Continuous Delivery (CD) and Deployment Strategies

What is Continuous Delivery (CD)?

Continuous Delivery (CD) is a software development approach that ensures software is always in a deployable state. The goal of CD is to make deployments predictable, frequent, and automated. CD focuses on minimizing risks associated with releases and maximizing software quality through automation and rigorous testing.

CD Ensures:

- Every successful build passes all automated tests.
- The build quality is not compromised.
- Code is deployed to test and QA environments before production.
- Every build that has passed all tests and quality gates is potentially deployable to production.

Understanding Continuous Delivery

Implementing Continuous Delivery requires:

- More than just automation—it requires collaboration between **development, operations, QA, and business teams**.
- A system that allows IT and business teams to deploy applications to a UAT (User Acceptance Testing) environment for testing.
- A traceable deployment pipeline that logs every deployment for audit purposes.
- Proper access control to ensure authorized deployments.
- Automated testing and risk management strategies to minimize deployment issues.

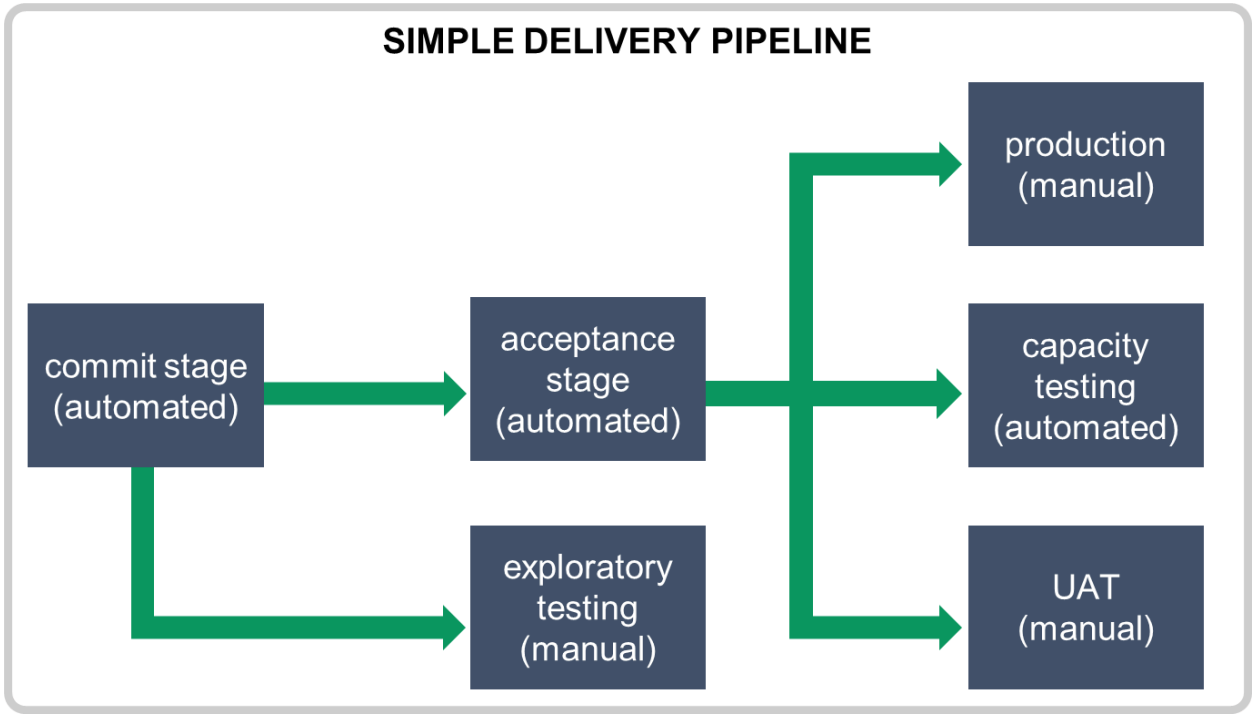
Definitions:

- **Development Team:** Responsible for writing and maintaining the application code, implementing new features, fixing bugs, and ensuring software stability.
- **Operations Team:** Manages infrastructure, deployments, and system reliability to ensure smooth application performance in production.
- **QA (Quality Assurance) Team:** Ensures software quality through rigorous testing, including functional, regression, performance, and security testing.
- **Business Team:** Focuses on aligning software development with business goals, ensuring that the product meets user needs and market requirements.

