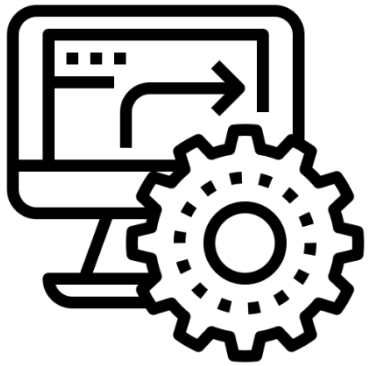
School of Computing and Information Systems

SMU SINGAPORE MANAGEMENT UNIVERSITY

Agile and DevSecOps

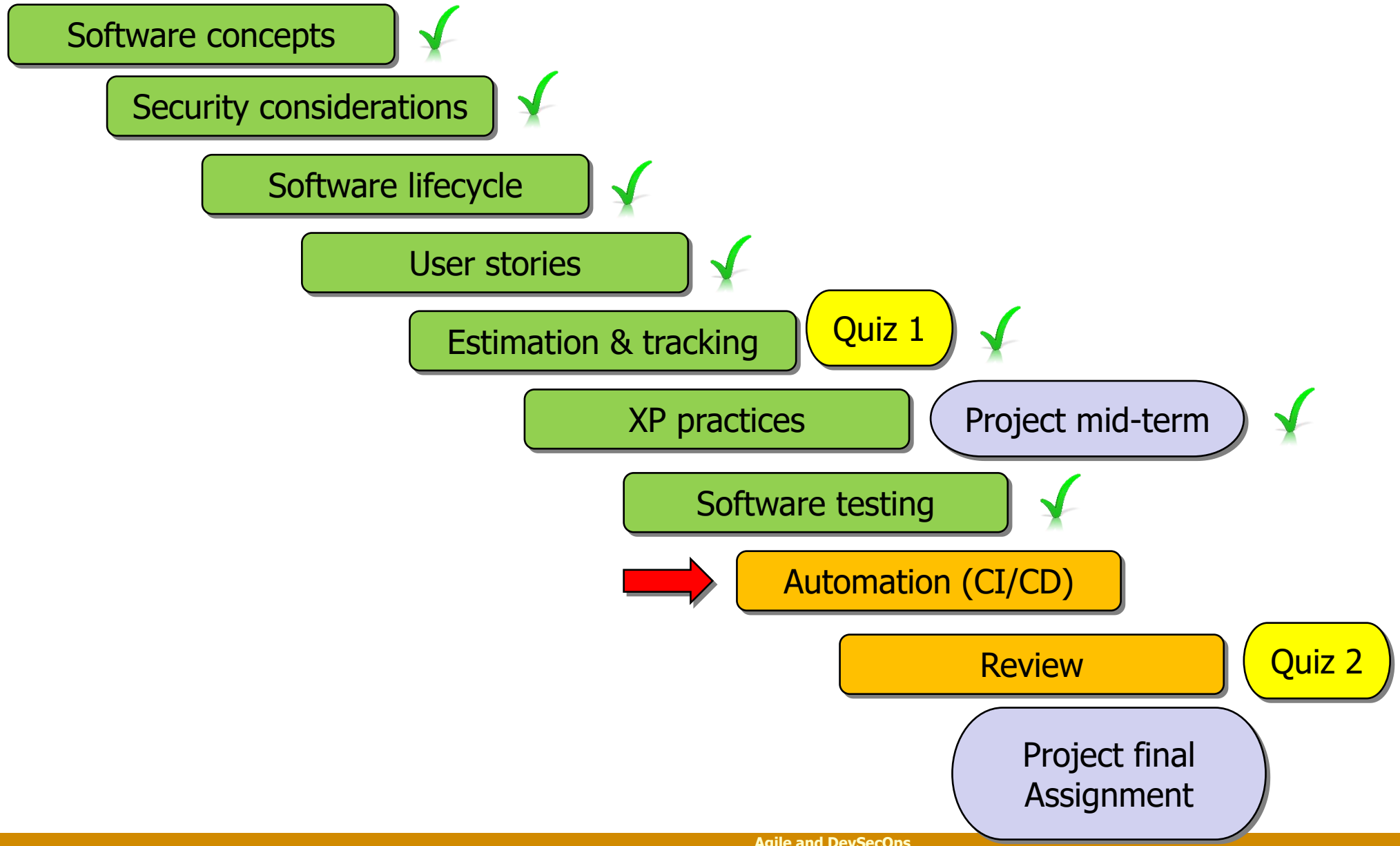
Automation (CI/CD)



"In software, when something is painful, the way to reduce the pain is to do it more frequently, not less."

David Farley

Learning Journey



Objectives

- To understand different software delivery models
- To understand the need for source code version control
- To learn about different software updating strategies
- To understand CI/CD pipeline
- To design a CI/CD pipeline for your project

- Topics covered:
  - Software delivery models (CI/CD)
  - Version control
  - Deployment pipeline
  - CI/CD pipeline demonstration

