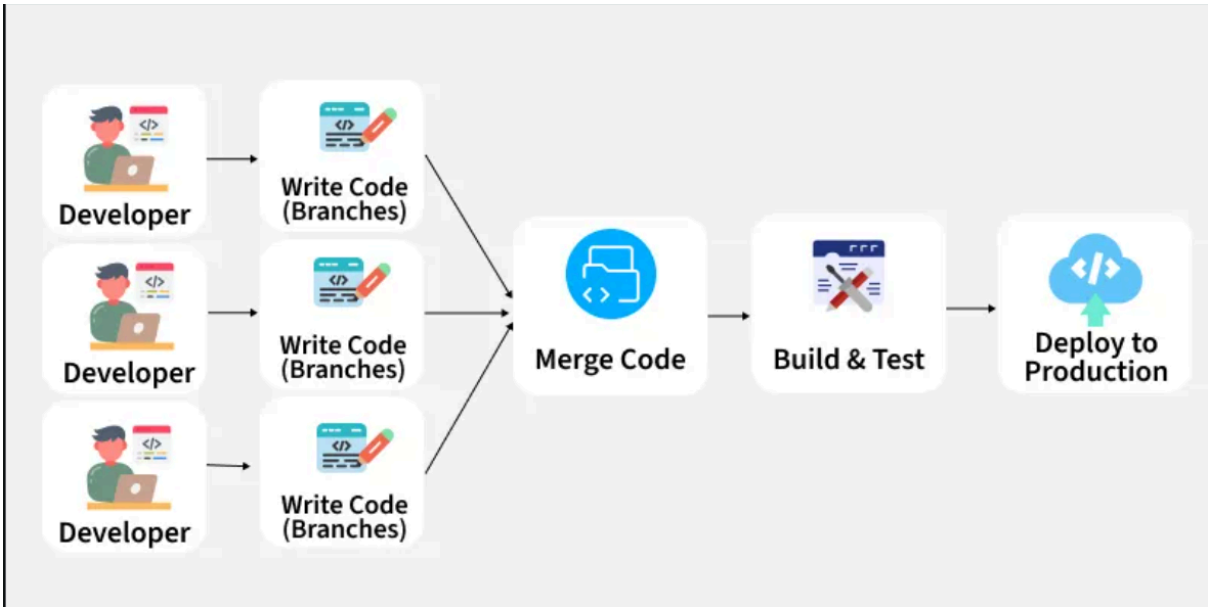
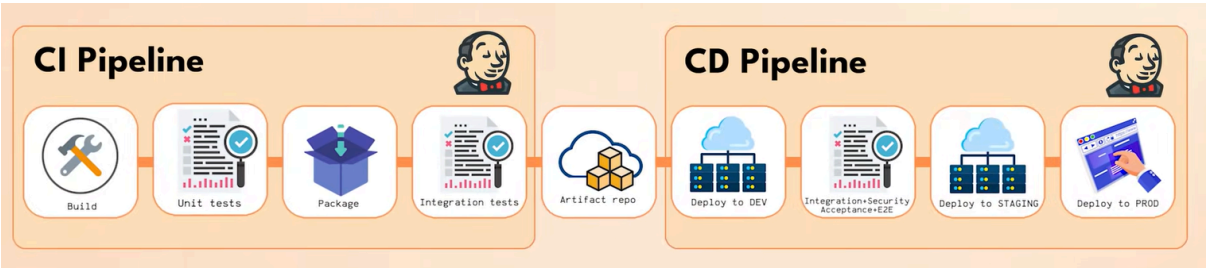
Before CI/CD

- Each Developer wrote code on their laptops.
- They didn't merge code frequently — sometimes only once every few weeks or months.
- When they finally merged code, there would be many Merge Conflicts.
- After the merge, the main branch always broke, so it needed to be fixed first, manually by resolving merge conflicts.
- After conflicts got resolved, usually near the sprint end → code freeze → Deploy to DEV.
- Intense manual e2e testing is performed on this deployment.
- If something is found while testing, immediate fixing is required since the clock is ticking on deployment.
- After the testing is done, someone senior from the team makes the changes to the docker compose file manually, and then deploys the changes.
- At each step, there is manual intervention, errors resulting from it, and delay.
- If something at PROD breaks after the deployment, the users are stuck.

