Microsoft

WorkshopPLUS: DevOps Fundamentals

Module 2: Development & Test

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet website references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

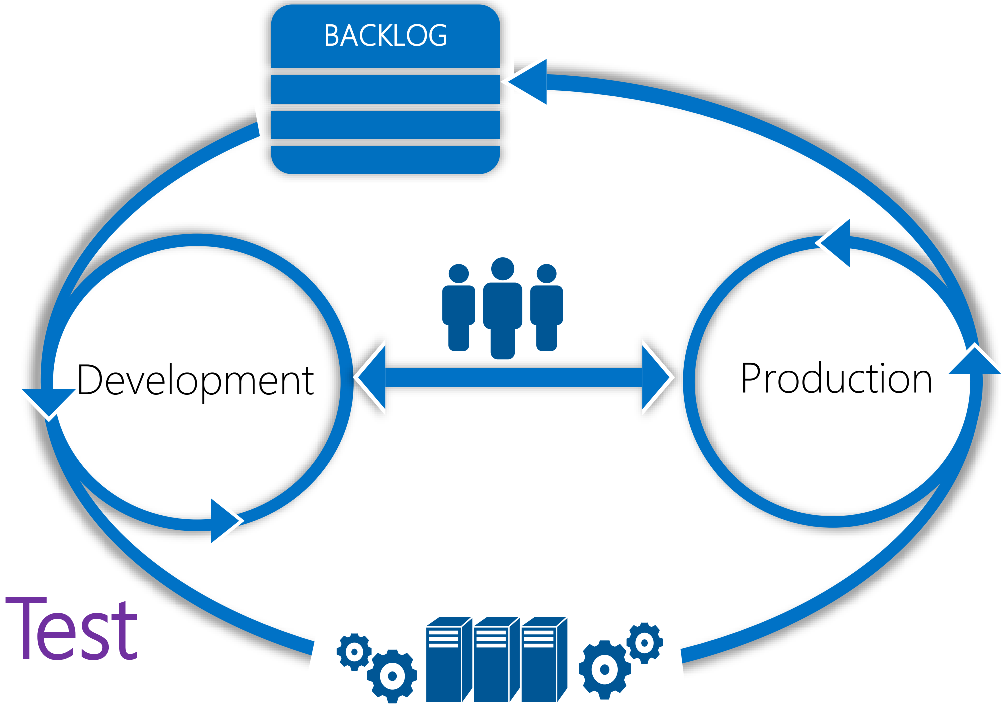
Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see **Use of Microsoft Copyrighted Content** at
https://www.microsoft.com/en-us/legal/intellectualproperty/permissions/default.aspx

Microsoft®, Internet Explorer®, Outlook®, SkyDrive®, Windows Vista®, Zune®, Xbox 360®, DirectX®, Windows Server® and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Module Objectives

- Understand the importance of a simple source control management strategy
- Learn how to add quality feedback loops early in your lifecycle
- Set the basis for continuous integration and continuous delivery



Dev & Test

Source Control Management using Git

Source Control Management Goals

- Harness collaboration
- Enable parallel development
- Minimize integration debt and merge conflicts
- Act as a quality gate

Git

- Distributed (DVCS)

- Repositories (including history) cloned locally

- Lightweight branches

- The most granular permissions you can apply is to a repository or a branch

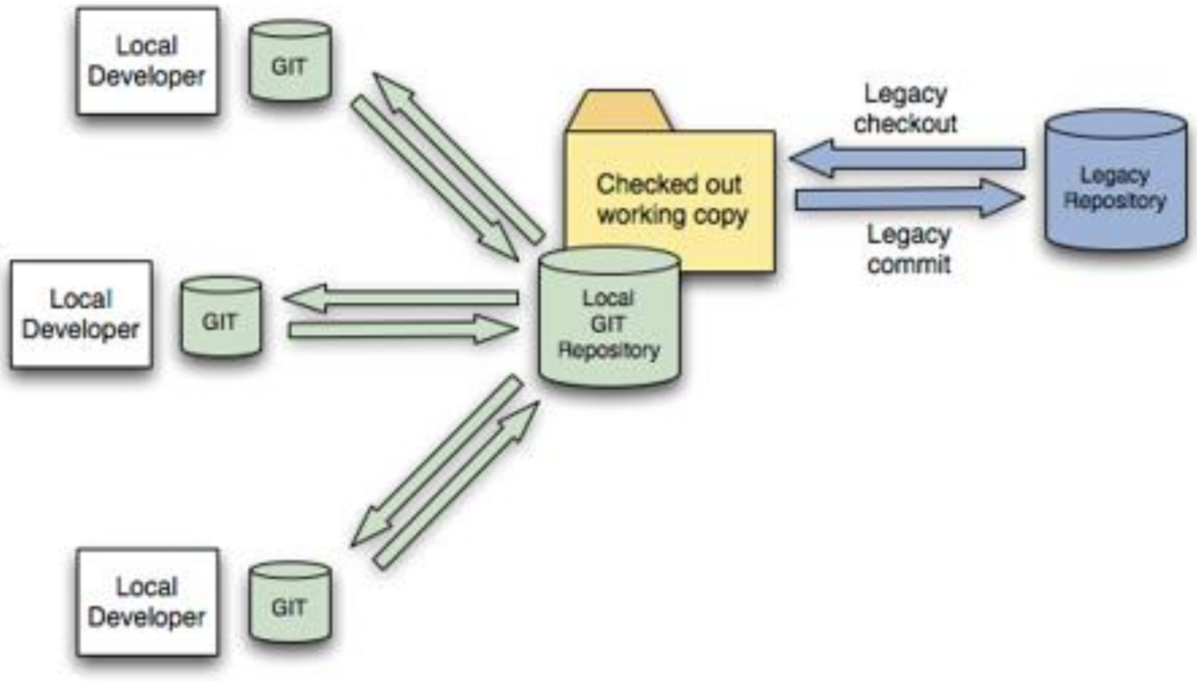
- You can quickly begin small projects. You can scale up to very large projects, but you must plan to modularize your codebase. You can create multiple repositories in a project

Git Source Control

- Git is the most used version control system today and is quickly becoming *the* standard for version control
- Git is a distributed version control system, meaning your local copy of code is a complete version control repository. These fully-functional local repositories make it is easy to work offline or remotely
- You commit your work locally, and then sync your copy of the repository with the copy on the server
- This paradigm differs from centralized version control where clients must synchronize code with a server before creating new versions of code
- Nearly every development environment has Git support and Git command line tools run on every major operating system

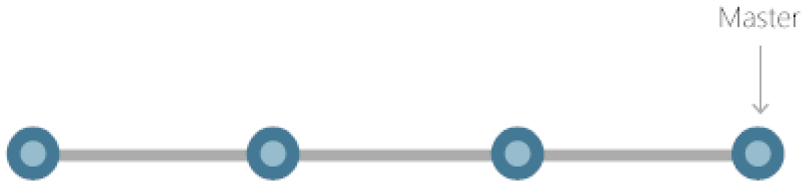
Git Benefits

- Simultaneous development
- Faster releases
- Built-In integration
- Strong community support
- Pull Requests
- Branch Policies



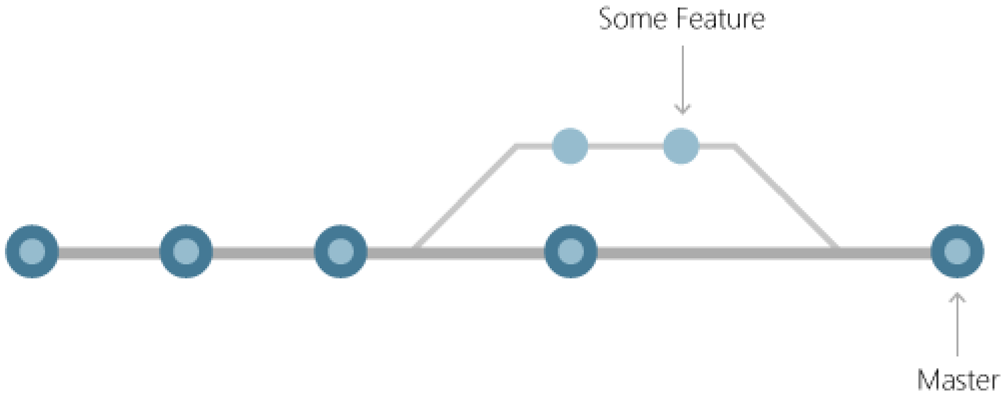
Git Basics - Commit

- A commit is a snapshot of all your files at a point in time, occurs every time you save your work
- If a file has not changed from one commit to the next, Git uses the previously stored file
- Commits are identified by a unique cryptographic hash of the contents
- Using these hashes Git can detect changes easily



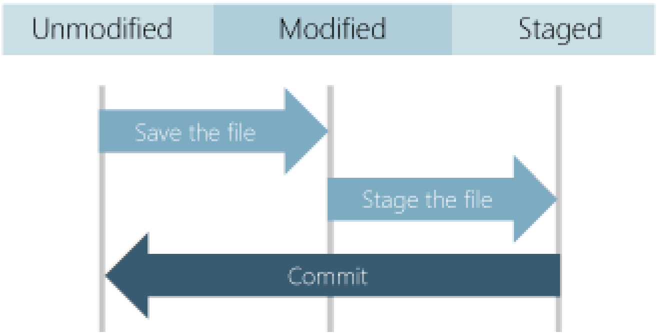
Git Basics - Branches

- Git provides tools for isolating changes and later merging them back together
- Branches, which are lightweight pointers to work in progress, manage this separation
- Once your work created in a branch is finished, merge it back into your team’s main branch



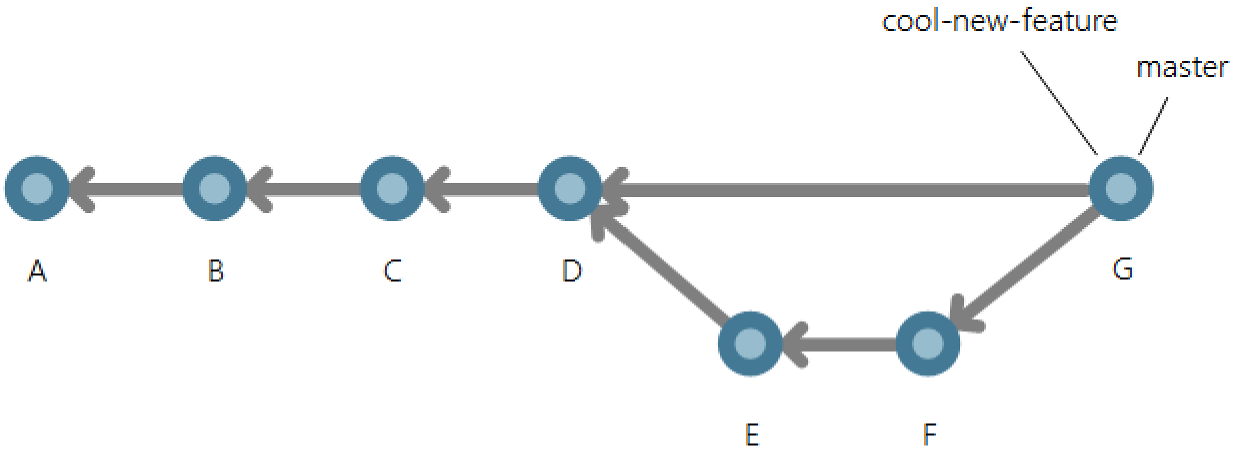
Git Basics – Files and Commits

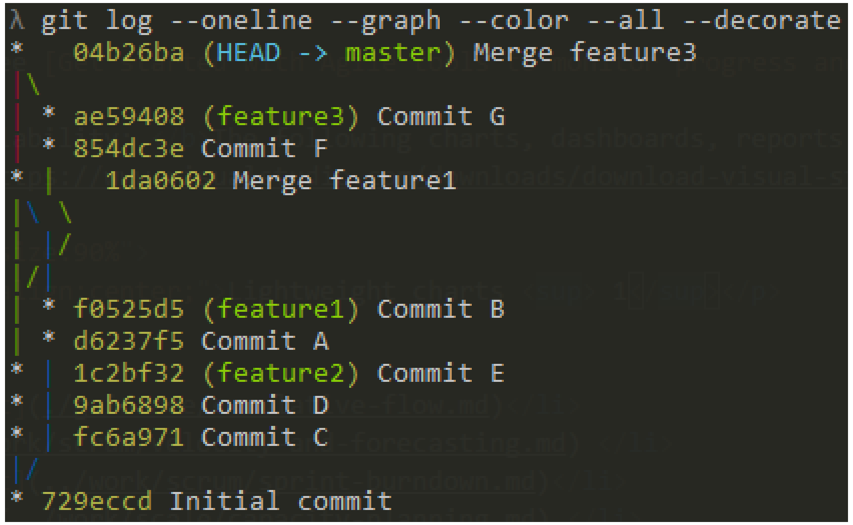
- Files in Git are in one of three states: modified, staged, or committed
- You must stage your changes to commit them; the staging area contains all the changes you wish to commit
- Once committed, these changes in the staged area become part of your history



Git Basics - History

- Centralized systems store a separate history for each file in a repository
- Git stores history as a graph of snapshots of the entire repository
- Snapshots, or commits, can have multiple parents creating a history that looks like a graph instead of a straight line

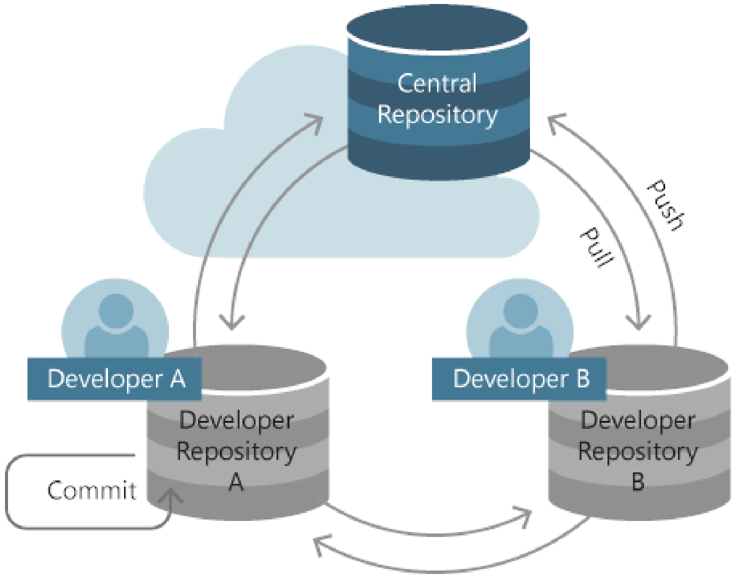




```
λ git log --oneline --graph --color --all --decorate
*   04b26ba (HEAD -> master) Merge feature3
|\
| * ae59408 (feature3) Commit G
| * 854dc3e Commit F
* |   1da0602 Merge feature1
|\ \
| |/
|/|
| * f0525d5 (feature1) Commit B
| * d6237f5 Commit A
* | 1c2bf32 (feature2) Commit E
* | 9ab6898 Commit D
* | fc6a971 Commit C
|/
* 729eccd Initial commit
```

