IIT Madras

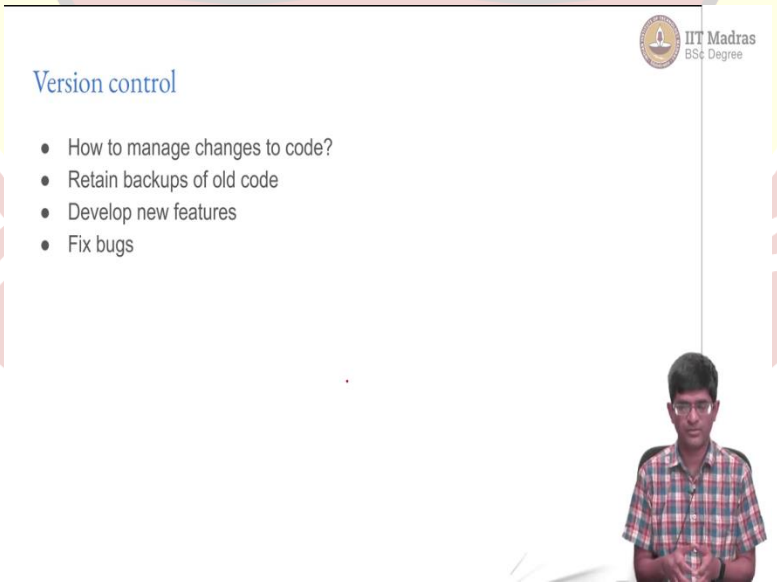
ONLINE DEGREE

Modern Application Development-I
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Deployment

Hello, everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:17)

Deployment



Version control

- How to manage changes to code?
- Retain backups of old code
- Develop new features
- Fix bugs

So, finally, after all of the platform services and so on that we have seen, let us look at the last stage, not the last stage as such, it is more something that has to play a role throughout your development process. But it is certain general topics related to how you deploy systems on to cloud based servers.

So, there are a number of different concepts that I am going to be talking about over here. And they are slightly, I mean, they are all related to the main topic of deploying software, but they are also slightly different from each other. So, there would be like a few different things that I mentioned, and then sort of summarize everything together at the end.

So, the first topic that I want to talk about is version control. Hopefully, all of you have at least heard the term Git. Hopefully, many of you have used Git in some form or the other. So, what is Git? It is a type of version control software. It is a version control system. It is more than software, it is a system.

And what exactly does that mean? It is a version control system? And what is version control? In the first place? So, version control relates to how do you manage changes to code? Because as we are writing code, what we are going to be doing is, I have an idea, let us say that I am developing some kind of grade book example, I first need to create, views and controllers for handling the users.

So, what does the user interface look like? Then comes a set of things for managing a course. And then something for sort of putting things together and saying, a particular student got a certain mark in a given course, how do I make that entry? How do I update that, and so on. I cannot do all of that at one shot, I cannot write all of the code, in one big block.

What I will do is I will develop it incrementally, and I want to keep testing as I am going around. So, there are a few things we would want to do. One is, of course, I need to be able to sort of test and make sure that as I proceed, I am not going to break any tests, I am not going to have regressions.

So, this is something we already looked at, in the topic on testing. But the question is, I am going on making modifications to the code, how do I manage these changes? If I have gone and broken something, or made a change that affected something in a bad way, I need to go back and reset something else.

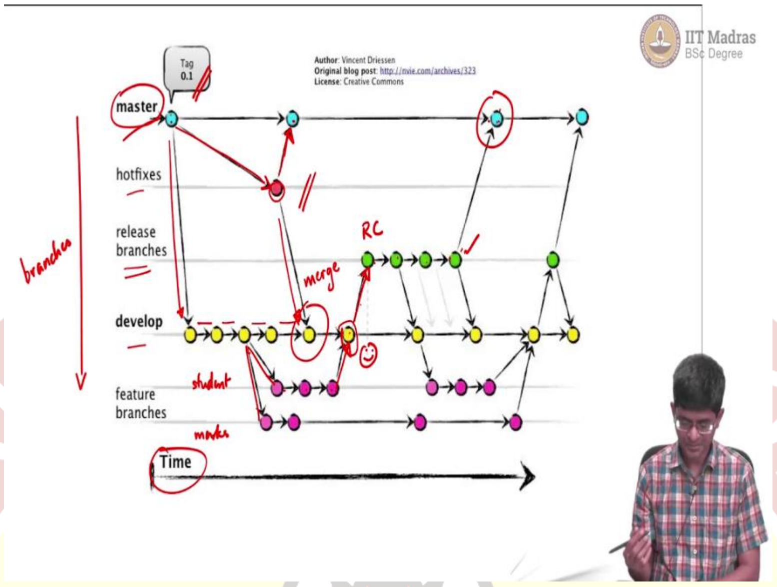
So, going back is the hard part, I need to have copies of the old code. I could always, maybe every day, take a backup of my code and say, at least once every day, I have, I know what my code was like yesterday. The problem is, after a while, I just have a lot of backup copies of the code. And I do not know which one is which I have like no information associated with those copies.

