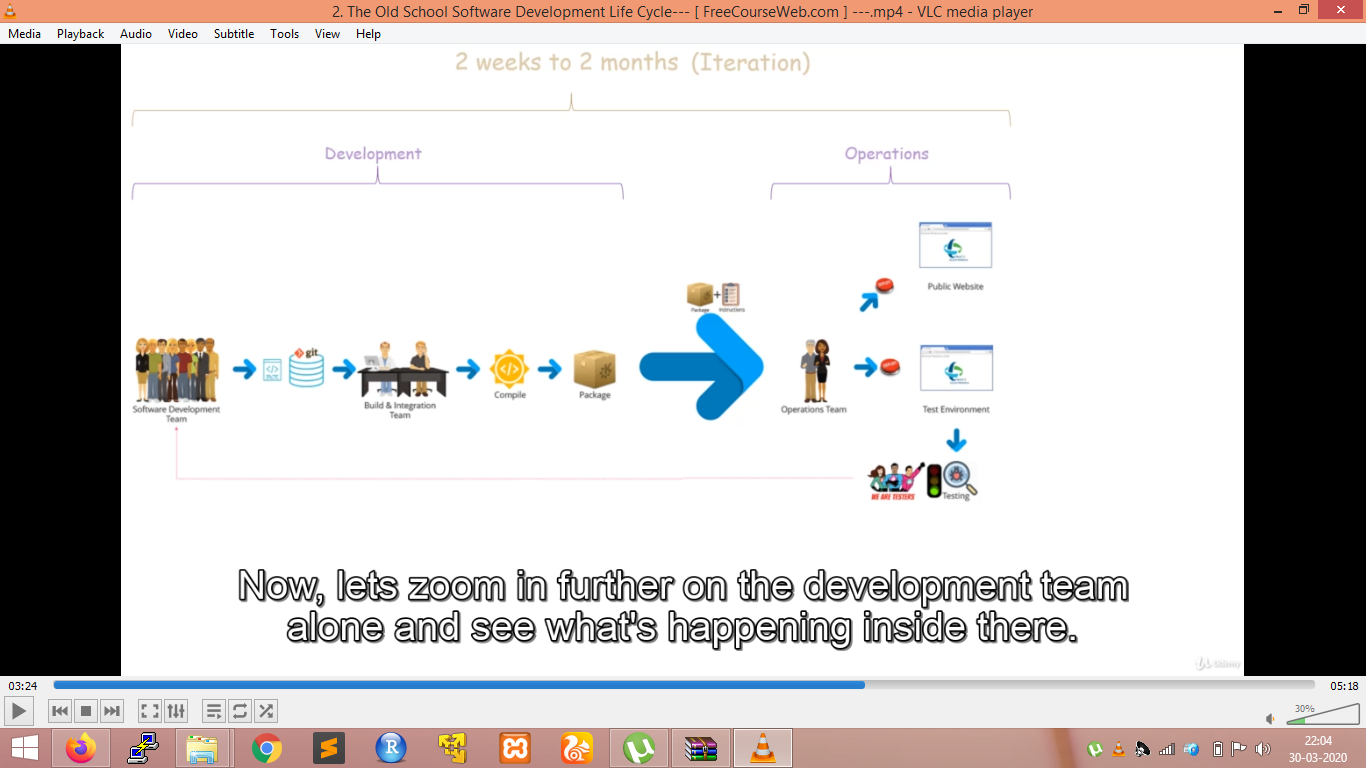
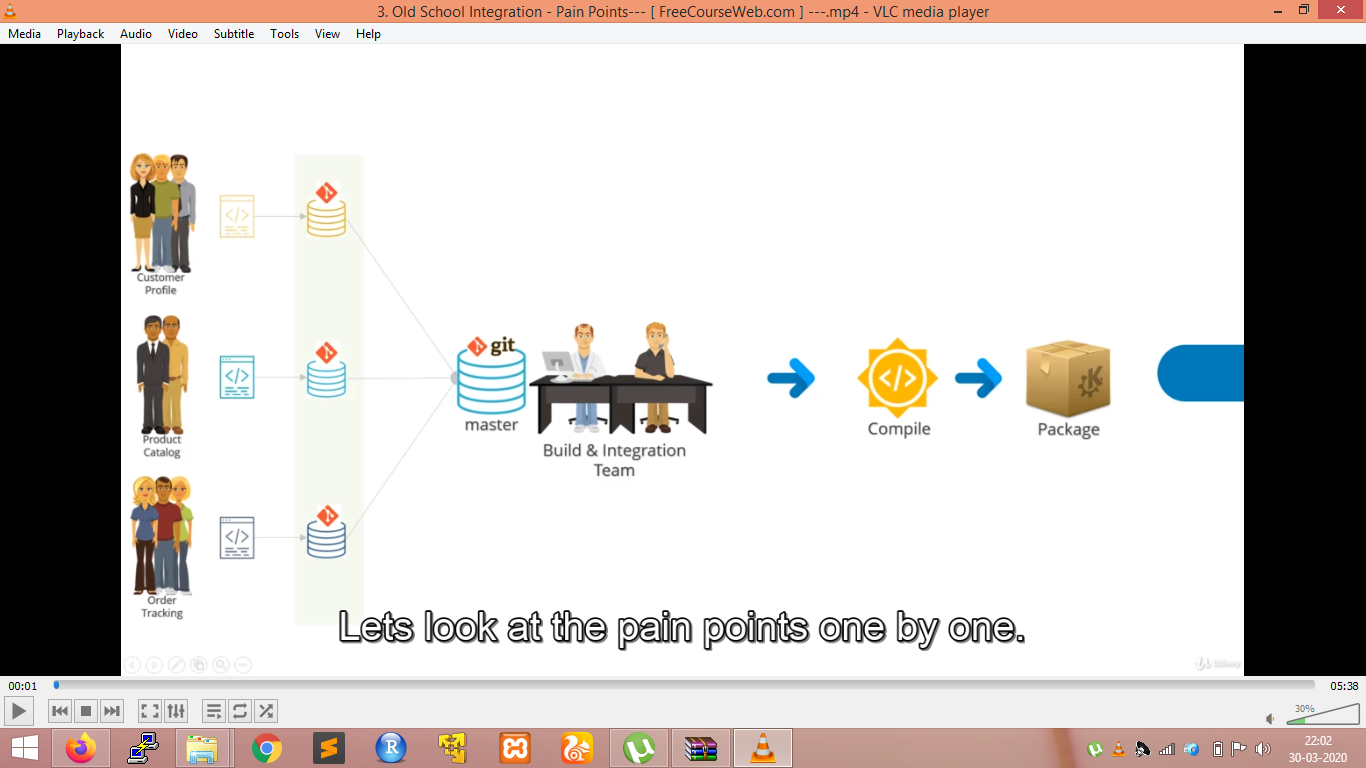
**TRADITONAL WAY OF INTEGRATION:**