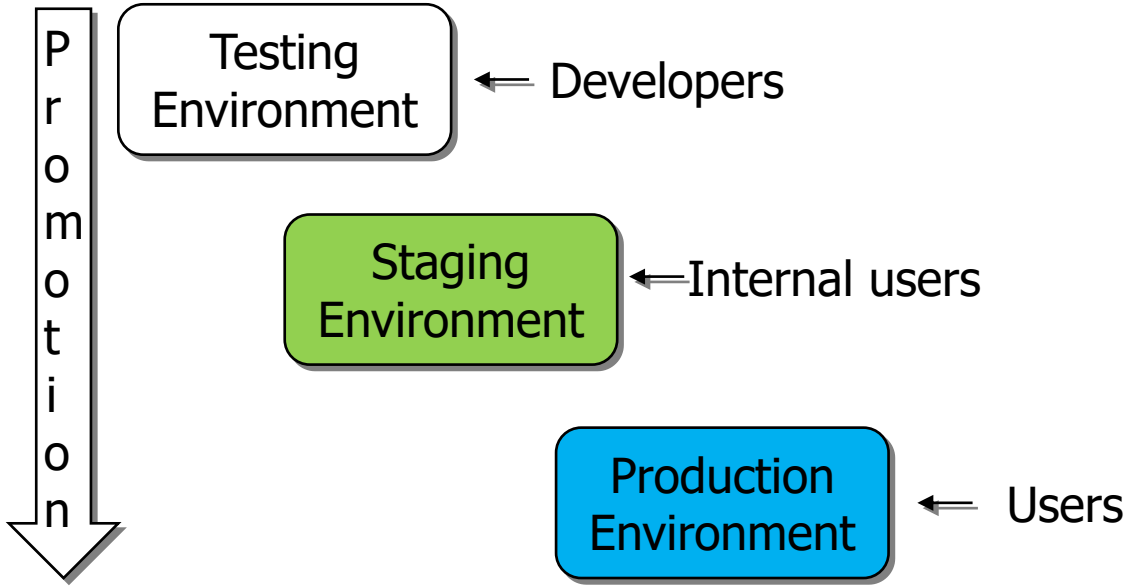
Software Delivery

- Developers finish work on a feature, then it's used in production
  - Production: feature is live with actual users
    - Running apps from your laptop is not production-ready!
  - Traditionally, production deployment takes months

- These days it can be minutes
  - New software features are used to generate value more quickly
    - Increasing the return on investment to build that feature
  - Provide short feedback loop for future improvements

Testing - Staging - Production Environments

- Not possible to simulate real environment when testing
  - Dev testing normally done on local machines
  - Staging: deploy to a complete production-like environment, but don't have everyone use it
    - Lower risk if a problem occurs in staging than in production
    - Additional cost for another environment
- Test-Stage-Production



Continuous Integration

- Automatically build and test a software in response to every change committed to the source code repository
  - "Commits" can come from different developers
  - Create deployable software artifacts frequently
  - Avoid integration hells!
- Need a version control system for code
  - Git is commonly used
  - GitHub is a managed service for Git
- Deployment pipeline:

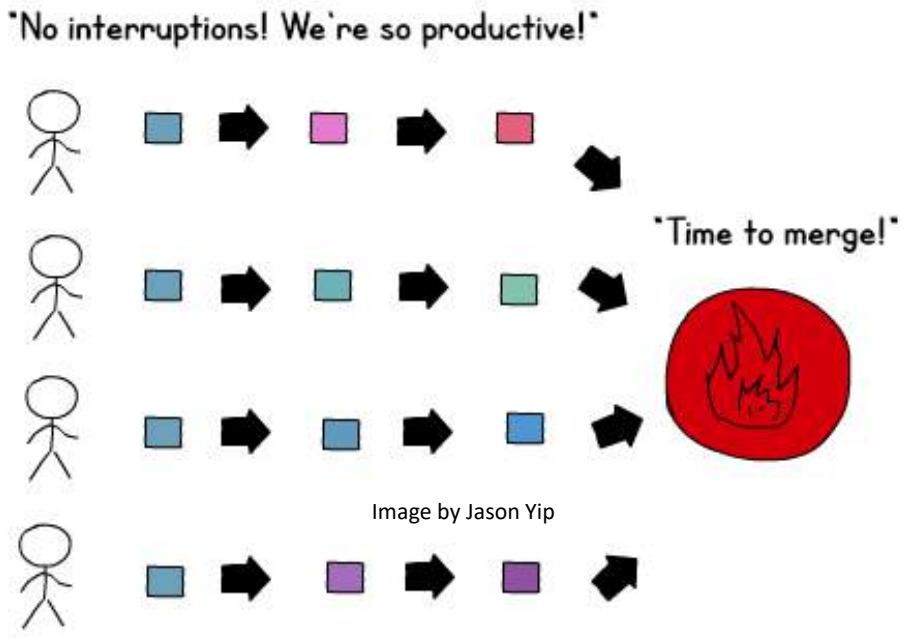
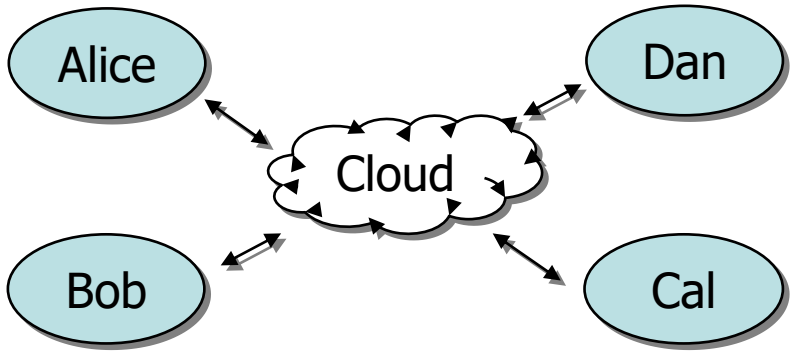


Image by Jason Yip

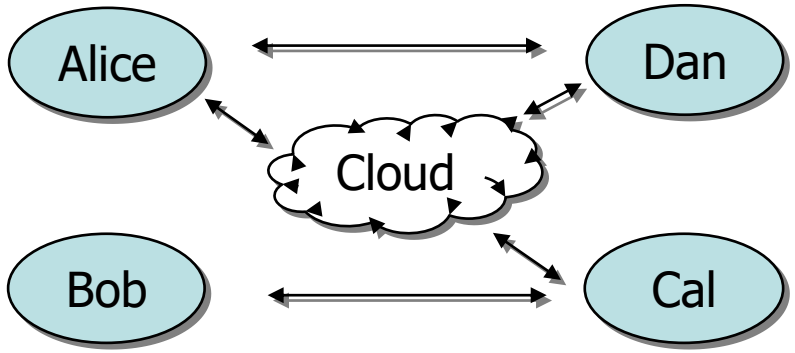
Version Control System

- Centralized versioning system
  - E.g., Subversion
  - A master server storing the only repository: the master
  - Each copy needs to talk to the master



