Lecture 15: Dynamic Analysis and Testing 2

CS 5150, Spring 2026

Lecture goals

- Compare quality assurance methods
- Use continuous integration to automate finding bugs
- Leverage dynamic analysis tools to find bugs

Kinds of testing

- Styles
  - Exploratory
  - Smoke tests
  - Black box
  - White box
  - Fuzz testing
  - Dynamic analysis

  Can synthesize with boundary value analysis, coverage feedback
- Scopes
  - Unit tests
  - Integration tests
  - End-to-end tests

- Sizes
  - **Small**: fast, deterministic (in-process)
  - **Medium**: multi-process, allow blocking calls (single machine)
  - **Large**: Multi-node
- Purpose
  - Prevent reoccurrence of bugs (**regression tests**)
  - Prepare for release (**acceptance tests, beta testing**)
  - Ensure operating health (**self-tests**)

Flaky vs. brittle tests

Flaky

- Non-deterministic failures
  - Multi-process/multi-node infrastructure failures
  - Timeouts
  - Randomness
    - Always log seed
  - Concurrency
    - Difficult to reproduce

Brittle

- "High maintenance"
  - Leverage private functionality
  - Depend on private state
  - Assume behavior beyond the spec
    - e.g., checking interactions instead of state

Example (Brittle/Flaky Test)

```
@Test
public void testCacheUsage() {
MyService service = new MyService();
service.getData("key");
assertTrue(service.cache.containsKey("key")); // accessing internal field
}
```

Brittle!

```
expect(document.querySelector("div > span:nth-child(2)").textContent).toBe("Submit");
```

```
@Test
public void testAsyncProcessing() throws InterruptedException {
service.startTask();
Thread.sleep(1000);
assertTrue(service.isDone());
}
```

Flaky!

Aside: random numbers

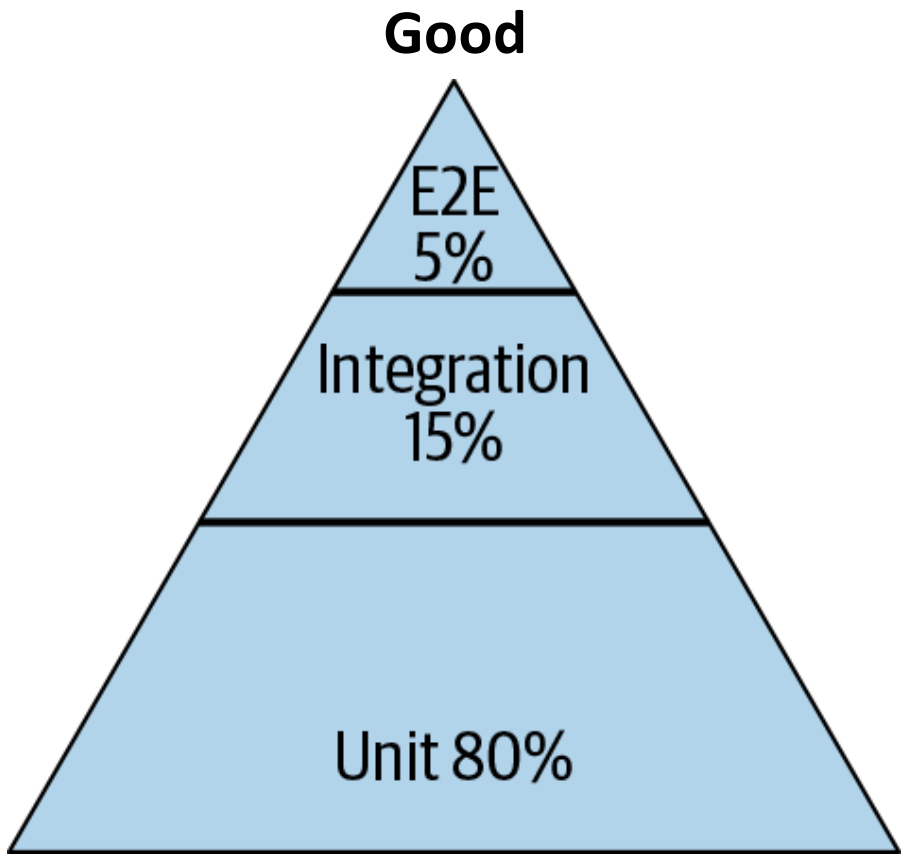
- In most settings, random numbers should be *deterministic*
  - Enables reproducibility, reduces test flakiness
  - Exceptions (in production): cryptography, gambling
- Recommended approach
  - Application starts with a specified global seed (and logs it)
  - Each component constructs a private RNG by combining global seed with unique instance name
  - Alternative for parallel computation: sequence queries, use RNG that can "fast forward" state
- Advantages
  - Results independent of amount of parallelism
  - Results do not change if "peripheral" components are added or removed