Let us say that I suddenly found out that some feature broke. And then I go back and look at yesterday's code, and it was broken there as well. Go back one week, it was still broken. But I go back one month and ago it was working there. But one month ago, I mean I had made so many changes in the one month after that, that I cannot just go back and take the code from one month and say, I will work with this. I need to be able to find out what was the specific change that broke. So, rather than just keeping like daily backups in some form, can I sort of manage those changes and have information about what changed where?

Similarly, if I just want to have backups, of course, that is also something that I might want to use. What happens if I want to try out a new feature? And before I put it onto the main server, I want to have some code which tries that out, but I am not yet confident that it will work. Similarly, I find that there is a bug and I want to fix it. So, all of these are different things, why you would want to change the code. And version control essentially provides you with a nice mechanism for managing.

Git is pretty much the most popular form of version control that is available today. It is not the only one, there are multiple different versions, I mean, there were many different ones that were available earlier. Nowadays, this happens to be the most popular, and with good reason. It has a lot of good features to it, and it is well worth sort of learning how to use it properly.

(Refer Slide Time: 04:43)



Now, this picture is from some, it is from a document, which talks about something called Git flow and has some very nice features that allow you to sort of, understand software development process. And one of the things that they talk about or here is essentially what we are looking at is these are what are called branches and the x-axis, as you can see is time.

So, now what are these branches? Why do you need different branches? And what are these arrows going around between the different branches. That is the main thing that we want to understand. Now, for those of who have worked with yet, you would have noticed that as soon as you start a new Git project, you start off with something called a master branch. Nowadays, it is sometimes called the main branch but either main branch or master branch.

The point is that this, ultimately, is what you want to keep as your golden reference, this is what you are going to say at any point is, what is for public release. So, if you are finally creating an app, and you are sort of, making it available for the public, it would be the master branch that you are going to deploy.

Now, let us look at this master branch. And as you can see, or here, there are not too many changes on the master branch, I mean, there is one change over here where you say there is a tag, which you just say, as a starting tag at some point. But apart from that, there is like one change quite a bit further down the line, then one more change much further down and one more change later.

On the other hand, these other things, the green release branches and this develop branch seem to be showing a lot of activity. And there is also this thing called the hotfixes branch, which is not, it is just there and then gone. So, let us sort of try and understand what these things are for. Essentially, what it is saying is a hotfix is a bug fix. You found a bug in your code, you have to fix it.

You cannot wait and say, I will release it after 6 months I have to fix it right now. When that happens, what you do is you create a branch directly from the master branch. And you push the changes directly back to the master branch. So, as far as this is concerned, what you are saying is I had some code, this code had a bug in it, I checked out part of that code, made the changes that I need in order to fix the bug, and pushed it back.

At which point with this new update on the blue line, the master branch, at least does not have that bug. But what I need to do is I need to make sure that in the meantime, what was happening, let us see what else happened with master. The first thing I did was, as soon as I had master, I also branched out into something else, that I am essentially calling my develop branch, this is where all the changes are going to happen.

And the develop branch is going on sort of having updates changes that are going along, day by day. Every day I write some code, I push it to the develop branch. Now, what happened is, after a certain amount of time, when this hotfix was done over here, I need to make sure the hotfix is reflected in the develop branch as well.

So, along with master, I need to also push it to the develop branch. And this is something called a merge. What I have done is I have merged two branches together. At this point, if I look at it, this hotfix branch is done. I can delete the branch, I no longer need it at this stage. Now, that is looks good, I have a master, I can fix bugs with it, I also have some place where I can do some development going on adding code to it.

