**12) How often team should do merge to master?**

The frequency of merging changes into the master branch depends on the development process and the size of the team. However, it is generally recommended to merge changes into the master branch frequently to avoid conflicts and ensure that the codebase remains stable.

In agile development methodologies, it is common to merge changes into the main branch at the end of each sprint, which typically lasts 1-4 weeks. This approach ensures that the codebase is stable and tested before new changes are introduced.

However, in more traditional development methodologies, changes may be merged into the master branch less frequently, such as at the end of a feature development phase or release cycle.

Ultimately, the frequency of merging changes into the master branch should be determined based on the needs of the project and the team's development process. It's essential to have clear communication and collaboration within the team to ensure that changes are integrated seamlessly and that the codebase remains stable.

**13) How GIT helps in CI/CD pipeline?**

WHAT IS CI/CD

CI/CD is an abbreviation for Continuous Integration and Continuous Delivery. It’s a process of delivering your code continuously using an automated pipeline as opposed to doing a manual.

Continuous Integration

Most of the software teams use a source control system like Git. We’ve several different branching strategies like [Git Flow](https://nvie.com/posts/a-successful-git-branching-model/). All of the strategies have one thing in common. There is a single mainline development branch. Each developer creates his/her own branch of the mainline and works independently on a feature. Once the feature is ready, the code is reviewed and merged back to the mainline.

Continuous Delivery

Now that the code is merged into the mainline, we need to go through the rest of the steps like deploying it, doing QA, and eventually, pushing the new code to production to serve some real users. Each step involves some build process that has to happen to build, test, and deliver the code to the right place. This is where an automated pipeline comes into play where each step is automated following some established conventions everyone on the team follows.

ADVANTAGES OF USING CI/CD

Speed

The main advantage of CI/CD is the increase in velocity. Once you have the process and the pipeline in place, you’ll be able to ship your changes faster. As with any process, it’s just more efficient to do the same steps over and over again rather than shipping changes ad-hoc differently each time. It removes the need for extra communication between the team members and between different teams. Everyone follows the same process. Developers know where to merge to code. QA team knows where to get a new build to test. It’s clear how to deploy to each environment including production and how to troubleshoot some issues.

Quality

CI/CD brings some automation allowing you to take advantage of automated tests, having your compiler catch bugs, and your automated deployments exercising your infrastructure. With all the automation in place, you start investing in the health of your system, getting some dividends with each test added instead of playing catchup using some manual testing. Every run of your pipeline runs hundreds (and sometimes even more) of automated tests to make sure your product is rock solid when it comes to production.

Consistency

As we all know, humans are inconsistent, and computers are great at doing the same thing over and over again. No human will be able to build, test, and deploy your changes multiple times a day consistently. Automated pipelines, when done right, always work for you. Consistency makes your compliance process easier to maintain. And while the CI server is running your pipeline, you have your developers writing some new features.

ANATOMY OF A CI/CD PIPELINE

Most of the automated pipelines consist of the following major steps:

Build

While developers make their changes, we want to make sure the code builds into a runnable product. Here, we run a compiler and/or linter for scripting based languages to make sure we’re able to run the code.

Test

The next step is to test your code. We run our automated tests to make sure the code we built does what we expect it to do. At this step, we’ll catch any bugs that happen along the way so the developer can fix them early in the process before they go out to the real world.

### Deploy

*Now the code is built and tested, let’s go ahead and deploy it to a real environment. We’ll package the code for deployment including the necessary libraries and the configuration. Once it’s ready for us to push, we deploy it directly to the cloud provider using their API.*

### Verify

*Once the code is deployed, we’ll run some comprehensive, non-destructive tests against a deployed environment to make sure the code runs in the cloud the way we expect it to run.*

### Notify

*Once the pipeline is finished running, we notify the developer of the successful deployment. Most of the notifications go to slack or email. Important to note that if failures happen along the way, the pipeline notifies the developer of the failed step so the action can be taken promptly.*

## HOW YOU CAN IMPLEMENT CI/CD

### Lock in Your Branching Strategy

*It is recommend following*[***Git Flow***](https://nvie.com/posts/a-successful-git-branching-model/)*or some variation of it. Making sure everyone follows the same process for branching makes everything easier. If you have multiple environments (like development, staging, and production) you may want to name your branches the same way so it’s easy to map the branches to the environments the code should be deployed to.*

### Build, Test, Deploy

*Chances are, you either already have a way to build, test, and deploy your code. The goal here is to distill it down to a single command you can run from your terminal. If your build process involves running multiple commands, no problem, we’ll just write a build script running each step. If you don’t have automated tests, the best first step is to set up a testing framework and add it to the CI pipeline. Once tests run automatically, it’s easy to add them one by one gradually. Last but not least, your deployment has to happen using a single command. What we’ve seen in the past is that a lot of software teams end up with a large spreadsheet of steps to run for each release. Worry not, each of those massive spreadsheets can be distilled down to a single command step by step.*

### Pipeline it

*Now you have all your steps in order, it’s easy to put it together into a pipeline. There are numerous technology choices starting from self-hosted options like*[***Jenkins***](https://jenkins.io/)*all the way to some hosted solutions like****[CircleCI](https://circleci.com/" \o "CircleCI website" \t "_blank)****and*[***Azure Pipelines***](https://azure.microsoft.com/en-us/services/devops/pipelines/)*. It all depends on your system and your current technology stack. At this point, it comes down to either creating an account for a hosted CI provider or standing up Jenkins inside your infrastructure and wiring the steps together with some configuration files using YAML or similar configuration language. Yes, we want our infrastructure to*[***run as code***](https://en.wikipedia.org/wiki/Infrastructure_as_code)*so we can stand it up at any time.*

## CONCLUSION

*CI/CD is key to increasing velocity, developer productivity, and it’s just more fun to focus on features. Which in turn drives the business forward. If you’re in a manual world without the pipeline or perhaps even without the automated tests in place, starting on CI/CD route might seem like an overwhelming task. Worry not, with the right processes in place and some persistence, you’ll get to a place where all you do is push some new code and let the pipeline do the rest of the magic.*