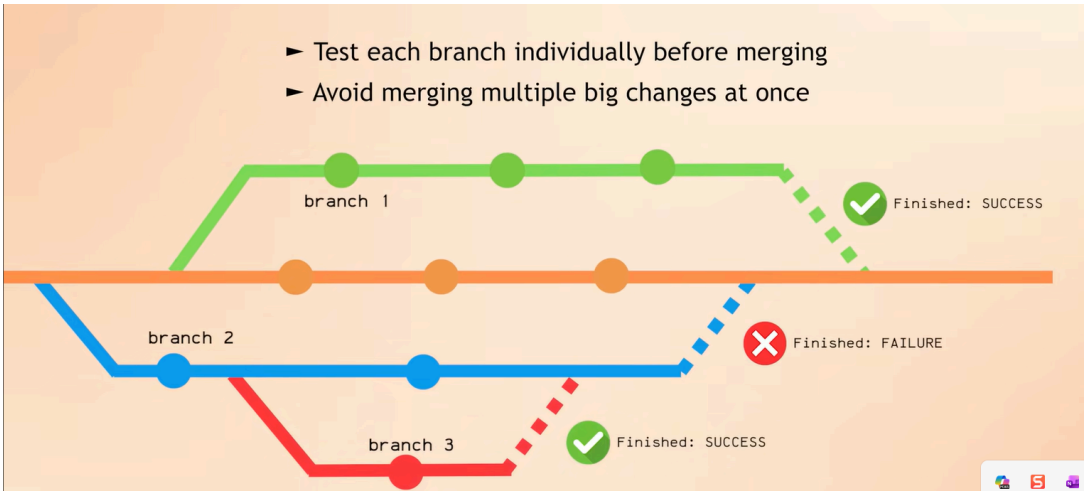


After CI/CD

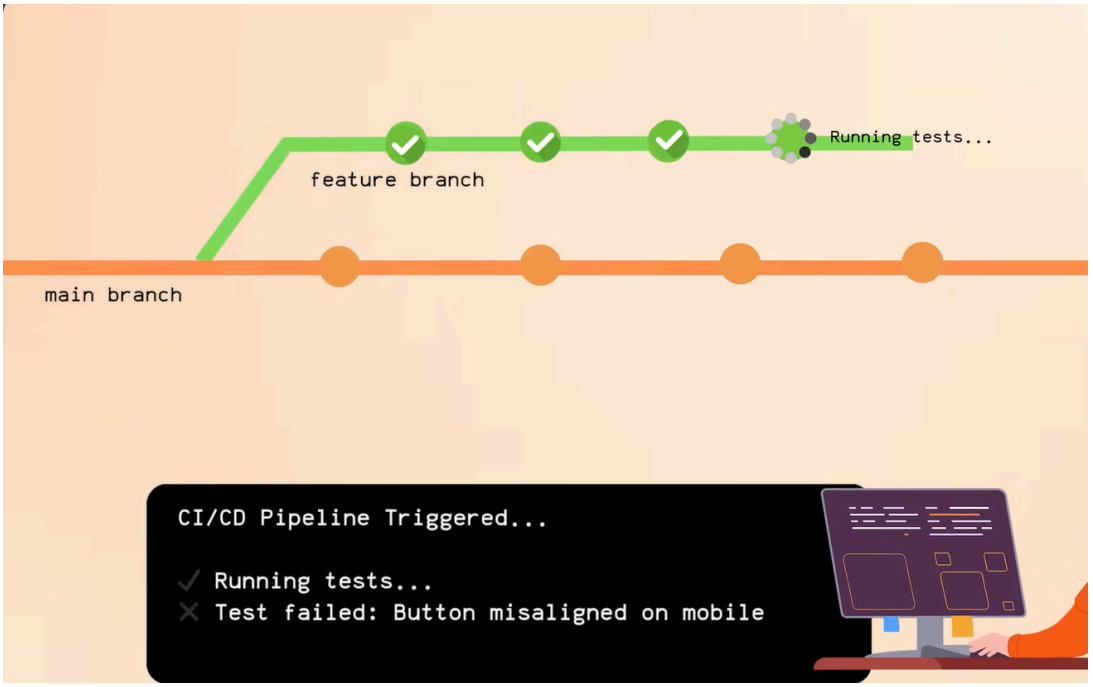
- We can actually reduce this Human element and add some automations here. This is where CI CD comes into the picture.

