Software Engineering 2025–26

Lecture 4: Quality Assurance & Reliable Delivery

Or: How do we deliver a quality product?

Prof. Robert Harle

Department of Computer Science and Technology
University of Cambridge

Easter Term

Part 1: Tools and coding

Managing the development environment and version control.

The Coordination Problem

Agile teams are fast. How do they stay in sync?

- Agile values *individuals and interactions*, but interactions generate conflict in the code.
- Two developers working on the same file will inevitably overwrite each other's changes without a technical coordinator.
- **Version Control Systems (VCS)** provide the single source of truth.
- **History:** Essential for auditing (who broke what?), compliance, and rolling back failures.

Centralized vs. Distributed VCS

Centralized (e.g., SVN, CVS)

- Single central server holds history.
- **Pros:** Simple model; easy to control access; "Single Source of Truth".
- **Cons:** Single point of failure; slow (every operation needs network).

Distributed (e.g., Git, Mercurial)

- Every dev has the full history locally.
- **Pros:** Fast (local ops); offline work; robust (no single point of failure).
- **Cons:** Steeper learning curve; local storage grows with history.

Enter Git

Distributed Version Control (Linus Torvalds, 2005)

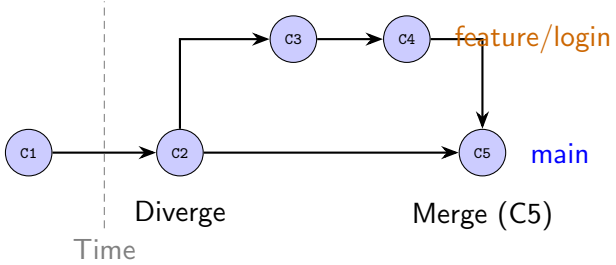
- **Commits:** A snapshot of the entire project. Each commit has a unique SHA-1 hash and a message explaining *why* the change happened.
- **Staging Area:** A "loading zone" where you pick which changes to include in the next commit.
- **Safety:** Git is nearly impossible to lose data from once it has been committed.
- **The Benefit:** You can experiment locally, fail, and instantly revert to a known good state.

Branching and Merging

Parallel Universes of Code

- **Main/Master Branch:** The stable, production-ready code.
- **Feature Branches:** Developers create a temporary branch for a specific task (e.g., `feature/login-fix`).
- **Merge:** Combining the changes from a feature branch back into main.
- **Merge Conflict:** When two branches changed the same line. Git pauses the merge and forces a human to resolve the logic manually.

Branching & Merging: Visualized



- **Isolation:** Work on `C3` and `C4` doesn't affect the stability of `main`.
- **Integration:** `C5` combines the work and resolves any conflicts.

Branching Strategies: How to stay in sync?

Git Flow (Feature-heavy)

- Long-lived branches (`develop`, `release`, `master`).
- Good for traditional releases (e.g., v1.0, v2.0).
- **Pain:** "Merge Hell" when merging long-lived branches.

Trunk-Based (Agile-heavy)

- Everyone works on `main` (the "trunk").
- Features are short-lived ($<$1 day).
- **Pros:** Fast integration; avoids merge conflicts.
- **Requirement:** Needs excellent automated tests to keep `main` stable.

The Pull Request (PR) Workflow

Social Coding

- A Pull Request is a request to merge your branch.
- It provides a dedicated space for **Code Review**.
- **Why Review?**
  - Knowledge sharing (others learn the new code).
  - Bug detection (four eyes are better than two).
  - Style consistency.
- This is how Agile teams maintain high standards without a central "Design Authority."

Bug Tracking & The Bug Lifecycle

Managing the Chaos with Issue Trackers (Jira, GitHub)

- **The Lifecycle:**
  1. **New:** Reported but not yet reviewed.
  2. **Triaged:** Confirmed as a bug and prioritized.
  3. **In Progress:** An engineer is actively working on it.
  4. **Resolved:** Code is written and passing tests.
  5. **Verified:** QA/Reporter confirms the fix works.
  6. **Closed:** Bug is dead and buried.
- **Traceability:** Linking commits to Issue IDs (e.g., "Fixes #42") connects the "How" to the "Why".