Git Basics - Repositories

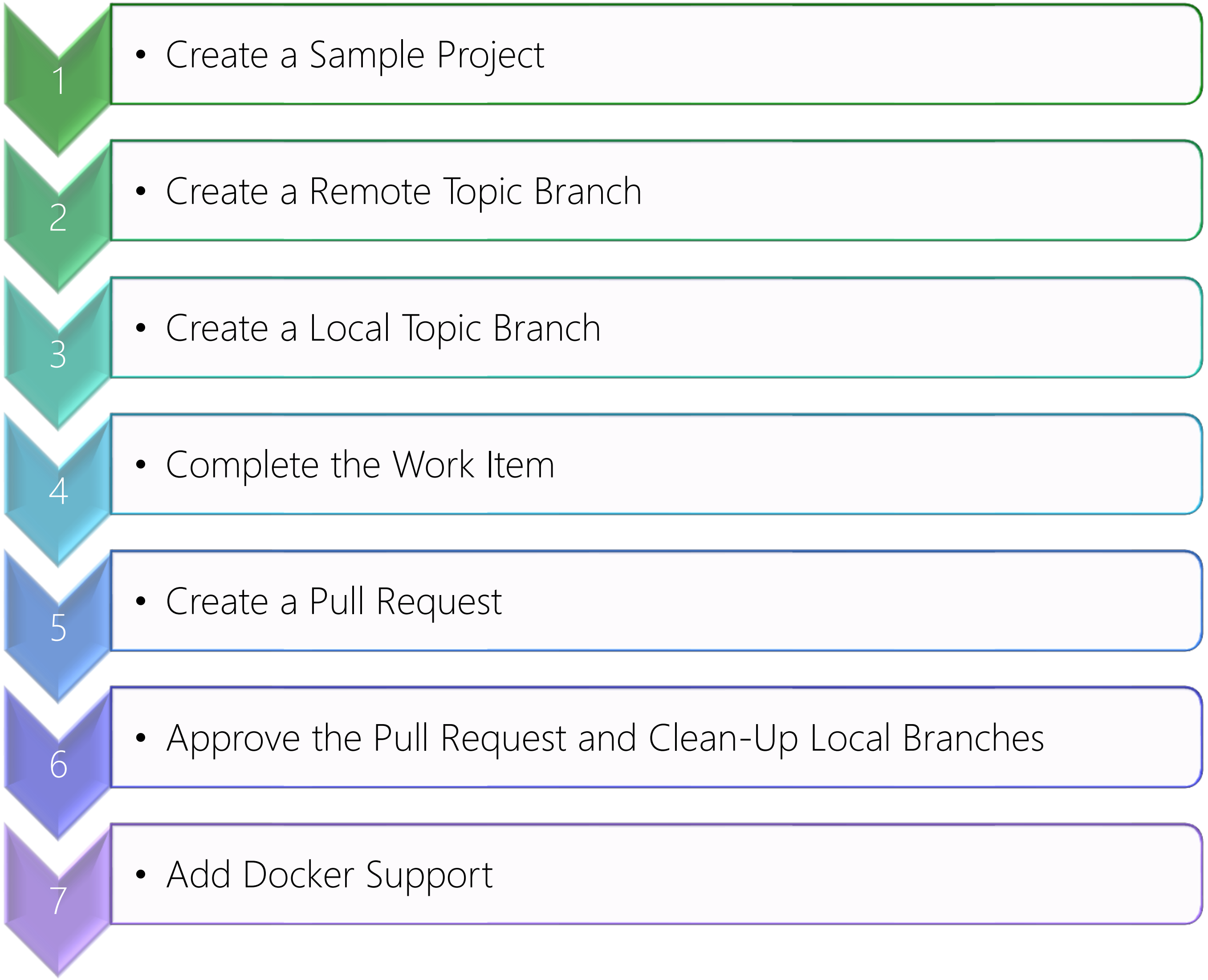
- A repository, or repo, is a folder you have told Git to track changes
- You can have any number of repositories on your system and each are independent
- Contains every version of every file saved in the repository in the hidden .git folder
- Most teams coordinate their changes using a shared repository
- Sets up the basis for continuous integration



Demonstration: Source Control Collaboration

We will use Git in Azure DevOps Services to showcase how your engineers can collaborate on features.

Demonstration Review



1. Create a Sample Project
2. Create a Remote Topic Branch
3. Create a Local Topic Branch
4. Complete the Work Item
5. Create a Pull Request
6. Approve the Pull Request and Clean-Up Local Branches
7. Add Docker Support

Module 2: Dev & Test

Lab 2: Source Control Management using Git

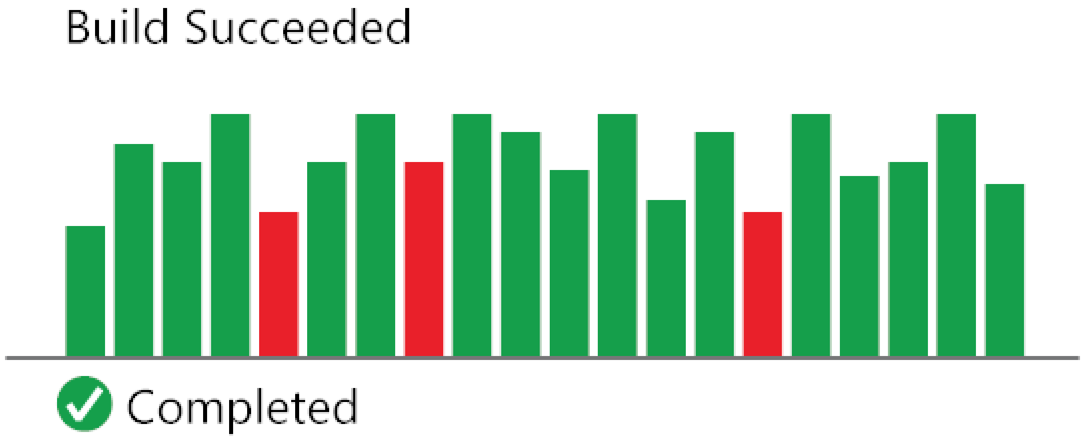
Exercise 1: Create a Sample Project
Exercise 2: Create a Remote Topic Branch
Exercise 3: Create a Local Topic Branch
Exercise 4: Complete the Work Item
Exercise 5: Create a Pull Request
Exercise 6: Approve the Pull Request and Clean Up
Exercise 7: Add Docker Support

Lab Time: 90 minutes (about 1 and a half hours)

CI/CD Strategy

Continuous Integration

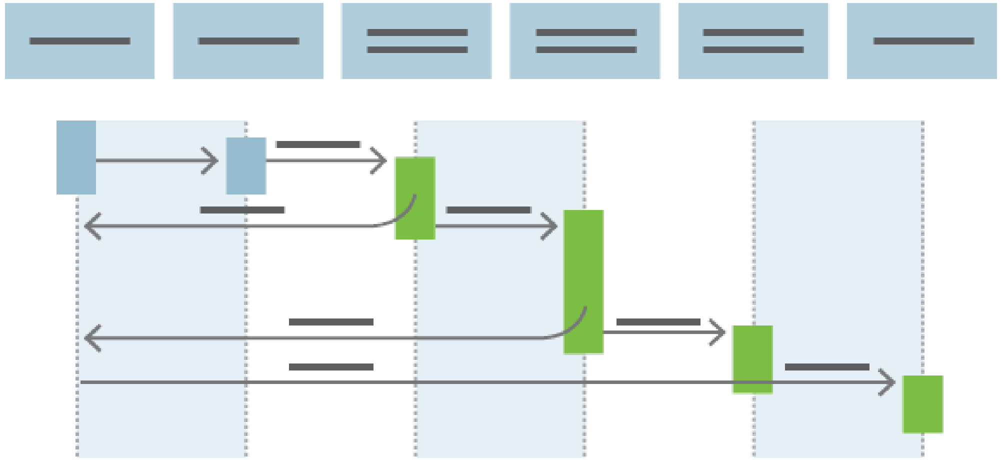
- Process of automating the build and testing of code every time a team member commits changes to version control
- Encourages developers to share their code and unit test by merging their changes into a shared version control repository after every small task completion



Source: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration

Continuous Delivery

- Process to build, test, configure and deploy from a build to a production environment with the goal of keeping production fresh
- Multiple testing or staging environments create a Release Pipeline to automate the creation of infrastructure and deployment
- Continuous Delivery may sequence multiple deployment "rings" for progressive exposure (known as "controlling the blast radius")



Source: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-delivery

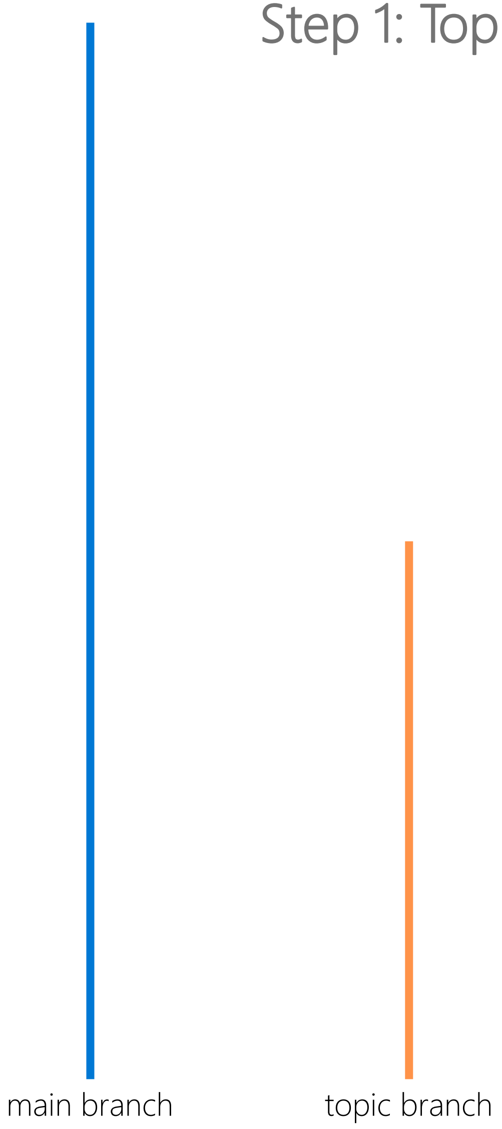
Suggested Strategy / Workflow

Goal: Build, Test, and Release to support
Agility, Testability, Operations, and Traceability

In the next slide series we will see how to use the Topic Branching source control methodology with Continuous Integration, Continuous Delivery and Shift-Left Testing.

CI/CD Workflow

Step 1: Topic branch created for every user story



Phase: Team start working on a story

Action: Team member creates a Topic Branch

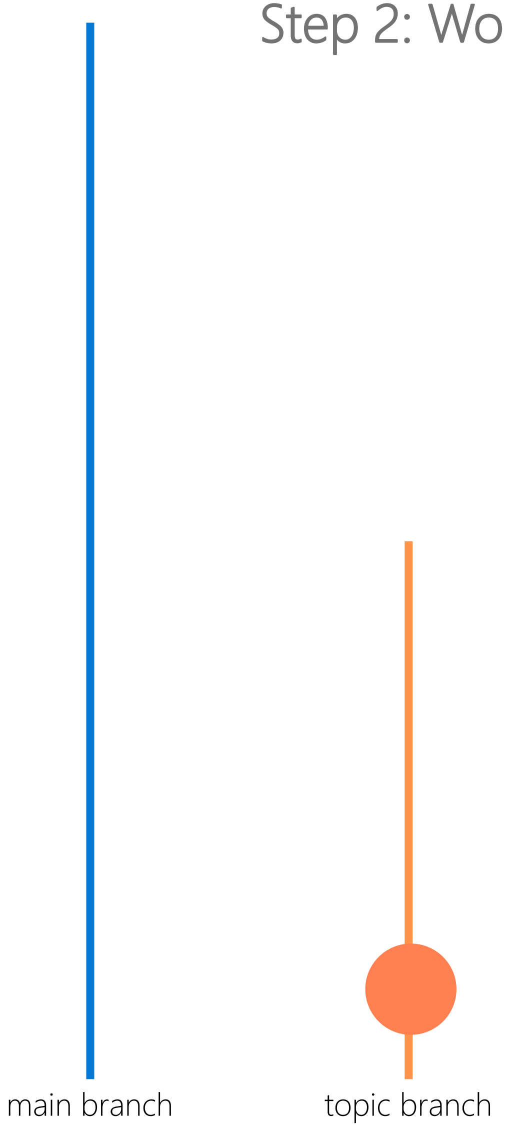
Goal: Parallel development

Assumptions:

Branch creation is done from the work item
Topic branches are short lived
Branch names include work items id (ex: tb-123)