Types of Testing in Continuous Delivery:

- **Functional Testing:** Ensures that the application works as intended by verifying each function against the requirement.
    - *Example:* Checking if a login form correctly accepts valid credentials and rejects invalid ones.
- **Regression Testing:** Ensures that new changes do not break existing functionalities.
    - *Example:* After adding a new feature to a shopping cart, testing if previous functionalities like checkout still work correctly.
- **Performance Testing:** Evaluates the application's speed, responsiveness, and stability under load.
    - *Example:* Simulating 10,000 users accessing a website simultaneously to check if the response time remains optimal.
- **Security Testing:** Identifies vulnerabilities and ensures data protection within the application.
    - *Example:* Checking if an application prevents SQL injection attacks when entering special characters into a login form.

Simple Delivery Pipeline



A simple delivery pipeline consists of:

1. **Commit Stage:** Code is committed, unit tests are run, and build notifications are generated.
2. **Testing Stage:** Acceptance tests and integration tests are executed.
3. **Deployment Stage:** The application is deployed to staging or production environments.

The deployment pipeline plays a crucial role in ensuring reliable software releases.

What is staging in **Continuous Deployment (CD)?**

In **Continuous Deployment (CD)**, **staging** refers to an environment where the application is deployed before it is released to production. It serves as a final verification stage where new changes are tested to ensure everything works as expected before being deployed to live users.

Here's how **staging** fits into the **CD pipeline**:

1. **Continuous Integration (CI)**: Developers push code changes, which are automatically tested (unit tests, integration tests) to ensure the application works correctly.

2. **Build**: The application is built from the latest codebase.

3. **Staging Deployment**: After the application passes the CI tests, it is deployed to the **staging** environment. This environment mimics the **production environment** as closely as possible. It often uses the same hardware, software, and configurations as the production environment.

4. **Testing in Staging**:

    - **User Acceptance Testing (UAT)**: The staging environment can be used for UAT where stakeholders or end-users test the system to verify it meets business requirements.
    - **Pre-production Testing**: QA teams may run additional tests here, like load testing, regression testing, or smoke testing to ensure that the application will behave properly under production conditions.
    - **Final Approval**: Once everything is validated and approved in staging, the code is ready for production deployment.
5. **Production Deployment**: After successful verification in staging, the application is deployed to production.

Why Staging is Important in CD:

- **Realistic Testing**: Staging provides a near-identical replica of the production environment to catch issues that might not have been detected in earlier testing phases.

- **Safety**: It ensures that updates won't negatively impact the user experience in production since they've already been validated in staging.
- **Streamlined Releases**: With a robust staging environment, CD ensures that releases are stable and less prone to failures or disruptions in production.