Coding Standards & Maintenance

Uniformity over Individualism

- **Maintenance:** 80% of a software's cost is maintenance. Uniform code is easier to read, meaning faster bug fixes.
- **Standards:** Agreed-upon rules for naming, indentation, and structure.
- **Linters (ESLint, Pylint):** Automated tools that scan code for style violations.
- **Why it matters:**
  - Reduces cognitive load (you don't waste time deciphering odd styles).
  - Prevents "nitpicking" in reviews (let the machine do the shouting).
  - Enables automated refactoring tools to work safely.

Part 2: Testing

Verification, validation, and automated quality assurance.

Why do we test?

Confidence and Documentation

- **Confidence:** Knowing you didn't break old features (Regressions).
- **Documentation:** Tests show how a function is *actually* supposed to be used (Executable Specs).
- **Refactoring:** You cannot safely clean up "spaghetti code" without a test suite to verify that the external behavior remains identical.
- **Design:** Writing tests often reveals that your code is too complex to test, forcing better architectural choices.

Automated vs. Manual Testing

Manual Testing

- **Pros:** Exploratory; catches visual glitches; uses human intuition.
- **Cons:** Slow; unrepeatable; error-prone; doesn't scale as the app grows.

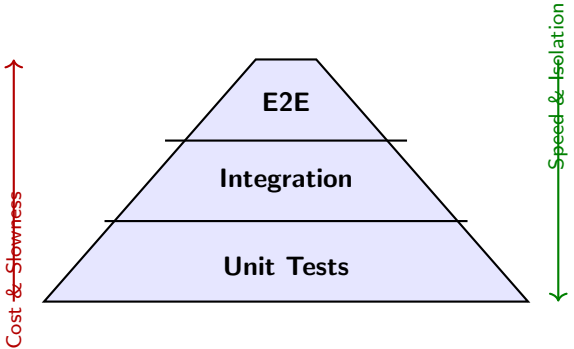
Automated Testing

- **Pros:** Fast; repeatable; runs on every commit; scales infinitely.
- **Cons:** High initial setup cost; tests require maintenance; can't "see" if UI is ugly.

The Testing Pyramid

Balancing Speed and Confidence

- **Base: Unit Tests (Many)**
  - *Example:* Testing that `sum(2, 2)` returns 4.
- **Middle: Integration Tests (Some)**
  - *Example:* Verifying that the User Service can save a record to a real PostgreSQL database.
- **Top: End-to-End Tests (Few)**
  - *Example:* Selenium script opens Chrome, logs in as 'Bob'.



Example: A Simple Unit Test

The Code (`maths.py`)

```python
def add(a, b):
    return a + b

def test_add():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
```

The Console

```
$ pytest maths.py
================ test session starts ================
platform linux -- Python 3.10.12
collected 1 item

maths.py .                                      [100%]

================ 1 passed in 0.01s =================
```

Unit Testing & Mocking

Isolation is Key

- **Unit Tests:** Test a single function in isolation. Must be fast (<10ms).
- **Mocking:** If your function calls a Payment API, don't call the real API (too slow/expensive).
- **Mocks/Stubs:** Fake objects that "stand in" for dependencies.
  - *Stub:* Returns a hardcoded value (e.g., always returns "Success").
  - *Mock:* Verifies interaction (e.g., "Ensure the email service was called exactly once").

Example: Mocking in Python

```python
from unittest.mock import MagicMock

# The function we want to test
def process_payment(payment_gateway, amount):
    if payment_gateway.charge(amount):
        return "Success"
    return "Failed"

# Our Unit Test
def test_process_payment_success():
    # 1. Create a Mock object
    mock_gateway = MagicMock()
    # 2. Configure 'stubbed' value
    mock_gateway.charge.return_value = True

    # 3. Call with the mock
    result = process_payment(mock_gateway, 100)

    # 4. Assertions
    assert result == "Success"
    # 5. Verify the interaction
    mock_gateway.charge.assert_called_once_with(100)
```

What is `MagicMock`?

- **Dynamic:** It automatically creates attributes and methods as you access them.
- **Stubbing:** Use `.return_value` to dictate what a method returns.
- **Spying:** It records every call made to it, including arguments.
- **Verification:** Use `.assert_called_*` to ensure the code under test interacted with the dependency correctly.

Code Coverage

How much of the code is actually tested?

- **Definition:** A metric that measures the percentage of your source code executed when your test suite runs.
- **Why it matters:**
  - Identifies "dead" or unreachable code.
  - Highlights complex logic paths (e.g., error handling) that you forgot to test.
  - Provides a baseline for team confidence.
- **The Trap:** 100% coverage means every line ran, but it **does not** mean every logical edge case is correct. (Coverage $\neq$ Quality).
- **Tools:** `pytest-cov` (Python), `Istanbul` (JS), `JaCoCo` (Java).