CI/CD Workflow

Step 2: Work is committed to the topic branch on task completion



Phase: Team members push partial work to the topic branch

Action: Optionally relate commits to tasks

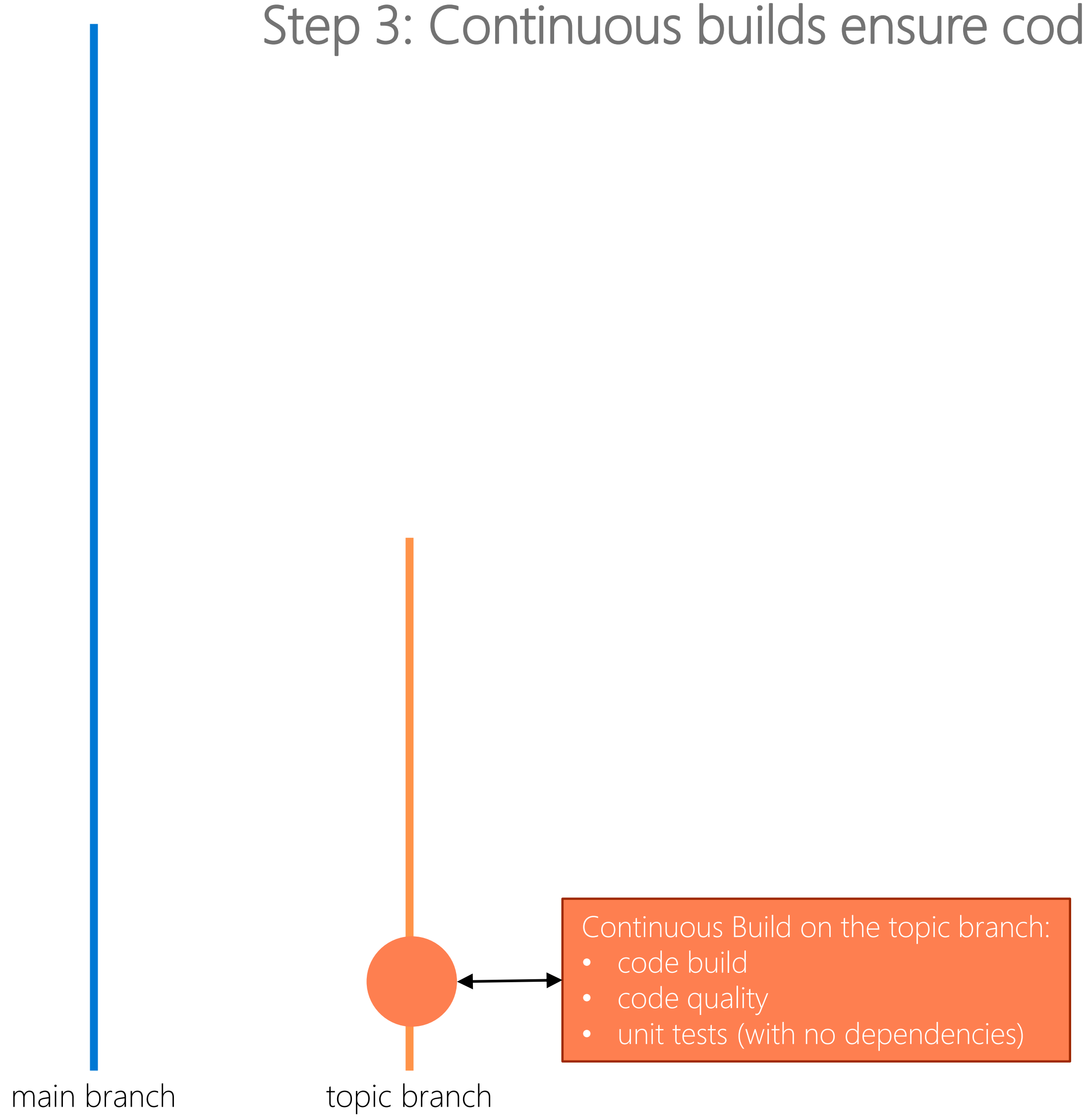
Goal: Parallel development and detailed traceability

Assumptions:

A feature flag might wrap the code for the overall feature.

CI/CD Workflow

Step 3: Continuous builds ensure code correctness, quality, security, and green unit tests



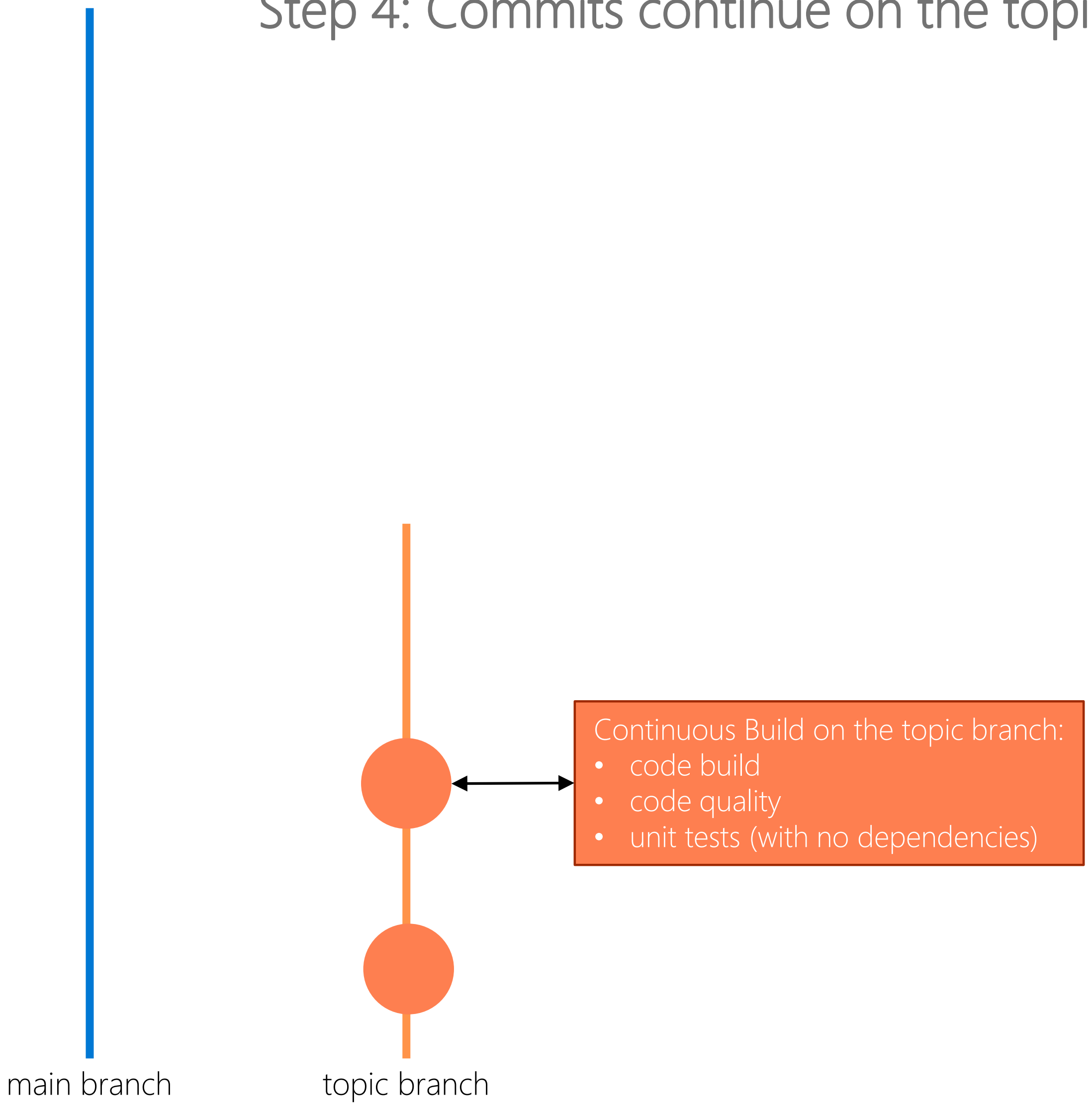
Continuous Build on the topic branch:

- code build
- code quality
- unit tests (with no dependencies)

Phase:	Build Test and Verification
Action:	A topic release pipeline is triggered for every successful build
Goal:	Commit functional and technical verification
Assumptions:	
	Ability to automate the create of team owned environments
	Environments can be discarded once the validation is done
	Tests are done in isolation from other topics

CI/CD Workflow

Step 4: Commits continue on the topic branch while tasks are completed



Continuous Build on the topic branch:
- code build
- code quality
- unit tests (with no dependencies)

Phase: User story implementation

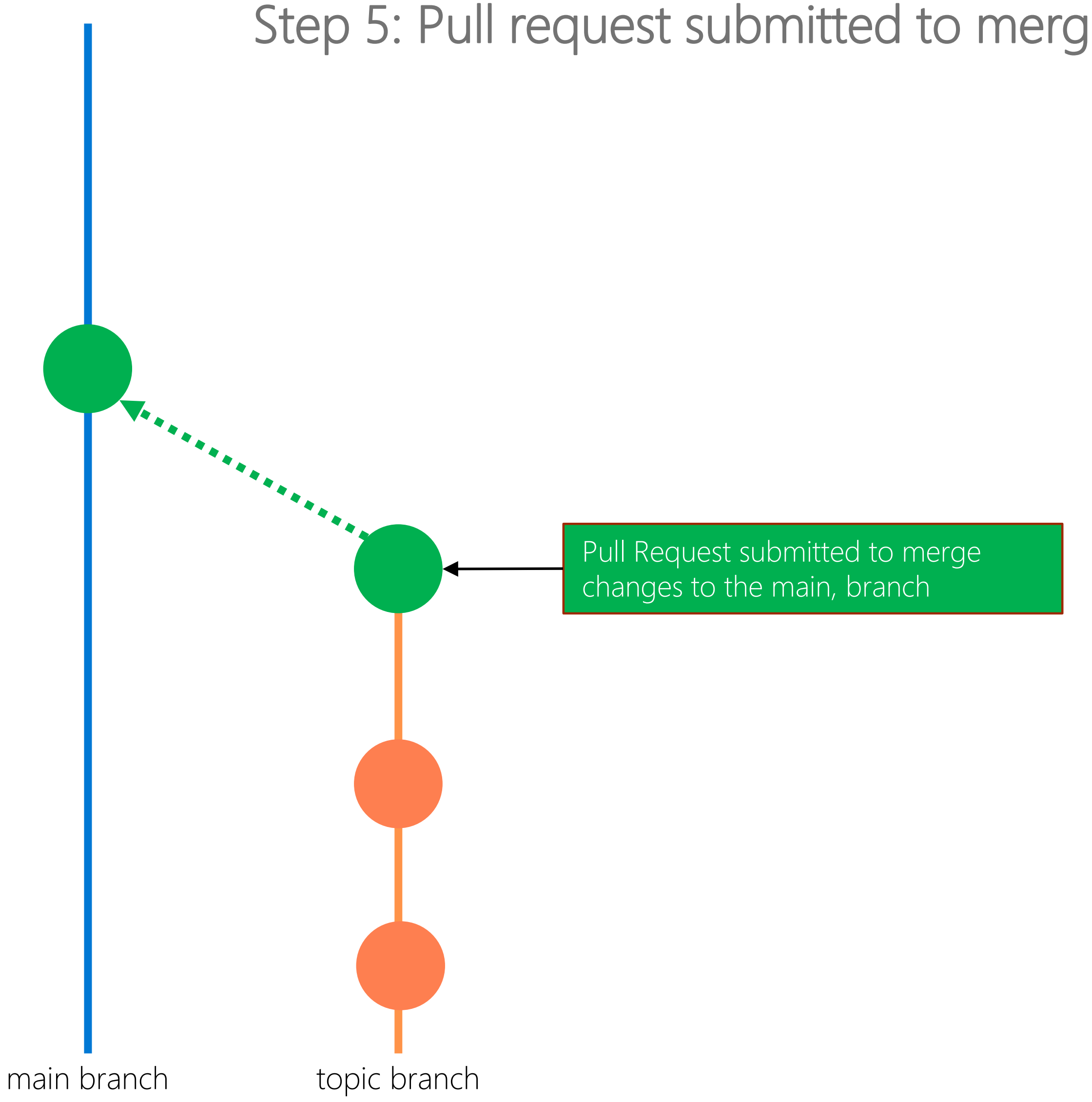
Action: Commits continue to the topic branch while work is in progress

Goal: Isolating incomplete work from main branch

Assumptions:
Topic branch is not kept for a long time

CI/CD Workflow

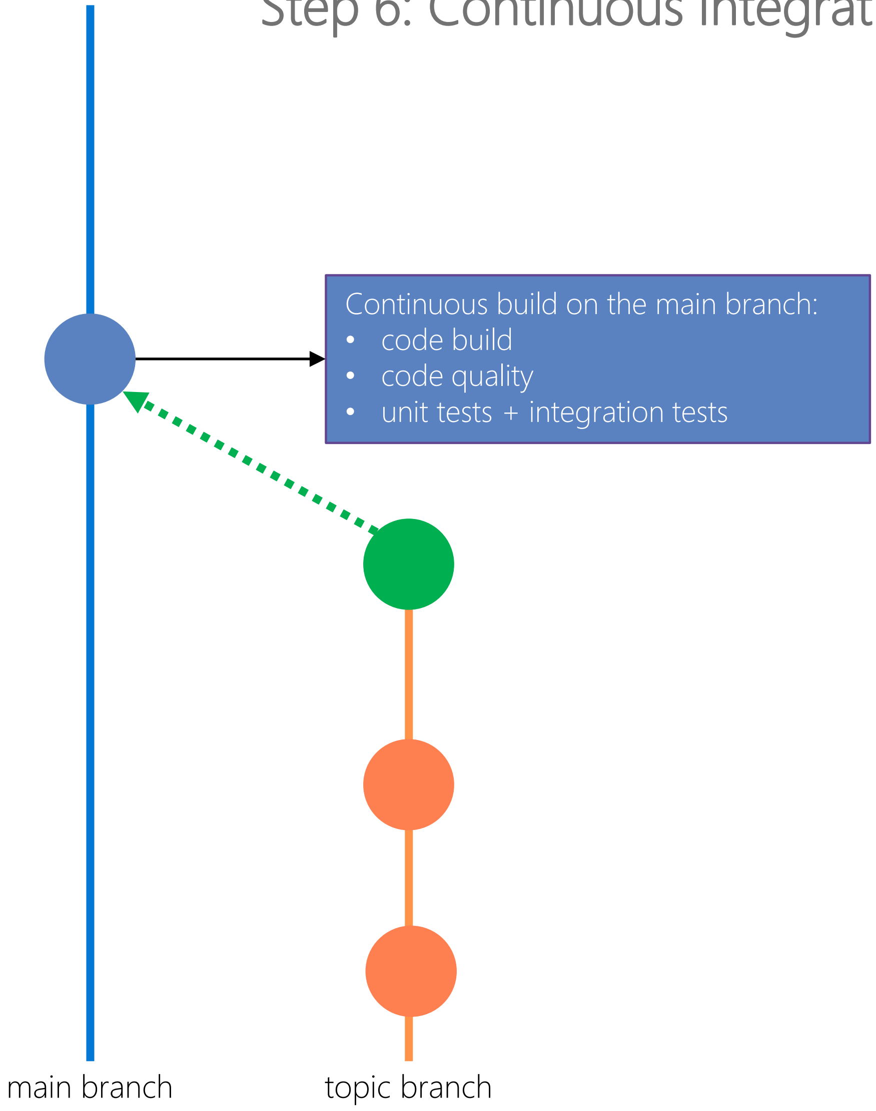
Step 5: Pull request submitted to merge changes to the main branch