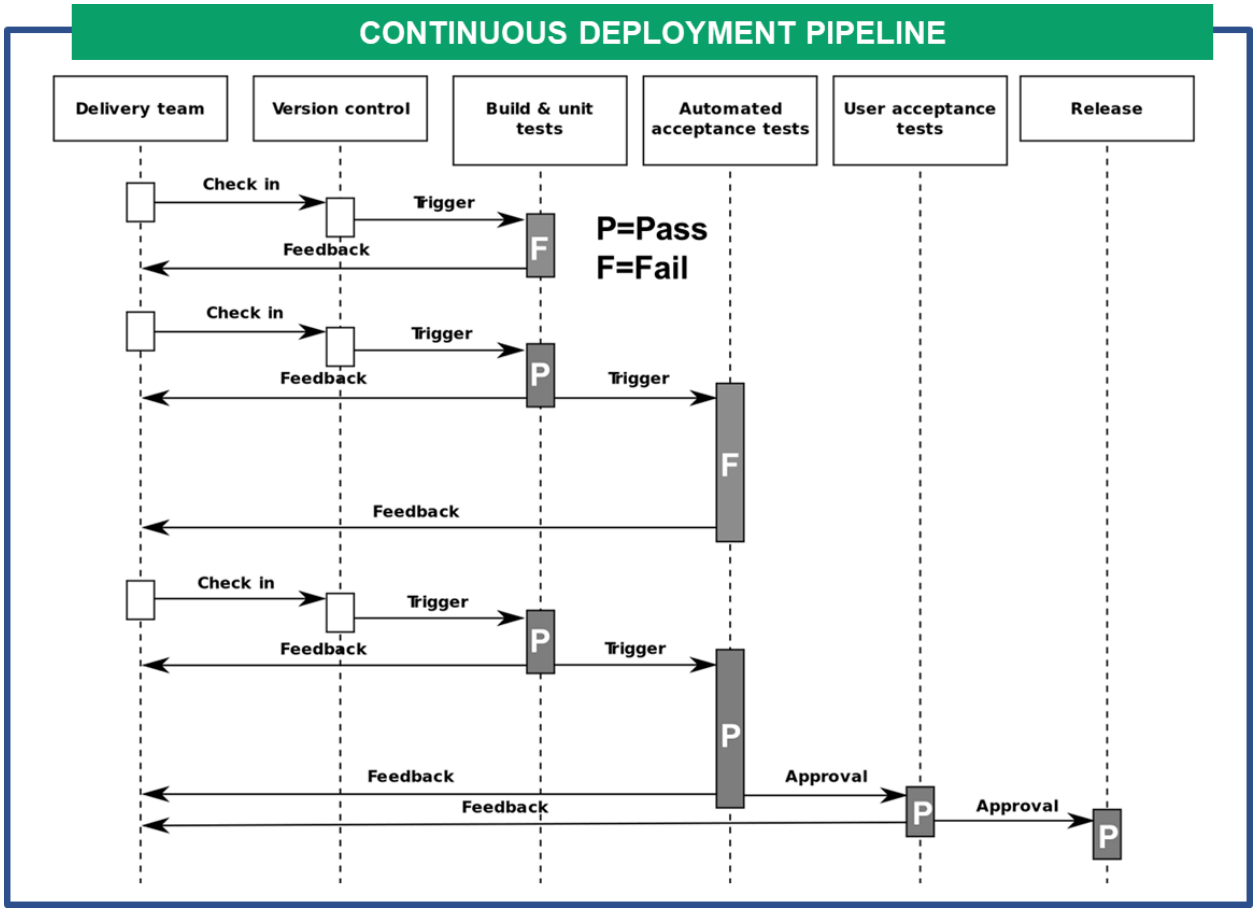
What is **User Acceptance Testing?**

A **UAT (User Acceptance Testing) environment** is a staging environment where software or applications are tested by the end users or clients to ensure that the system meets their business requirements. It is the final testing phase before the software goes live. During UAT, users test the application to verify if it works as expected in real-world scenarios and if it satisfies the agreed-upon requirements.

Key points about UAT environments:

1. **Real-world testing**: It mimics the actual user environment, so real users can test the functionality of the system.
2. **Business requirements**: It ensures that the application meets business needs, not just technical specifications.
3. **End-user involvement**: The actual users (or clients) are involved in testing to ensure the system is ready for production.
4. **Issue identification**: Bugs or issues found here are typically related to functionality, usability, or user interface, rather than the core technical aspects.
5. **Approval process**: The UAT environment is where the client signs off on the system before it moves to production.

Continuous Deployment Pipeline



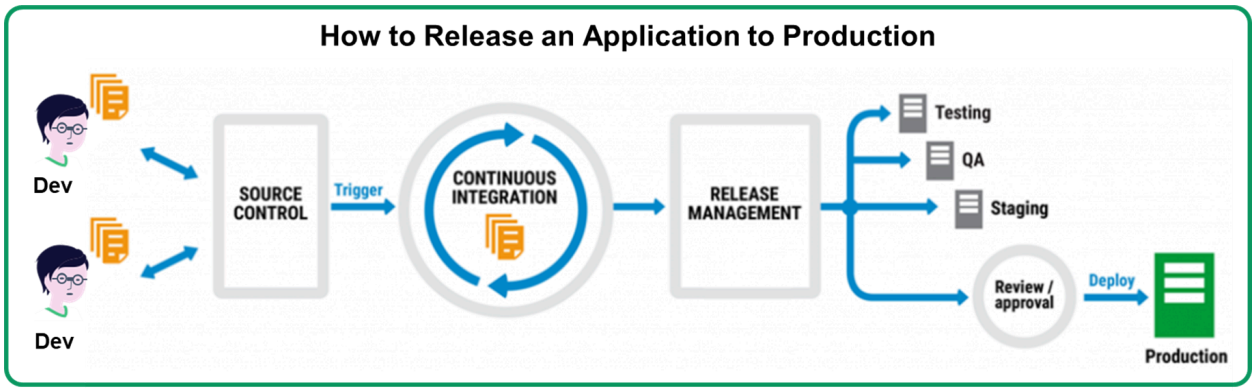
A Continuous Deployment Pipeline consists of:

- **Component Phase:** Code reviews, unit testing, and static code analysis.
- **Subsystem Phase:** Functional testing and negative test scenarios.
- **System Phase:** Integration, performance, and security testing.
- **Production Phase:** Pipelines for deployment with zero downtime.

Key Features:

- Provides visibility into production readiness.
- Enables self-service deployments.
- Detects bugs before the final release.
- Ensures a high level of software quality.

Releasing an Application to Production



Releasing an application into production requires proper planning and execution:

1. **Preparation and Planning:** Ensure the production setup is properly documented.
2. **Environment Consistency:** The production environment should mirror the staging/testing environment.
3. **Rollback Strategy:** Plan for contingencies in case of failure.