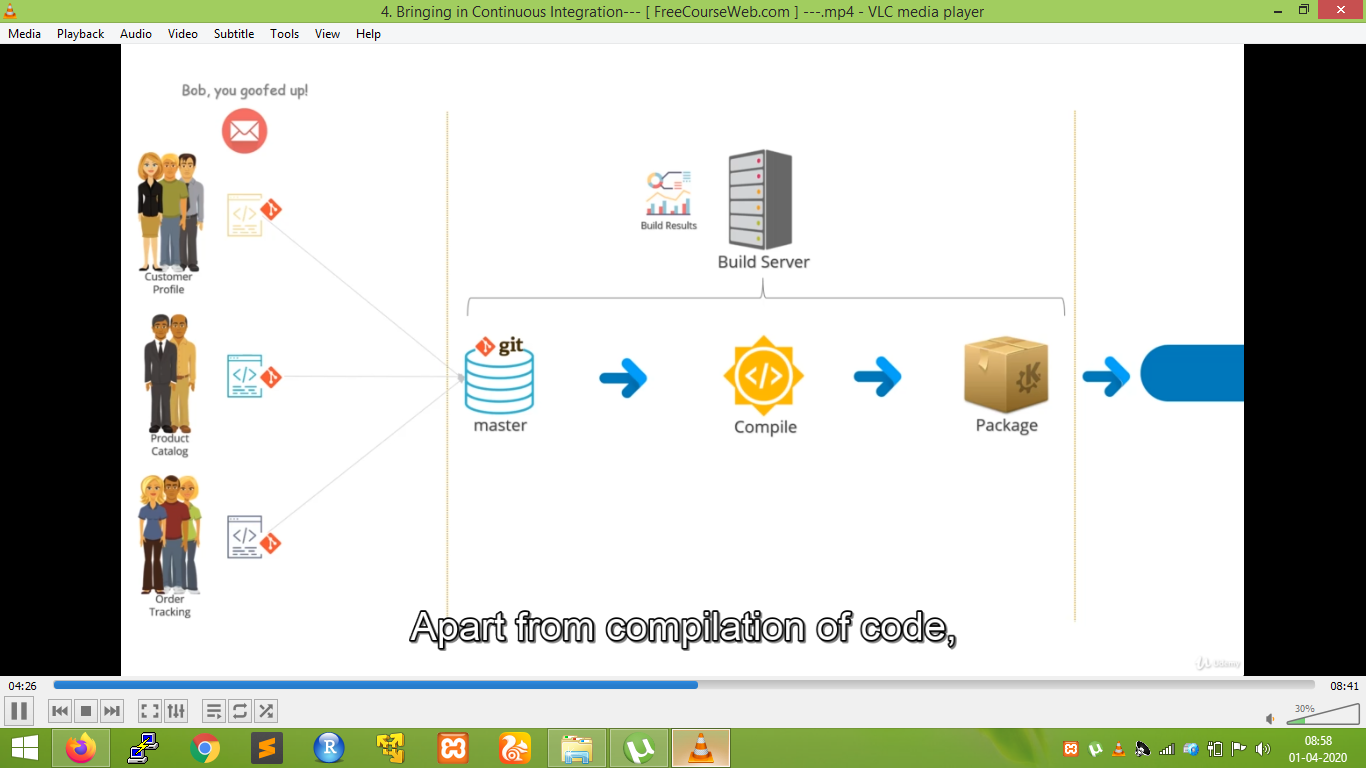


* Development team develops an application or the features of an appl.
* Production management team sets up the functional roadmap of the product by doing market research and customers’ feedback. They come up with new functionalities or modifications of existing functionalities. They interact closely with software development team.
* Software dev team takes inputs from the product management and works on implementing the requested features.
* All the code is maintained in a source code repository (GIT-vcs).
* Build and integration team which integrates the codes from all the developers and then they compile it and build a software package. This software is passed along with some instructions to the operations team who deploy by passing it to the testing environment.
* Then it is passed on to quality assurance team who makes sure that the features work as expected. Any defaults found are reported back to the programmers, then the whole process is repeated. Quality assurance teams role are very close to product management to make sure product management is getting what they asked for.
* Once the QAT gives a go-ahead, the operations team takes the latest release package, follows instructions given to them by developers and deploys it into production.
* For developing an application, functions are divided into modules and each module is given to different developing teams.
* Integration of Code:
* It includes merging the code from source control branches onto mainline branch and integrating it .It happens only once in an iteration. The integration team often sits in a room with at least one developer representing each of the module teams to figure out how to integrate the code into a single working unit. They come across many challenges:

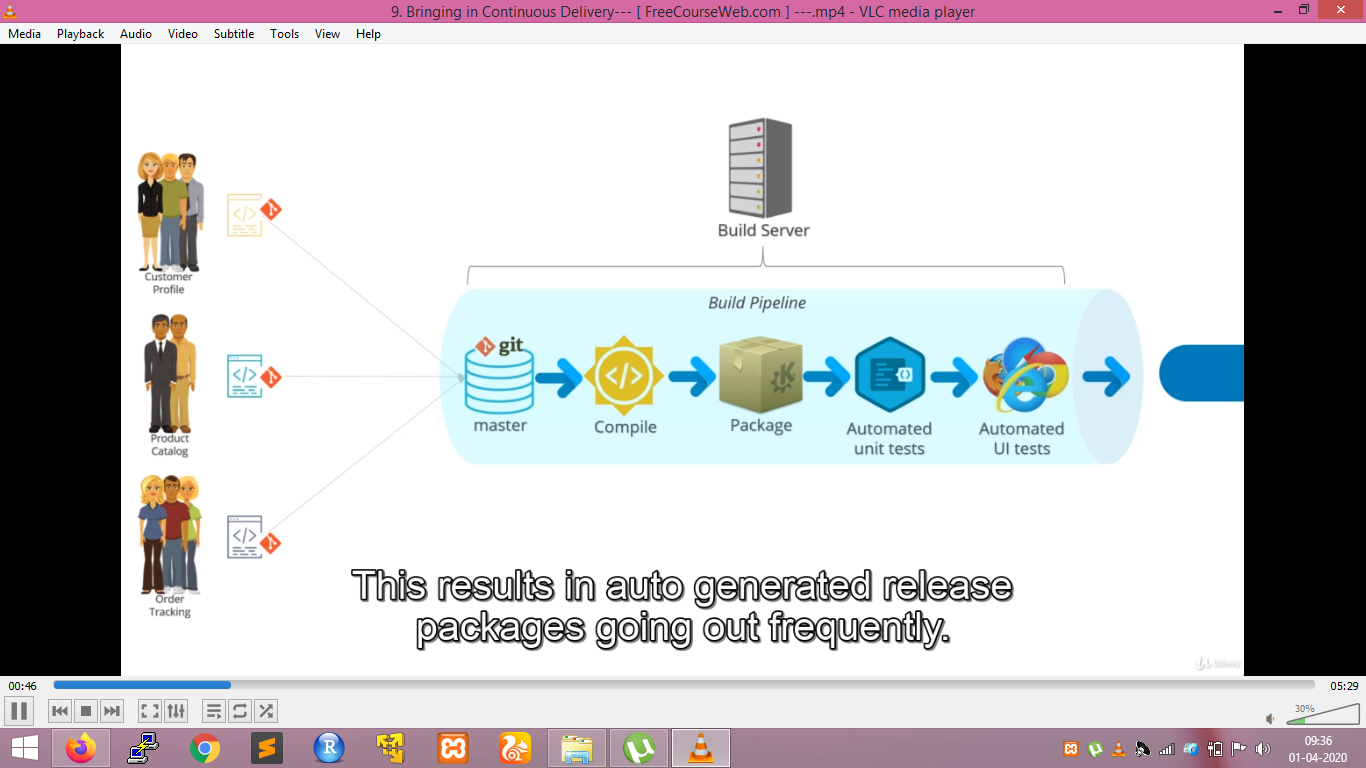
1. They consume a lot of effort and the process is very much error-prone as it is mostly a manual process.
2. AT the end of the iteration only it can be checked whether the modules work with other modules or not if not there is a need of rework. Also developers might be troubling recalling the details related to the detected defects. It will consume a lot of time and effort to go back to those pieces of code and debug
3. Merging to the mainline happens once each iteration, there might be intermediate merging required. Any such code merges could produce compilation errors in the merged code. This can lead to broken builds. Resolving such issues uses a lot of time of all until the issue is resolved.
4. If a developer introduces a functional defect into the code, there is chance that it will not be visible for longer time. Although they don’t give compilation errors, but during proper testing on deployed application, they are caught. The developer do their unit testing but there is a limit what unit testing can catch. The developer’s codes has to be merged, integrated, and compiled to get a stable build. Then the build package needs to be passed on to operations team which deploys it to a test server. It is only at this point the developer can see his code working in totality and get notified by the QA team. This is a long feedback cycle. Ideally, the developer should see his code in actin as soon as possible. So long feedback cycles are not advisable for developers.
5. Iterations time usage is very long due to which the customers has to wait in the era where the market gets disrupted with a blink of eye. Hence, longer iterations lead to higher time-to-market for business.

