CI/CD

DA 410/510 Cloud Computing

Continuous Integration Fundamentals

Overview

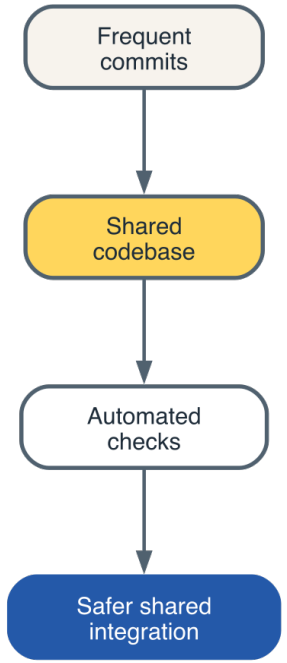
- This lecture introduces the core activities that make continuous integration useful.
- The main focus is on automated builds, unit tests, artifact creation, and fast feedback.

Why Continuous Integration Exists

- Software teams create risk when changes remain isolated for too long and then collide during late integration.
- Continuous integration addresses that problem by making integration frequent, visible, and supported by automation instead of by last-minute manual effort.

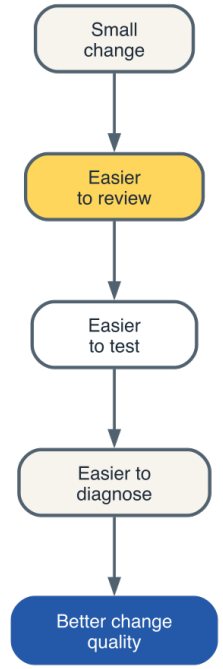
What Continuous Integration Means

- **Continuous integration (CI)** is the practice of merging code changes into a shared codebase regularly and verifying those changes through automated checks.
- CI matters because it reduces the delay between writing code and discovering whether that code works safely with the rest of the system.



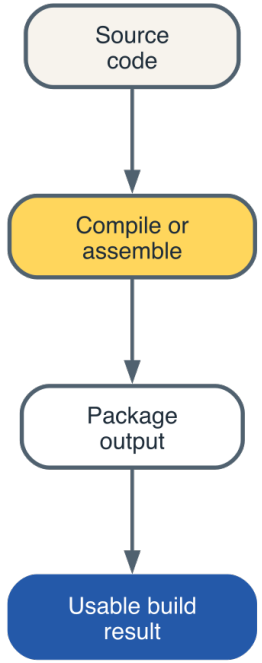
Why Frequent Integration Improves Quality

- Frequent integration improves quality because smaller changes are easier to review, easier to test, and easier to diagnose when something fails.
- It also helps teams detect conflicts and broken assumptions before those problems expand across many days of work.



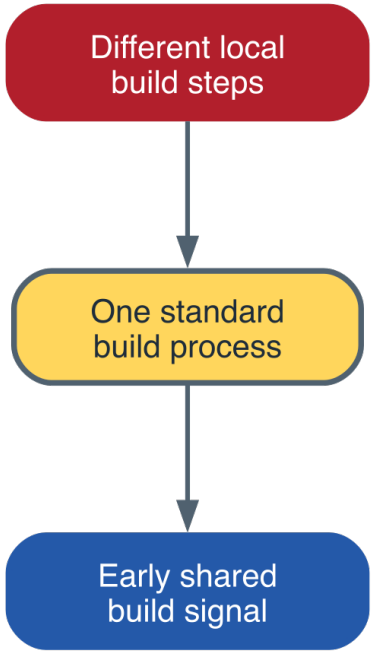
What “Automated Build” Means

- An **Automated build** is the machine-executed process that transforms source code into a usable output such as a compiled application, package, or deployable bundle.
- Automated builds reduce reliance on inconsistent local steps and make the build process repeatable across team members and environments.



Why Automated Builds Matter in CI

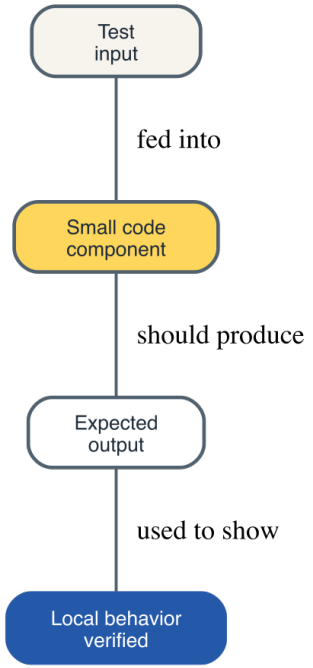
- Automated builds matter because a team cannot trust its release workflow if every developer builds the software differently.
- In CI, the build becomes one of the earliest signals that a recent change may have broken the shared application.