Phase:	Work on the topic is done
Action:	A pull request is initiated to request a merge to main
Goal:	No work is committed to main without due diligence
Assumptions:	
	Users cannot approve their own pull requests
	Pull requests are associated to a work item
	Branch policies protect the main branch
	Code coverage metrics are tracked for code related work
	Code quality metrics are tracked for code related work

CI/CD Workflow

Step 6: Continuous Integration

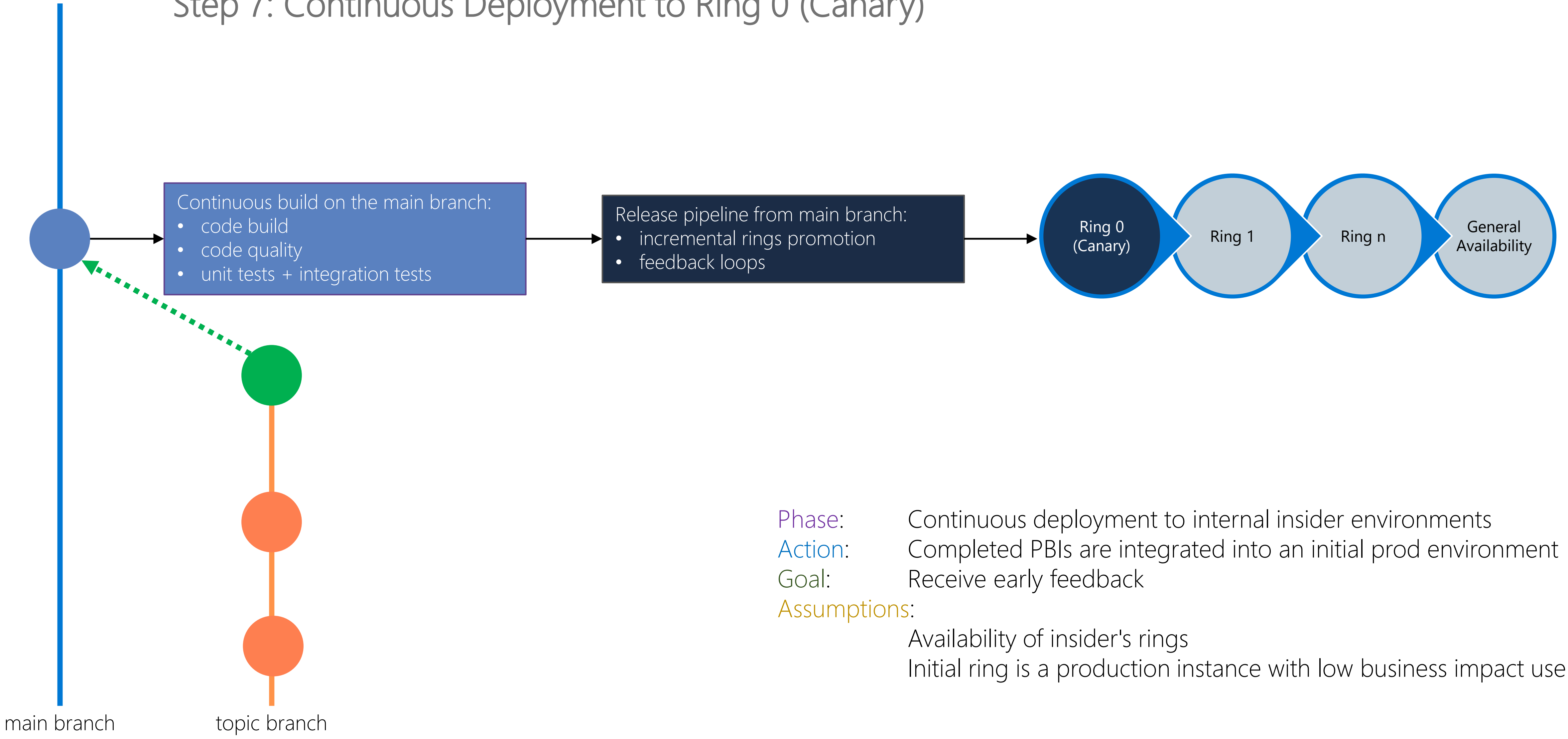


Continuous build on the main branch:
- code build
- code quality
- unit tests + integration tests

Phase:	Work is done on the work item
Action:	The pull request is approved, and a CI build is triggered
Goal:	Continuously integrate, and always work in main
Assumptions:	
	CI builds validate the correctness and completeness of the PBI

CI/CD Workflow

Step 7: Continuous Deployment to Ring 0 (Canary)



Continuous build on the main branch:

- code build
- code quality
- unit tests + integration tests

Release pipeline from main branch:

- incremental rings promotion
- feedback loops

Phase: Continuous deployment to internal insider environments

Action: Completed PBIs are integrated into an initial prod environment

Goal: Receive early feedback

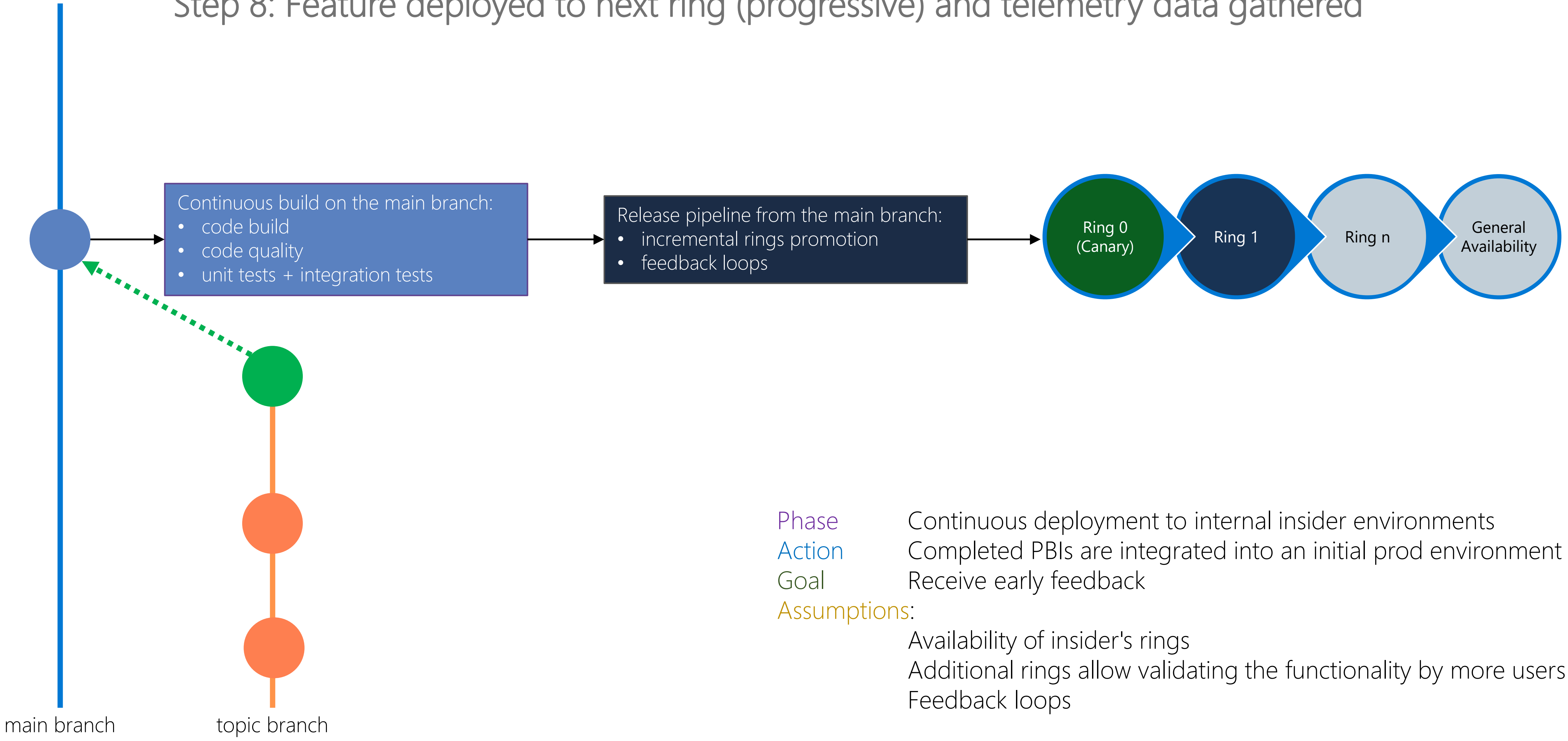
Assumptions:

Availability of insider's rings

Initial ring is a production instance with low business impact users

CI/CD Workflow

Step 8: Feature deployed to next ring (progressive) and telemetry data gathered



Continuous build on the main branch:

- code build
- code quality
- unit tests + integration tests

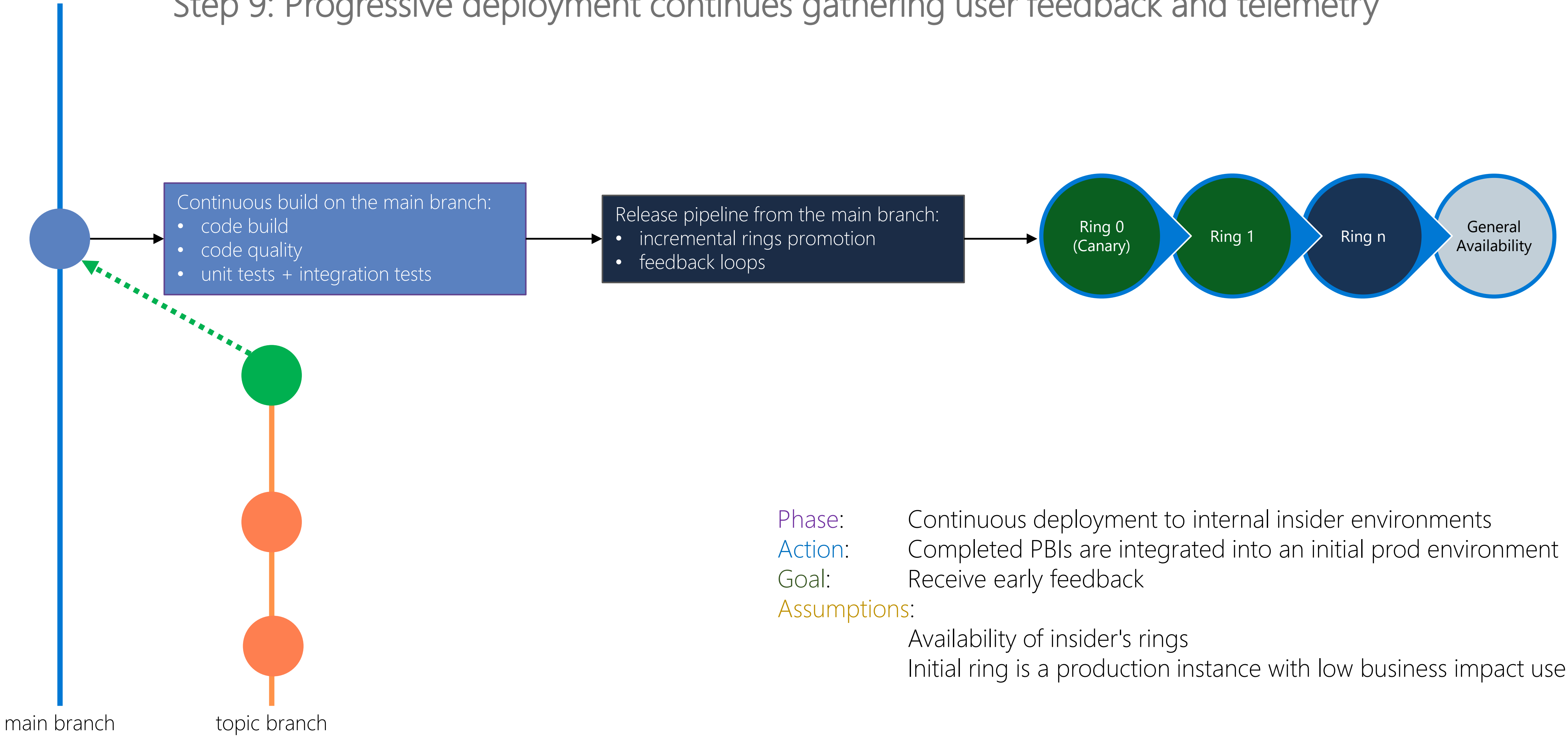
Release pipeline from the main branch:

- incremental rings promotion
- feedback loops

Phase	Continuous deployment to internal insider environments
Action	Completed PBIs are integrated into an initial prod environment
Goal	Receive early feedback
Assumptions:	
	Availability of insider's rings
	Additional rings allow validating the functionality by more users
	Feedback loops

CI/CD Workflow

Step 9: Progressive deployment continues gathering user feedback and telemetry



Continuous build on the main branch:

- code build
- code quality
- unit tests + integration tests

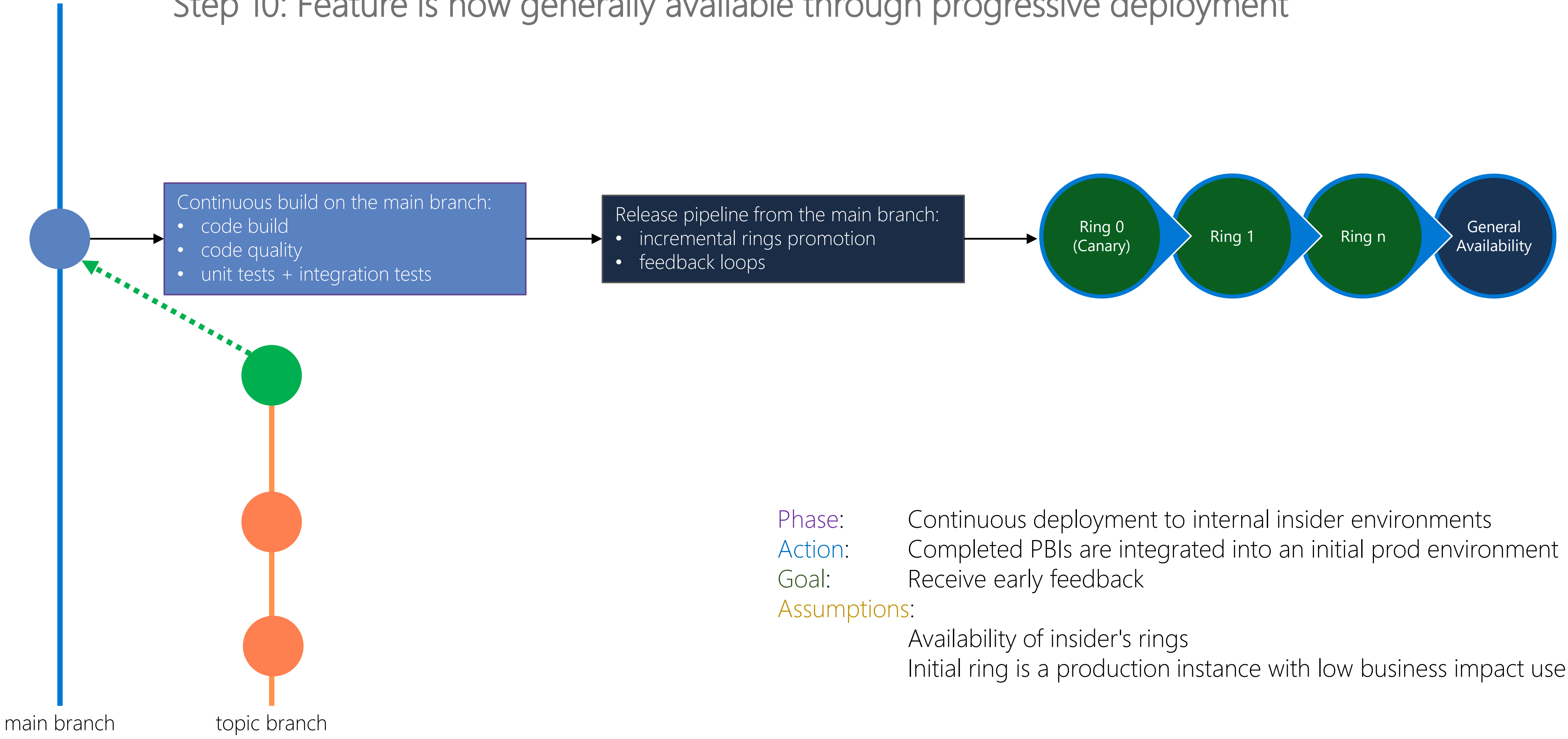
Release pipeline from the main branch:

- incremental rings promotion
- feedback loops

Phase: Continuous deployment to internal insider environments
Action: Completed PBIs are integrated into an initial prod environment
Goal: Receive early feedback
Assumptions:
Availability of insider's rings
Initial ring is a production instance with low business impact users

CI/CD Workflow

Step 10: Feature is now generally available through progressive deployment



Continuous build on the main branch:

- code build
- code quality
- unit tests + integration tests

Release pipeline from the main branch:

- incremental rings promotion
- feedback loops

Phase: Continuous deployment to internal insider environments
Action: Completed PBIs are integrated into an initial prod environment
Goal: Receive early feedback
Assumptions:
Availability of insider's rings
Initial ring is a production instance with low business impact users

Shift-Left Testing

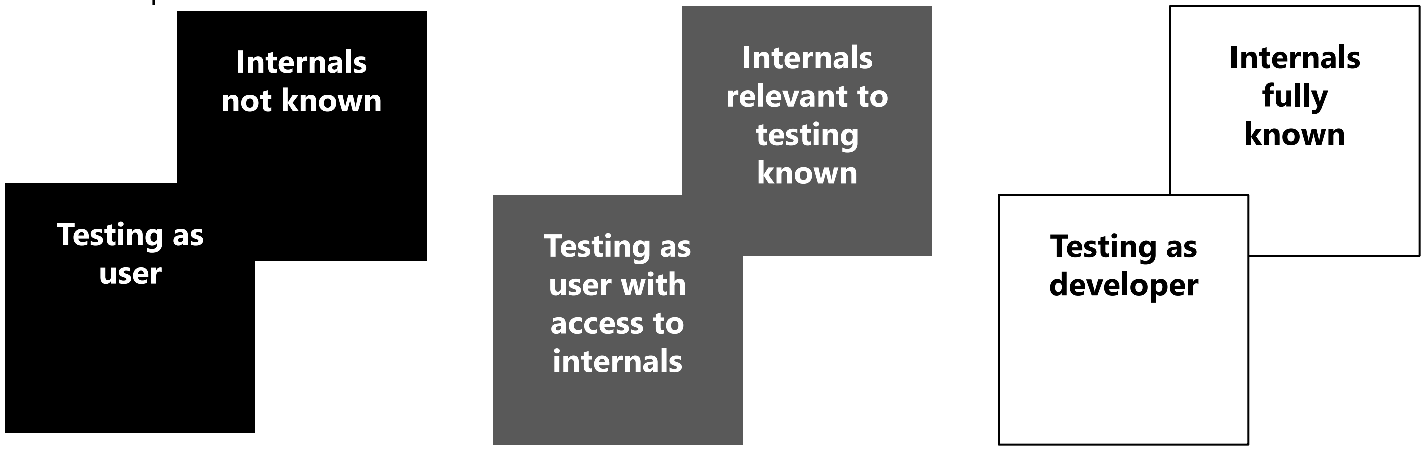
An approach to software testing and system testing in which testing is performed ***earlier*** *in the lifecycle (i.e. moved left on the project timeline).*

Software Testing

- Software testing is the art of measuring and maintaining software quality to ensure that user expectations and requirements, business value, non-functional requirements, such as security, reliability and recoverability, and operational policies are all met
- Testing is a team effort to achieve the well understood and agreed upon minimum quality bar and definition of "done."

Traditional Testing Strategies

- Black Box
  - The inside of the box “solution implementation” is not known. Testers focus only on input and output. Typically, when performing system and user acceptance testing
- White Box
  - The inside of the box is known and analyzed as part of the testing
- Gray Box
  - Combines black box and white box strategies and typically used to test edge cases. Requires understanding of the internals and expected behavior.



Internals not known

Testing as user

Internals relevant to testing known

Testing as user with access to internals

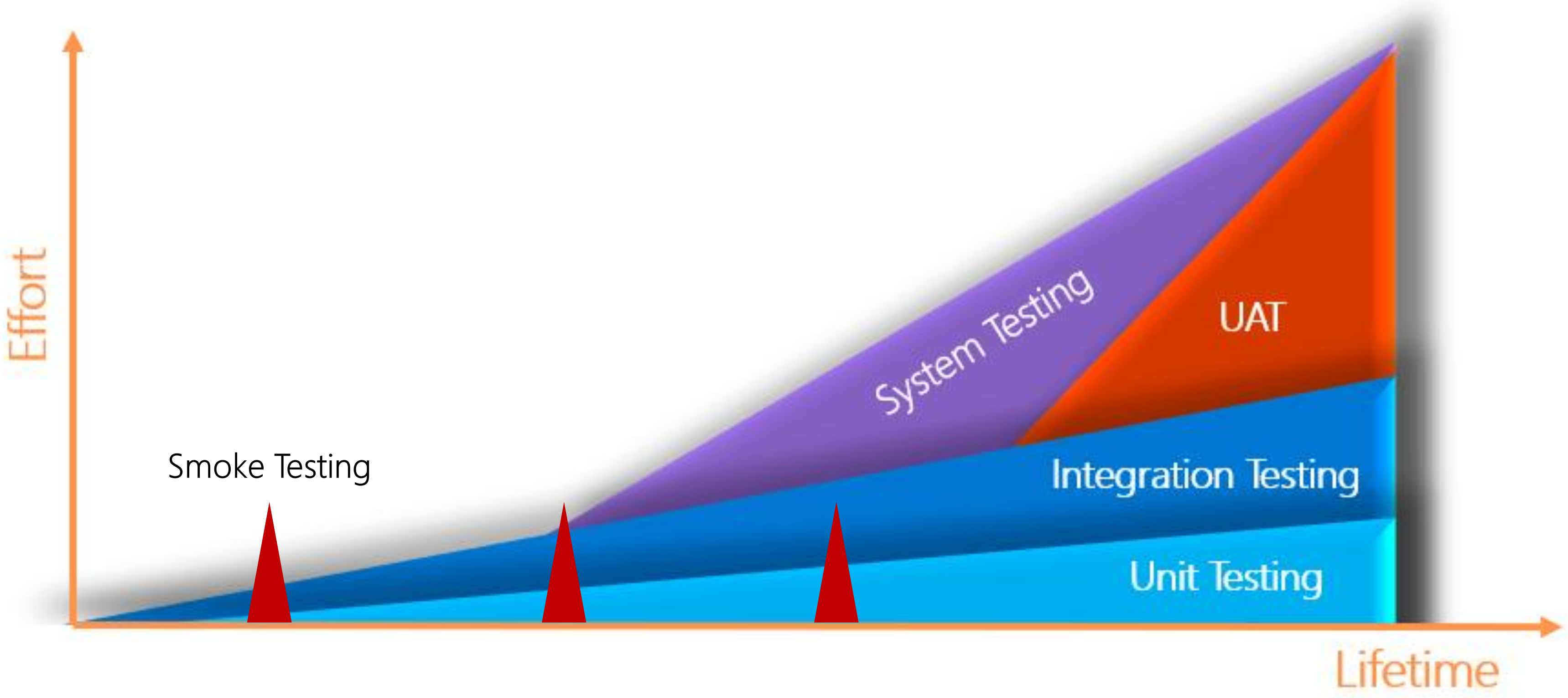
Internals fully known

Testing as developer

Some Testing Types

- Exploratory Tests – No predefined tests
- Integration Tests – Test components working together
- Load / Stress Tests – Test under load in a controlled environment
- Regression Tests – Test entire system for the same quality
- Smoke Tests – Test a new feature or idea before committing code
- System Tests – Test entire system against expected features
- Unit Tests – Test the smallest unit of code (method / class)
- User Acceptance Tests – Users review, and test based on requirements

Traditional Usage Testing Types



Unit Test or Not?

- Having bug detection or deterministic quality signal means you can find bugs early
- Increases your confidence in making big changes
- A test is flaky when it passes sometimes and fails sometimes
- Flaky tests can impact the value of an automated regression suite

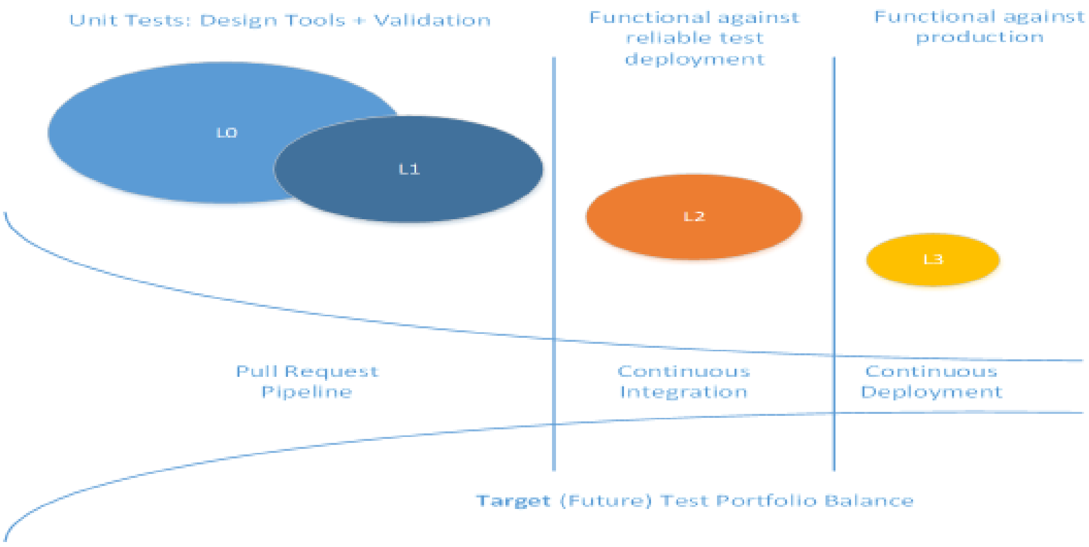
I have an ongoing fear that all my unit tests pass, but they're meaningless.

Source: www.devops.com

Do we have to test differently in a DevOps context?