The Coverage Trap: 100% $\neq$ Correct

The Code (`utils.py`)

```
def average(scores):
    # Sums and divides
    return sum(scores) / len(scores)

def test_average():
    # Only one test case
    assert average([10, 20]) == 15
```

- **Coverage:** 100% (every line in `average` was executed by the test).
- **The Bug:** What if `scores` is an empty list `[]`?
- **ZeroDivisionError:** The code crashes, but your tests passed with 100% coverage!
- **Lesson:** Coverage tracks which lines ran, not which **logical edge cases** were verified.

Mutation Testing: "Test your Tests"

Is your test suite actually effective?

- **The Problem:** High code coverage only means the code was *executed*, not that it was *verified*.
- **The Concept:** Automatically inject small bugs ("mutants") into your source code:
  - Change $>$ to $>=$.
  - Change + to −.
  - Delete a function call.
- **The Goal:** If your tests still pass, the mutant "survived" $\rightarrow$ **Your tests are weak.**
- **Success:** A strong test suite should "kill" all mutants by failing when they are present.

Test-Driven Development (TDD)

Write the Test First TDD enforces a strict Red-Green-Refactor loop:

1. **Red:** Write a failing test for a new feature.
2. **Green:** Write the minimum code to make it pass (no matter how ugly).
3. **Refactor:** Clean up the code. The test ensures you don't break the feature while cleaning.

Result: 100% test coverage by default and simpler code.

Continuous Integration (CI)

Goal: Find bugs as soon as they are written.

- **The Rule:** Developers merge their code into the main branch at least once a day.
- **The Automation:** Every merge triggers an automated build and runs the full test suite.
- **Fast Feedback:** If a test fails, the developer is notified within minutes ("Breaking the Build").
- **Benefit:** Eliminates "Merge Hell" where developers spend days resolving conflicts from long-lived branches.

Continuous Delivery (CDE)

Goal: Any version of the code is ready to ship at any time.

- **The Buffer:** Automated tests pass, and the code is automatically deployed to a **Staging** environment.
- **Human in the Loop:** Moving code to the live Production environment is a **manual business decision**.
- **The Button:** Releasing a feature becomes a "push-button" non-event rather than a high-stress midnight ceremony.
- **Benefit:** Decouples technical readiness from business timing.

Continuous Deployment (CDP)

Goal: Fully automated path from code to customer.

- **No Human Oversight:** If the code passes all automated tests in the pipeline, it is **automatically deployed** to real users.
- **High Bar:** Requires extreme confidence in your automated test suite and monitoring tools.
- **Cycle Time:** Shrinks the time between "Commit" and "Live User" to minutes.
- **Risk Management:** Relies on automated rollbacks if the system detects an error spike after deployment.

CI, CDE, and CDP vs. Manual Deployment

Manual Deployment

- **Pros:** Human oversight; feels safer for small, low-risk apps.
- **Cons:** "It works on my machine" syndrome; slow; high risk of human error; hard to roll back.

CI, CDE, and CDP Pipelines

- **Pros:** Predictable; repeatable; fast; automated rollbacks.
- **Cons:** Complex to set up; requires high trust in your test suite.

The Anatomy of a CI/CD Pipeline

- **Fast Feedback:** Developers know within minutes if they broke the build.
- **Quality Gates:** Every step must pass for the code to advance.
- **Artifacts:** The "Build" step creates a versioned package (e.g., a Docker image) that is used in all subsequent environments.

The Promotion Pipeline: Environments

Code moves through stages of increasing "Production-likeness"

- **Development (Local):** The wild west. Untested, experimental.
- **CI / Test:** Clean, ephemeral environments for automated tests.
- **Staging / UAT:**
  - Identical to Production (same DB size, same OS, same scale).
  - Used for **User Acceptance Testing (UAT)** and smoke tests.
  - The "Final Rehearsal" before the live show.
- **Production:** The live system. Real users, real money, real consequences.

Google's Release Strategy: "xFood"

Progressive internal testing before public release

- **Fishfood:** Testing with the **immediate developers** (20–30 people).
  - Highly unstable; low bar for accepting changes; fast iteration.
- **Teamfood:** Testing with the **wider team/department** (50–100 people).
  - More stable; uses "friendly" testers who understand the context.
- **Dogfood:** Open to **all employees** (hundreds/thousands).
  - Testing the next **Release Candidate**.
  - Provides a signal closest to real users before the public launch.