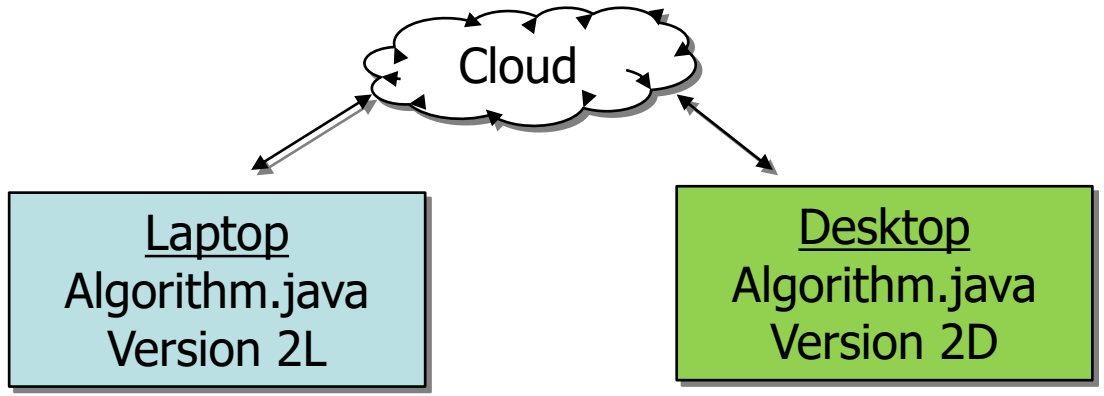
Collaboration graph: who is sharing what with whom

- Distributed versioning:
  - Git and Mercurial support diverse collaboration graphs
  - All repos have the same role
    - Team can designate one Git repo as a "central" one



Version Control Example: A Single Developer

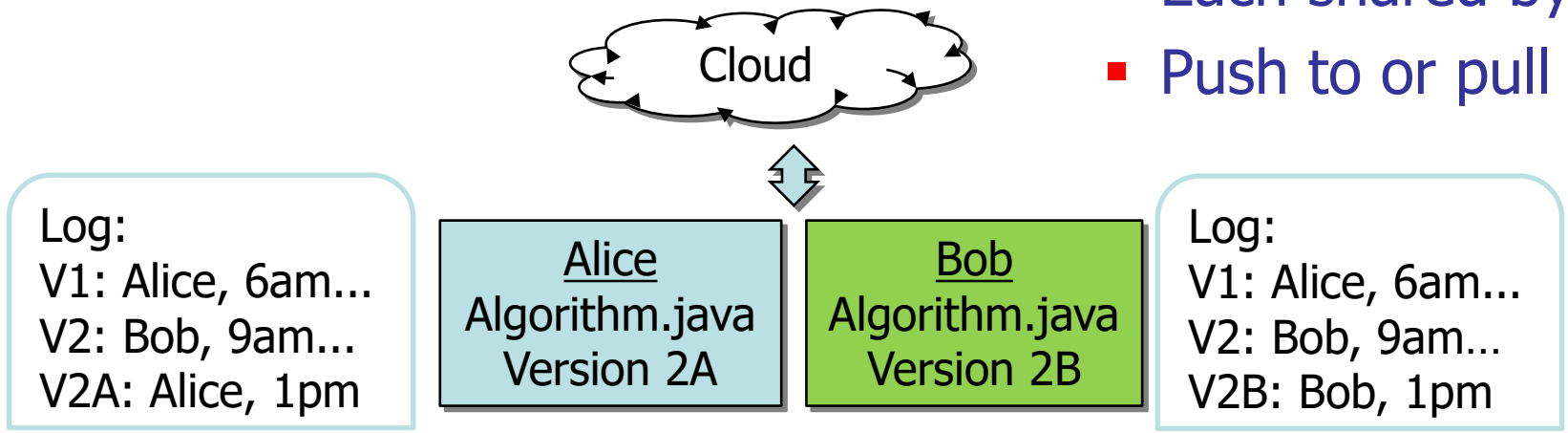
- Keep multiple versions of a file
  - Need to revert to a past version
  - How to compare different versions
    - Using tools like diff
    - To retain good changes
- Back up files to the cloud
  - Push files from the local working directory to the remote backup



- Example: different machines
  - Laptop: version 2L of a file
    - Not synced to the cloud yet (e.g., due to no network connection)
  - Desktop: version 2D of the same file, different changes
    - Push 2D to the cloud
    - Later, pull 2D to the laptop: changes in 2L might be overwritten
  - The need to merge changes:
    - To create a new version based on 2L and 2D

Version Control Example: Add More Collaborators

- Assume two developers
  - They must coordinate in terms of versioning
  - Same problems as before
- Need log files for changes
  - Who, when, what – to blame!
- Merging log files: manually?
- Solution: multiple branches
  - A dev works on his own branch:
    - Parallel code universe: to experiment with new features
    - Others do not want to pull in the changes if not done yet
- Flexibility in collaboration
  - Many branch locations at once
  - Each shared by several persons
  - Push to or pull from any location



Git Workflow

First, clone code from the project's shared repository hosted on a server, e.g., GitHub

Local development and test
1) Dev writes code and test on local setup
2) Run unit testing
3) Commit to git locally
4) Get code from others and merge
5) Run tests again

