Chapter 30: Continuous Delivery and Deployment

1. Introduction to Continuous Delivery and Deployment (CD)

- Once a code change and its tests are merged, it needs to be released to production.
- **Manual release processes** are problematic:
  - They don't happen frequently, leading to batched changes over days or weeks.
  - This *increases the likelihood of release failure.*
  - When a release fails, it's *harder to pinpoint the breaking change*, slowing down the team.
  - Developers must constantly monitor dashboards and alerts.
- Manual deployments are a *terrible use of engineering time*, especially with many services.
- The only way to release changes safely and efficiently is to *automate the entire process.*
- A change merged to a repository should *automatically be rolled out to production safely.*
- The **Continuous Delivery and Deployment (CD) pipeline** automates the entire release process, including rollbacks.
- Releasing changes is a main source of failures, so CD requires significant investment in *safeguards, monitoring, and automation.*
- If a regression is detected, the deployable component (artifact) is either *rolled back* to the previous version or *rolled forward* to a version with a hotfix.
- There's a balance between rollout safety and release time; a good CD pipeline strives for a good trade-off.

2. CD Pipeline Stages: Review and Build

- A code change goes through a pipeline of four stages to be released to production: **review, build, pre-production rollout, and production rollout**.

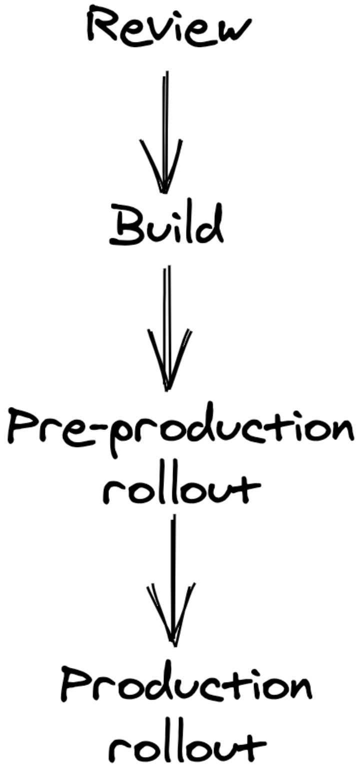

Figure 1: Continuous delivery and deployment pipeline stages

2.1 Review Stage

- It begins with a *pull request (PR)* submitted for review by a developer.
- When the PR is submitted, it needs to be *compiled, statically analyzed, and validated with a battery of tests.* This should take no longer than a few minutes.
- To increase test speed and minimize intermittent failures, tests at this stage should be *small enough to run on a single process or node*, while larger tests

run later.

- The PR must be *reviewed and approved by a team member* before merging.
- The reviewer validates whether the change is correct and safe for automatic release by the CD pipeline.
- A checklist can assist reviewers:
  - Does the change include unit, integration, and end-to-end tests as needed?
  - Does the change include metrics, logs, and traces?
  - Can this change break production (e.g., backward-incompatible change, hitting service limits)?
  - Can the change be rolled back safely if needed?
- Not just code changes, but also *static assets, end-to-end tests, and configuration files* should go through this review process and be version-controlled.
- A service can have *multiple CD pipelines*, one for each repository, potentially running in parallel.
- It's critical to review and release *configuration changes* with a CD pipeline, as they are a common cause of production failures when applied globally without prior review or testing.
- Applications running in the cloud should declare their infrastructure dependencies (VMs, data stores, load balancers) with code (**Infrastructure as Code - IaC**) using tools like Terraform. This automates infrastructure provisioning and treats infrastructure changes like software changes.

2.2 Build Stage

- Once a change is merged into the main branch, the CD pipeline moves to the build stage.
- Here, the repository's content is *built and packaged into a deployable release*

artifact.

3. **Pre-production Rollout**

   - During this stage, the artifact is deployed to a *synthetic pre-production environment.*
   - Although it lacks production realism, it's useful to verify:
     - No hard failures are triggered (e.g., null pointer exception at startup due to missing configuration).
     - End-to-end tests succeed.
   - Releasing to pre-production is *significantly faster* than to production, allowing early bug detection.
   - There can be *multiple pre-production environments*: from those created from scratch for each artifact for smoke tests, to persistent ones mirroring production and receiving a small fraction of mirrored requests.
   - Ideally, the CD pipeline should assess artifact health in pre-production using the *same health signals used in production* (metrics, alerts, tests) to ensure consistent health coverage.

4. **Production Rollout**

   - After successful pre-production rollout, the CD pipeline proceeds to release the artifact to production.
   - **Initial Release:** It should start by releasing to a *small number of production instances first.*
     - The goal is to surface problems not yet detected as quickly as possible before widespread damage.
   - **Incremental Release:** If the initial release is healthy, the artifact is *incre-*

mentally released to the rest of the fleet.

- **Capacity Management:** During rollout, a fraction of the fleet can't serve traffic, so remaining instances must pick up the slack. *Enough capacity* must be available to sustain the incremental release without performance degradation.
- **Multi-Region Deployment:** If the service is in multiple regions, the CD pipeline should:
  - Start with a *low-traffic region* to reduce the impact of a faulty release.
  - Divide releasing to remaining regions into *sequential stages* to minimize risks.
  - The more stages, the longer the pipeline takes. This can be mitigated by *increasing release speed* in later stages once confidence is built.
  - Example: Stage 1 to a single region, Stage 2 to a larger region, Stage 3 to N regions simultaneously.

5. Rollbacks

- After each step, the CD pipeline must *assess the deployed artifact's health* and, if unhealthy, *stop the release and roll it back.*
- **Health Signals for Decision Making:**
  - Results of end-to-end tests.
  - Health metrics (e.g., latencies, errors).
  - Alerts.
- **Monitoring Scope:** Monitoring only the health signals of the service being rolled out is *not enough.* The CD pipeline should also monitor the health of *upstream and downstream services* to detect indirect impacts.
- **Bake Time:** Allow enough *bake time* between steps to ensure success, as

some issues (e.g., performance degradation) appear only after time or at peak load.

  - Bake time can be reduced after early successes.
  - It can also be gated on the number of requests for specific API endpoints to ensure proper exercise of the API surface.

- **Degradation Response:** When a health signal reports degradation, the CD pipeline stops.
  - It can *automatically roll back* the artifact.
  - Or, it can *trigger an alert* to engage the on-call engineer to decide if a rollback is warranted.
  - Based on engineer input, the pipeline might retry the failed stage or roll back entirely.
- **Rolling Forward vs. Rolling Back:** An operator can stop the pipeline and wait for a new artifact with a hotfix to be rolled forward. This is necessary if the release can't be rolled back due to a *backward-incompatible change.*
- **Rule of Thumb:** Since rolling forward is riskier than rolling back, *any change should always be backward compatible.*
- **Common Cause of Backward Incompatibility:** Changing the serialization format used for persistence or IPC.
- **Safely Introducing Backward-Incompatible Changes:** Break them down into multiple backward-compatible changes.
  - Example: Changing a messaging schema between a producer and consumer:
    1. **Prepare change:** Consumer is modified to support *both new and old* messaging formats.
    2. **Activate change:** Producer is modified to *write messages in the new format.*

    3. **Cleanup change:** Consumer *stops supporting the old format altogether*. This is released only when there's enough confidence the activated change won't need rollback.

- An automated *upgrade-downgrade test* as part of the CD pipeline in pre-production can validate if a change is truly safe to roll back.