But now comes the idea that, maybe now I want to add a particular feature. This is the student page or this is something for marks. So, there are two different features that I want to take care of. For that what I do is I branch off from my development branch, develop each one of them independently. And at some point, I merge back again, over here into the develop branch never into master.

I mean, you can do it into master the point is, this is a particular flow that is logical and makes sense. You merge back into developer after that it looks good at this stage, I am quite happy with how things look. So, I go for a release candidate. This is something called a release branch. I iterate through the release candidates fixing some bugs and, making sure everything is working properly.

At some point, I am like, happy about this. I push it and merge into master again. So, all of this saying that, I am going to push code I am going to merge and so on, if you look at it code is just a bunch of text files. So, merging just means taking the contents of two text files, making sure they do not have any conflicts with each other and putting the files together maybe adding a new file with new functionality.

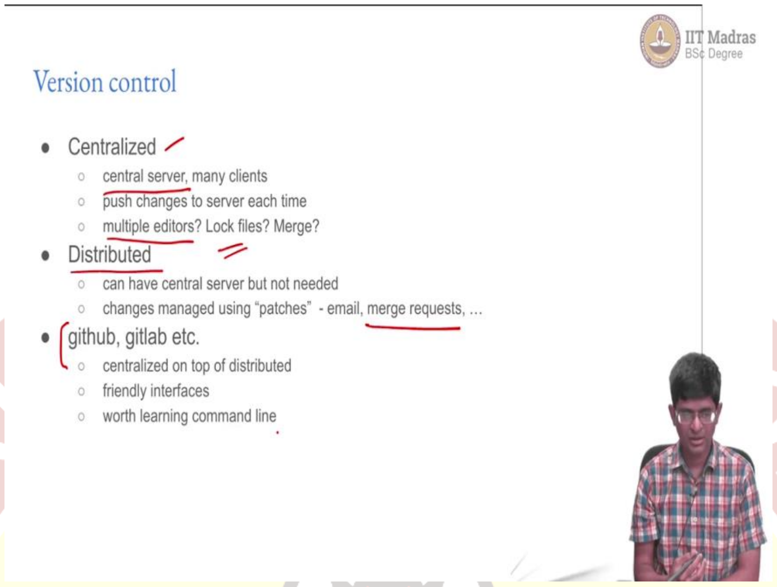
Pushing means, this branch had certain files with certain data in it. Now, I am also going to merge that or push that into the server, the main branch. So, push or merge, both of those ultimately have the same kind of functionality. The way it is done ultimately leads to different ways of looking at it, you could either say it is a push to the main branch, or I could generate something called a pull request.

And the pull request says that, somebody else is in charge of master, I cannot directly push into master. So, what I can do instead is I say that everything is and ask my manager to say, take a look at my code and merge it in. So, now the manager has to pull it. It is either a pull request or a merge request.

So, all of this terminology is something that you will see when you start using Git in a regular workflow a lot more. Once again, like I said, that is pretty much out of scope of this particular course, it is a whole area of software engineering in itself. But it is well worth sort of trying to learn at least the basics on your own to start with, and there is a lot of documentation online, on how to, on best practices.

The problem is, there is a lot of documentation online, so you do not know which one to follow either. And whatever I specify as being, this is a good way of doing it might not be what a particular other company finds to be best practice. So, at some point, you will need to get familiar with the basics. And from there on, be able to adapt to different kinds of flows that you have.

(Refer Slide Time: 11:37)



Version control

- Centralized
  - central server, many clients
  - push changes to server each time
  - multiple editors? Lock files? Merge?
- Distributed
  - can have central server but not needed
  - changes managed using "patches" - email, merge requests, ...
- github, gitlab etc.
  - centralized on top of distributed
  - friendly interfaces
  - worth learning command line