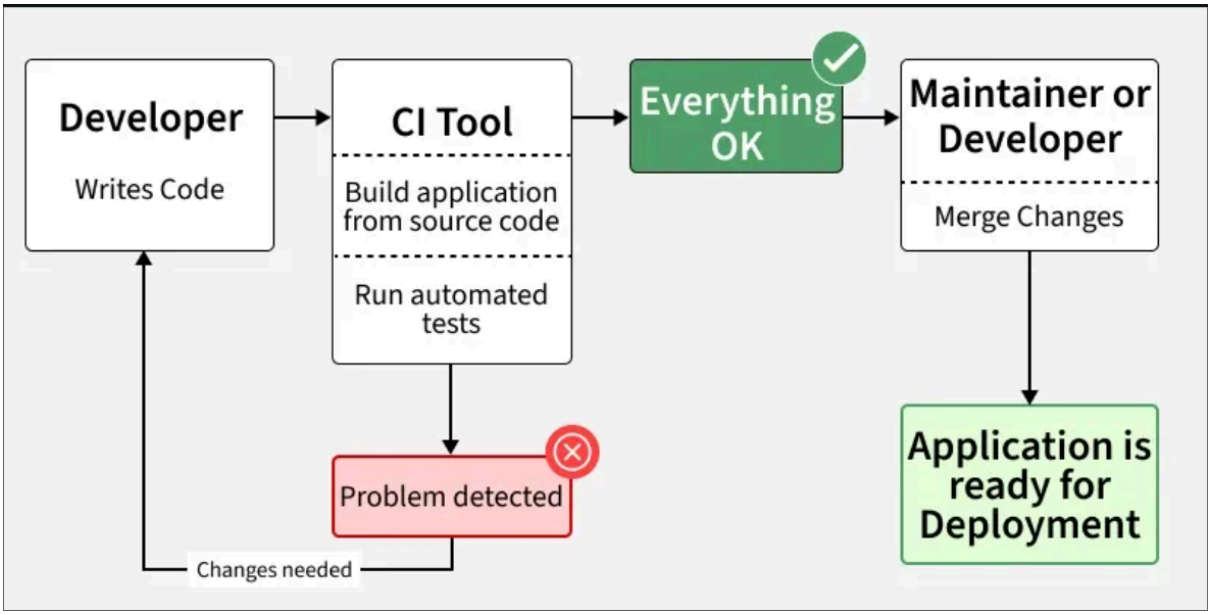


- Our merge conflicts problem, why are we fixing after merge, why can we test on the feature branch itself



- Push the changes to remote more often and check each commit by running, this continuous Integration will isolate the possible issue early.





Every commit is automatically built, tested, and validated in an isolated environment before merging.

What actually happens in CI?
When you push a commit (or open a pull request), your CI system (GitHub Actions, GitLab CI) automatically:

- Checks out your code
- Installs dependencies
- Runs your configured CI steps

Those steps can include (depending on the configuration):

- Linting
- Unit tests
- Integration tests
- Type checking
- Security scans
- Running small parts of the app (if configured)

Where does it run?

- CI runs in a temporary virtual machine or container provided by the CI service.