**Continuos Integration:**

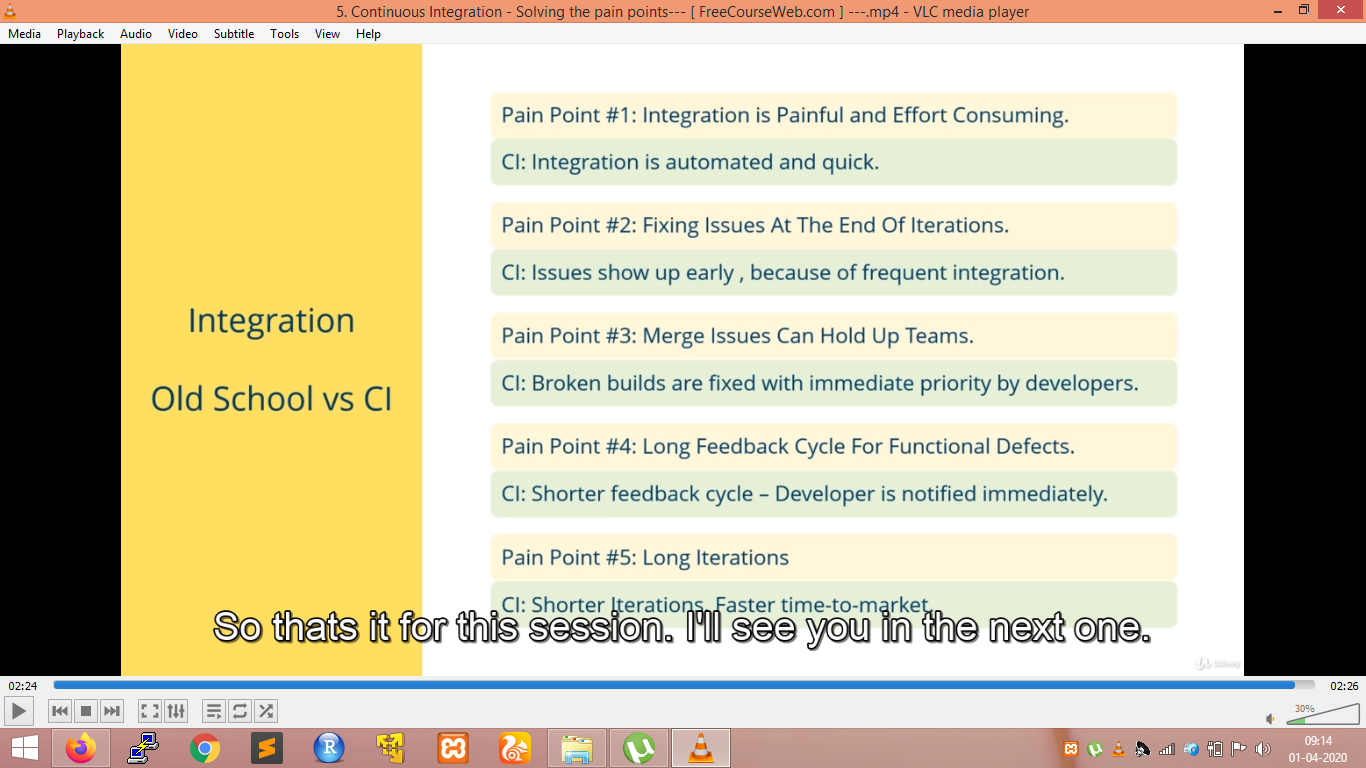
* It is a development practice that requires developers to integrate code into a shared repository several times a day.
* In the traditional way of working we need to make some changes. The developers need to use same repository while developing their codes. They can use different branches but they should frequently merge to the mainline branch often multiple times a day. This will require to integrate code themselves. In short we will divide the responsibility of build and integration team among all the developers. To make sure the codes from developer’s works properly it should be compiled regularly that is after each code commit by any developer any time. The build team is replaced with build server now. Whenever a developer checks a code it should be compiled on this build server. So the whole build process must automatically triggered whenever a developer does a commit. 
* So the build process is centralized and happens on the same build server. All the developers have access to build server so that they can see if their code compiled properly or there is compilation error. All the results will be posted automatically by the build server and all team members have access to these portal results.



* Apart from compilation of code the build should have automated tests that do various checks to ensure the correctness and quality of code. These tests will run against the latest versions of the code. These tests can be a combination of automated unit tests to check if the code yields the expected results and automated UI tests to check if the app UI responses as expected.
* For the automated UI tests to run the package code would need to get deployed onto some temporary local environment.
* An important requirement for continuous integration is that the whole build process which includes compilation and running the test should be quick not more than few mins. This way the results of the tests can be communicated to the developers quickly, who can fix issues that pop-up. This keeps code base stable and lets the developers to prioritize to fix the error as others are waiting for the results.
* The UI tests can give functional defects which can be fixed by developers.
* Automation is an enabling step in continuous integration.
* Continuous integration means the way developers work.