Key Steps in Releasing to Production

- Use the same deployment process for staging and production.
- Automate deployment without manual intervention.
- Maintain a rollback strategy to revert in case of failure.
- Utilize techniques such as **blue-green deployments** and **canary releases** for seamless transitions.

Releasing an Application to Production

- Final stage for any product.
- Two key pillars: **Preparation and Planning**.
- Production setup must be well-documented.
- Production environment should closely match the staging/testing environment.
- Always have a clear rollback strategy.

Deployment vs. Release

- **Deployment**: Installing software into an environment.
- **Release**: Making the software available to users.
- The key difference is the ease of rollback in case of issues.

Releasing Steps

- Use automated deployment pipelines, consistent across all environments.
- Minimal manual intervention (only small details like IP and port may differ).
- Deployments to all environments should be automated and uniform.

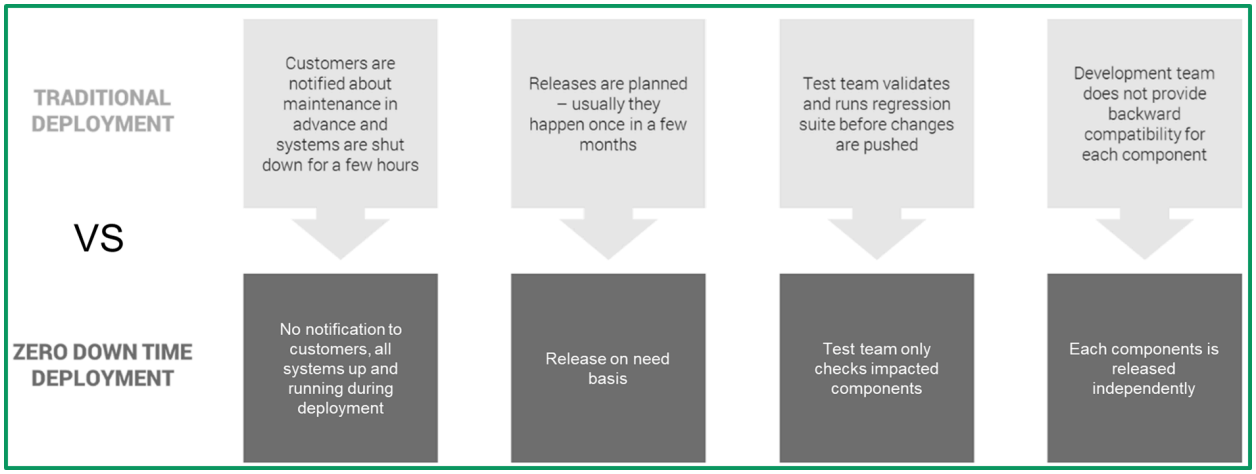
Essential Components of Production Release

1. **Creating a Release Strategy**
    - Clearly define how rollbacks are handled.
    - Integrate deployment, testing, and rollback processes into the pipeline.
2. **Having a Release Plan**
    - Define product vision.
    - Rank product backlog.
    - Conduct release planning meeting.
    - Finalize and communicate the release calendar.
3. **Releasing the Product**
    - Backup and restore application state.
    - Upgrade application without losing state.
    - Restart/redeploy if failures occur.
    - Monitor application performance and behavior.