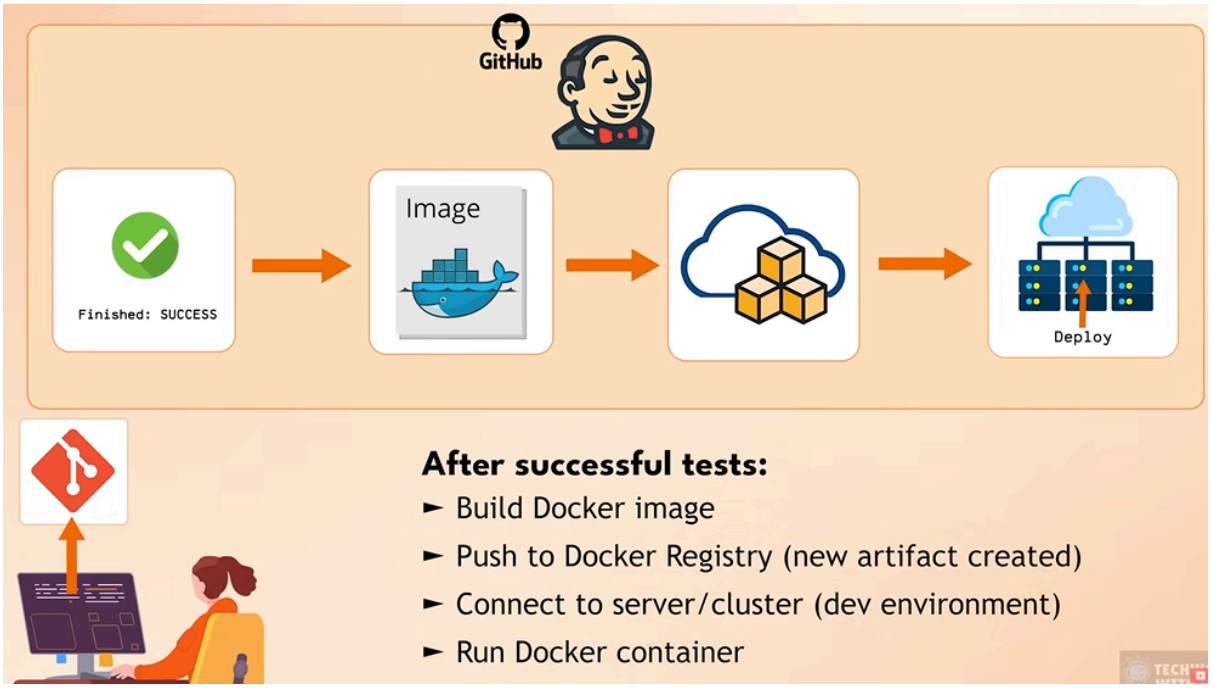
This is CI (Continuous Integration). “Keep changes small and integrate regularly.”

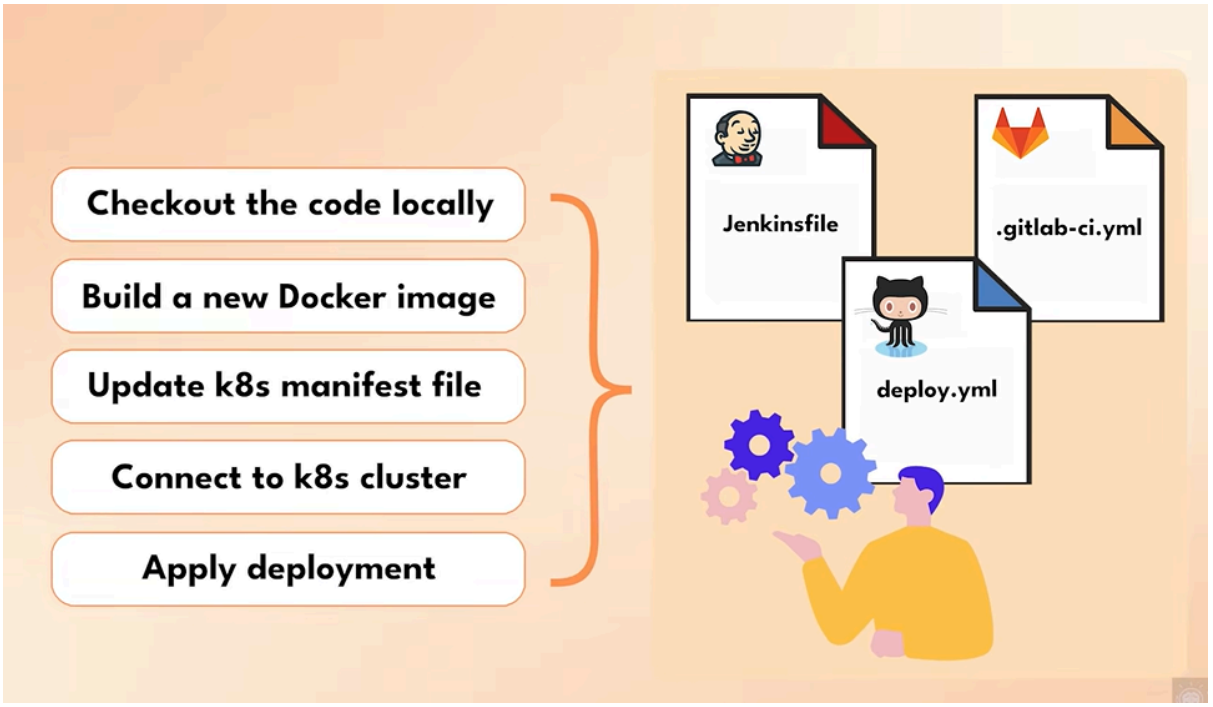
Impact of CI:

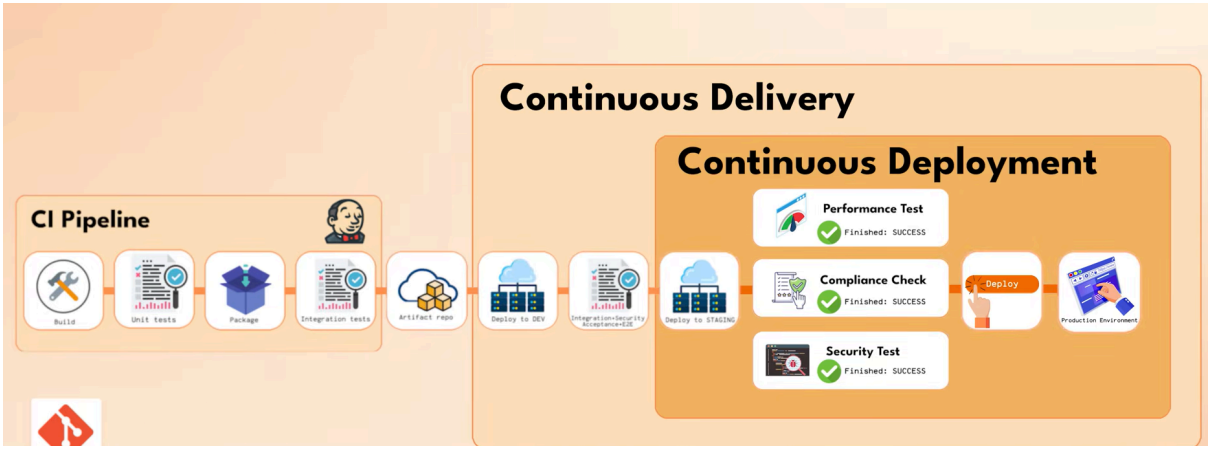
- We have earlier feedback, so issues are caught early.
- Less stress at the time of release since the merges are clean, and tested a lot of times.

What is Continuous Delivery (CD)?

- After all tests are run in CI phase,
- Build a new docker image say, update the version tag
- Push to docker registry and run it. This is deployment.
- This all is not done manually, it's automated in the following files: .yml files.







Artifacts:
Artifacts in CI/CD are files produced by your pipeline that you save for later use — not always executable apps.
They are basically output files generated during the build or test process.

Artifacts can be:

✅ Build outputs
Compiled binaries
APKs, IPAs, JARs, DLLs
Bundled frontend apps (dist folder)

✅ Test outputs
Test reports (JUnit, HTML)
Coverage reports
Logs
Screenshots
Videos

At the end of the CI pipeline, you typically generate production-ready artifacts, such as:

Docker images
Compiled binaries (JAR, DLL, EXE, etc.)
Frontend build directory (dist/)
Test reports
Versioned ZIP/tar files
The important ones for CD are the deployable artifacts (e.g., Docker image or build bundle).