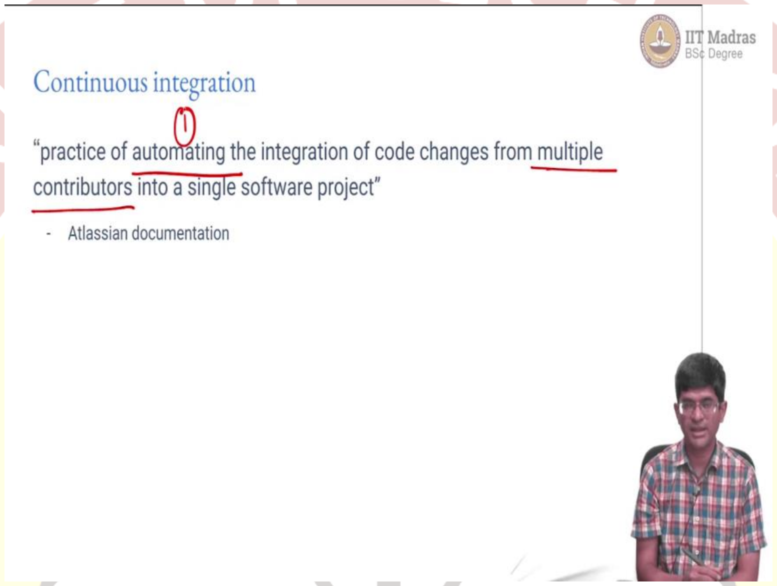
So, version control, in other words, you could have multiple ways of doing it. One of them is the so called centralized version, which is the relatively the older versions of version control SVN and CVS before that. What they did was they had a central server, and many different clients, and each of those clients would sort of check out a version of the code, make changes, and push changes back.

And the reason for having this checkout is that what happens if multiple people try editing a file, and they did not want that to happen. So, what they would do is, they would say that when you check out code, you are locking it, so nobody else can make changes to it. Now, that is good, because, you make sure that two people do not change the same file in different ways. The problem is, somebody might check it out, and then forget they check it out and nobody else can work on it in the meantime.

The other approach that was taken by git is that everything is done using so called merges, branches and merges. So, this follows something called a distributed form of version control. You can have a central server if needed, but it is not required. That is the main point. And changes are managed by essentially people exchanging what are called patches, that exchanging could be done over email, or through a centralized server. And this is where services like github and gitlab have come into play. What they do is they sort of provide that centralized overlay on top of a distributed version control system, which is Git.

And very friendly interfaces, they take care of the web interface, they take care of SSH based interfaces, they provide you with functionality, like hooks, and tokens for deployment and various other things, which are not part of the basic Git. And it is well worth sort of learning how to use it from the command line, not just from the web interface, simply because there are certain things, especially when you start scaling up become easier using the command line.

(Refer Slide Time: 13:41)



Continuous integration

"practice of automating the integration of code changes from multiple contributors into a single software project"

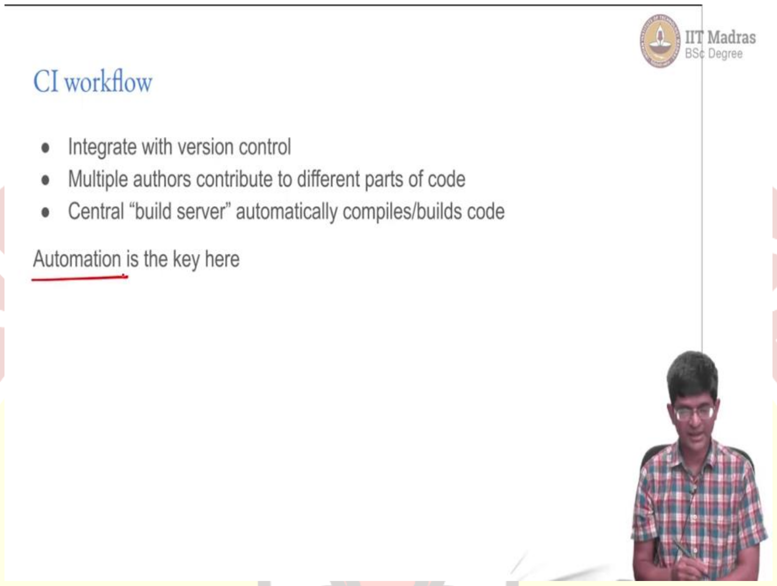
- Atlassian documentation

So, now that we have a reasonable idea of what the version control side of things is doing, how does that sort of, tie in with the next steps, which I am going one of them is called continuous integration. And what is the definition of continuous integration? This is one sample definition from the documentation of a company called Atlassian.

Atlassian provides you with various, they provide, they are the owners of bitbucket, which is yet another version control setup, which can build on top of Git. It uses Git at the back end, but, it provides you with the web interface and so on. They also do various other things like issue tracking, wikis, documentation, lots of other things, and it is a good sort of example of software being provided as a service.

