charts, risk logs, etc.

The version control is useful for speeding up the creation of new environment, ensuring the base configurations definition. And also allows the user to *recover, the exact snapshot of the state of the entire system* that existed in the development environment to the production environment and to step back to a recent known good version of the various artifacts as required. Version controls benefits more with good number of updations and commits done regularly.

The addition *of new updates can be stored as separate branches* and once the changes are satisfied, these branches are merged into the development branch. The integration is a very complex and may create conflicts. Though Automated merging tools can be used to avoid semantic conflicts, though the better approach is to commit the main trunk on a regular and frequent basis to avoid large merge conflicts and integration problems resulting the development of higher quality software.

The *version control system supports to add description to each of the commit messages* with details of the built breaks, the problems associated with the build, who has created the break and why it happened. Providing sufficient description in the commit messages will make complex debug problem simpler under tight deadlines situations.

Managing dependencies

The common dependencies to be managed include ***managing the external dependencies*** and also ***managing the components in a development process*** in a large project. The most common external dependencies within an application is the *third party libraries*, deployed in the form of binary files which are not changed by the application development team. The relationship between the components and modules of a project are underactive development by the team and change quite frequently. This can be managed by installing the libraries globally on this system using a package management tool like Ruby gems or pearl modules. The organization can maintain a repository of

approved version of libraries to be used. Large projects are split into small components for more efficient development process.

Managing Software Configuration

Managing the configuration of a software is key part of an application. The ***change in configuration information***, affects/determines the behavior of the software during the deploy and run time. The configuration options are *set by the delivery team* and should ensure the consistency of the configuration across the components of the application and technologies.

In terms of **configuration flexibility**, *ultimate configurability* is in demand, but this comes with a cost and many significant pitfalls. High *configurability results in a complex system* and high effort are involved. This reduces the benefits in terms of flexibility. Therefore, it is always *better to focus on delivering the high value functionality*, with little configuration and then to add configuration option later, when it's necessary.

The application receives the configuration information at various points like, the *build time, packaging, deploy time, testing and* release process. **The various types of configuration are:**

- *During the build time*, the build scripts can pull configuration information into the binaries.
- *During the packaging*, the packaging software can inject configuration information into assemblies, ears (Enterprise Archives) or gems.
- *During the installation process,* the deployment scripts launched or the installers/Agent softwares used can fetch necessary information and pass it to the application.
- Ultimately, the application itself can fetch configuration at startup time or run time.

Always the changes between the deployments needs to be captured as configuration when the application is compiled or packaged.

Accessing Configuration Information

- The easiest way to access the configuration information of an application is through the *file system*, as file system is cross-platform and can be supported in every language.
- An alternative is from a *centralized repository such as a RDBMS, LDAP or a web service*. ESCAPE is an open-source tool to access the configuration information.
- Configuration information can be also fetched by performing *HTTP GET* operation.

Modelling Configuration Information

The applications configuration information is modeled as a *tuple*. Therefore, the configuration as a *set of tuples* and their value depends on the application itself, version of the application, and the environment in which it runs.

Configuration of the application *at the* ***deploy time*** requires to know which is the database messaging servers or external systems that the application belongs to. If the ***runtime configuration*** of an application is store in a database, it should it should pass the database connection parameters to the application for the start-up at the deploy-time. In a ***production environment***, the deployment scripts can fetch the configuration information and supply it to application. In ***package management***, the software's default configuration will be present in the package, but sometimes it needs to be overridden at deployment time for testing purpose. Finally, to configure the application at the ***startup time****, the* ***runtime startup configuration*** is supplied from the environment in the form of *environment variables* or as arguments to the command line, used to start the system. This includes the registry settings, database values, configuration files or any external configuration service.

While ***managing the applications configuration*** information, three main things include (i) How to represent the configuration information (ii) How do the deployments scripts access it (iii) How does the configuration information vary between environments, applications and versions of applications?

The choices used to store the application information of the software are *database, a version control system, or a directory or registry*. From which the easiest option is a version control system. Application specific configuration information regarding the testing, production and environments is stored in a sub-repository separate from the source code is the best way. The *advantage of storing the configuration information and databases directories and registries* are because they are convenient places to access the configuration even from the remote place. Storing the history of changes to the configuration is also useful for the purpose of audit, as well as rollback.

