TOPIC:

Continuous Delivery: Principles of Software delivery, Introduction and concepts.

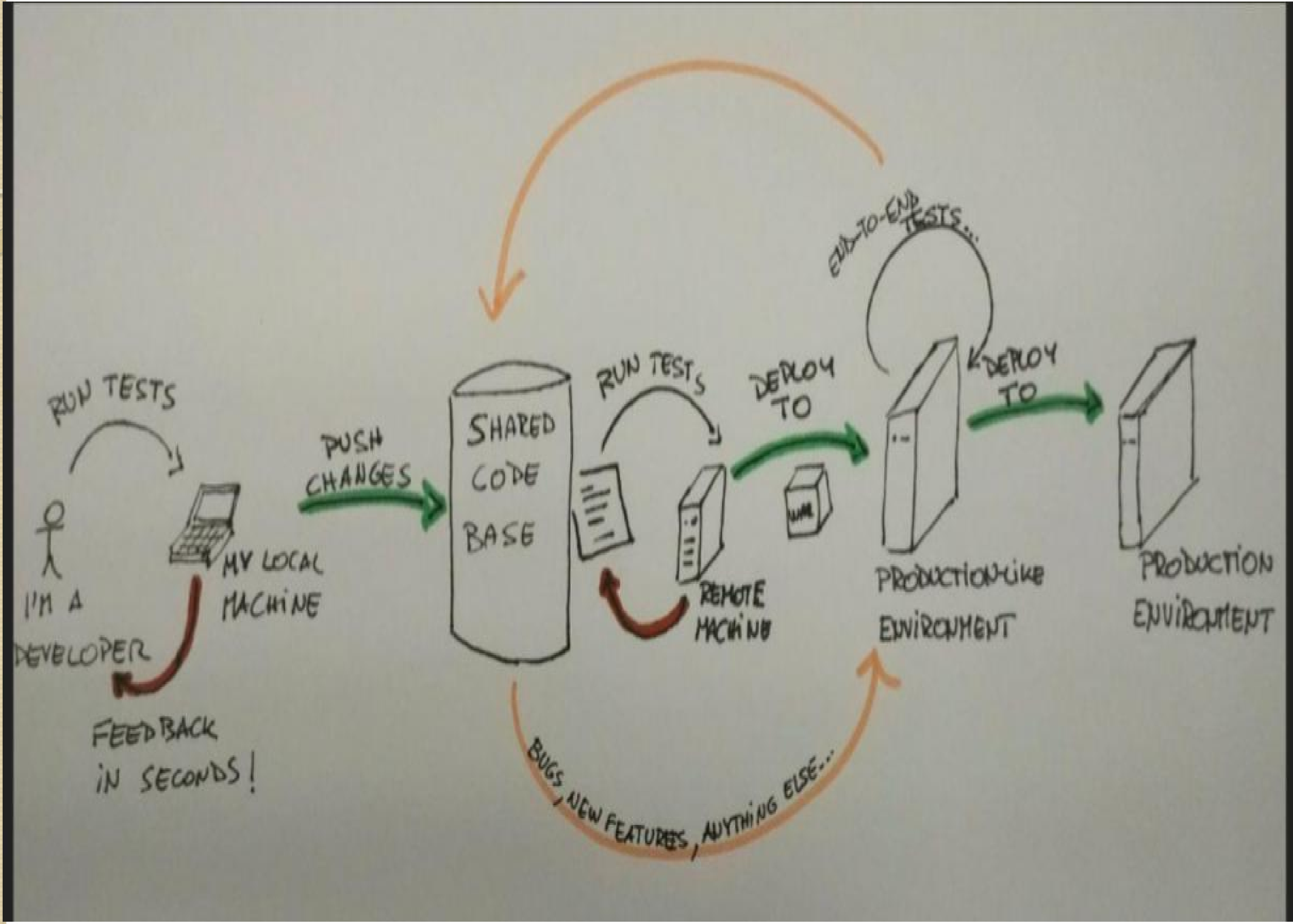
Continuous Delivery

Continuous delivery is an extension of continuous integration. It focuses on automating the software delivery process so that teams can easily and confidently deploy their code to production at any time