Push to shared repository

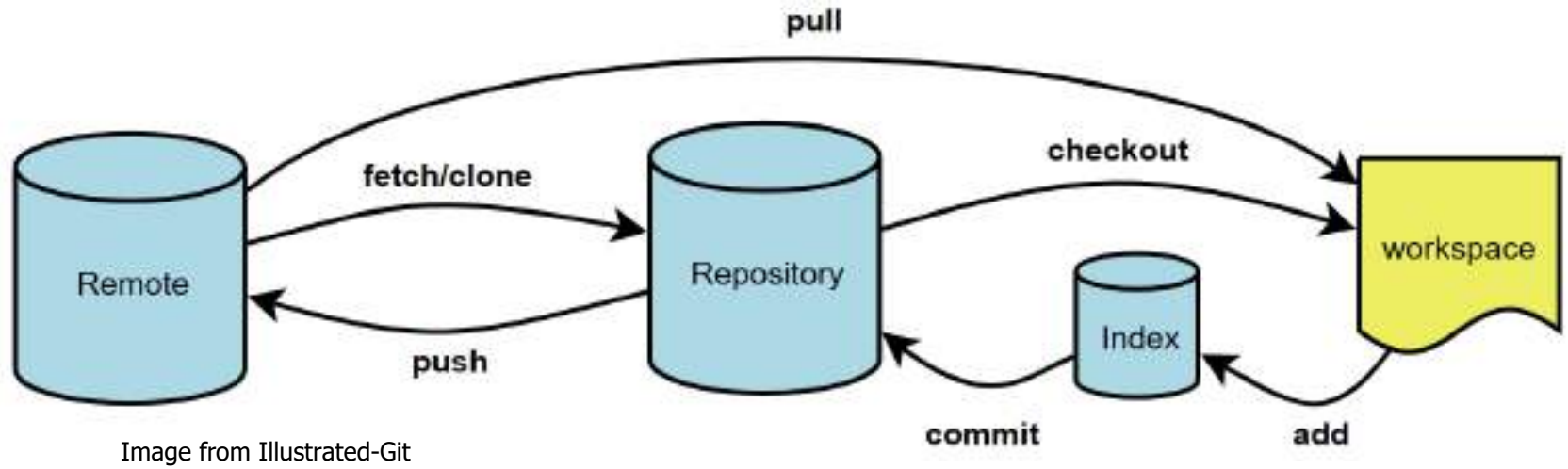


Image from Illustrated-Git

Git: Branching and Merging

- Example
  - Do some work on the project
  - Create a branch for a new user story
  - Do some work in that branch

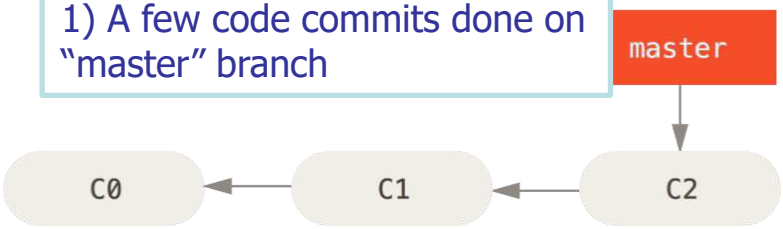
Origin Master

- A critical issue happens
  - Switch to production branch
  - Create a new branch to add the fix
  - After it's tested, merge the fix branch, and push to production
  - Switch back to the original user story and continue working

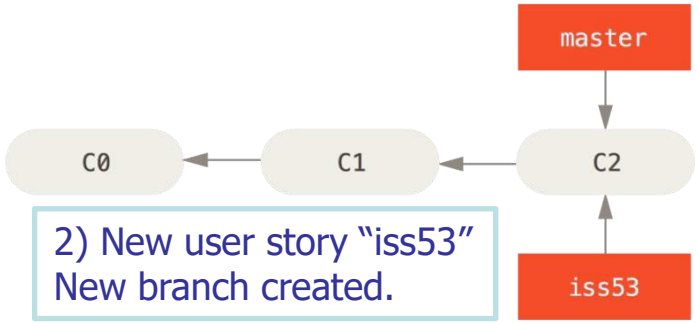
```
>git merge
```

Git: Branching and Merging

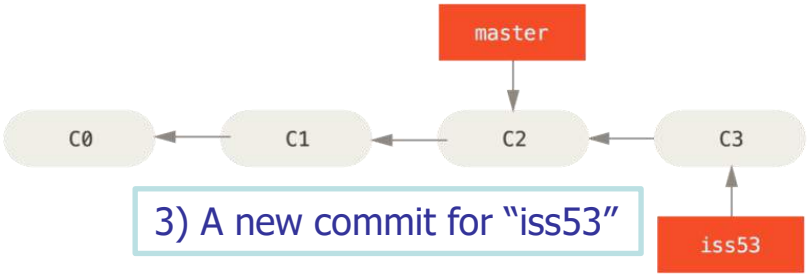
1) A few code commits done on "master" branch



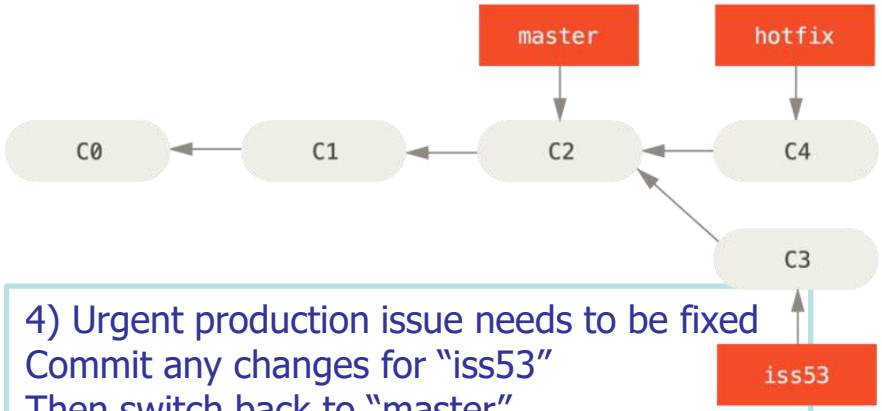
2) New user story "iss53"
New branch created.