To access the configuration information, it is always efficient to manage the configuration using a central service through which every application can get the configuration it needs. This is true for both packaged software's and other software applications used.

i) Modeling the configuration information, the ***use-cases include adding a new environment*** which includes a new developer workstation or a capacity testing environment. In this case, a new set of values for the application deployed into this new environment needs to be specified.

ii) Also, while creating a ***new version of the application***, it should be ensured that the new version is deployed to production to get its new settings.

iii) While ***promoting a new version of the application from one environment to a different environment***, it should be ensured that the new settings are available in the new environment.

iv) While ***re-locating a database server***, it should be updatable. Every configuration setting that references the first database should point to the new one.

v) Finally, to ***manage the environment using virtualization***, the virtualization management tool should be able to create new instance of a particular environment with Virtual machines, configured correctly.

Similar to the way an application and the build script needs testing, the configuration settings also need testing. There are two parts to testing the configuration are: a) In the first to stage, it should ensure that the references to the external services in the configuration setting are fine. b) The second stage is to do Run test as once the application is installed, it is to make sure it is operating as expected, this includes some test exercising the functionality which depends on the configuration setting.

In order to manage the *configuration across the various applications* in an organization, a catalog of all the configuration options of each of the applications, has to be maintained. It should be possible to automatically generate the configuration information from the respective applications code as part of the build process. The applications configuration information can also be accessed from the **operations team production monitoring system**. Which should have the information such as version of each application deployed in each environment. The tools such as Nagios, openNMS provide services to record such information.

Principles behind managing Application Configuration

The application configuration information should be treated similar to the code and should be managed properly and tested.

1. Decide the *stage in the applications life cycle to be used to inject the configuration information*, whether it is the packaging, deployment or installation or at the startup time, or at runtime.

2. Keep all the configuration information of the application in the same repository as where the source code is kept, But the *passwords and other sensitive* values need not to be stored in Version Control.

3. The *configuration should be always managed by an automated process* using the values taken from the configuration repository.
4. The configuration system should identify different values to different applications that includes its packaging, installation and development scripts based on the application, its version and the environment in which it is deployed.

5. Avoiding cryptic naming, *clear naming conventions* needs to be used for the configuration options.

6. Each configuration information should be modular and encapsulated so that changes in one place do not affect unrelated pieces of configuration.

7. Define all the concepts, to *define the elements in each concept using a single representation* in the configuration information.

8. Keep the *configuration information to minimal and as simple as possible*, keeping configuration only where there is a requirement.

9. *Avoid over engineering the configuration system*.

10. *Testing needs to be ensured for the configuration that run at the deployment or installation time*. Also test the services that are required by the application are available.

Managing the Application Environment

Any Applications *depend the hardware, software, infrastructure, and external services* to work and forms the Application Environment in which it works. Therefore, while an application runs, *managing the environment of the application is an important task*.
For example, if the application being installed depends on a *messaging bus,* it is important that the bus needs to be configured correctly or the application will not work. Similarly, the *operating system* configuration is very important. All the *details of the previous configuration in which the application has worked needs to be preserved*. For any reason, if the new configuration doesn't work, it's difficult to return to a known good state if the record of the previous configuration is not available.

The problems related with Managing the Application Environment are listed below.

- The *large collection of configuration information*.
- Any *small change leading to the break of the whole application* or can degrade its performance.
- Once application is broken, finding the problem and fixing it takes time and requires senior personal.
- Difficulty in precisely reproducing the manually configured environment for testing purpose.

Therefore, a **fully automated process for *managing the environment*** *of the application is* the key to aspect. An Automated Application management can solve the issues like,

- *Storing the environment configuration* related details in a repository will remove the problem leading to stoppage leading to a significant downtime.
- Long time to Fixing any problems with an environment.
- Need to rebuild the environment in a predictable amount of time by using the copies of configuration and the respective version stored in the repository, to reinstate the application in a good state in a decent time.

Different types of configuration information related with an application.

- The various operating systems in the environment, that includes the operating system version, patch levels and configuration setting.
- The additional software packages. Installed in and enrollment to support the application.
- The networking topology required for your application to work.

- All the externals services that the application depends on including the version and configuration.
- And finally, any data or other state that is present in the production databases.

Effective, **configuration management strategy follows 2 principles**.

- Keep the binary files, independent from configuration information and keep all the configuration information in one place.
- An environment that is in a properly deployed state is known as *baseline configuration*. The automated environment provisioning system must be able to establish, reestablish any given baseline that has existed in the reason history of the project. It stores the changes creating new version of the baseline.