Consequences of Testing Late in the Cycle

Shift-Left and move quality upstream


Unit Tests: Design Tools + Validation
Functional against reliable test deployment
Functional against production
L0
L1
L2
L3
Pull Request Pipeline
Continuous Integration
Continuous Deployment
Target (Future) Test Portfolio Balance

- ✕ Not enough resources to test
- ⚠ Defects are uncovered after significant effort is wasted
- Regression tests are more difficult
- Less time available for fixing bugs
- Higher deployment and maintenance costs
- ☹ Lower team morale

Shift-Left Strategies

- Tests should be written at the lowest level possible
- Write once, run anywhere including production system
- Product is designed for testability
- Test code *is* product code, only reliable tests survive
- Testing infrastructure is a shared service
  - Tests run in the build process and other processes
- Test ownership follows product ownership
  - Tests sit right next to product code

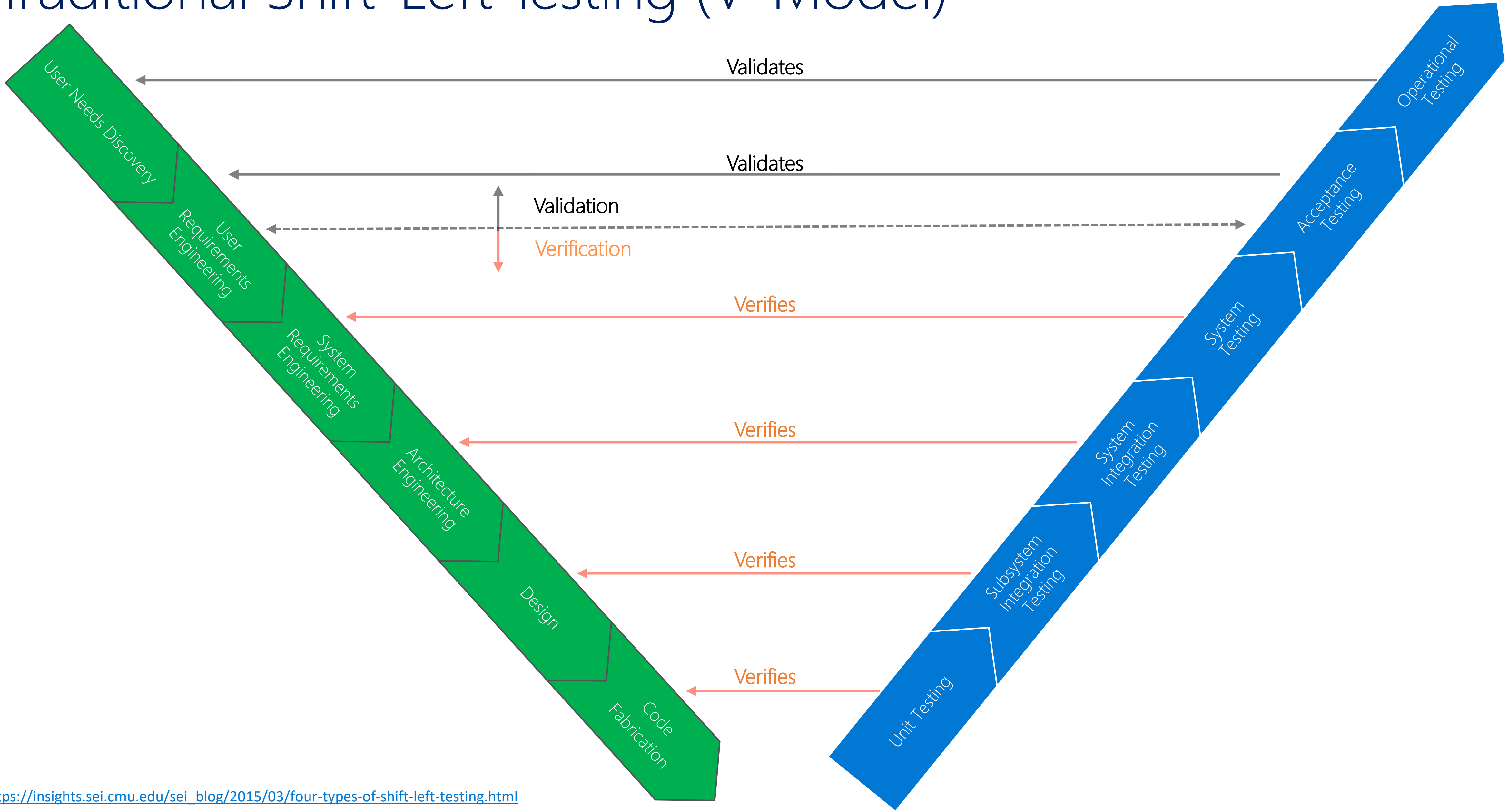
Creating a Test Taxonomy

- Categorize tests to represent external dependencies
- Establish test rules:
  - Do not allow a L0 test to exceed 2 seconds
  - Chart your test execution

L0/L1 – Unit tests
L0 – Broad class of fast in-memory unit tests. A test that depends on code in the assembly under test and nothing else.
L1 – A L1 test might require the assembly plus SQL or the file system.

L2/L3 – Functional tests
L2 – Functional tests run against "testable" service deployment. It is a functional test category that requires a service deployment but may have key service dependencies stubbed out in some way.
L3 – Restricted class of integration tests that run against production. They require a full product deployment.

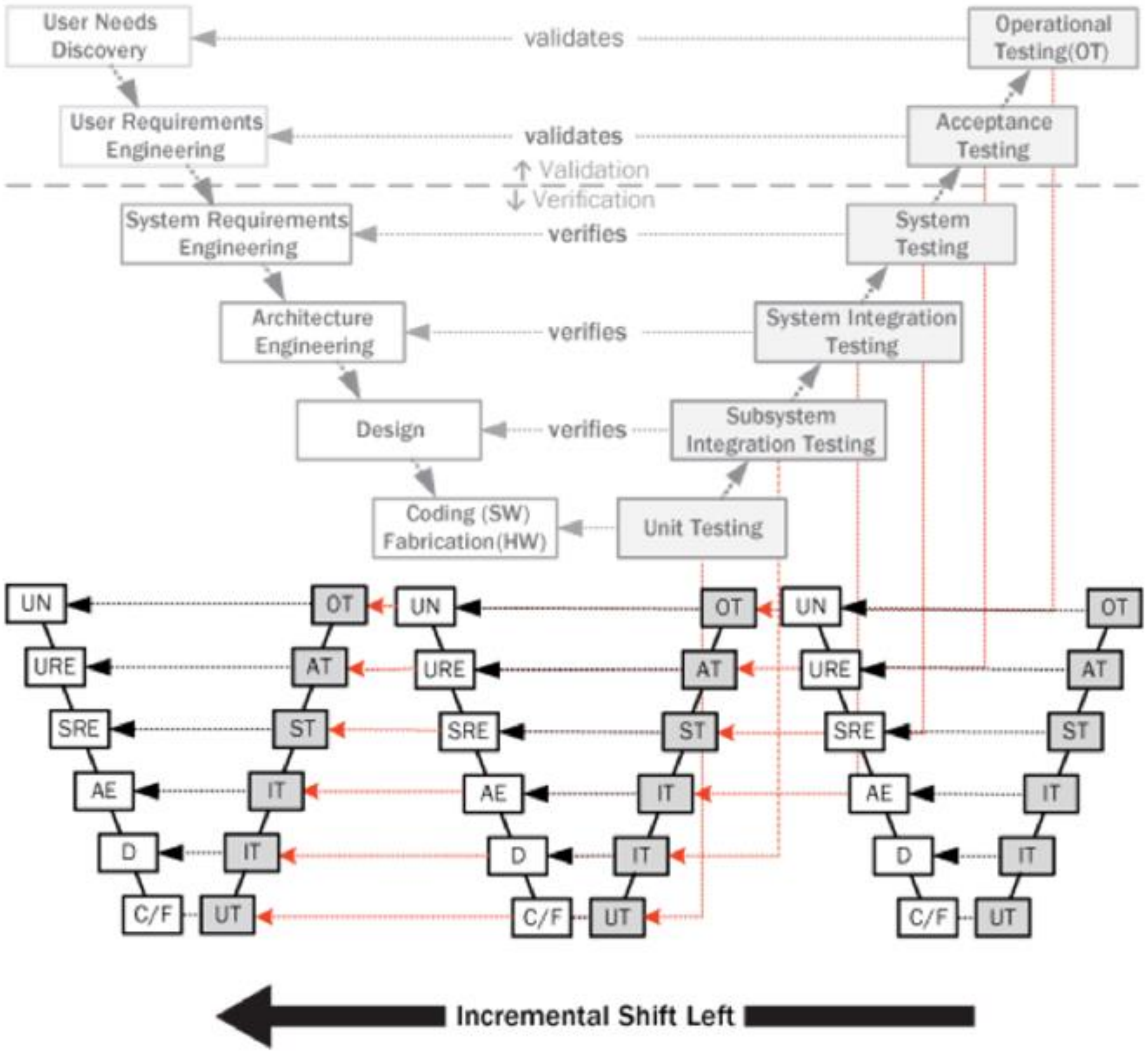
Traditional Shift-Left Testing (V-Model)



Source : https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html

Incremental Shift-Left Testing

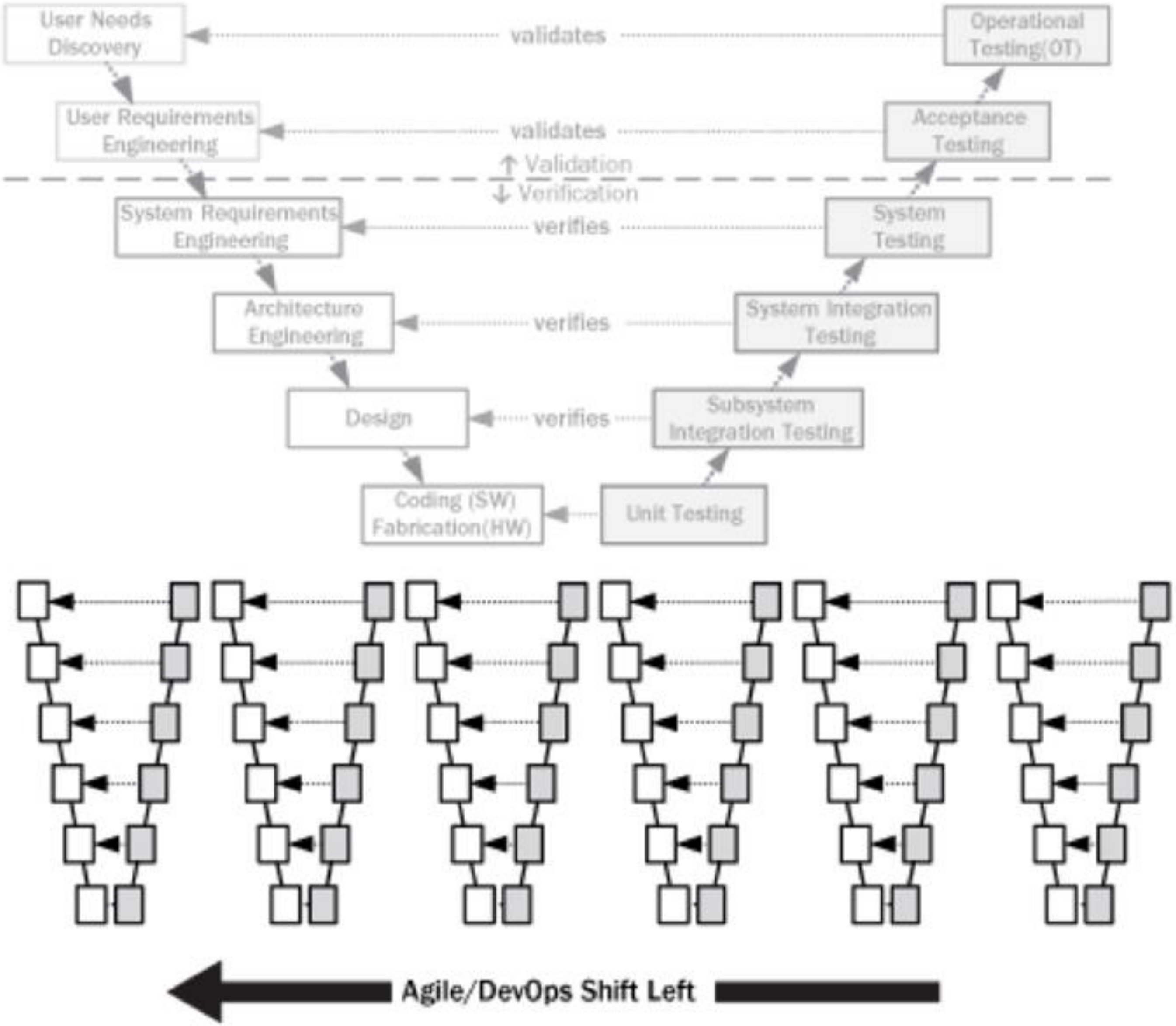
- Large software development has decomposed into increments, smaller V's
- Parts of the large testing shift-left into the corresponding increments
- Each increment is a delivery to the customer and operations, you shift-left developmental testing and operational testing



Source : https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html

Agile / DevOps Shift-Left Testing

- Shift-left occurs because the types of testing on the right sides of these tiny Vs are to the left of the corresponding types of testing on right side of the larger V(s) they replace.