Goal: Catch "annoyance" bugs and UX friction in increasing concentric circles.

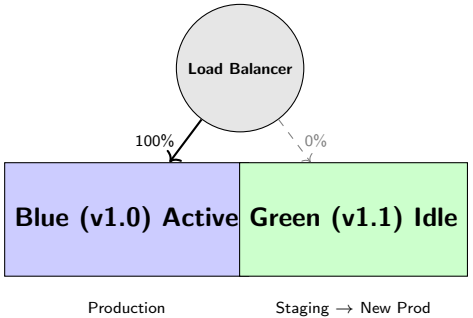
Deployment Strategies

- **Blue-Green:** Two identical environments. Switch traffic from Blue (old) to Green (new). Zero downtime.
- **Canary Releases:** Deploy to 5% of users. If errors spike, auto-rollback. If stable, roll out to 100%.
- **Feature Flags:** Code is deployed but hidden behind a toggle. Decouples "Deployment" (technical) from "Release" (marketing).

Deployment Strategies: Blue-Green

Blue-Green Deployment

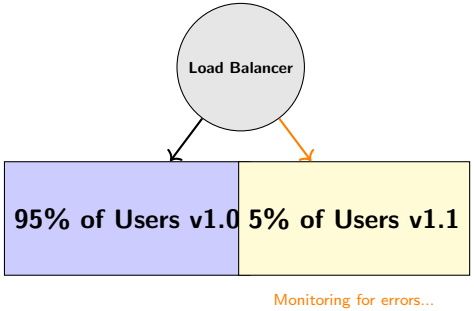
- Two identical environments.
- **Blue:** Running v1.0.
- **Green:** Deploy v1.1.
- Switch the Router/LB to Green.
- **Benefit:** Instant rollback (just switch back).



Deployment Strategies: Canary

Canary Deployment

- Deploy v1.1 to a small subset of servers.
- Route a tiny % of traffic to them.
- Monitor metrics (errors, latency).
- If healthy, expand to 100%.



A/B Testing: Measuring Business Value

Is the new feature actually better for the user?

- **Hypothesis:** "Changing the 'Buy' button to red will increase sales."
- **Method:** Split users into two groups:
  - **Group A (Control):** Sees the current version (Blue button).
  - **Group B (Treatment):** Sees the new version (Red button).
- **Measurement:** Use statistical analysis to see if the difference in behavior is significant.
- **Key Point:** A/B Testing is for **Product/Business** validation; Canary is for **Technical** validation.

Incident Response & Reliability

- **SLO (Service Level Objective):** Target for reliability (e.g., 99.9% uptime).
- **Error Budget:** The 0.1% "allowed" failure. If you have budget left, you can deploy risky features. If it's gone, freeze all releases.
- **Blameless Post-mortems:** "Why did the system allow the human to make this mistake?" rather than "Who do we fire?"

What is DevOps?

Development + Operations

- **Origin:** A portmanteau of *Development* and *Operations*.
- **History:** Coined by **Patrick Debois** in 2009 (the first "DevOpsDays" conference) to address the toxic silos between software creators and software deployers.
- **The Conflict:** Developers were rewarded for *change* (shipping features); Operations were rewarded for *stability* (uptime).
- **The Solution:** Merge the teams. If you build the feature, you are also responsible for its stability in production ("You build it, you run it").

Part 3: Cloud systems

Infrastructure, platforms, and scaling in the modern web.

The Shift to Cloud

Rent, Don't Buy

- **On-Premises:** High CapEx (Capital Expenditure). You own the hardware, power, and cooling.
- **Cloud:** OpEx (Operating Expenditure). Pay-as-you-go. Scale from 1 to 1,000,000 users in seconds.
- **Elasticity:** The system grows and shrinks automatically based on demand.

IaaS: Infrastructure as a Service

Raw Compute Resources

- You rent virtual machines, storage, and networks.
- **Example: AWS EC2**, Google Compute Engine, Azure VMs.
- **Pros:** Maximum control; you choose the OS and runtime.
- **Cons:** High management overhead (you must patch the OS and handle security updates).