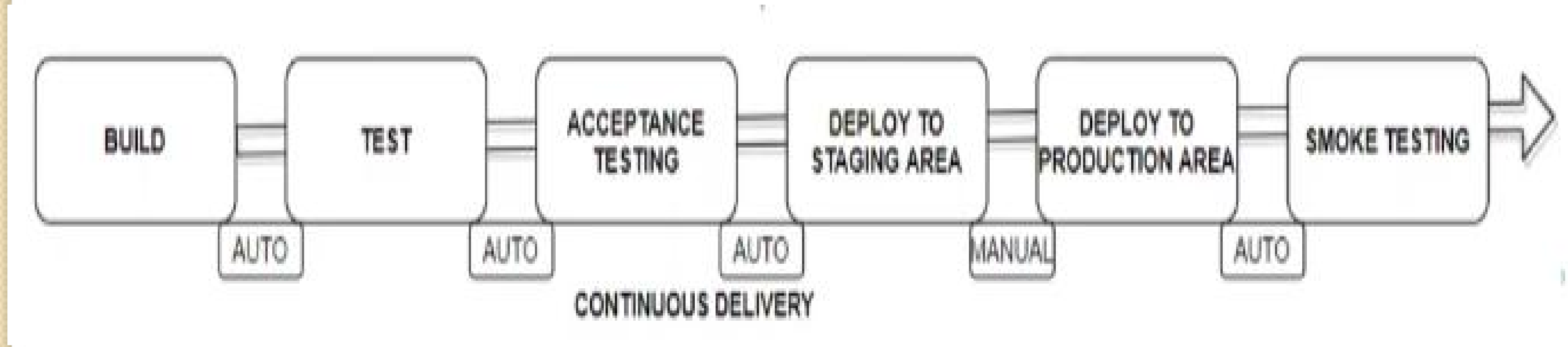
Teams can be confident that they can release whenever they need to without complex coordination or late-stage testing

Continuous delivery leans heavily on deployment pipelines to automate the testing and deployment processes

A **deployment pipeline** is an automated system that runs increasingly rigorous test suites against a build as a series of sequential stages.


RUN TESTS
I'M A DEVELOPER
MY LOCAL MACHINE
FEEDBACK IN SECONDS!
PUSH CHANGES
SHARED CODE BASE
RUN TESTS
REMOTE MACHINE
DEPLOY TO
END-TO-END TESTS...
PRODUCTION-LIKE ENVIRONMENT
DEPLOY TO
PRODUCTION ENVIRONMENT
BUGS, NEW FEATURES, ANYTHING ELSE...

Continuous delivery helps you build a refined version of the software by continuously implementing fixes and feedback until finally, you decide to push it out to production



Principles of continuous delivery

Continuous delivery promotes the adoption of an automated deployment pipeline to release software into production reliably and quickly.

Its goal is to establish an optimized end-to-end process, enhance the development to production cycles, lower the risk of release problems and provide a quicker time to market

There are 8 principles of continuous delivery

Repeatable Reliable Process

Use the same release process in all environments. If a feature or enhancement has to work through one process on its way into the integration environment, and another process into QA, issues find a way of popping up.

Automate Everything

Automate your builds, your testing, your releases, your configuration changes and everything else.

Once you automate a process, less effort is needed to run it and monitor its progress

Version Control Everything

Code, configuration, scripts, databases, documentation.

Bring the Pain Forward

Deal with the hard stuff first. Time-consuming or error prone tasks should be dealt with as soon as we can

Build-in Quality

Create short feedback loops to deal with bugs as soon as they are created. By having issues looped back to developers as soon as they fail post-build test, it will enable them to produce higher quality code quicker.

"Done' Means Released

A feature is done only when it is in production. Having a clear definition of "done" right from the start will help everyone communicate better, and realize the value in each feature