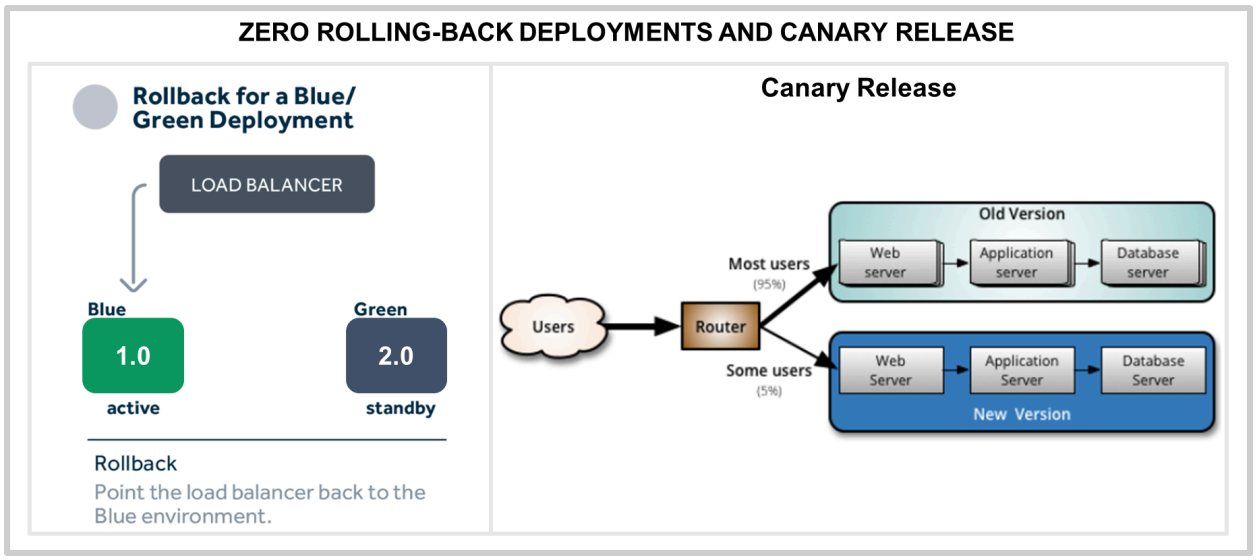
Zero-Downtime Releases

A **zero-downtime release** (or hot deployment) allows seamless transitions between software versions without disrupting users.

- Requires decoupling of services to minimize dependencies.
- Ensures instant rollbacks to a previous stable version in case of failure.
- Uses techniques such as **database migrations** and **feature toggles** to ensure smooth updates.

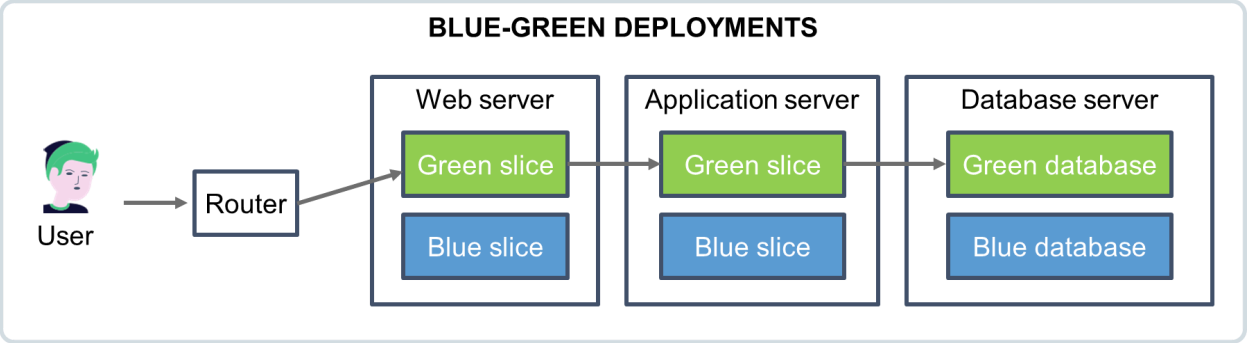


Rolling Back Deployments



- **Rollback Mechanisms:** If a deployment causes issues, rollback strategies allow reverting to a previous stable version.
- **Challenges:** Data changes and system integrations can complicate rollbacks.
- **Best Practices:** Maintain **versioned deployments** and **database versioning** for easy recovery.

Blue-Green Deployment Strategy



A blue-green deployment model maintains two identical production environments.

- **Blue Environment:** Runs the current live version.
- **Green Environment:** Deploys the new version for testing.
- **Switching Traffic:** Once verified, traffic is switched from blue to green, ensuring zero downtime.