Now, their definition of continuous integration is it is the practice of automating the integration of code changes from multiple contributors into a single software project. And this is a fairly standard definition. So, it is not like there is a strange definition or anything like that. The key words being automating, and of course, the integration of code changes from multiple contributors. This is also there, but the automation is probably the number one thing to keep in mind.

(Refer Slide Time: 15:05)



CI workflow

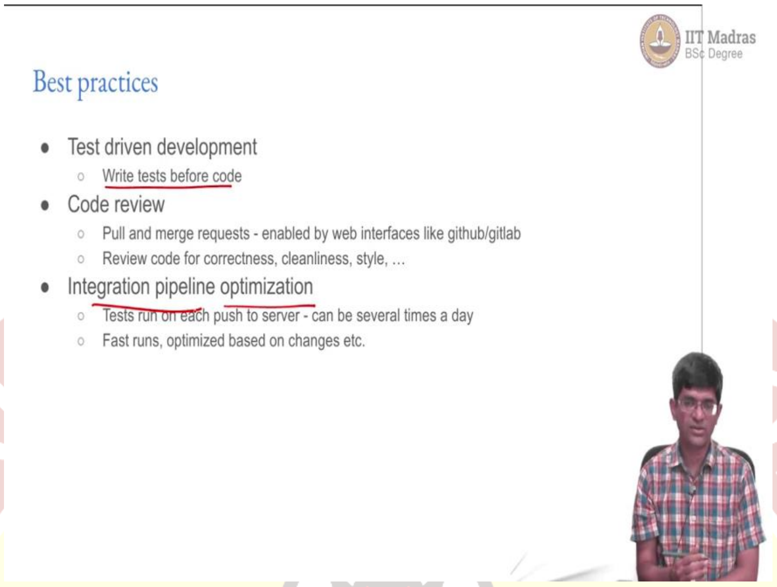
- Integrate with version control
- Multiple authors contribute to different parts of code
- Central "build server" automatically compiles/builds code

Automation is the key here

So, the idea behind continuous integration is that you want to integrate with the version control. Multiple authors can work using version control. And they can, create various kinds of pull requests, merge requests, and so on. But any time that you push your commits your changes back onto the centralized server, there is some kind of functionality on the central server that will automatically compile the code, build it.

It is called a build, when you compile everything and generate, what is the final, deployable package. And will run all the tests for you. And like I said earlier, automation is the key here this is what you are really interested in, the fact that it is on a central server and so on is, good. But the fact that is done automatically is the main point.

(Refer Slide Time: 15:59)



Best practices

- Test driven development
  - Write tests before code
- Code review
  - Pull and merge requests - enabled by web interfaces like github/gitlab
  - Review code for correctness, cleanliness, style, ...
- Integration pipeline optimization
  - Tests run on each push to server - can be several times a day
  - Fast runs, optimized based on changes etc.

So, there are a number of best practices that are mentioned in the context of CI, one of the first being test driven development, always write your tests before code. Now, there are people who sort of argue both for and against this, but just from a pure logical point of view, this makes sense.

Before you start writing your code, always think about how you are going to test it. Because otherwise, you have written a lot of code, and then you are sitting there staring at it and saying, does it work correctly, even if you want to just try it out and say, is this the writing, that is a test. So, you should have those tests in mind saying, how am I going to test?

Am I going to, run the entire code, open a browser, look at it, that is a test. It is not a particularly easy to automate test, but it is a test nevertheless. But if you can write tests that can be automated, that is much better. Because what it means is, you can in fact, write those tests upfront, create the entire test suite, and keep it ready for you. Then write your code and make sure it passes. Each of those tests may be one by one or all at a time.