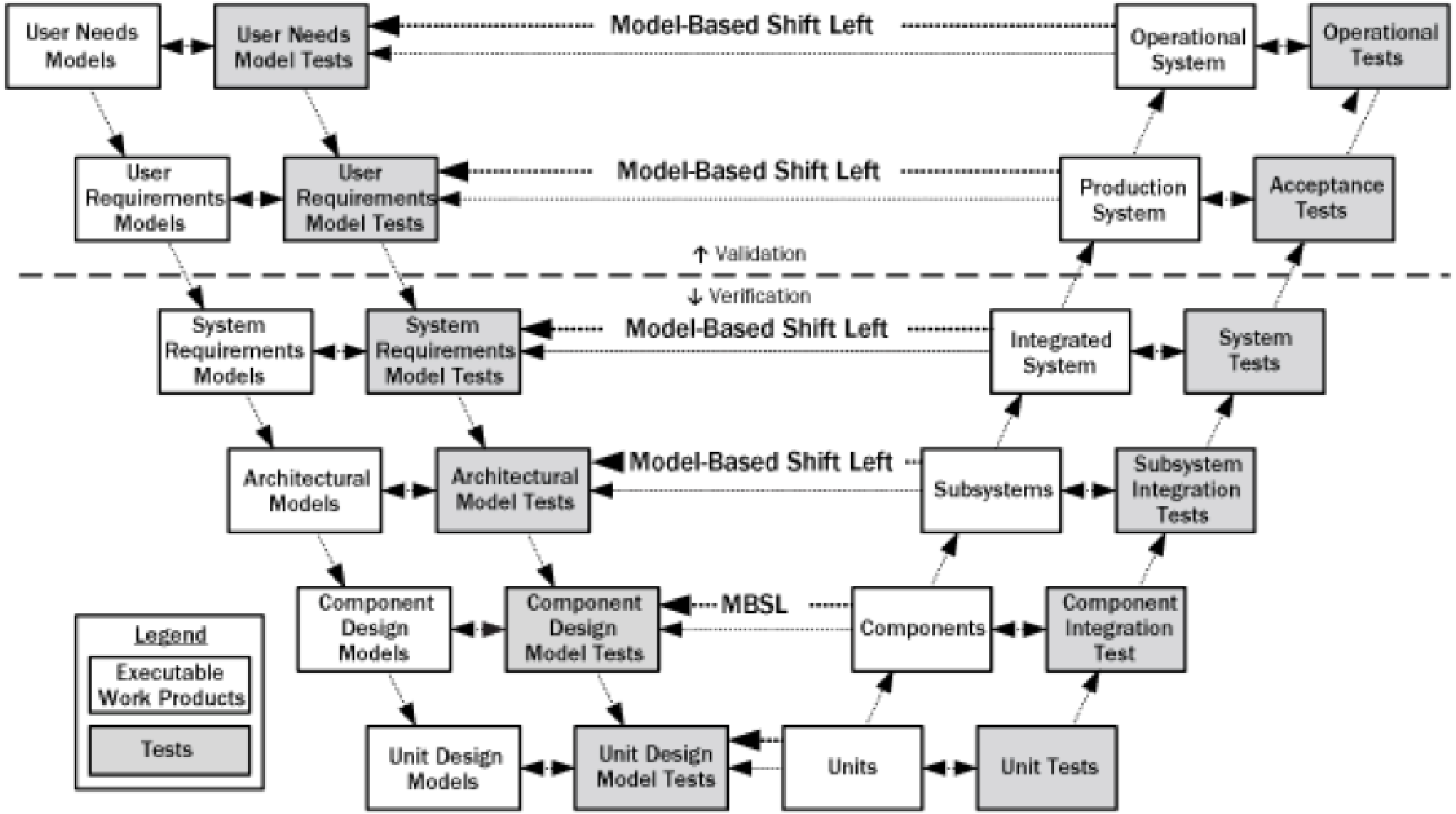


Source : https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html

Model-Based Shift-Left Testing

- Model-Based shift left testing introduces the testing of executable requirements, architecture, and design models.

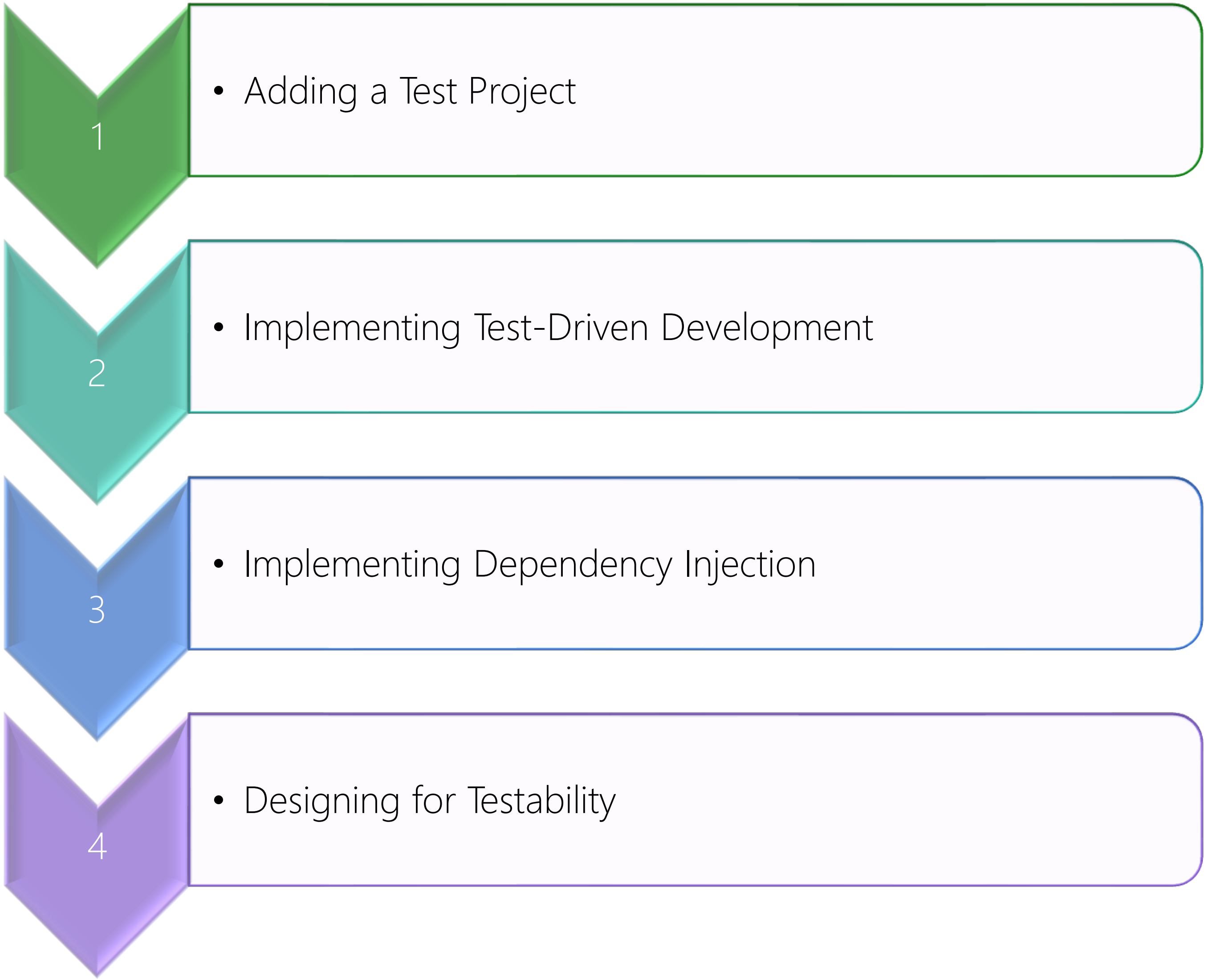
- This approach is used to test requirements, which typically do have errors.



Demonstration: Shift-Left Testing

We will introduce some basic testing techniques that help to shift-left in your testing approach.

Demonstration Review



1

- Adding a Test Project

2

- Implementing Test-Driven Development

3

- Implementing Dependency Injection

4

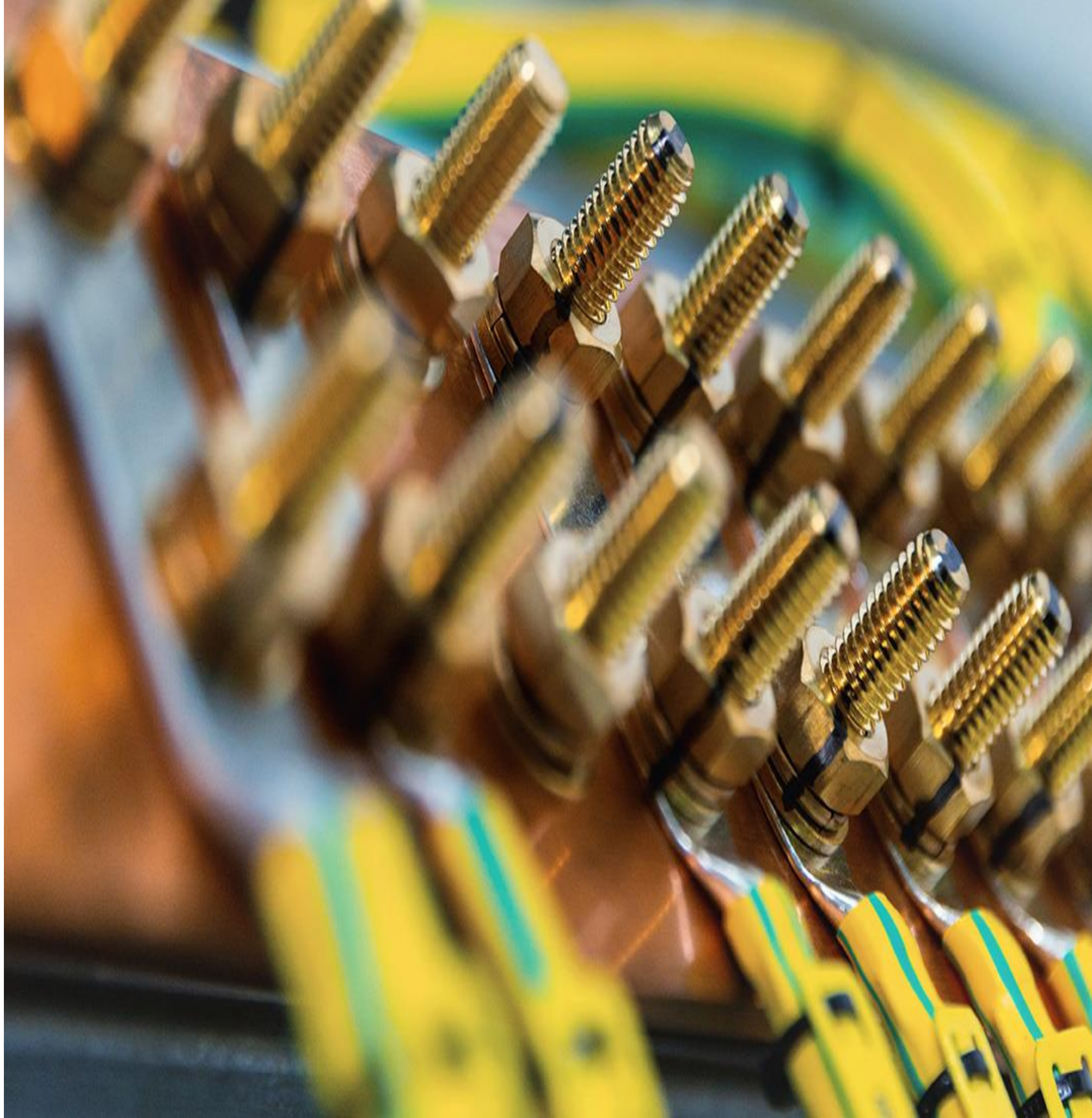
- Designing for Testability

Module 2: Dev & Test

Lab 3: Shift-Left Testing

Exercise 1: Adding a Test Project
Exercise 2: Test-Driven Development
Exercise 3: Dependency Injection
Exercise 4: Designing for Testability

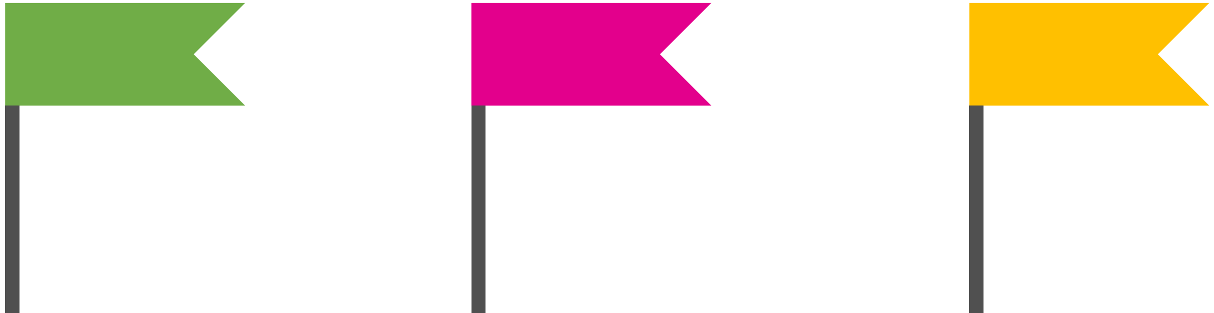
Lab Time: 90 minutes (about 1 and a half hours)



Feature Flags

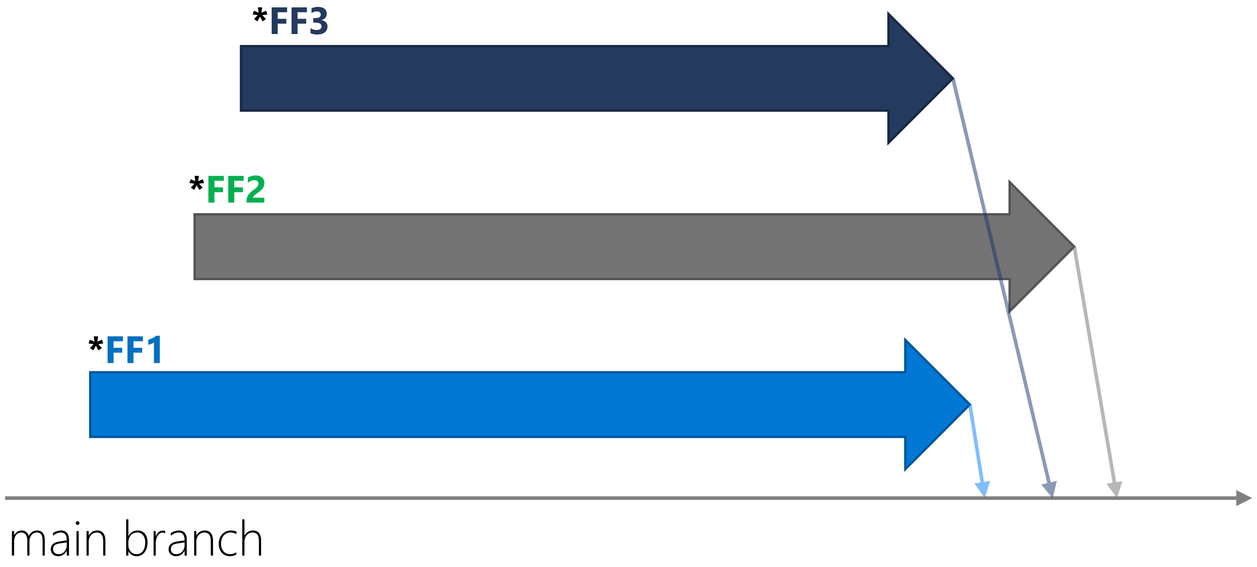
What are Feature Flags?