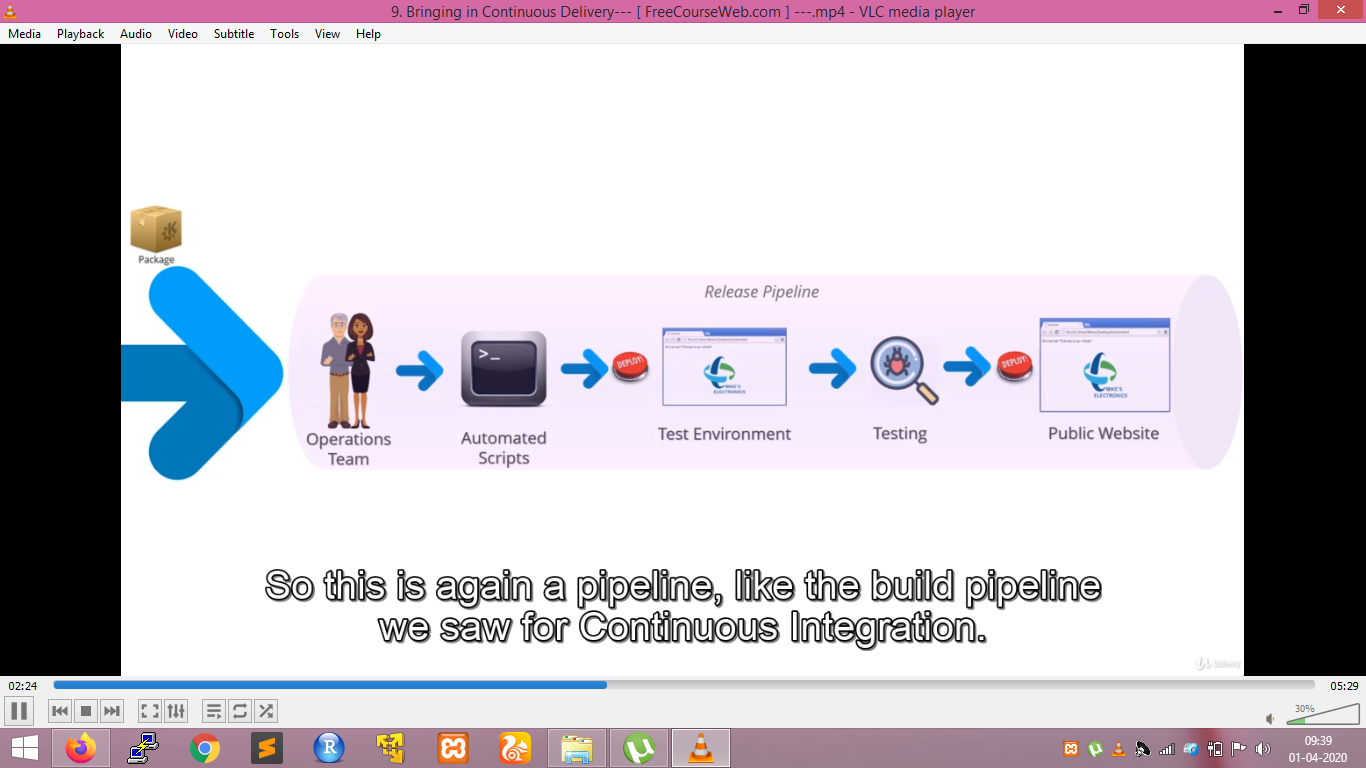
**Time before continuous delivery:**

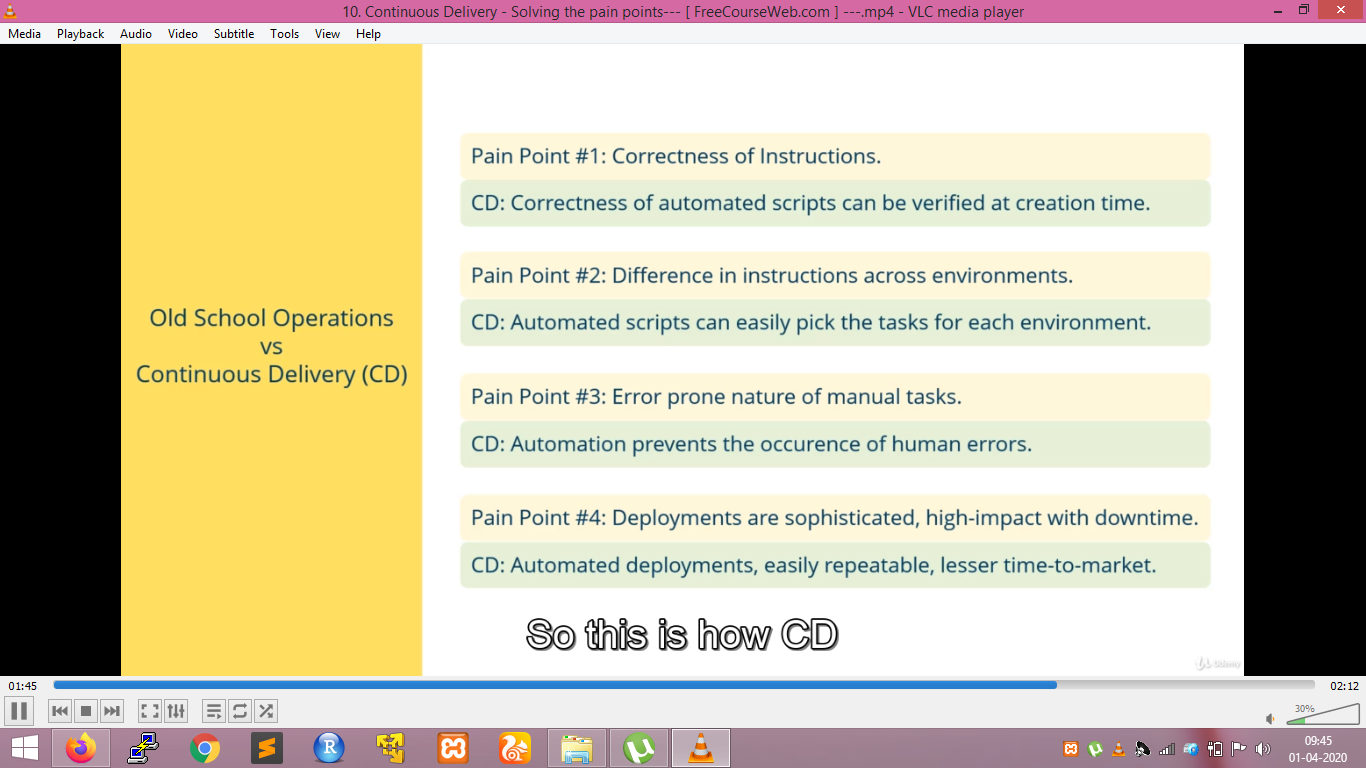
* The package with a set of instructions are sent to the operations team so that it can be deployed on different environments. The operations team go through the instructions and make a test environment, if the test passes it is then sent to QAT. QAT do a round f testing again, if they find the code okay they release for production or else shoot up the error to developers.