Testing Code

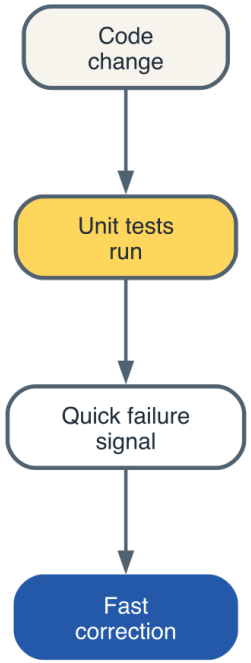
What “Unit Test” Means

- A **Unit test** checks a small, focused part of the codebase to verify that it behaves as expected.
- Unit tests are useful in CI because they provide fast feedback on whether a change broke a local behavior or contract inside the application.



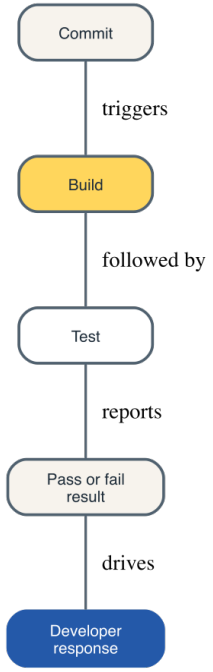
Why Unit Tests Support Safe Change

- Unit tests support safe change because they help engineers discover errors soon after a commit instead of after the application has already moved much farther toward release.
- They are not the only quality measure in a CI system, but they are often the fastest and most frequent signal of change safety.



Builds and Tests Create Fast Feedback

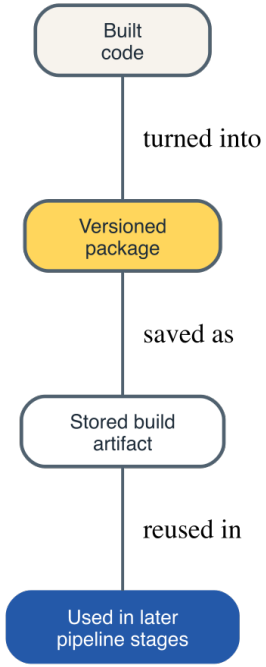
- **Fast feedback** is one of the main values of CI because developers learn quickly whether a recent change introduced a problem.
- When builds and tests run automatically, the team does not need to wait for manual verification to discover obvious integration issues.



Artifacts & Basic Workflow

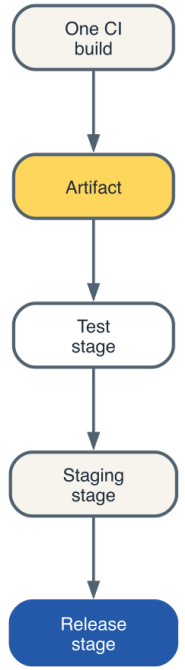
What an Artifact Means

- An **Artifact** is the output produced by the build process that can be stored, tested further, or delivered to later stages of the pipeline.
- Artifacts matter because they give the team a concrete, versioned result of the CI process rather than only a pass or fail signal.



Why Artifact Creation Matters

- Artifact creation matters because reliable delivery depends on promoting the same built output through later stages instead of rebuilding the software differently at each step.
- This reduces ambiguity about what exactly was tested and what exactly is being deployed.

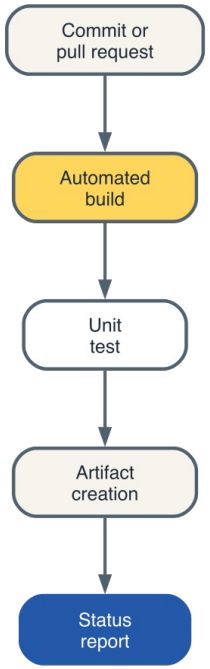


Common Artifact Types

- Artifacts can take different forms, including compiled binaries, package archives, container images, or other deployable outputs.
- The exact format depends on the application, but the operational purpose is the same: produce a stable output that later stages can trust.

The Basic CI Flow

- A typical CI flow begins when a code change reaches a shared branch or pull request, which triggers an automated build.
- The system then runs unit tests, creates an artifact if the checks succeed, and reports the result back to the team.



What CI Does Not Guarantee

- A successful CI run does not mean the software is fully ready for production. It means the current change passed the automated checks included at this stage.
- CI improves confidence, but it does not eliminate the need for later testing, release controls, or operational judgment.