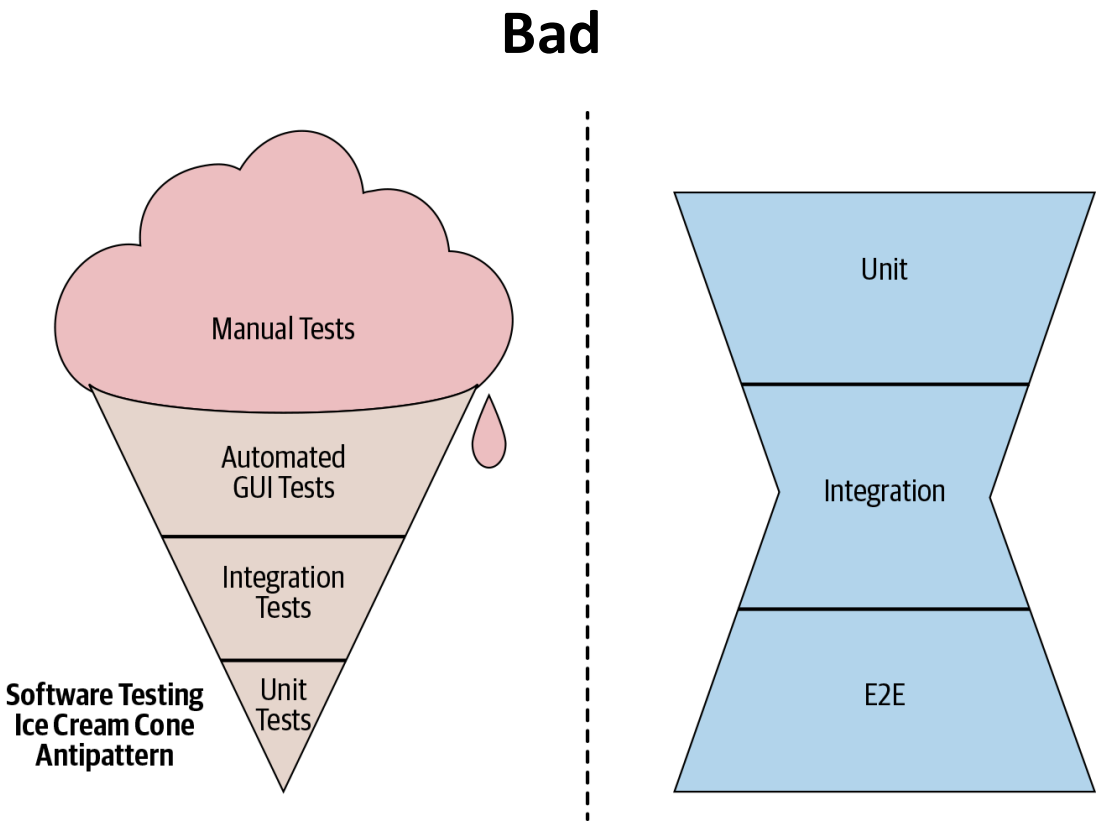
Example Tests

```
Random rand = new Random();
int value = rand.nextInt(100);
assertEquals(81, value); // non-deterministic
```

```
Random rand = new Random(42);
int value = rand.nextInt(100);
assertEquals(81, value); // deterministic
```