1. The instructions sent by the developers might always not be correct.
2. There can be difference in instructions across environments
3. This reading of instructions and building an environment is a manual work and hence prone to many errors.
4. Deploying is a sophisticated process as it goes through n instructions and n steps. This can take few hours to even a day. This leads to downtime.

**Continuous delivery:**

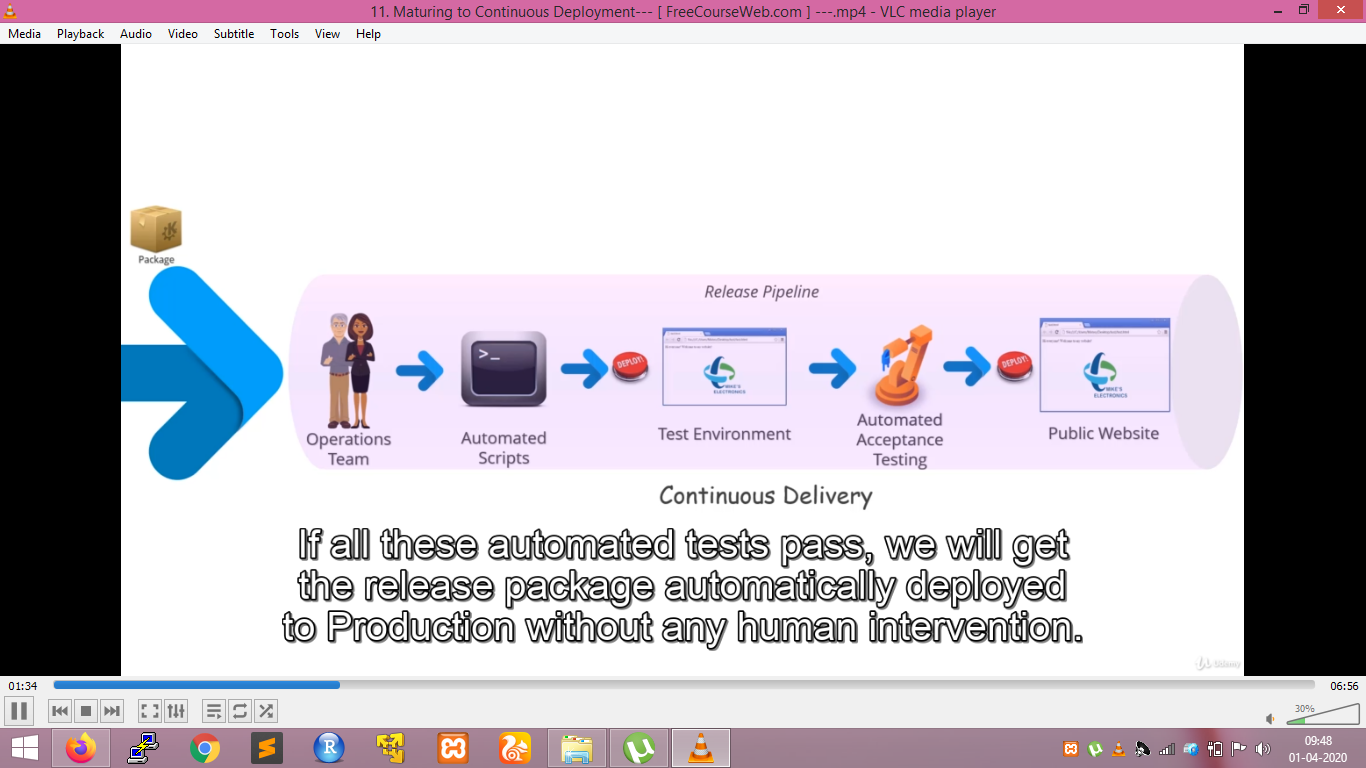
* Continuous delivery is a software deployment practice where software can be released to production any time. It is a choice.
* It means a software should be production ready.
* A pre-requisite for Continuous delivery is CI
* As there will be continuous integration so everything is automated and hence with release package there will be no instructions sent. The automation is script based automation.
* Most of the steps are scripted and automated but the final decision requires human intervention (QAT has to certify).
* It is quick and efficient and also repeatable and the downtime is very less.