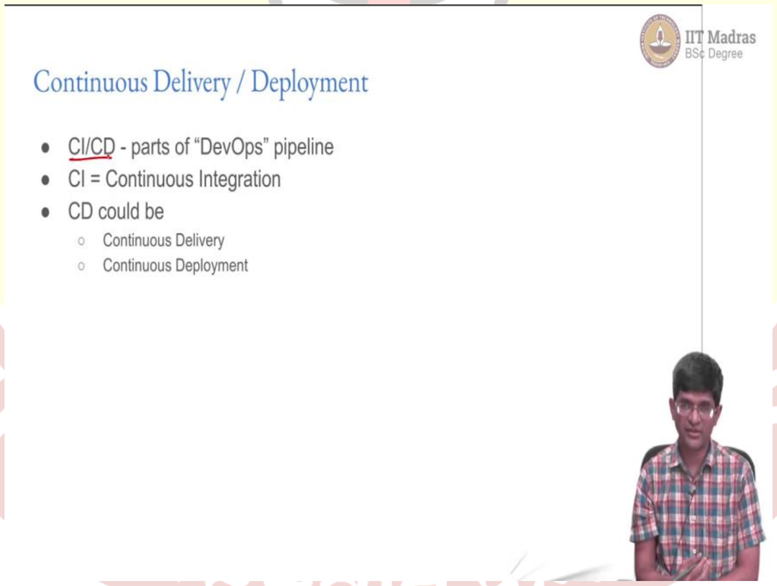
Then code review. So, continuous integration is good in the sense that, it automatically runs all the tests for you. But you should also not get into the practice of just pushing tests and let the server sort of find out that there are problems with your code.

So, code review is a practice whereby people who have more experience writing code, look at code written by people with less experience, presumably, and are able to give feedback saying, look, this is not a particular good way of writing the code. It might be to do with functionality, it might be to do with performance issues, it might just be to do with style. I mean, you are not written this cleanly enough, you need to documented better or indented better, or do something else. But go make some changes, that helps to sort of bring down the number of tests that need to get rerun every time.

And finally, remember that continuous integration means that every time you are pushing code, all the tests have to be run again, which means that how the tests are created, what tests are run at any given time, you need to do some kind of optimization. That is usually called the integration pipeline. Because what happens in the integration pipeline is you push, and there are a number of steps that need to be followed in order to run the tests.

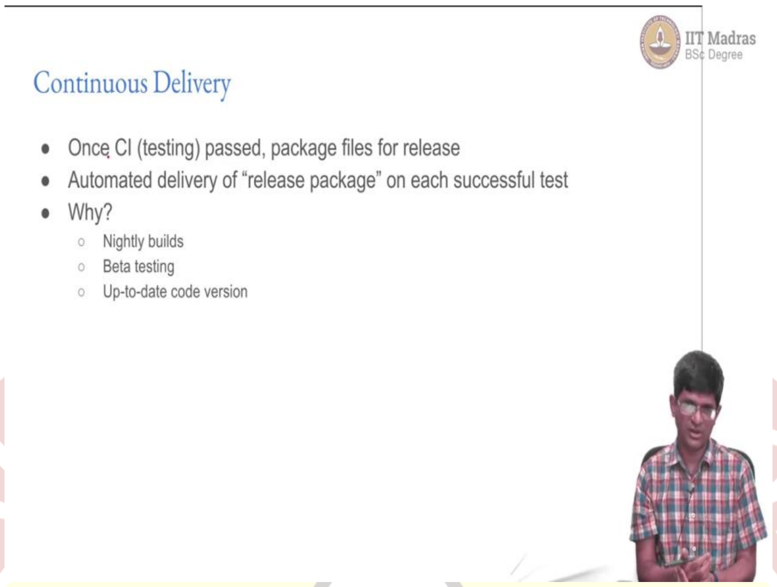
Those steps are called the integration pipeline, optimizing those and seeing, are you able to run just the tests that are needed at any given point in time and do those tests run fast, is a key to having a successful continuous integration.

(Refer Slide Time: 18:39)



Continuous Delivery / Deployment

- CI/CD - parts of "DevOps" pipeline
- CI = Continuous Integration
- CD could be
  - Continuous Delivery
  - Continuous Deployment


Continuous Delivery

- Once CI (testing) passed, package files for release
- Automated delivery of "release package" on each successful test
- Why?
  - Nightly builds
  - Beta testing
  - Up-to-date code version

So, after continuous integration comes what is called CD? And there is this thing, term CI, CD that you might have come across in lots of places, it is part of what is usually called the DevOps pipeline? DevOps sort of stands for development operations something like that. And DevOps is now one of the common words that you keep hearing in the context of, how to deploy applications.