Test scope





From *Software Engineering at Google*, Ch. 11

Test scope

Small scope

- Limited coverage (per test)
  - But coverage is orthogonal
- May require awkward setup (dependency injection, mock objects)
- Can be written simultaneously with the code-under-test
- Easy to diagnose
  - Limited amount of code is executed
  - Easier to understand procedure and results
- Typically faster: can run more often

Large scope

- Extensive coverage (per test)
  - Much coverage is redundant
  - Most results are not checked (false sense of security)
- May be easier to set up than mid-scoped tests
  - But total configuration harder to reason about
- Depends on whole system
  - Bugs may not be found until later
- Difficult to diagnose
  - Slows down debugging when bugs are found
- Typically slower

Exploratory testing

- Applications
  - Developers check how existing code behaves
  - Developers "gut check" new code
  - Demonstrate functionality in a scenario of interest with complicated setup
  - QA testing (test behaviors developers often overlook)
- Tools
  - Application itself
  - REPL (JShell, iPython)
  - Dynamic analysis tools (valgrind, callgrind)
- Drawbacks
  - Not reproducible
    - Results may depend on unique context
    - Good habit to log all interactions
  - Good to think about expectations before running test, but if you can express what you expect, just write a unit test
  - Quality varies with tester
    - Can't measure coverage
- Other tools: **Selenium** for browsers

Unit tests

- Narrow scope (typically a single function or a single class)
- Focus on publicly-visible, fully-specified behavior
  - Check state, not process
- Write for clarity
  - Okay to be repetitive
  - Avoid new abstractions or logic

- Bad example:
  - When registering a new user, the system first generates a password, then tries to insert a new auth table row, throwing an exception if insertion failed (name already taken)
- Better example:
  - After registering a new user whose name is not taken, a new row will exist in the database with their username and password
  - If attempting to register a new user whose name is already taken, an exception is thrown

Behavior-driven development (BDD)

- Structuring tests around methods can make them brittle, hard to read
  - Try to test too many behaviors at once
- Better to structure tests around scenarios
- **Arrange-act-assert** format
  - "Given …, when …, then …"
  - Analogous to User Stories preamble

- "Given two accounts, the first of which has at least $100, when transferring $100 from the first to the second account, then both account balances should reflect the transfer"
- Test frameworks can help make tests self-documenting

BDD example

```
"A Stack" should "pop values in last-in-first-out order" in {
  val stack = new Stack[Int]
  stack.push(1)
  stack.push(2)
  stack.pop() should be (2)
  stack.pop() should be (1)
}

it should "throw NoSuchElementException if an empty stack is popped" in {
  val emptyStack = new Stack[Int]
  a [NoSuchElementException] should be thrownBy {
    emptyStack.pop()
  }
}
```

BDD example output

```
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack
  is popped
Run completed in 76 milliseconds.
Total number of tests run: 2
Suites: completed 1, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0,
  pending 0
All tests passed.
```

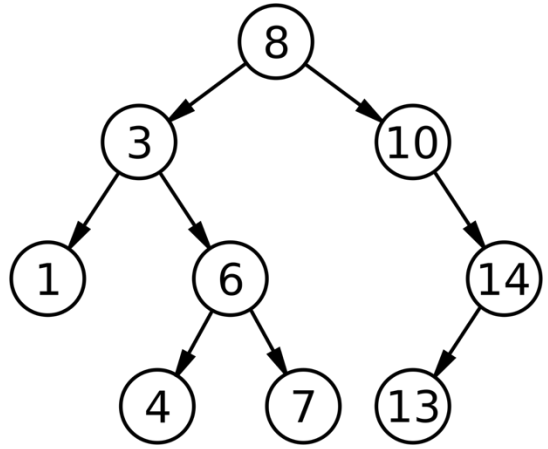
BDD example 2

```
info("As a TV set owner")
info("I want to be able to turn the TV on and off")
info("So I can watch TV when I want")
info("And save energy when I'm not watching TV")

Feature("TV power button") {
  Scenario("User presses power button when TV is off") {
    Given("a TV set that is switched off")
    val tv = new TVSet
    assert(!tv.isOn)
    When("the power button is pressed")
    tv.pressPowerButton()
    Then("the TV should switch on")
    assert(tv.isOn)
  }
  Scenario("User presses power button when TV is on") {
    Given("a TV set that is switched on")
    val tv = new TVSet
    tv.pressPowerButton()
    assert(tv.isOn)
    When("the power button is pressed")
    tv.pressPowerButton()
    Then("the TV should switch off")
    assert(!tv.isOn)
  }
}
```