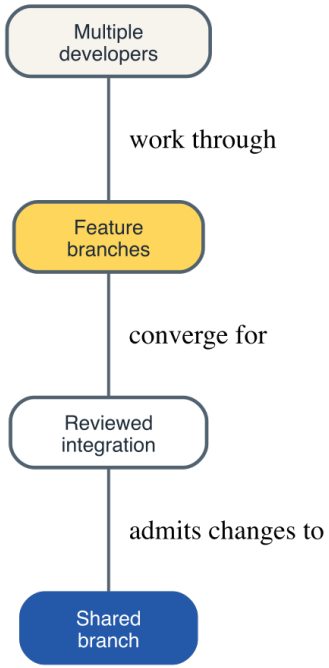
Design Guidance for You

- You should ask whether your CI workflow gives quick quality signals, produces trustworthy artifacts, and runs often enough to keep integration problems small.
- You should also ask whether the pipeline tells the team something meaningful about change safety rather than merely running automation for its own sake.

CI Workflow and Triggers

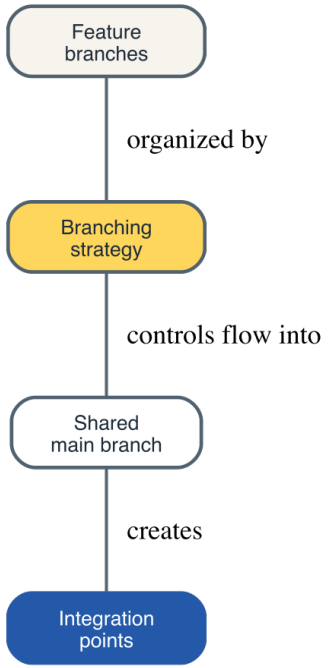
CI Is Also a Collaboration System

- Continuous integration is not only a build server pattern. It is also a collaboration model that determines how developers introduce change into a shared codebase.
- This means team policy around branches, merges, and dependencies is part of CI quality, not separate from it.



What Branching Strategy Means

- A **Branching strategy** is the planned way a team organizes code changes across branches before those changes are integrated into the shared codebase.
- Branching strategy matters because it influences how often changes meet one another and how large each integration event becomes.

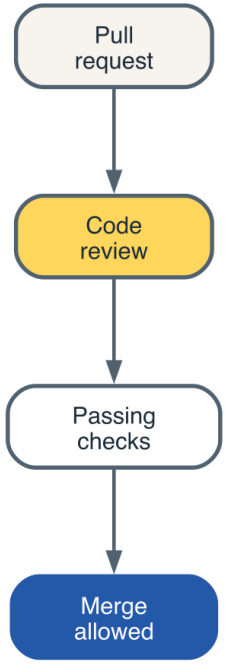


Why Branching Strategy Affects CI

- Short-lived branches usually fit CI better because they encourage small, frequent integrations that produce faster and clearer feedback.
- Long-lived branches often make integration harder because they accumulate larger differences that are more likely to conflict or fail unexpectedly.

What Merge Policy Means

- A **Merge policy** is the set of rules that determines what conditions must be met before code can be merged into an important shared branch.
- Merge policies often include review expectations, required passing checks, and restrictions on direct unreviewed changes.



Why Merge Policies Matter

- Merge policies matter because CI loses value if unsafe changes can bypass the quality signals the team depends on.
- A disciplined merge policy protects the integrity of the shared branch and helps ensure that successful integration actually means something.

Branching and Merge Policy Together

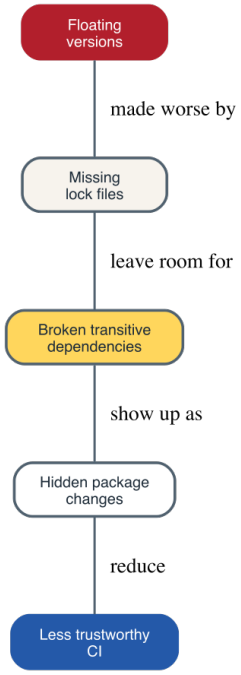
- Branching strategy and merge policy work together. One controls how changes are prepared, and the other controls when they are allowed into the shared codebase.
- Teams need both because frequent integration without review can be risky, while strong review without practical integration patterns can become slow and frustrating.

What Dependency Management Means

- **Dependency management** is the practice of controlling which external libraries, packages, or components the build relies on and which versions are used.
- Dependency management matters in CI because the team needs consistent build inputs if it wants consistent build outputs.