PaaS: Platform as a Service

Focused on the App

- You upload your code; the provider handles the OS, runtime, and scaling.
- **Example: Heroku**, AWS Elastic Beanstalk, Google App Engine.
- **Pros:** Fast time-to-market; focus entirely on features, not servers.
- **Cons:** "Vendor Lock-in"; less control over the underlying environment.

SaaS: Software as a Service

Consuming the Product

- Fully functional applications delivered via the web.
- **Example: Salesforce**, **Slack**, Gmail, Microsoft 365.
- **Pros:** Zero maintenance; accessible from anywhere; instant value.
- **Cons:** Zero control over features or data storage; subscription costs.

Shared Responsibility Model

Who is at fault when things go wrong?

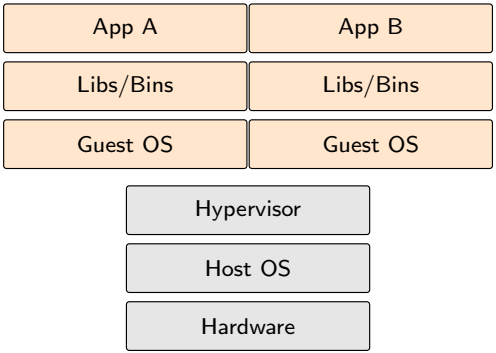
- **Provider:** Responsible for security **OF** the cloud (Physical data centers, hardware, basic networking).
- **Customer:** Responsible for security **IN** the cloud (Your application code, your data, your firewall settings).
- *Analogy:* The landlord provides the lock (Provider), but you must remember to turn the key (Customer).

Containers (Docker)

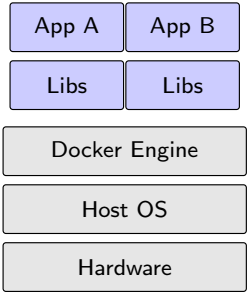
Packaging the Environment

- **The Problem:** "It works on my machine but not in production."
- **Docker:** Packages code and **all** its dependencies (libraries, config, OS files) into a single image.
- **Containers vs VMs:** Containers share the host OS kernel. They are lightweight, boot in milliseconds, and use 1/10th the resources of a VM.

VMs vs. Containers: Architecture



Virtual Machines

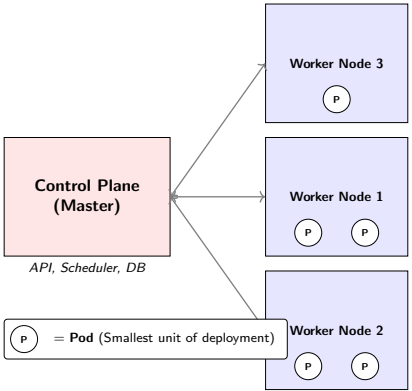


Containers

Container Orchestration (Kubernetes)

- Docker runs a single container. But how do you manage 1,000 containers?
- **Kubernetes (K8s):**
  - **Self-healing:** Restarts containers that fail.
  - **Auto-scaling:** Adds more containers when traffic spikes.
  - **Load Balancing:** Distributes traffic across healthy containers.
- Originally built by Google to run their global infrastructure.

Kubernetes: Cluster Architecture



Cluster: A single logical computer made of many physical ones.

Progressive Delivery & Feature Flags

Decoupling Deployment from Release

- **Deployment:** Moving code to production servers.
- **Release:** Making the feature visible to the user.
- **Feature Flags (Toggles):**
  - Wrap new code in an `if` statement controlled by a database toggle.
  - Turn features ON for developers, then Beta testers, then everyone.
  - Allows "emergency shutoff" without a code rollback.

Observability: The Feedback Loop

How do we know if a deployment is healthy?

- **Metrics:** Quantitative data (CPU usage, 500-error counts).
- **Logs:** Qualitative records (textual history of what happened).
- **Traces:** Following a single user request across multiple microservices.
- **Dashboards:** Visualizing health in real-time (e.g., Grafana).

You cannot manage what you cannot measure.

Summary of Lecture 4

- **Coordination:** Distributed VCS (Git) and PRs enable parallel work and peer review.
- **Testing:** Use the Testing Pyramid to balance cost and speed.
- **DevOps:** CI, CDE, and CDP automates the path to production.
- **Deployment:** Use Blue/Green or Canary to minimize risk; use A/B testing to measure business value.
- **Progressive Delivery:** Decouple deployment from release using Feature Flags.
- **Portability:** Containers (Docker) and Orchestration (Kubernetes) ensure code runs identically at scale.