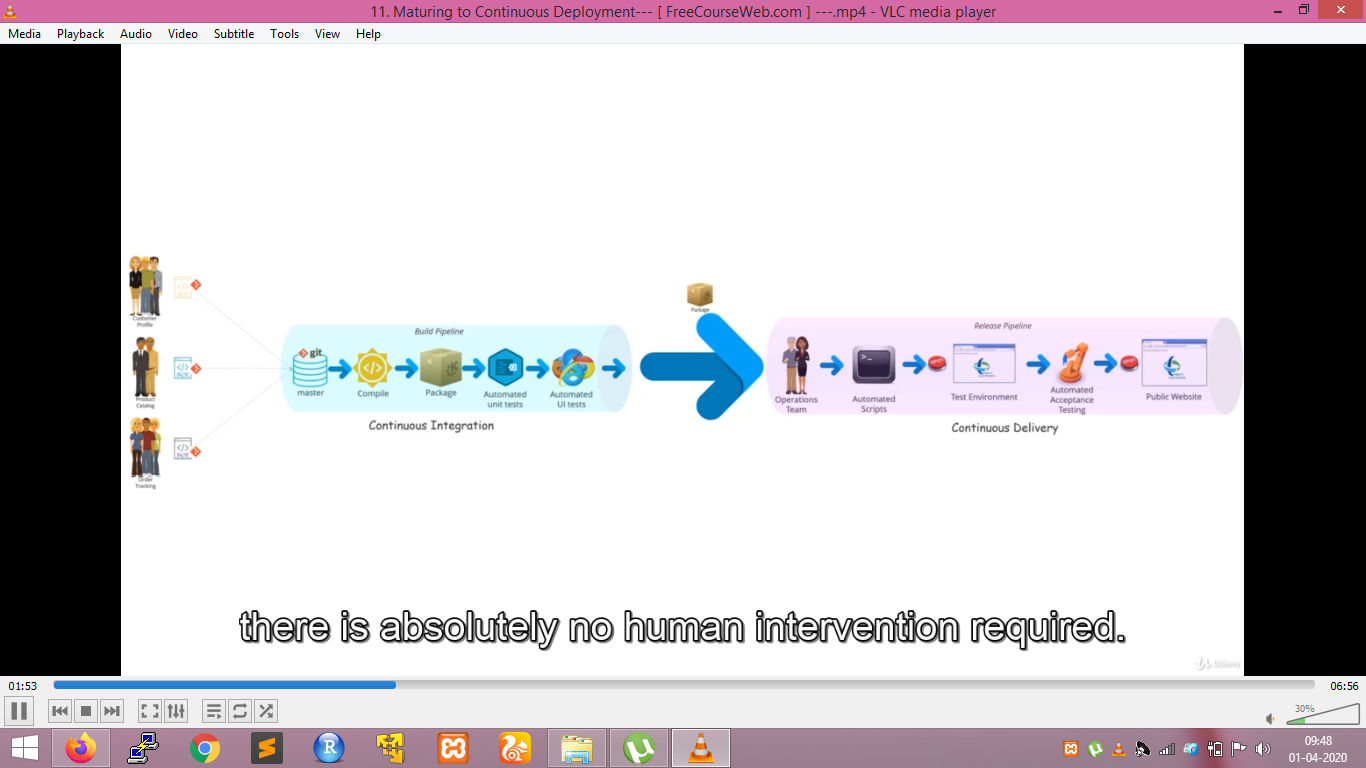


**Continuous Deployment:**

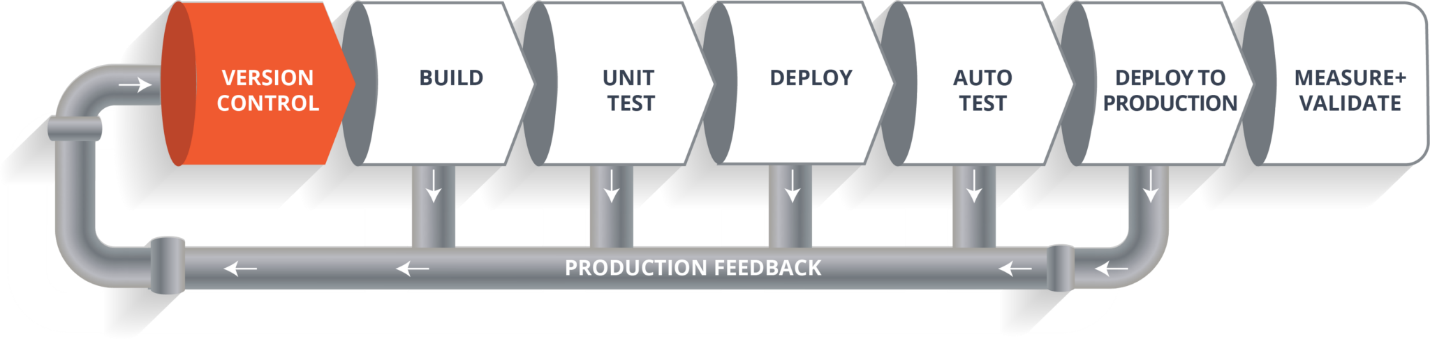
* In this the QAT is replaced with automated Acceptance testing on the deployed application on the test environment.
* So the whole process now becomes automated and there is no human intervention at all.
* Whenever a developer makes a commit into mainline branch it is going to get deployed to production in maybe a few minutes.provided of course it passes all the automated test. All the changes made by the developer is going to be visible to the users of the public website across the globe. Only few developers are allowed to make commits.
* CD is a software development practice where code is automatically released to production continuously.
* So there is no choice every time the code passes an automated tests there is production all the time.
* When all the automated tests has been passed it moves to production. We can make changes to only a particular geography



For e.g.: we want to deploy change in Canadian IP addresses. If it goes well , we will roll the changes to USA IP addresses as well. If it goes well for few days we can also roll it out to all IPs. If some risks arise in public IPs we can roll back.