Why Dependencies Affect Reproducibility

- **Reproducibility** means the same source and build process should produce the same expected result when rerun under controlled conditions.
- If dependencies change unpredictably between runs, the team may struggle to tell whether a failure came from its own code or from an external version shift.



Common Dependency Problems

- Common dependency problems include floating versions, missing lock files, broken transitive dependencies, and hidden changes in external packages.
- These problems make CI less trustworthy because the pipeline result may vary even when the team’s own source code has not changed.

Dependency Discipline in Practice

- Good dependency management treats changes to important dependencies as reviewable events rather than as background noise.
- This helps the team preserve stable builds and understand when reproducibility has been threatened by the broader software ecosystem.

What Trigger Means

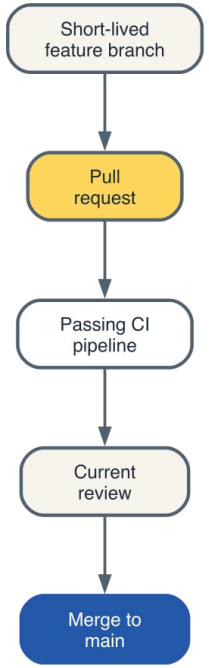
- A **Trigger** is the event that causes the CI pipeline to start, such as a code push, a pull request update, a scheduled run, or a manual action.
- Trigger design matters because it controls when the team receives feedback and which kinds of changes are checked automatically.

Common CI Triggers

- Push and pull-request triggers are common because they align CI directly with everyday development activity.
- Scheduled triggers can also be useful for periodic checks, while manual triggers may support exceptional workflows that should not run automatically every time.

Trigger Choice Shapes Pipeline Structure

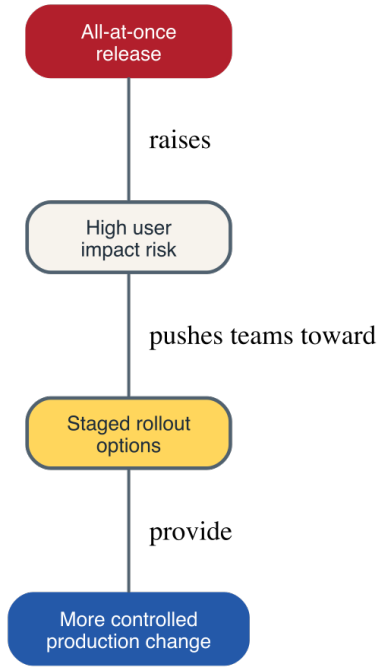
- Trigger choice shapes pipeline structure because not every CI action belongs on every event.
- A pull request may run fast verification checks, while a merge to the main branch may run a broader build or artifact publication workflow.



Safe Release Strategies

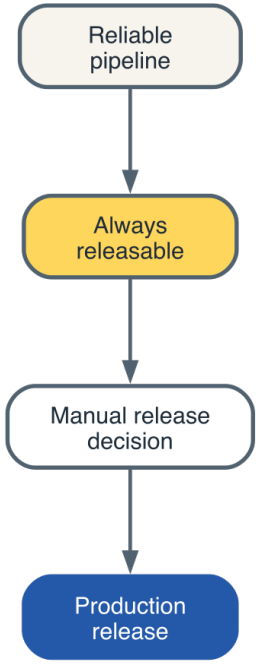
Why Release Strategy Matters

- Releasing software to production is one of the highest-risk moments in the delivery lifecycle because user-facing behavior changes under real load and real data.
- Safe rollout strategy matters because teams need ways to introduce change without turning every deployment into an all-or-nothing event.



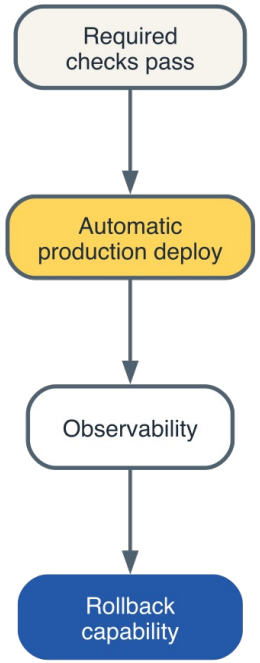
What Continuous Delivery Means

- **Continuous delivery** means the system is built so that software can be released to production at any time through a reliable, repeatable process.
- In continuous delivery, the pipeline prepares the software for release, but a human or organizational decision may still control the final production deployment.



What Continuous Deployment Means

- **Continuous deployment** goes one step further by automatically deploying changes to production when the required pipeline checks succeed.
- This model can increase speed and feedback, but it also requires strong confidence in testing, observability, and rollback capability.