There are several ***tools to manage the environments*** in an automatic way. *Puppet and cfEngine* are two examples of tools that manage operating system configuration. The agents running on the various machines will pull the latest configuration and update the operating system and the software installed on it on a regular basis. *Virtualization* can improve the efficiency of environment management process.
Instead of creating new environment in an automated process, Virtualization can copy each box in the environment and store it as a baseline. By creating this new environment, it can be done by clicking a button to manage the change in the environment. All the changes recorded in the repository must be tested before it goes into the production.

Managing the change process in an environment.

The changes in a set production environment should be completely following the organization's change management process. All the change has to be approved and taken to the production environment in an automated fashion. Any change in the environment should go through all the faces as similar to the software like built deploy, Build and release process. Configuration management is the foundation of the deployment cycle and this possible with continuous integration release management and deployment by planning. This makes collaboration possible and a positive impact on the delivery teams. This allows complete recreation of the production system using the version-controlled assets providing the possibility to revert to a known good state of your application.

Topic-4 Continuous Integration

In the *manual Software development process*, a significant proportion of the development time is in unusable state. Until the developer checks in the changes or run an automated test, nobody starts the application in a production like environment. Therefore, the projects retain branches, for a longer period of until the end. This causes acceptance testing to be deferred until the end, and the integration period takes extremely long time. Contrary to this, in an automated test environment, projects spend most of the time working with the latest changes. This is possible with the continuous integration process. In a continuous integration model, it requires to commit any change, whenever

the change is made, the entire application is built, which undergoes a comprehensive set of automated test. In this process, If the development or built or test process fails, the team stops then and there and fixes the problem immediately. The *goal of continuous integration is that the software is in working state all the time.* The *continuous integration uses* regular integration means integration all the time, meaning, every single time anybody commits any change to the version control system. Continuous integration represent efficiency that is able to deliver software, much faster with fewer bucks in a collaborative environment. The bugs are caught *much earlier in the delivery process*, when they are cheaper to fix providing *significant savings in time and cost*.

Implementation of continuous integration

The practice of continuous integration requires certain *prerequisites and tools* are available to perform continuous integration.

Requirements to needed start: Three things are required to start with continuous integration.

- Version control.
- Automated Build
- Agreement of the team.

Version control: All the artifacts in the project must be checked into a single version Control repository, that includes code, test cases, database script, build and deployment setting scripts and everything needed to create, install run, and test the application.

Automated Build: We must be able to start the build of the application from the command line with the ability to run a series of tests with the collection of multi-stage build scripts that calls one another. This mechanism needs either a person or a computer to run the built test and deployment process in an automated fashion from the command line. The advantages include the ability to build the process in an automated way from the continuous integration and environment which can be audited. All the build scripts should be tested and should be easy to understand maintain and debug. This allows better collaboration with the operation's people.

Agreement of the team: The continuous integration is a practice where everyone needs to check in any small incremental changes, very frequently to the main line and agree that the highest priority task on the project is to fix any change, that breaks the application.

There are several ***continuous integration tools*** available the open-source options include Hudson, Cruise control family with Cruise control, cruise control.NET, and cruise control.rb. The Cruise control is a lightweight Continuous Integration Software written in Ruby. In a continuous integration system. Once the continuous integration tool of choice is installed, based on the preconditions, it should be possible to get started in a few minutes finding the source control repository executing the script to compile and the automate the commit test required for the application. Once all the actions in the latest change is made the next step requires to use continuous integration server. On ready to check-in the latest change,

- It requires to check whether the built is already running. If not finished, wait for it and if the already running built has failed, need to wait till the built is corrected.

- Once it has finished and the tests have passed, then update the code in the development and the environment from the latest version from the version-controlled repository.
- Before passing the updated code to the repository, it is mandatory to run the build script and test on the development machine to verify everything is working correctly on the developer's computer.
- If it passes the local built process, then check in the code into the version control.
- Now wait for the continuous integration tool to run the build with your changes if it fails stop the process and fix the problem immediately on the development machine and go back to step 3.
- When the build passes, move to the next task.

Pre-requisites of continuous integration: The **Pre-requisites** of continuous integration include:

- Check in regularly
- Creation and use of comprehensive automated Test Suite
- Use of short Built and Test process.
- Managing development workspace

***Check in regularly*:** The most important practice in continuous integration is frequent check-ins to the trunk, or mainline at least a couple of times in a day. This reduces the chance of build break. And if it breaks also, it gives a chance to revert back to the recent non-good version. Ensuring that the changes altering a lot of files unless likely to conflict with others work.