Activity: Design tests using BDD

```
class BinarySearchTree {
private Node root; // root node
private int size; // number of nodes in the tree
static class Node {
private Node left; // left child
private Node right; // right child
}
public BinarySearchTree insert(int N);
public BinarySearchTree delete(int N);
public BinarySearchTree search(int N);
public BinarySearchTree succ(int N);
public BinarySearchTree pred(int N);
public int getSize();
```



Task: What kind of tests would you add?

S1: given empty tree, when you insert a node, the size should be 1

S2:

Test doubles

- How to write unit-scoped tests with complex dependencies?
  - Using external services makes tests "larger"
    - Depending on specialty hardware is very constraining
  - Can be difficult to get complex objects into appropriate state
  - Can be difficult to trigger a corner-case response (e.g. I/O errors)
- Examples of external dependencies?

- Options
  - Use real dependencies anyway (highest fidelity and coverage)
  - Use fakes & simulators (good option; requires investment)
  - Use stubbing/mocks (convenient, but dangerous)
    - Beware temptation of interaction testing
- Design for testing
  - Dependency injection: *pass in* dependencies instead of using Singletons or constructing your own

Stubbing and mocking frameworks

- Create subclasses of dependencies whose methods return values specified by the test
  - Frameworks like **Mockito** make this easy, even with static types
- Enables interaction testing
  - Checking whether code-under-test calls methods on dependencies in the way we expect

Example:

```
var userAuth = new UserAuthorizer(
    mockPermissionDb);
when(mockPermissionDb.getPermission(
    user1, ACCESS)).thenReturn(EMPTY);

userAuth.grantPermission(ACCESS);

verify(mockPermissionDb).addPermission(
    user1, ACCESS);
```

Dangers of stubbing & interaction testing

- Increases brittleness
  - When refactoring the real dependency, must also change everyone's stubs
- Reduced fidelity
- Decreases clarity
  - Pollutes tests for one class with a different class's API
- Depends on implementation details rather than on observable state
  - May be appropriate to test for "side effects"

Integration tests

- Broader scope
  - Check that multiple components interface correctly
  - Check behavior of subsystems
- Tend to be larger in size
  - SoA requires multiple processes
  - Non-trivial data, config can be slow
  - Aim for smallest test possible
    - Split pipelines into pairwise interactions
- Larger tests require non-trivial infrastructure, can be flaky
  - Fakes
  - Lightweight substitutions
    - In-memory databases
  - Hermetic services
    - Leverage virtualization to deploy isolated instances of service dependencies
  - Record/replay I/O
    - Trades flakiness for brittleness

Integration environments

- Production
  - Highest fidelity, esp. for load
  - Failures affect real users
  - Canarying: deploy to subset of production systems
    - E.g., internal users, early access
    - Can lead to version skew – incompatibility between concurrently-running components
  - Feature flags: Allow operators to quickly toggle between new and old implementation
- Staging
  - Ideally configured just like production
  - Potentially high infrastructure cost, limited availability
  - Often can't duplicate production load
  - Failures do not harm users
  - Can practice disaster recovery

Chaos engineering

- Originated at Netflix (**ChaosMonkey**)
- High-reliability, distributed systems **must tolerate failure**
- Recovery procedures are often not sufficiently rehearsed – painful, risky
- **Hypothesis**: What should happen *when X fails*?