**THE WHOLE PROCESS IS NOW AUTOMATED AND LOOKS LIKE:**

**CI /CD Pipeine:**



* **Version Control**:The developers code is committed into a version control system(such as git).
* **Build Phase**: Next, it goes through the **build phase** which is the first phase of the pipeline, where developers put in their code and then again code goes to the version control system having a proper version tag. Through the version control phase, it again goes to build phase where it gets compiled. You get all the features of that code from various branches of the repository, which merge them and finally use a compiler to compile it. This whole process is called the **build phase**.
* **Testing Phase:** Once the build phase is over, then you move on to the **testing phase.** In this phase, we have various kinds of testing, one of them is the unit test (where you test the units of software).
* **Deploy Phase:** When the test is completed, you move on to the **deploy phase**, where you deploy it into a staging or a test server. Here, you can view the code or you can view the app in a simulator(A **simulator** is a software that helps your computer run certain programs built for a different Operating System).
* **Auto Test Phase:** Once the code is deployed successfully, you can run another set of a sanity test. If everything is accepted, then it can be deployed to production.
* **Deploy to Production:** Meanwhile in every step, if there is some error, you can shoot a mail back to the development team so that they can fix them. Then they will push it into the version control system and goes back into the pipeline. Once again if there is any error reported during testing, again the feedback goes to the dev team where they fix it and the process re-iterates if required.
* **Measure + Validate:** So, this lifecycle continues until we get a code or a product which can be deployed in the production server where we measure and validate the code.

We want the whole pipeline process to be on automated mode. For this we use Jenkins tool. This Jenkins tool provides us with various interfaces to make the process automated.



* We have a git repository where the development team will commit the code. Then Jenkins takes over from there which is front-end tool where you can define your entire job or the task.
* From Git, Jenkins pulls the code and then moves it to the **commit phase**, where the code is committed from every branch. Then Jenkins moves it into the **build phase** where we compile the code. If it is Java code, we use tools like maven in Jenkins and then compile that code, which we can be deployed to run a series of tests. These test cases are overseen by Jenkins.
* Then it moves on to the staging server to deploy it using **docker**. After a series of Unit Tests or sanity test, it moves to the production.

**JENKINS PIPELINE**

* Jenkins Pipeline is a combination of plugins that support the integration and implementation of **continuous delivery pipelines** using Jenkins.
* Jenkins pipelines can be defined using a text file called **Jenkinsfile.**You can implement pipeline as code using Jenkinsfile, and this can be defined by using a domain specific language (DSL). With Jenkinsfile, you can write the steps needed for running a Jenkins pipeline.

The benefits of using J**enkinsfile are**:

* You can create pipelines automatically for all branches and execute pull requests with just one **Jenkinsfile.**
* You can review your code on the pipeline
* You can audit your Jenkins pipeline
* This is the singular source for your pipeline and can be modified by multiple users.

Keywords in pipeline:

* Agent :The agent section specifies where the entire pipeline or a specific stage will execute in Jenkins environment depending on the agent section position. There are two types of agents:
  + (a). Top Level Agents: In agents declared at the outermost level of the Pipeline, the options are invoked **after** entering the agent.
  + (b). Stage Agents: In agents declared within a stage, the options are invoked **before** entering the agent and **before** checking any when conditions.

### Node: A node is a machine which is part of the Jenkins environment

### and is capable of executing a Pipeline.

### Stage: A stage block defines a conceptually distinct subset of tasks performed through the entire Pipeline

### Step: A single task. Fundamentally, a step tells Jenkins what to do at a particular point in time

There are two types of syntax used for defining your JenkinsFile.

1. Declarative
2. Scripted

**Declarative:**

Declarative pipeline syntax offers an easy way to create pipelines. It contains a predefined hierarchy to create Jenkins pipelines. It gives you the ability to control all aspects of a pipeline execution in a simple, straight-forward manner.

In declarative blocks must only consist of [Sections](https://jenkins.io/doc/book/pipeline/syntax/#declarative-sections), [Directives](https://jenkins.io/doc/book/pipeline/syntax/#declarative-directives), [Steps](https://jenkins.io/doc/book/pipeline/syntax/#declarative-steps), or assignment statements.

Syntax:

pipeline {

agent any

stages {

stage('Build') {

steps {

//

}

}

**Scripted:**

Scripted Jenkins pipeline runs on the Jenkins master with the help of a lightweight executor. It uses very few resources to translate the pipeline into atomic commands. Both declarative and scripted syntax are different from each other and are defined totally differently.

Here are the reasons why you use should use Jenkins pipeline:

* Jenkins pipeline is implemented as a code which allows multiple users to edit and execute the pipeline process.
* Pipelines are robust. So if your server undergoes an unforeseen restart, the pipeline will be automatically resumed.
* You can pause the pipeline process and make it wait to resume until there is an input from the user.
* Jenkins Pipelines support big projects. You can run multiple jobs, and even use pipelines in a loop.

In Scripted Pipeline syntax, one or more node blocks do the core work throughout the entire Pipeline. The nodes inside the pipeline schedules the steps contained within the block to run by adding an item to the Jenkins queue. As soon as an executor is free on a node, the steps will run. It also Creates a workspace where work can be done on files checked out from source control.

Scripted pipeline syntax is :

node {

stage('Build') {

//

}

}