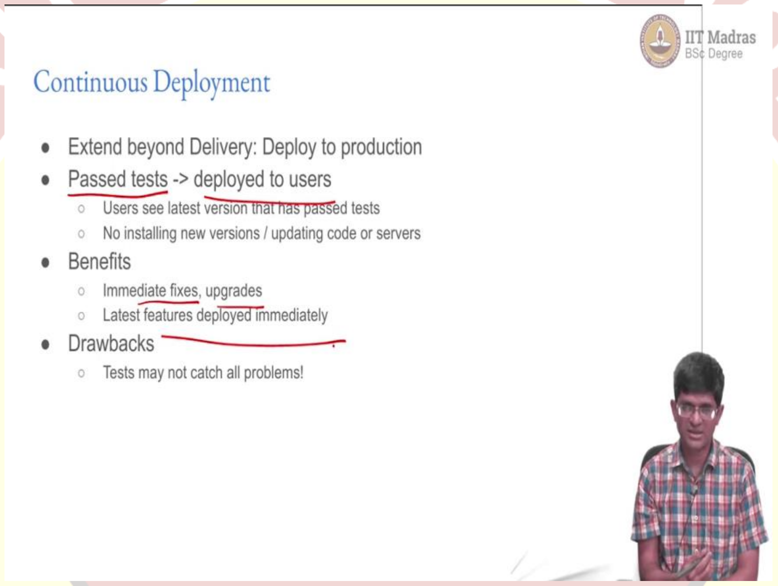
CI, of course, is the continuous integration part of it. The CD has like multiple definitions. One of them is continuous delivery, the other is continuous deployment. Let us look at what those mean. So, the continuous delivery part, what it says is, your CI has passed, all the tests passed. Now what you need to do is to say, whoever is your end user, whether it is a package that you are trying to create, which is, let us say an app that people download and run on their local machines, or it is an app that gets deployed on a server and, is run over there. That is the delivery part, packaging everything together and putting it into one bundle.

So, what you can do is you can have scripts at the shell script or a python script or whatever it is, which once your tests have passed, then take all the files that you need, puts them together in a standard way, and creates, let us say, a zip file which somebody else can download. That is a delivery part.

Now, why would you want to do something like this, there are applications, even something like, libra officer whatever, where you can find that there is something called a nightly build. Every night, there is a build that is created of the latest version of the software. And people who are really interested in having access to the latest features can download the nightly build and work with that.

You get up to date versions of code, you might want to do beta testing, are there new features that people are really testing out and are not sure whether they want to integrate into the final thing, but you are really keen on having access to all of that, then great, go with something like this. So, this continuous delivery can facilitate things of that sort.

(Refer Slide Time: 20:53)



Continuous Deployment

- Extend beyond Delivery: Deploy to production
- Passed tests -> deployed to users
  - Users see latest version that has passed tests
  - No installing new versions / updating code or servers
- Benefits
  - Immediate fixes, upgrades
  - Latest features deployed immediately
- Drawbacks
  - Tests may not catch all problems!

And continuous deployment, is applicable more for web based services. So, continuous delivery could even be for something like libra officer visual studio code. Where you have nightly builds, continuous deployment is probably more meaningful in the context of a specific service that needs to get deployed on a server.

And what it says is, you first need to package the delivery package, but then also push it to a server, unpacked over there, restart the services, if needed, so that from now on, let us say, it was a python flask based application, you restart it, so that now it is running the new code, in production.

And continuous deployment says, the moment you have passed all the tests, it will package it, push it out and deploy it to users, which is great in the sense that the users will see the latest version of the code that has passed all the tests, they never need to install a new version, they never need to say something was old. I need to update my software, that does not come into the picture. Because the software on the server just gets updated as soon as it has passed all the tests. So, the next time a user comes and visits that website, they are seeing the new version of the code. So, the benefits are immediate fixes if you have a bug.

Also upgrades, and latest features can be deployed immediately. But the flip side is, you could potentially have tests that have not caught all possible bugs. Because testing is not some kind of bulletproof process that guarantees correct functionality. It means that whatever tests you wrote, passed, but what if you did not have a test for some particular bug or some other problem crept in which you had not thought of before.

That means that when you have continuous deployment, you can break a system, because it would automatically push it out. And maybe some tests that you did not think of gets triggered and causes the system to crash. Or some people have a poor performance because of it. So, continuous deployment has benefits. It also has certain drawbacks do you really want to go for that is something that you need to think about and decide.