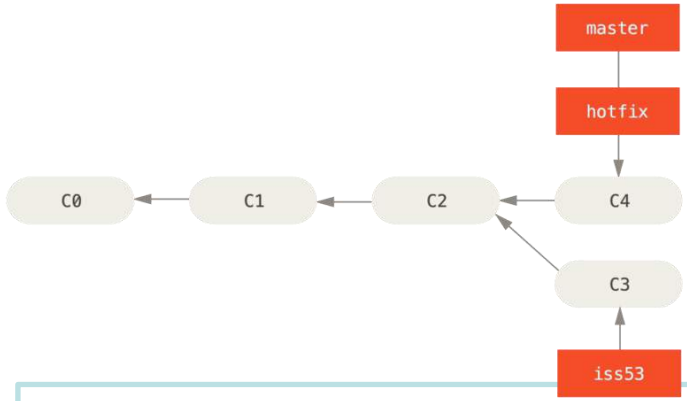
3) A new commit for "iss53"



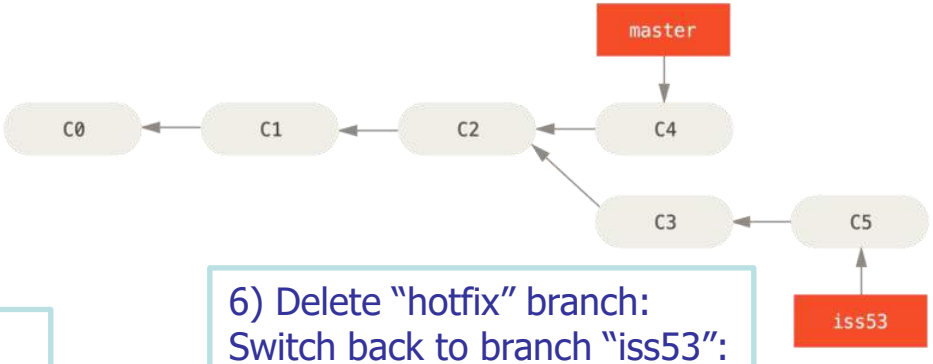
4) Urgent production issue needs to be fixed
Commit any changes for "iss53"
Then switch back to "master"
Create new branch "hotfix" based on "master"
Commit new changes to "hotfix" (C4)



5) Completed change with "hotfix"
Fast-forward merge hotfix into "master"
Ready to deploy "master" into production

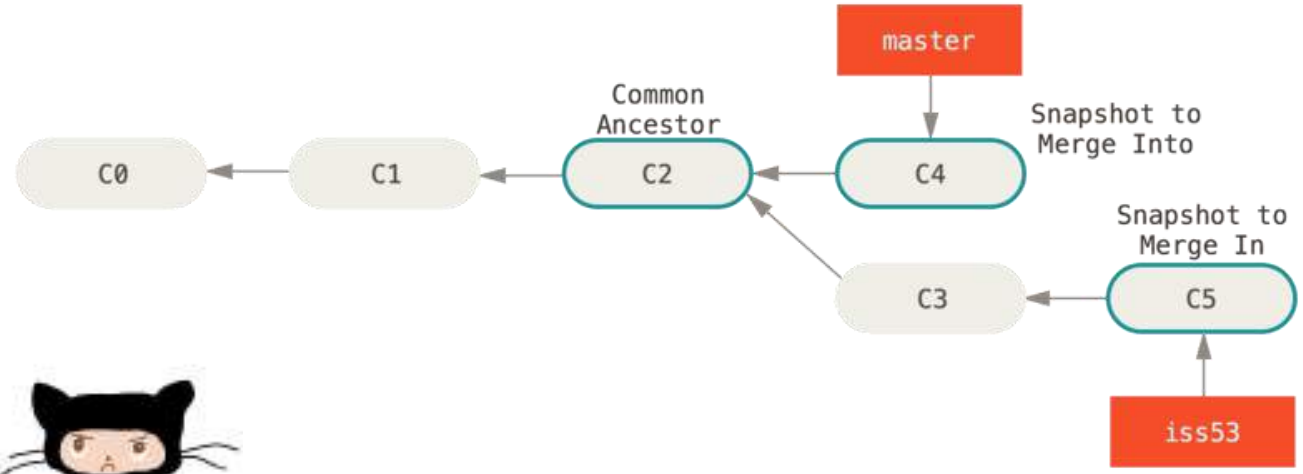


6) Delete "hotfix" branch:
Switch back to branch "iss53":
Do a new code commit (C5)



Note that each Git commit, e.g., C0, C1, etc., is a complete version of the project.
A branch is just a pointer to a commit.

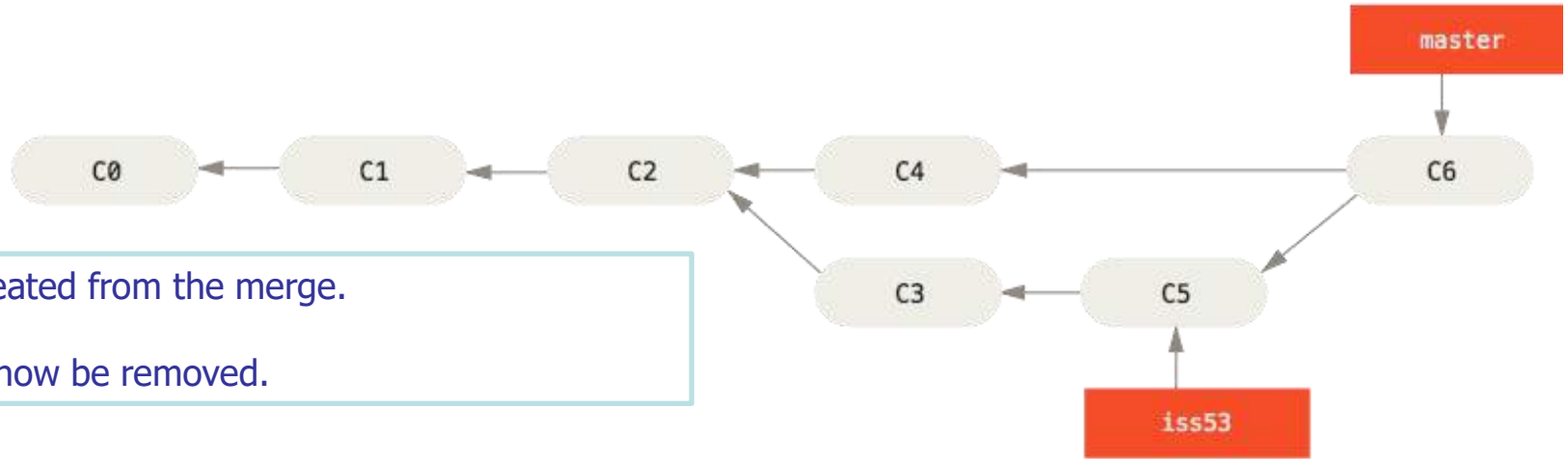
Git: Three-Way Merge



1) Merge "iss53" into "master" branch, note that "iss53" is on a different branch.