CD uses those artifacts to deploy. CD does NOT rebuild the app.
This is the core principle of CI/CD: build once, deploy many times.
Instead, it deploys the exact same artifacts that CI created on different envs.

This gives:
✔️ Consistency -> The code that passed CI tests is exactly the code that goes to staging/production.

✔️ Safety -> If you need a rollback, CD redeploys the previous artifact — not rebuild.
✔️ Speed -> CD is faster because it doesn't run package/build again.

Overall CI/CD (Typical Workflow)

1️⃣ **Developer pushes commit**

→ CI starts

2️⃣ **CI runs**

Checkout code
Install dependencies
Build the app
Run tests
Package artifacts
Store artifacts

3️⃣ **CD pulls those artifacts**

Deploys to staging
Runs health checks
Deploys to production (manual or automatic)

Benefits of CI CD:

- More focus on building, less on deployment, since automated processes are there.

Continuous Delivery Vs Continuous Deployment

Continuous Delivery

- Code is automatically built, tested, and prepared for release — but requires manual approval to deploy to production.
- Key idea: "The app is always ready to deploy, but a human decides when."
- Deployment happens only when someone clicks Deploy or approves a release. So, manual steps still exist here.

Continuous Deployment

- Every commit that passes CI is automatically deployed to production — with NO human approval.
- Key idea: "If tests pass, it goes live automatically." No manual steps.
- But if something breaks, there must be a rollback mechanism to revert to the last stable version.
- This is why most companies don't prefer this.

Deployment strategies

In case, any bugs reach the end user in production after the CI/CD, we can still tackle it via certain deployment strategies so that there is zero downtime.

Blue Green deployment strategy

1️⃣ Blue environment is live
- Users are currently using the Blue version of your app.

2️⃣ Deploy new version to Green
- You deploy and test the new release in the Green environment:
- Run automated smoke tests
- Perform manual QA
- Check logs
- Run health checks
- Green is not exposed to users yet.

3️⃣ Switch traffic from Blue → Green
- When Green is fully validated:
- Update the load balancer / router
- All new user traffic goes to Green
- Blue is now idle but still running
- This switch is instant and has no downtime.

4️⃣ If something goes wrong → instant rollback
- Just flip traffic back:
- Redirect users back to Blue
- Fix issues in Green
- Try again later
- Rollback takes seconds.

Blue = Standby in case of failures
Green = new env deployed.

Disadvantage: Double infra cost due to double deployment.
Advantage: Zero downtime.

Canary Deployment strategy

We roll out a new version to a small percentage of users first, monitor it, and then gradually increase traffic until everyone is using the new version.
It's named after the old practice of sending a canary into a coal mine—if the canary died, miners knew something was wrong.
In software, the "canary users" detect issues before the full rollout.

Rolling Deployment strategy

A Rolling Deployment updates your application gradually across your servers or pods, one batch at a time, until all instances run the new version.

You replace old instances with new ones incrementally, instead of all at once.

1️⃣ Start with N running instances (pods/VMs/containers)

Example: You have 10 replicas of your service running version 1.0.

2️⃣ Deploy new version to a small batch

For example: Take 1 instance out of service
Replace it with new version (1.1)
Wait until it's healthy

Now you have: 9 instances of v1.0
1 instance of v1.1

3️⃣ Continue replacing instances one batch at a time

Replace another 1 or 2 instances
Wait for readiness/health checks
Continue the rollout

4️⃣ Eventually all instances run the new version

0 instances of v1.0
10 instances of v1.1

5️⃣ If something goes wrong → rollout paused or rolled back

Orchestration tools detect problems and:
Stop the rollout Or replace new instances with old ones

Actually, Rolling Deployments use less resources than Blue–Green, not more.

Blue–Green = requires 2 full environments (100% + 100%)

You literally have:
Blue → entire production environment
Green → an identical full environment

So for a period of time, you are running double the infrastructure.
👉 Blue–Green deployment cost = ~200% capacity

Rolling = replaces instances gradually, does not duplicate full environment

Rolling deployment works like this:
Remove 1 old instance
Add 1 new instance
Wait
Repeat until all are updated
At most, rolling deployment may briefly require:
1 extra instance (if you configure “max surge = 1”)