Everyone is Responsible

Responsibility should extend all the way to production

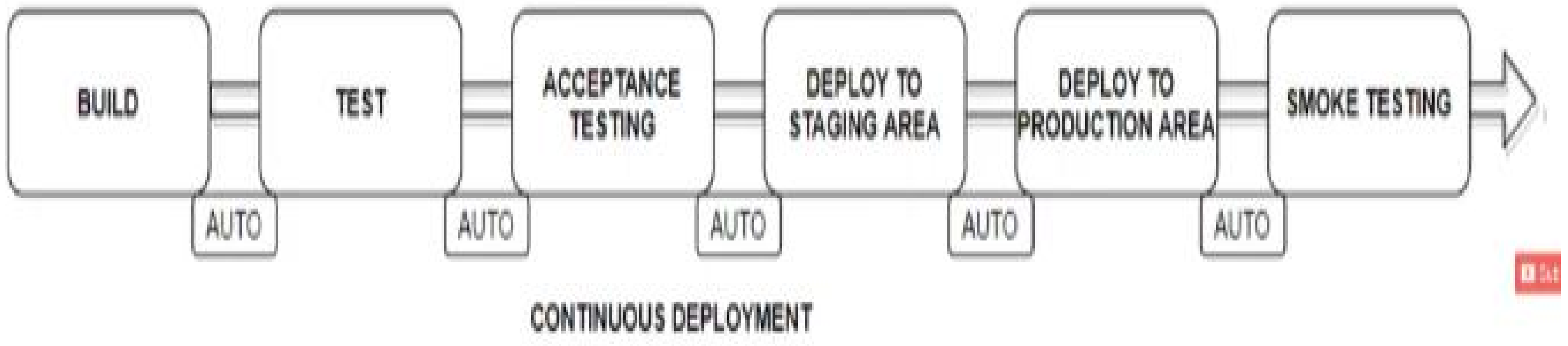
Continuous Improvement

Continuous Delivery cannot be accomplished without continuous improvement.

Implementing continuous delivery for an organisation’s processes will go a long way towards developing efficient, robust and high-quality software production, deployment and delivery practices at substantially lesser cost and effort

Continuous Deployment

In continuous deployment ,every changes goes through an automated pipeline and a working version of the application is automatically pushed to production



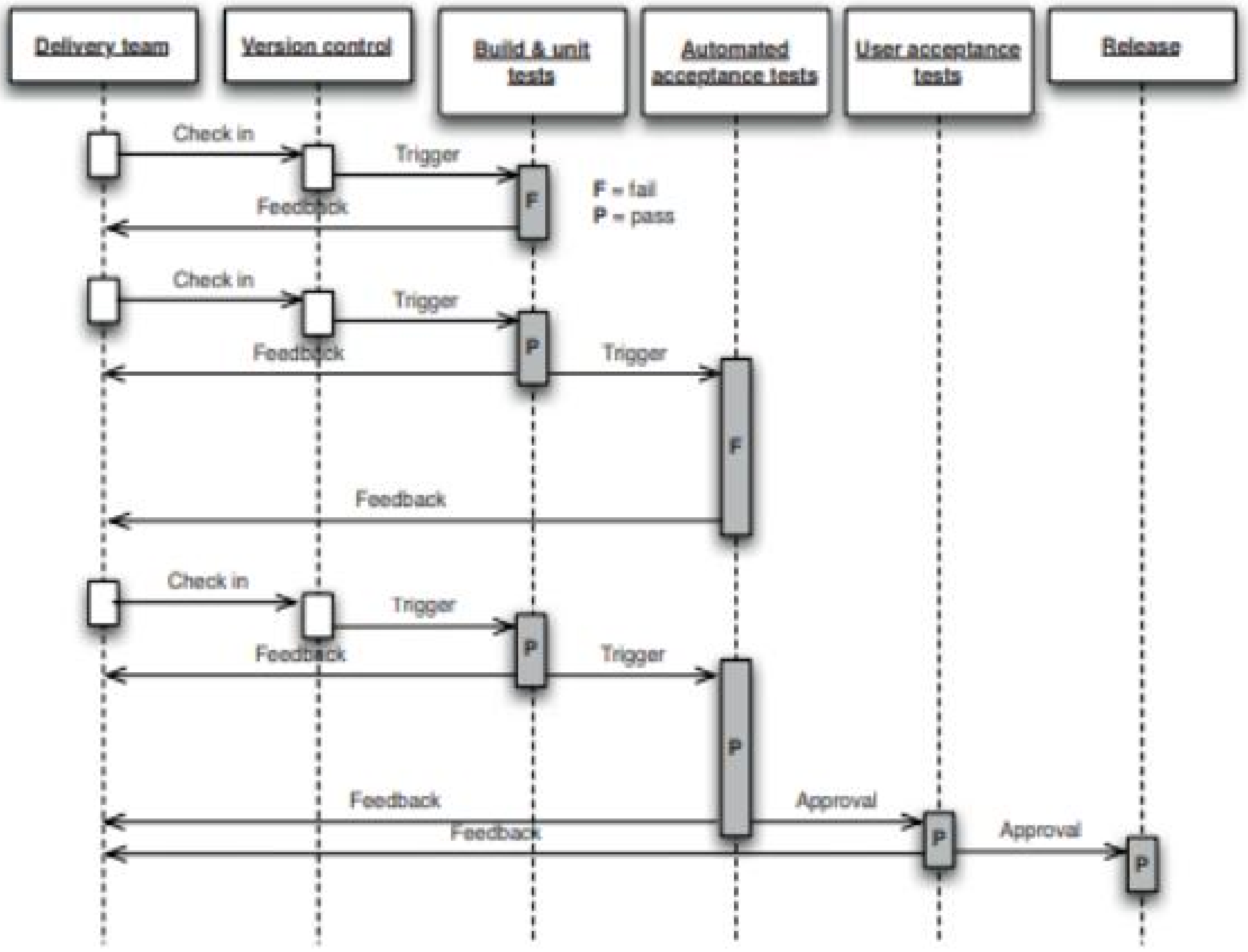
Deployment pipeline

At an abstract level, a deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users.