Create a comprehensive automated test suite: In the absence of automated test suite passing a built process *means only that the "application ready to be compiled and assembled"*. But the automated testing provides the confidence that the application will actually work. There are *different types of automated tests* available. Such as:

- ✓ Unit Test
- ✓ Component Test and
- ✓ Acceptance Test.

The ***Unit tests*** are written to test the behavior of small pieces of the application in isolation such as a *method or a function or the interaction between small group of methods or function*. The Unit Test can usually run without starting the whole application. They do not primarily hit the use of database, the file system, or the network, and does not require the application to be running in a production like environment. Where as in the ***Component Test***, it tests the *behavior of several components of the application* in combination. This does not require the starting of whole application. But they may use database file system or other systems and also *take longer time* to run compared to unit test. The ***Acceptance Test*** tests that the application meets the acceptance criteria decided by the business. This guarantees that the application *can perform in terms of functionality proposed* and also provides other characteristics like capacity, availability, security, etc. The acceptance testing takes longer amount of time. The three set of tests combined provide extremely high level of confidence that any change introduced will not block the break the existing functionality.

Keep the built and test process short: Taking too much of time for building the code and running the Tests has several problems as the people will not want to do the full build and run the test more in number. This results in the built and run process less often and leads to build failures. Later when

before taking a next build, would have completed multiple commits, giving more chances to break the build. Therefore, *the compile and test process should be done on a continuous Integration server and should take only few minutes to complete.* The modern CA servers makes it easy to stage the build, run multiple tasks concurrently and aggregate the results. So that the results of each state of the built can be seen.

Managing the development workspace: Managing the development environment should be done carefully, so that they are able to run the built comma, execute the automated test and deploy the application in an environment under control. Running any application in a local development environment should use the same automated process, which is used in a continuous integration and testing environment or in a production environment.

- ***Managing the workspace*** also requires the *careful configuration management*. This requires managing the source code, test data, database script, build script, and deployments script all of this must be stored in the version control. The Version Control should enable smooth and easy reverting to a known good state.
- The second step in the ***managing the development workspace*** requires configuration management of third-party dependencies, libraries and components, with the correct versions of the libraries or component suitable for the work in the version of the source code is required. Several open source. Tools are available to manage third party dependencies like Maven and Ivy.
- Finally, **automated test** must be run on developer machines.

Use of Continuous Integration Software

The most basic functionality of a continuous integration software is to *pull the version control system*, by verifying for any commits that has occurred, and if so, checks-out the latest version of the software to run the build script, to compile the software, run the test, and then notify the results to the developer. The Basic operation is done by the continuous integration server. The Software has got two components which include:

- Execution of a simple workflow at regular intervals.
- Viewing the results of the process that have been run notifying, the success or failure of the built and test runs and providing access to test reports installers etc.

The major task of the CI workflow is to pull the version control system at regular intervals. Detecting any change, it checks-out a copy of the project to the directory on the server or the directory on a build agent. This is followed by the execution of the script specified that executes the command to build the application and run relevant automated test.

The CI Server includes a Web Server to

- Show the list of builds run or completed.
- Show the reports that define the success or failure of build run or completed

The build instructions culminate or leads to the production and storage of artifacts such as binaries or installation packages for the testers and clients to download as the latest good version of the software. Beyond the basic functionality, the continuous integrations has provision to get the status of the most recent built by sending it to external device. It uses mechanism like

- flashing lights
- sirens into action to indicate the progress of built processes
- Another way is to use text to speech to read out the name of the person who has resulted in the block of a build.
- Some CI Servers display the status of build along with the avatars of the people who checked in displayed on the big screen.

These methods allow everyone to see the status of the build completed for visibility in using the CI server and are very useful for teams that work in a distributed environment.

There are ***certain essential practices to ensure the basic objective of CI system***, which make sure that the software is working all the time. These practices include:

i) Never check in on a broken build
ii) Always run all the commit test locally in the Developer machine before committing to CI server.
iii) Complete all the commit tests before moving on to the next new task.
iv) Never leave a build broken without attending it
v) Always maintain the preparation to revert to a previous known safe revision of the application.
vi) Fix Time duration before performing the reverting
vii) Never comment out a failed test
viii) Find it Important to own the responsibility for any breakages that result from the changes done by you.
ix) Maintain a test-driven development.

Never check-in onto a broken build: In a continuous integration environment when the built breaks, the developers stay responsible and should wait to fix it by identify the cause of breakage and should fix it as soon as possible.