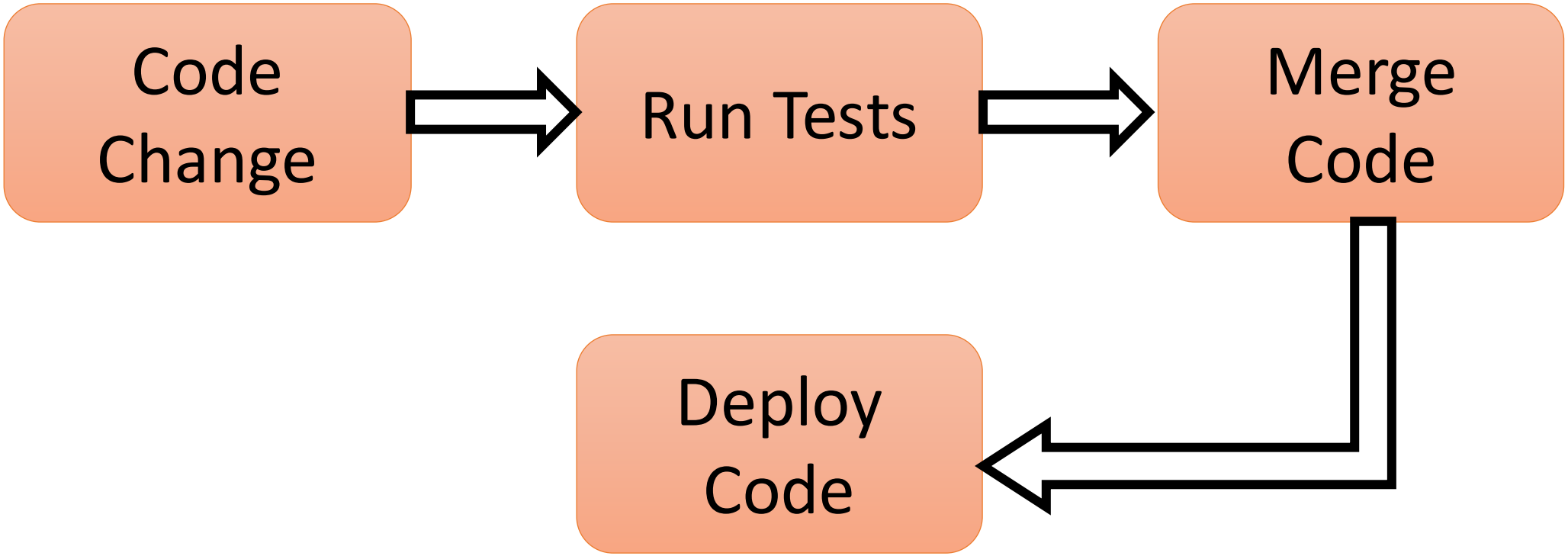
- Deliberately **inject failures** *in production environment*
  - Tests system resiliency under **realistic load**
  - Encourages recovery automation
  - **Failures**: delays, faults, traffic,

https://www.ibm.com/think/topics/chaos-engineering

How to automate finding bugs?

CI/CD Pipelines

- Continuous Integration/Continuous Delivery
- Catch mistakes before you push code



History of CI

- **1999**: Extreme Programming (XP) rule: Integrate Often
- **2000**: Martin Fowler posts “Continuous Integration” blog
- **2001**: First CI tool: Cruise Control
- **2005**: Hudson/Jenkins
- **2011**: Travis CI
- **2019**: Github Actions