Every change to your software goes through a complex process on its way to being released, which includes building the software, followed by the progress of these builds through multiple stages of testing and deployment.

Also requires collaboration between many individuals, and perhaps several teams.

All these process can be implemented using deployment pipeline,where continuous integration and release management tool is what allows us to see and control the progress of each change as it moves from version control through various sets of tests and deployments to release to users.



Changes moving through the deployment pipeline

Input to the pipeline is a particular revision in version control.

Every change creates a build , pass through a sequence of tests of, and challenges to, its viability as a production release.

As the build passes each test of its fitness, confidence in it increases.

Environments the build passes through become progressively more production like.

The objective is to eliminate unfit release candidates as early in the process as we can and get feedback on the root cause of failure to the team as rapidly as possible. To this end, any build that fails a stage in the process will not generally

be promoted to the next.

To achieve an enviable state, we must automate a suite of tests that prove that our release candidates are fit for their purpose.

We must automate deployment to testing, staging, and production environments to remove manually intensive, error-prone steps. For many systems, other forms of testing and so other stages in the release process are also needed, but the subset that is common to all projects .

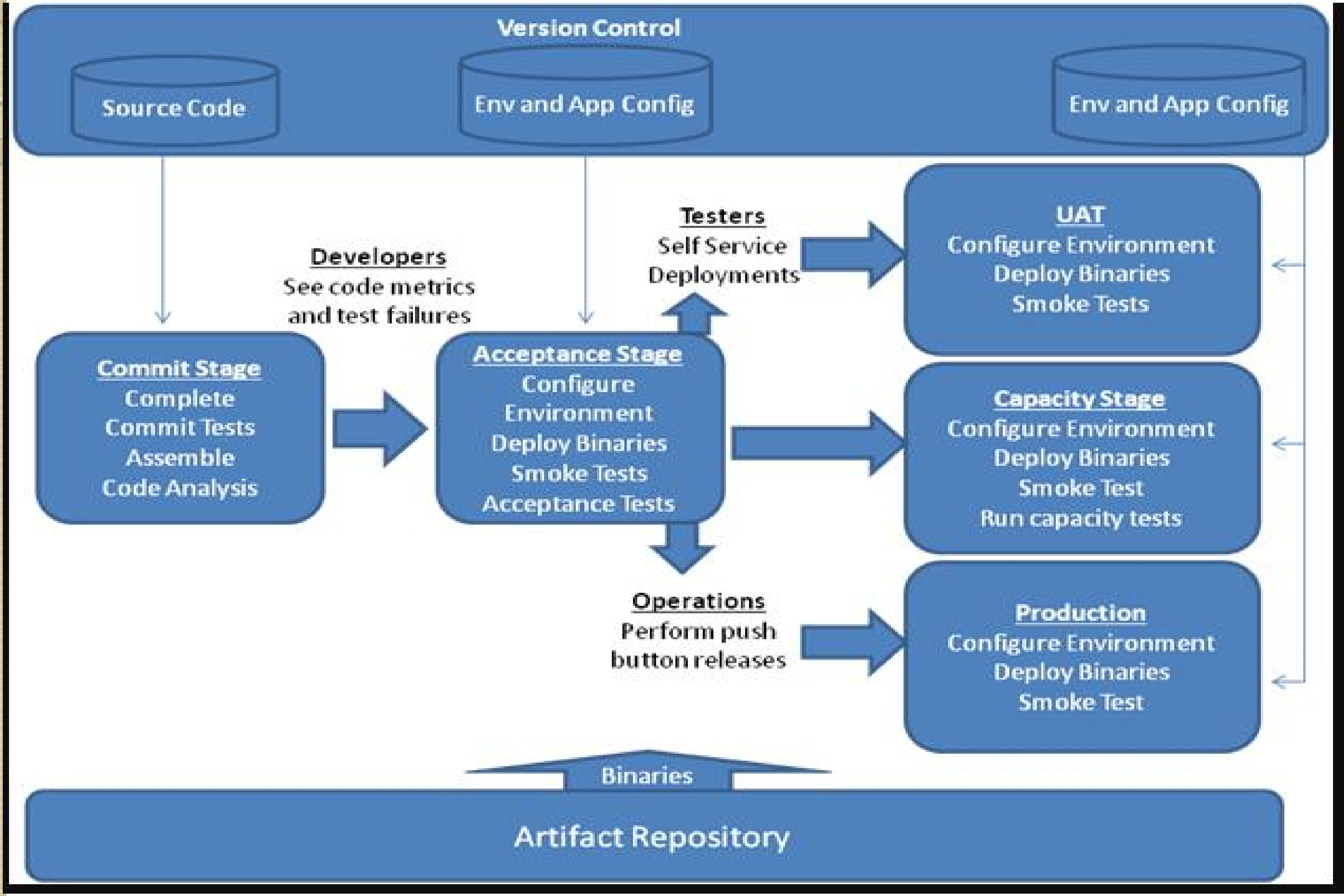
The commit stage asserts that the system works at the technical level. It compiles, passes a suite of automated tests, and runs code analysis.