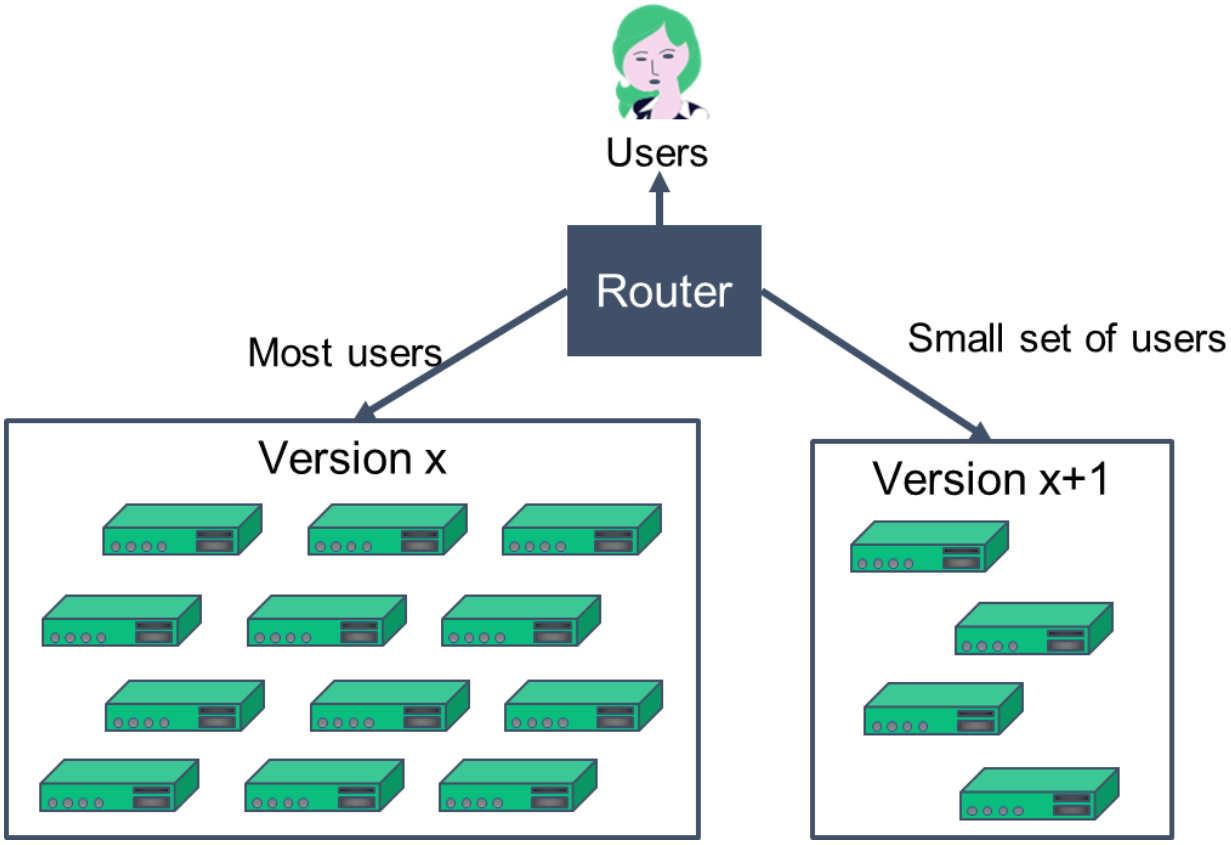
● Blue-Green is one of the most powerful release management techniques available. The concept is to keep the production environment in two identical versions, which is called as call blue and green deployment model.

● In the figure above, system users are routed to the green environment, which is the production that is currently being designated. We want a new version of the application to be released. It is deployed in the blue environment, allowing the application to start up. This in no way affects the green environment operation.

● To check that it works properly, we can run smoke tests against the blue environment. Moving to the newer version when everything is ready is as simple as changing the configuration of the router to point to the blue environment rather than the green environment. Therefore, the blue environment becomes production. It is typically possible to switch to a new version in much less than a second.

● We simply switch the router back to the green environment if something goes wrong. We can then debug on the blue environment what went wrong.

Canary Releasing Strategy



A canary release involves deploying a new version to a subset of users before a full rollout.

● Canary releasing, as shown in the figure above, involves rolling out a new version of an application for quick feedback to a subset of production servers. This quickly uncovers any problems with the new version without affecting the majority of users like a canary in a coal mine. Canary release is the best way to reduce the risk of a new version being released.

● Like blue-green deployments, you need to initially deploy the new version of the application to a set of servers where no users are routed to. You can then do smoke tests and, if desired, capacity tests, on the new version. Finally, you can start to route selected users to the new version of the application. Some companies select “power users” to hit the new version of the application first. You can even have multiple versions of your application in production at the same time, routing different groups of users to different versions as required.

● Canary releasing is not for everyone, though. It is harder to use it where the users have your software installed on their own computers. There is a solution to this problem (one used in grid computing)—enable your client software or desktop application to automatically update itself to a known-good version hosted by your servers.