Basic three-way merge:

- Use a common ancestor of the two branches: C2
- The other twos are the branches to be merged
- Different from the previous fast-forward merge



2) New commit C6 is created from the merge.

The "iss53" branch can now be removed.

CI Pipeline Stages

- Compile code and build software artifacts, e.g., exe or libs
- Run tests:
  - Should run on every change that is made to the main repository
  - What kind of tests to run?
    - Unit test suites: should be fast to complete
    - Smoke tests: sanity checks if the application can start
    - Security tests: scanning container images, known vulnerabilities, etc.
    - Integration/acceptance tests: can take hours to run
- Usually, VM/container images are then created and stored for further testing and deployment

CI Implementation

- Use CI server solutions: monitor and react to new commits
  - Trigger the build/test process
  - E.g., Jenkins can be run on-premises
- Cloud based solutions are easier to setup with minimal management
  - Bitbucket Pipelines
  - GitHub Actions
  - AWS CodePipeline
  - GitLab

1.Write tests for critical parts of code.

2.Setup CI server to run tests automatically on every push to the main repository.

3.Make sure that your team integrates their changes everyday.

4.Fix the build as soon as it's broken.

5.Write tests for every new story that you implement.

Continuous integration in 5 steps, by Atlassian

Build status for a code repository

Continuous Delivery

- Extend continuous integration
- Auto-deploy new changes that pass CI to production-like environments
- Manual deployment to production
  - Can be automated as well
- The goal is that code should always be *ready* to deploy into production.
- Setup staging environments
  - Consistency is key
  - Automatic setup via Infrastructure as Code

Infrastructure-as-Code

- Define computing and network infrastructure via code
- On-demand cloud servers
  - Setup and tear down with code
  - Avoid snowflake servers

"It can be finicky business to keep a production server running. You have to ensure the operating system and any other dependent software is properly patched to keep it up to date. Hosted applications need to be upgraded regularly. Configuration changes are regularly needed to tweak the environment so that it runs efficiently and communicates properly with other systems. This requires some mix of command-line invocations, jumping between GUI screens, and editing text files.
The result is a unique snowflake - good for a ski resort, bad for a data center." – Martin Fowler.

- Recall: immutable servers
  - Once deployed: are never modified, only replaced with newly updated servers
  - Any updates are applied to the base image, not to running servers

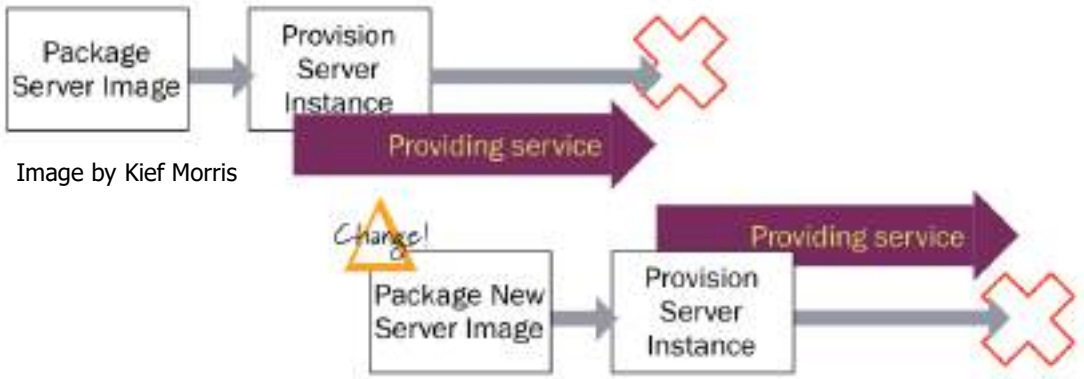


Image by Kief Morris

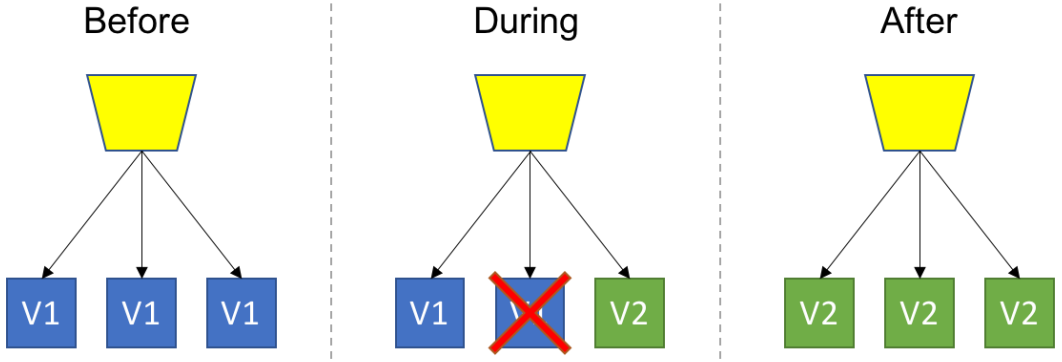
Continuous Deployment

- Extend continuous delivery
  - Incremental software changes are automatically tested, vetted, and deployed to production environments
- No human-intervention
  - Only failed tests preventing deployment
- Accelerate the feedback loop with actual customers

- Reduced risk
  - Deploying smaller changes, less to go wrong, easier to fix
- Real progress
  - Features deployed to real users
- User feedback
  - Ensure that we're building something useful

Update Running Services

- Updating a service that's already in production
  - Need to consider downtime and other risks
  - Rolling update, blue-green deployment, and canary release strategies.
- Rolling update:
  - Slowly replace previous versions of a service with the new version by completely replacing the infrastructure on which the service is running
  - Deploy new version which starts serving users right away
  - The previous version is then removed
  - Problem: both old and new version might run concurrently during the update.