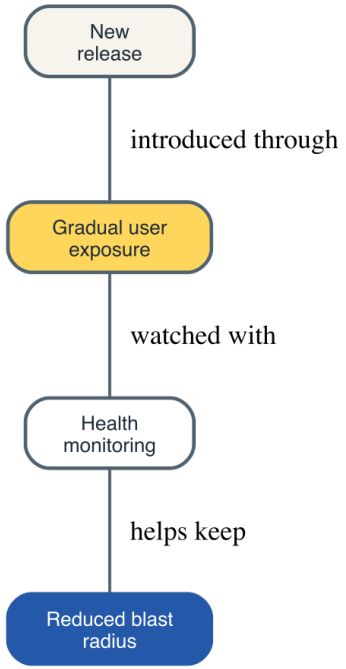


Delivery Versus Deployment

- Continuous delivery and continuous deployment share automation, but they differ in where the final production decision occurs.
- That distinction matters because some organizations need a human gate for regulatory, operational, or business reasons even when the rest of the pipeline is automated.

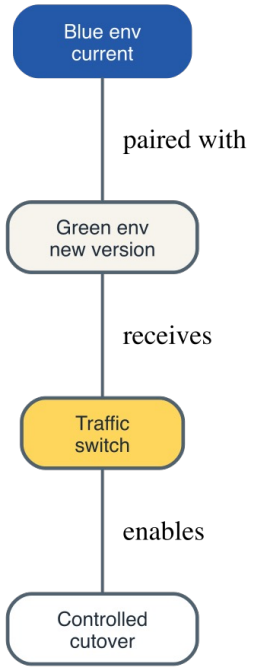
What a Safe Rollout Means

- A **Safe rollout** is a deployment approach that limits the blast radius of change and gives the team a better chance to detect problems before all users are affected.
- Safe rollout strategies are especially valuable when availability, user trust, or rollback speed matters.



What Blue/Green Deployment Means

- A **Blue/green deployment** uses two parallel environments, where one is currently serving production traffic and the other holds the new version.
- When the new environment is ready, traffic is switched from the old environment to the new one in a controlled transition.



Blue/Green Trade-offs

- Blue/green deployment can simplify rollback because traffic can often be switched back quickly if the new environment behaves poorly.
- The trade-off is that running two environments may increase cost and require stronger consistency management between them.

What a Rolling Deployment Means

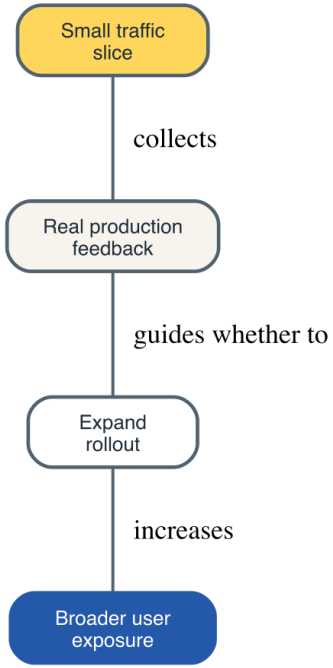
- A **Rolling deployment** updates instances gradually, replacing the old version with the new version over time instead of switching everything at once.
- This pattern reduces the need for a fully duplicated environment, but it requires careful health management while old and new versions coexist temporarily.

Rolling Deployment Trade-offs

- Rolling deployment can be efficient because it uses existing capacity more incrementally.
- The trade-off is that the system may run mixed versions during the rollout, which can create compatibility and monitoring challenges if the release is not designed carefully.

What a Canary Deployment Means

- A **Canary deployment** sends the new version to a small subset of traffic before expanding rollout to a broader audience.
- This pattern is valuable when the team wants real production feedback while still limiting the number of affected users during the earliest release stage.



Canary Deployment Trade-offs

- Canary deployment can reduce rollout risk because failures may become visible on a smaller user segment before the full release proceeds.
- The trade-off is that the team needs strong monitoring and clear success criteria to decide whether the rollout should continue, pause, or reverse.

Choosing a Rollout Strategy

- Teams should choose a rollout strategy by asking how quickly they need rollback, how much duplicate capacity they can afford, and how confidently they can observe partial failures.
- The answer may differ across services in the same organization because workload criticality and architecture are not always uniform.

Monitoring During Rollout

- Safe rollout only works if the team can observe whether the new version is behaving correctly during the deployment.
- Metrics such as error rate, latency, and health-check behavior become part of the rollout strategy because they influence whether the release continues or stops.