Automated acceptance test stages assert that the system works at the functional and non functional level, that behaviorally it meets the needs of its users and the specifications of the customer.

Manual test stages assert that the system is usable and fulfills its requirements, detect any defects not caught by automated tests and verify that it provides value to its users

Release stage delivers the system to users, either as packaged software or by deploying it into a production or staging environment

A Basic Deployment Pipeline



The deployment pipeline begins when a developer commits code to a joint versioning system.

Prior to doing this commit, the developer will have performed a series of pre-commit tests on their local environment. The failure of the pre-commit tests of course means that the commit does not take place.

A commit then triggers an integration build of the service being developed. This build is tested by the integration tests.

If these tests are successful, the build is promoted to a quasi-production environment – the staging environment.

Then it is promoted to production under close supervision.

After another period of close supervision, it is promoted to normal production.

STAGE 1

The Process starts with the developers committing changes into their version control. At this point, the continuous integration management systems responds to the commit by triggering a new instance of our pipeline. The first (commit) stage of the pipeline

- o compiles the code,
- o runs unit tests,
- o performs code analysis and
- o creates installers

If the unit tests all pass and the code is up to scratch, we assemble the executable code into binaries and store them in an artifact repository so that it is easily accessible both to the users and to the later stages in the pipeline.

STAGE 2

The second stage is typically composed of longer-running **automated acceptance tests**. Again, CI server split these tests into suites which can be executed in parallel to increase their speed and give feedback faster

This stage will be triggered automatically by the successful completion of the first stage in your pipeline.

At this point, the pipeline branches to enable independent deployment of your build to various environments like , UAT (user acceptance testing), capacity testing, and production.

If you don’t want these stages to be automatically triggered by the successful completion of your acceptance test stage, can do it manually by writing automated script that performs deployment.

In order to get the benefits from pipeline approach there are some practices we should follow

1) **Only build your Binaries Once**

The code will be compiled repeatedly in different contexts: during the commit process, again at acceptance test time, again for capacity testing, and often once for each separate deployment target.

Every time you compile the code, you run the risk of introducing some difference.

The version of the compiler installed in the later stages may be different from the version that we used for our commit tests may leads to see bugs.

So, you should only build your binaries once, during the commit stage of the build. These binaries should be stored on a filesystem somewhere where it is easy to retrieve them for later stages in the pipeline.

Most of the cases CI server will perform these job

2) Deploy the same way to every environment

if your deployment script is different for different environments, you have no way of knowing that what you are testing will actually work when you go live

Instead,if you use the same process to deploy everywhere ,when a deployment doesn’t work to a particular environment you can narrow it down to one of three causes

A setting in your application’s environment-specific configuration file

- A problem with your infrastructure or one of the services on which your application depends

The configuration of your environment

Smoke test your deployments

When you deploy your application, you should have an automated script that does a smoke test to make sure that it is up and running.

This could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content.

Smoke test should also check that any services your application depends on are up and running—such as a database, messaging bus, or external service.

Deploy into a Copy of Production

The other main problem many teams experience going live is that their production environment is significantly different from their testing and development environments.

To get a good level of confidence that going live will actually work, you need to do your testing and continuous integration on environments that are as similar as possible to your production environment.

Propagate through the Pipeline Instantly

In the deployment pipeline ,the first stage should be triggered upon every check-in, and each stage should trigger the next one immediately upon successful completion.

If Any Part of the Pipeline Fails, Stop the Line

If a deployment to an environment fails, the whole team owns that failure. They should stop and fix it before doing anything else