Rolling update illustration by I. Shakury

Blue-Green Deployment

- To reduce downtime and risk
  - Use two identical production environments
  - One of them, e.g., Blue, is live
  - New software released for testing on Green environment
  - Switch the load balancer: Green is live now to handle actual traffic
  - Blue now becomes idle
  - Roll-back if anything goes wrong: routing traffic to the previously live server

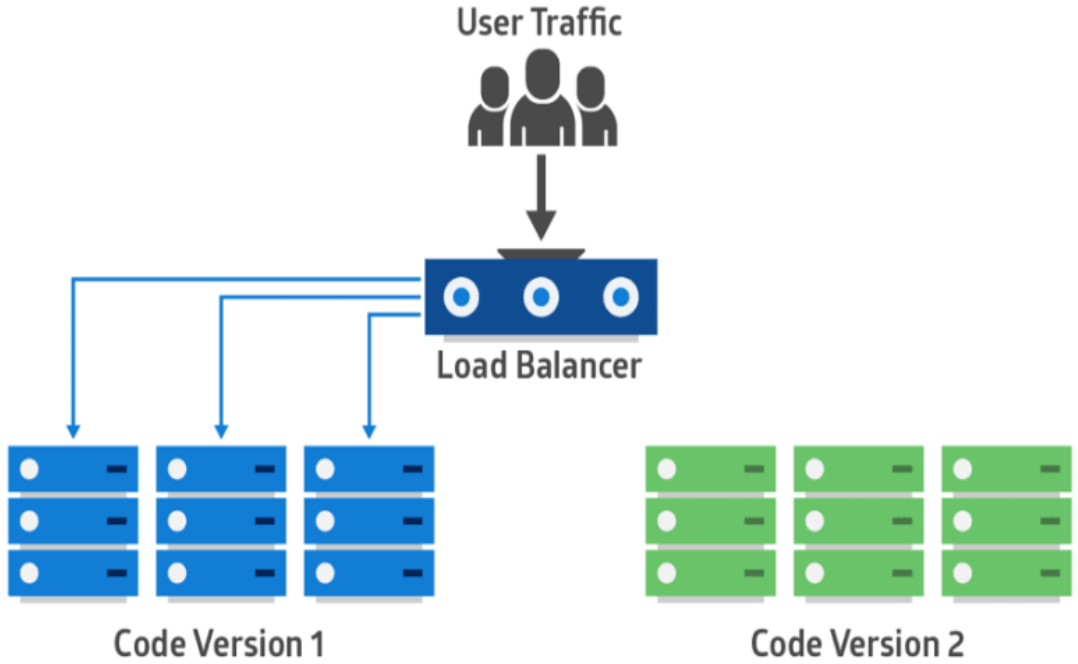


Image by dev.to

Canary Deployment

- Similar to Blue-Green, but more risk-averse
  - Deploy a new app in a small part of the production infrastructure.
  - A subset of users are routed to the new app
    - E.g.: expose internal users first, release app in certain geographic locations, unlock new features based on user groups, etc.
    - Minimize impacts of errors if any
  - New app can then be gradually rolled out to the rest of users

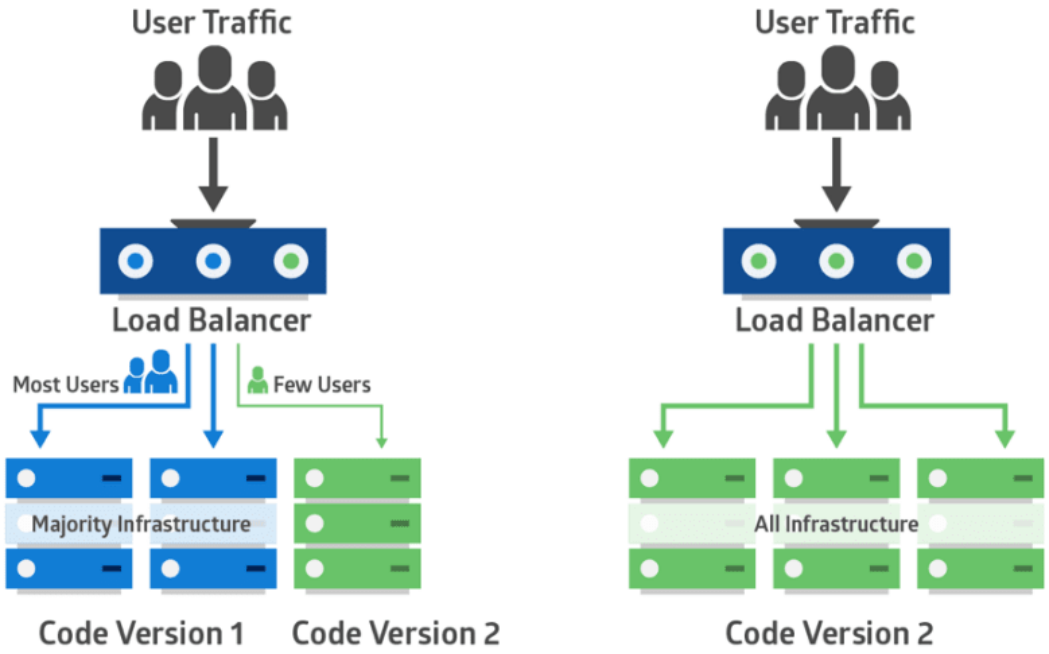


Image by dev.to

Deployment with Feature Flags

- Modify system behavior without code change
- Used in canary release:
  - Turn new feature on for a small percentage of users - "canary" cohort.
- A/B testing
  - Run production experiments
    - Turn the feature on for a cohort of users, then study how those behave compared to a "control" set of users
    - To see which are the better features
- Kill switches
  - Instantly hide features