It is always a good practice to run all the ***commits locally before committing it to the repository***. This means that the developers should initiate a local built and run the commit test and only when this is successful, the developer should commit the changes to the version control system. This can help to identify the problem without breaking the whole application. In addition to committing the changes to the repository, addition of all new artifacts required to build the process is essential like the configuration file.

Many of the modern CA servers offer feature known ***as pretested commit*** personal built or preflight build using this facility instead of checking the built process by the developer itself. The CA server will take the local changes, run a built with them on the CI grid. If the built process passes, the CA server will check the changes. And if the build fails, it will update the developer regarding what went wrong.

Complete all the commit tests before moving on. The CA system is a shared resource for all the team members. Though, the build breakages are normal and expected part of the process, the CI system is aimed to find errors and eliminate them quickly as possible. Until all the check ends are compiled and all the commit tests are passed. The developers should not start into a new task.

Never leave a broken build in the process. Leaving a built broken will take significantly longer time to understand and fix the problem. This will generally result in the magnification of the problem especially in distributed development setup. ***Therefore,*** it is always better to fix the break or problem immediately.

Be Prepared for revert operation: While fixing a problem or broken belt, if we are not able to find where the problem lies, it is always important to get everything to working stage as quickly as possible. For whatever reason, not able to do so. Reverting back to a previous change set held in the revision control and to find the remedy for the problem in the local environment. if we are not able to find where the problem lies in a fixed amount of time***, it*** Is always recommended fix it by reverting it back to a safe assured state.

Fix Time duration before performing the reverting: The continuous integration always has a team role that whenever the built breaks. Fixing should be as soon as possible within 10 minutes. And if you are not able to finish for a solution, then always have the decision to revert back to a previous version from your version control system.

Don't comment failing test: While dealing with a built break or solving a problem, the developers always comment out the failing test in order to record their changes. But It is always recommended to comment occasionally in some serious development work or otherwise in a case which requires some extended discussion with the customer or when a test is pending out.

Take responsibility for all the breakages which has resulted from your changes: Even when all the changes are committed and that test return has passed. But while integrating, if the others code has broken. Then it is a responsibility of the developer who has made the change and ha to fix it at the earliest. All the tests that are not passing as a result of the change, which is made by the developer. Therefore, to do the continuous integration effectively, the team needs to access needs to have access to whole of the code base.

Test driven development: There are several comprehensive tests suite essential for continuous integration. This automated test suits provide fast feedback, which is extremely core for continuous integration. Providing excellent unit testing, acceptance test in a test-driven development. The ***test developed test driven development*** means that when developing a new piece of functionality or fixing a bug, the developer first creates a test that is an executable specification of the expected behavior of code to be written.

Distributed teams and Distributed VC system Versus Centralized continuous integration.

For continuous integration with A distributed team, the team is not sitting together in the same room but working from different times zones. In a technical perspective, this uses a shared and continuous integration system. The distributed teams can impact, the continuous integration

process in different ways. If the team in San Francisco breaks, the bill and leaves home, this can seriously impact the team in Beijing. In very large projects with distributed teams' tools like Voice Over IP (VOIP) such as Skype for example and instant messaging is of importance to enabled fine-grained communication to keep up things running smoothly. It is also possible to do retrospectives showcases, stand-ups and other regular meetings using video conferencing.

The *Distributed version control systems (DVCS) like git and mercurial* makes it easy to pull patches back and forth between developers and teams manage Branches and merge work streams. DVCs also allows easy working in offline, commit changes locally. This allows to shell the code before pushing them to the users. In DVCs, every repository contains the entire history of the project. Any changes to the local working copy must be checked into the or pushed to the repositories and updated from other. Repositories must be merged with the local copy repository before you can update the working copy. DVCs offers many new, and powerful ways to collaborate GitHub, is an example, is a new pioneered model for collaboration for open source project.

Centralized Continuous Integration uses Powerful continuous integration servers has which has facilities such as centrally, managed and sophisticated authorization, key schemes that allow the developers to provide CI as a centralized service to large and distributed teams. These systems make the continuous integration system easy without having to obtain their own hardware. The operations team consolidate server resources control configuration of continuous integration and testing environments to ensure consistent and similar production. This forces good practices such as managing configuration of third party Libraries providing pre-installed tools for gathering consistent metrics and monitoring across the projects. This supports managers and delivery teams with the ability to create dashboards to monitor code quality. It program level this also allows virtualization that can work well in conjunction with centralized CA services, the virtualization can be used to make provisioning new environments in a completely automated process.