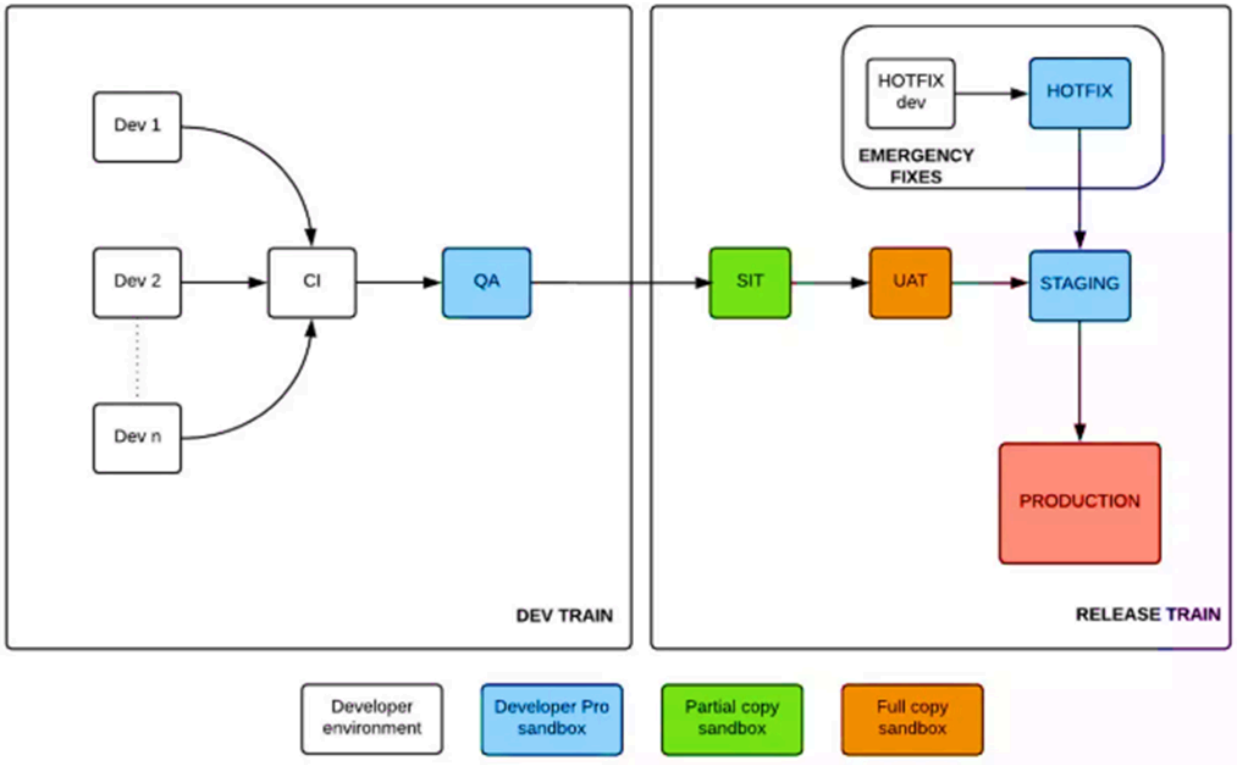
● Canary releasing imposes further constraints on database upgrades (which also apply to other shared resources, such as shared session caches or external services): Any shared resource needs to work with all versions of the application you want to have in production. The alternative

approach is to use a shared-nothing architecture where each node is truly independent of other nodes, with no shared database or services,3 or some hybrid of the two approaches.

- **Advantages:**
    - Reduces the impact of potential failures.
    - Allows early feedback before a full-scale release.
    - Can be combined with **A/B testing** to compare versions.

Emergency Fixes in Production

The following image displays Emergency fixes.