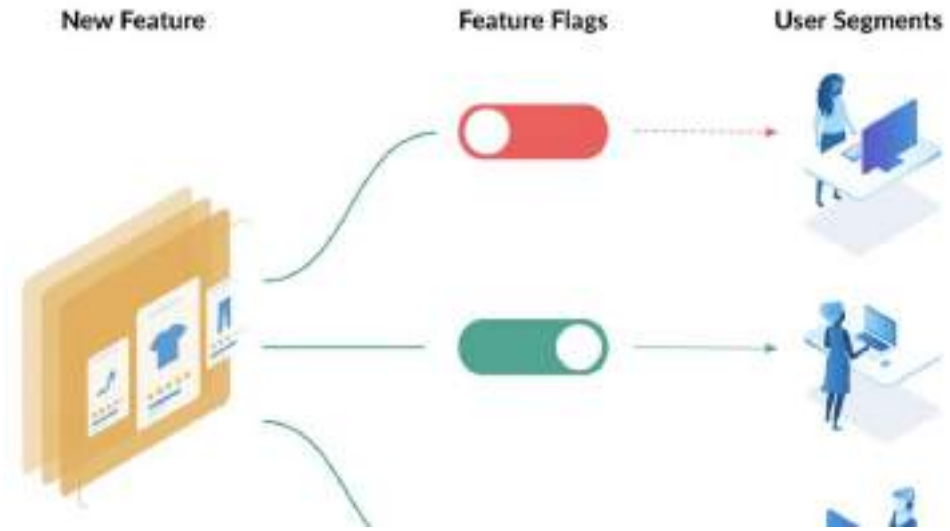
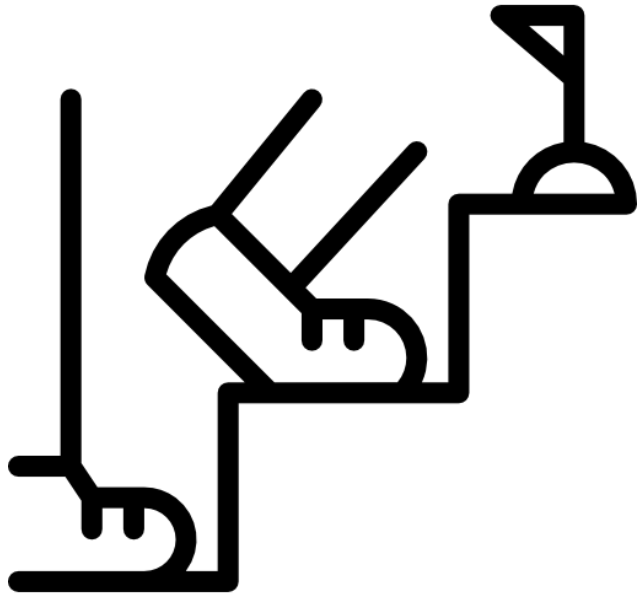


Image by dev.to

Approach to CI/CD Realization

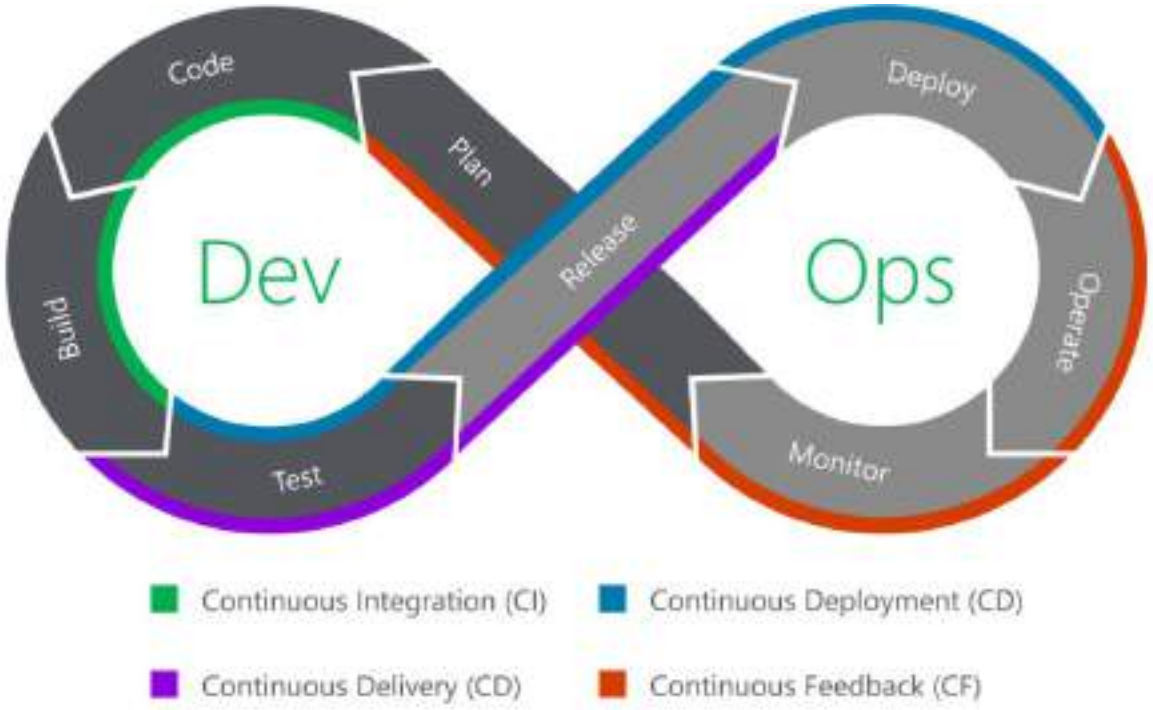
- A new project with no users
  - Can be easy to deploy every commit to production

- Steps for existing applications with customers:
  - Should start with CI first
    - Implement basic, automated unit testing
    - Complex end-to-end tests can be considered later
  - Try auto deployments to staging environments
  - Start releasing software on a daily basis:
    - Then look into implementing CD
    - Ensure the rest of your organization is ready too:
      - Documentation, support, marketing, etc.



Summary

- Continuous integration, delivery and deployment
- Distributed version control
- CI/CD pipeline
- Software updating strategies
- Approach to CI/CD realization



References

- Pro Git, available at https://git-scm.com/book/en/v2

- Continuous Delivery Pipelines: How To Build Better Software Faster, by D. Farley