- Nearly eliminate integration debt
- Toggle features to hide, disable, or enable features at run-time
- Revert a deployed change without rolling back your release
- Present users with variants of a feature to determine which one works best



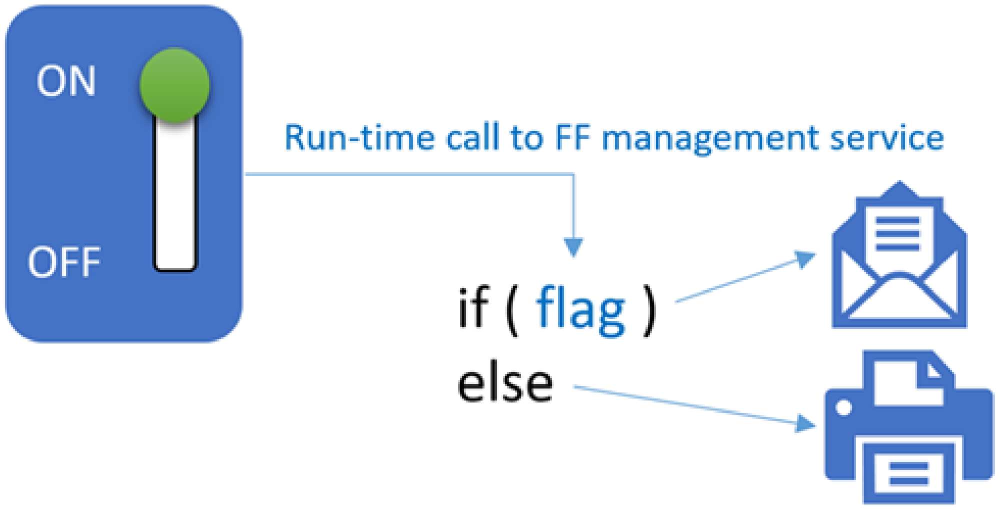
Eliminating Integration Debt

- Every new feature is wrapped with a flag
- Allows for isolation from the rest of the system
- Supports safe deployment



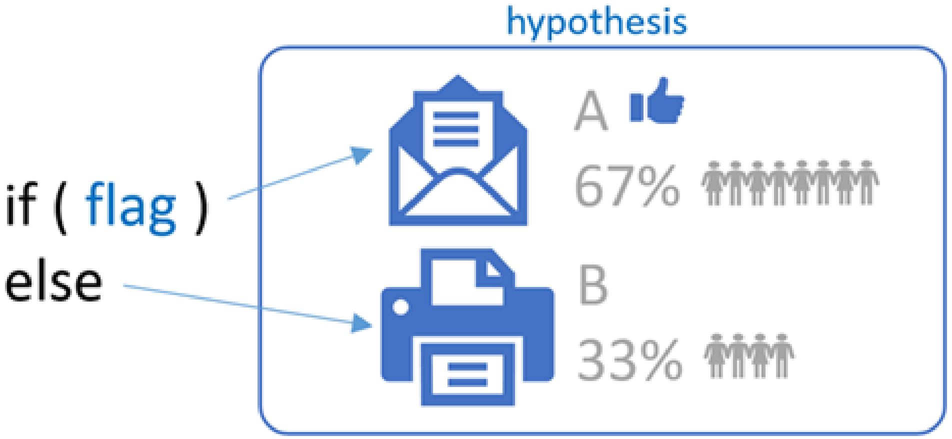
Feature Flags (ON | OFF)

- Feature flags act as an ON | OFF switch for a specific feature
- We can deploy a solution to production that includes both an email and a print feature.
- If the feature flag is set (ON), we'll email. If reset (OFF) we'll print



Hypothesis (A|B Testing)

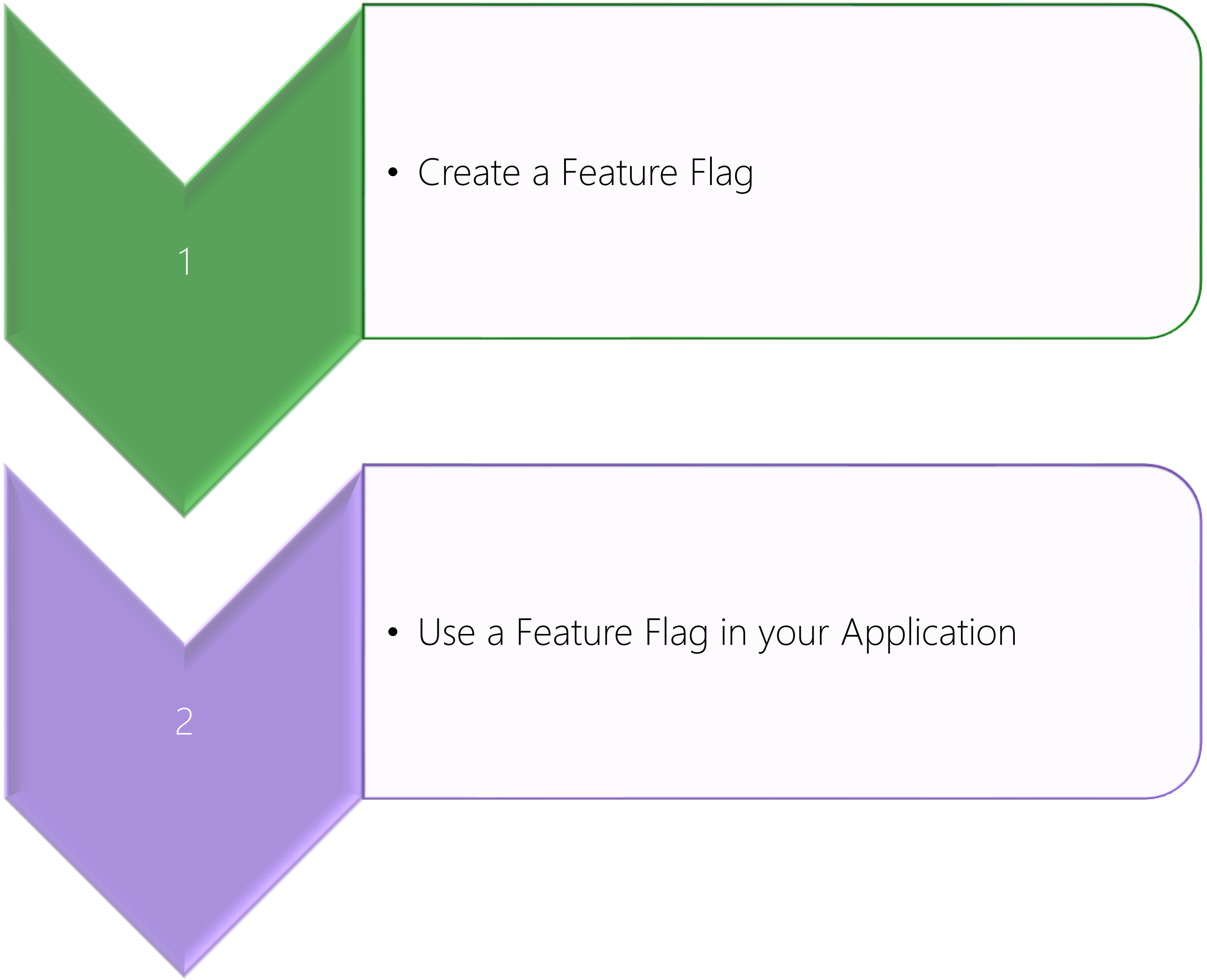
- Combine a feature flag with an experiment, *led by a hypothesis*, we introduce A|B testing
- For example, we could run an experiment to determine if the email (A) or the print (B) feature will result in a higher user satisfaction



Demonstration:
Feature Flags

We will introduce feature flags to discover how they help with integration debt and feature availability.

Demonstration Review



1

- Create a Feature Flag

2

- Use a Feature Flag in your Application

Module 2: Dev & Test

Lab 4: Feature Flags

Exercise 1: Create a Feature Flag
Exercise 2: Use a Feature Flag in your Application

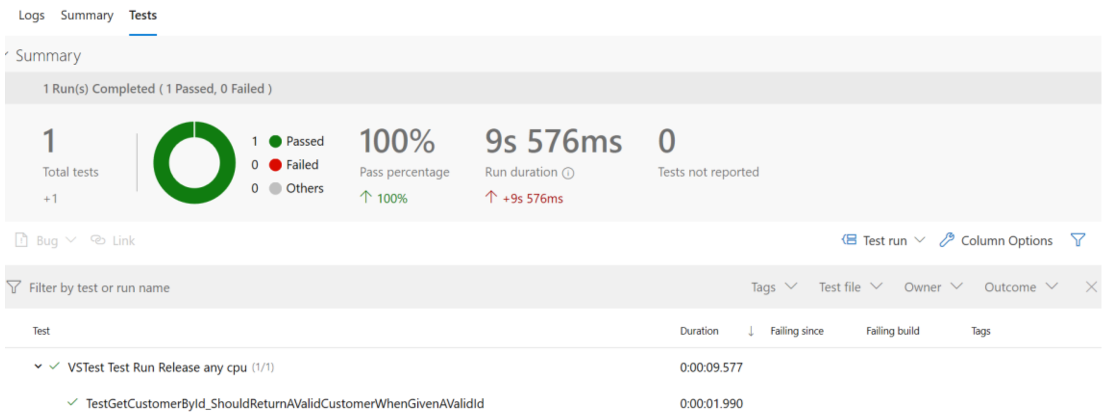
Lab Time: 30 minutes (half an hour)

Build Management

What is a Product Build?

Ideally, a single command should partially or completely:

- validate the system,
- validate functional correctness,
- package the product, and
- make sure it is ready to be shipped.



1 Run(s) Completed (1 Passed, 0 Failed)

1 Total tests +1

1 ● Passed
0 ● Failed
0 ● Others

100% Pass percentage ↑ 100%

9s 576ms Run duration ↑ +9s 576ms

0 Tests not reported

Test	Duration ↓	Failing since	Failing build	Tags
✓ VSTest Test Run Release any cpu (1/1)	0:00:09.577			
✓ TestGetCustomerById_ShouldReturnAValidCustomerWhenGivenAValidId	0:00:01.990			

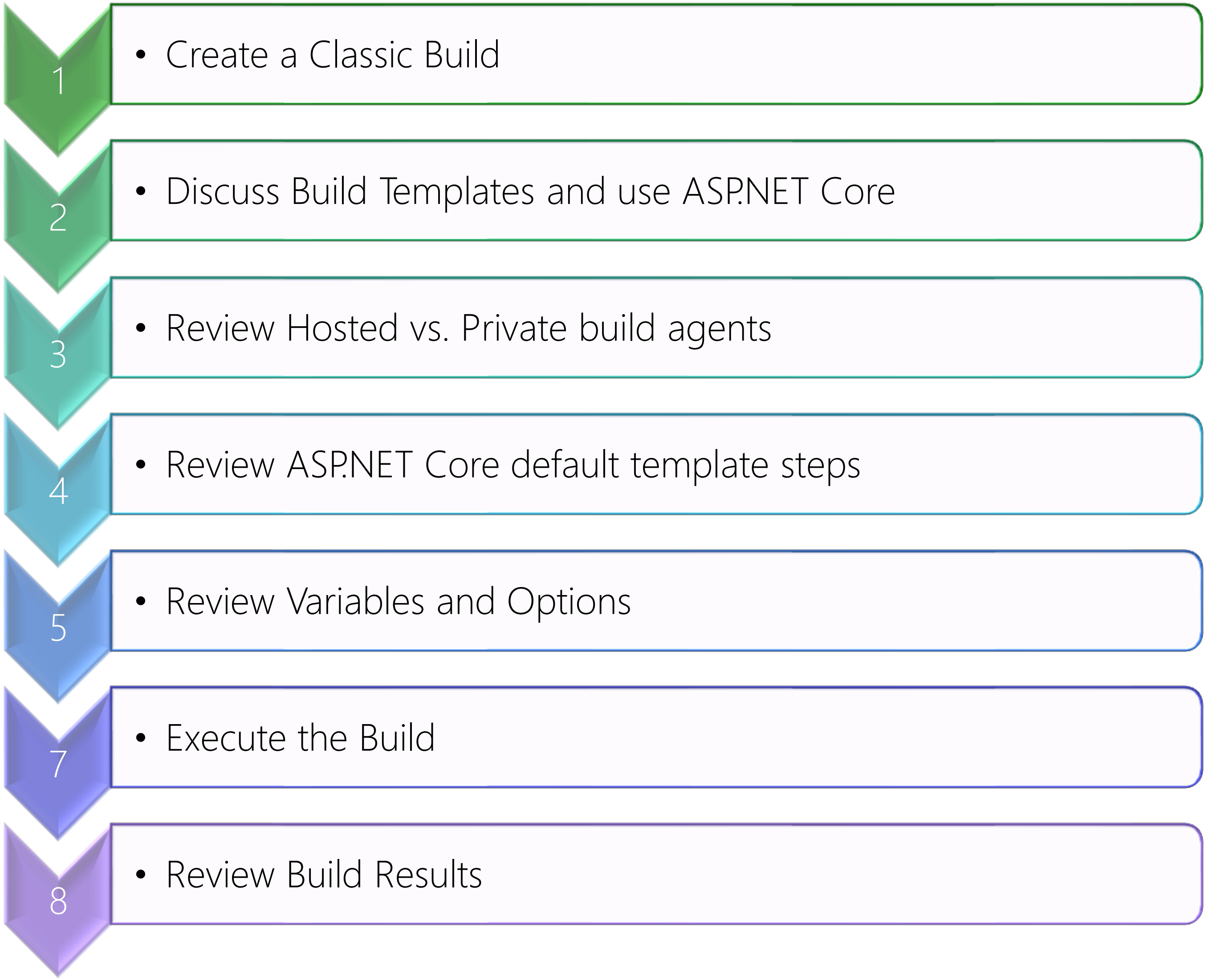
What is Build Management?

- Build consistently
- Changes in the build process are tracked
- Build results are triaged for quality
- Link artifacts to build events

Demonstration: Build Management

We will introduce builds and build management by creating a classic build process and executing the build.

Demonstration Review



1

- Create a Classic Build

2

- Discuss Build Templates and use ASP.NET Core

3

- Review Hosted vs. Private build agents

4

- Review ASP.NET Core default template steps

5

- Review Variables and Options

7

- Execute the Build

8

- Review Build Results

Knowledge Check

Question #1: What is Continuous Integration?

A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day.

Question #2: What is Continuous Delivery?

A software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably release at any time.

Question #3: What is the importance of Feature Flags?

Help reduce integration debt, conduct A|B testing, implement features that can be turned on or off for new customer experiences.

Microsoft