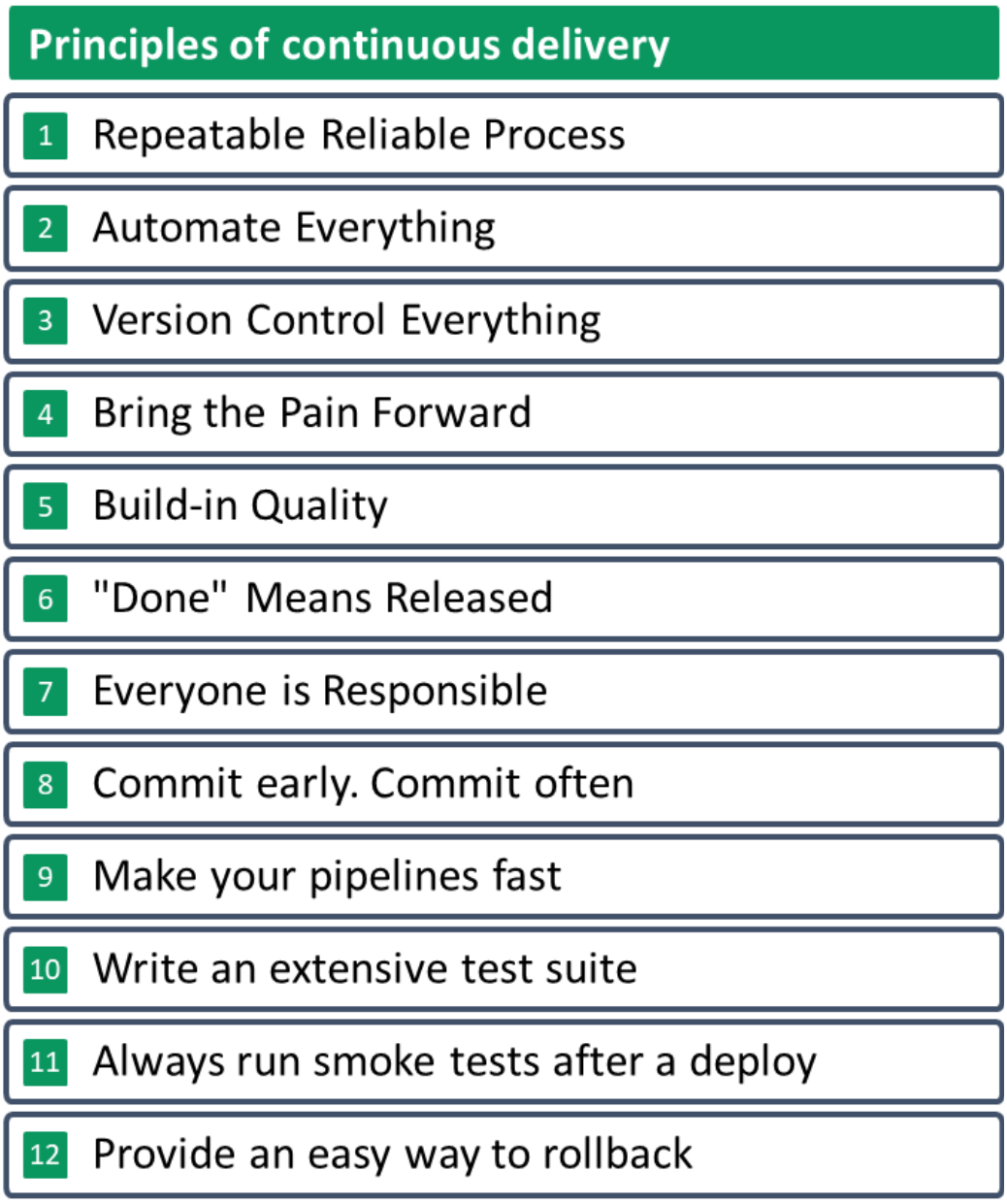


- Emergency fixes should follow the standard deployment process to avoid untested changes.
- Avoid directly modifying production environments.
- Ensure emergency fixes are tested in staging environments before deployment.

Best Practices for Managing Emergency Fixes:

- **Automate error detection** using monitoring tools.
- **Ensure every fix is documented** and version-controlled.
- **Rollback if necessary** instead of deploying an unstable fix.

1.3.2 Continuous Delivery Engineering Practices



•**Continuous Delivery (CD)** leverages an automated deployment pipeline to release software into production **quickly and reliably**.

The primary goals of CD include:

•Establishing an optimized end-to-end delivery process

•Reducing the risk of deployment issues
•Accelerating development-to-production cycles
•Ensuring faster time-to-market

Principles of Continuous Delivery

1.**Repeatable Reliable Process**

A consistent and standardized approach where the **same release process** is used across **all environments** (development, testing, staging, production).

Ensures predictable and stable deployments, reduces errors, and facilitates troubleshooting.

2. Automate Everything

Automate your builds, your testing, your releases, your configuration changes and everything else

3. Version Control Everything

Code, configuration, scripts, databases, documentation. Maintaining everything in one source that is a reliable one

4. Bring the Pain Forward

Deal with the hard stuff first. The tasks that are time-consuming or error prone should be dealt with as soon as possible.

5. Build-in Quality

Create feedback methods to deal with the bugs as soon as they are created. By routing the issues back to developers as soon as they fail post-build test, it will enable them to produce higher quality code quicker.

6. “Done” Means Released

A feature is marked as completely done only when it is in production. Setting a clear definition of “done” criteria right from the beginning will help everyone communicate better, and realize the value in each feature.

7. Everyone is Responsible

It works on development machine, is never a acceptable excuse. Responsibility should extend all the way to production. Cultural change can be the hardest to implement.

8. Commit early. Commit frequently.

This is a fundamental principle to be able to implement CI/CD in your organization. We cannot ignore this.

9. Make your pipelines fast.

This is a very important requirement for the team to be more productive and to enable fast turn around if incase of any issues. If a team has to wait an hour long for the pipeline to finish just to be able to deploy a one line change they will get frustrated very quickly.

10. Write an extensive test suite (unit and integration tests)

This is one of the hardest parts to get right apart from organizational changes. Your team needs to develop a culture of writing tests for each code change that they integrate into the master branch in order to be

successful with CI / CD. This includes creating the unit tests, regression, integration and smoke.

11. Always run smoke tests after a deploy.

A rapid, preliminary check performed after deployment to verify the critical functionalities of the software.

It is very useful to be able to verify that the successful deployment actually works as expected and has not broken any common user flows.

12. Provide an easy way to rollback.

This usually involves idempotent deployments where doing a rollback simply means redeploying a previous release.