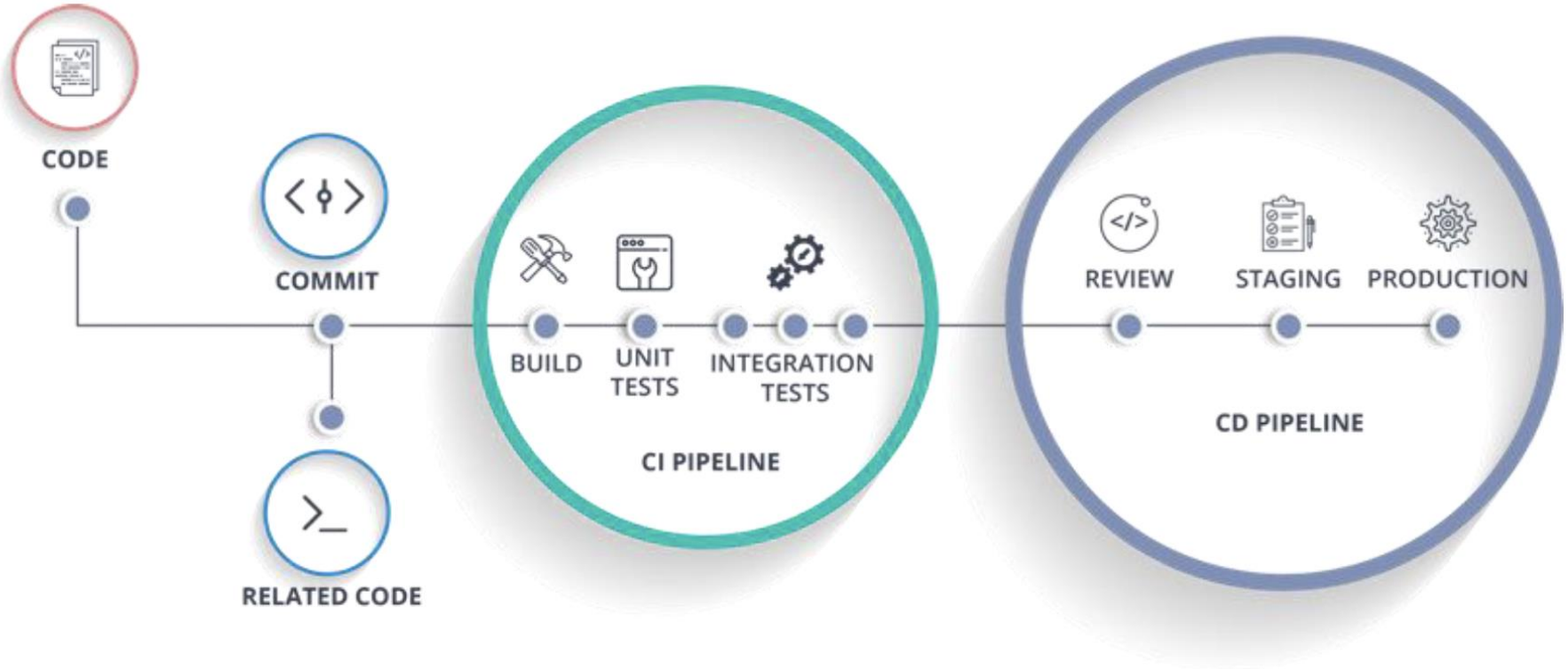
Jenkins

Travis CI

Continuous integration ("CI")

- Build and test whole systems regularly
  - Discover issues earlier
  - Reduce integration pain through automation and isolation of issues
  - Test beyond single developer's resources
  - Eliminate reliance on developers' discipline
  - Continuously monitor readiness of code
- Applies to both development and release
  - Continuous build+test
  - Continuous delivery

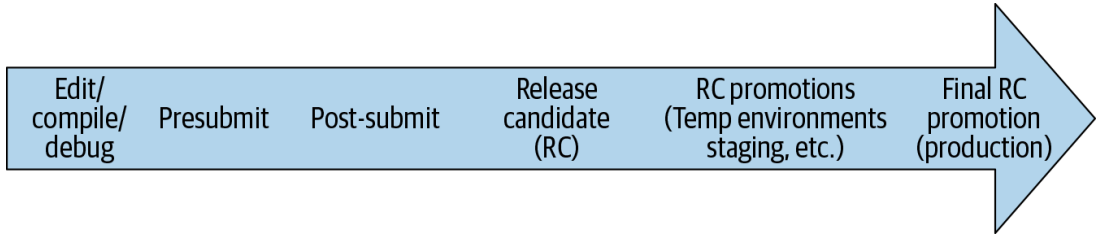
Example CI/CD Pipeline



Shift left

Heavyweight

Lightweight



Poll: pre-submit vs. post-submit tests

PollEv.com/cs5150sp26