

**MASTERING SOFTWARE
DEVELOPMENT**

A Guide For Early Career Engineers



Balwinder Sodhi

Software Testing and Quality Assurance

Balwinder Sodhi

What Is Software Testing & QA?

- **Software Testing**

- A set of activities that evaluate software behavior and verify that it meets requirements.
- Focuses on finding defects, validating functionality, and ensuring reliability.

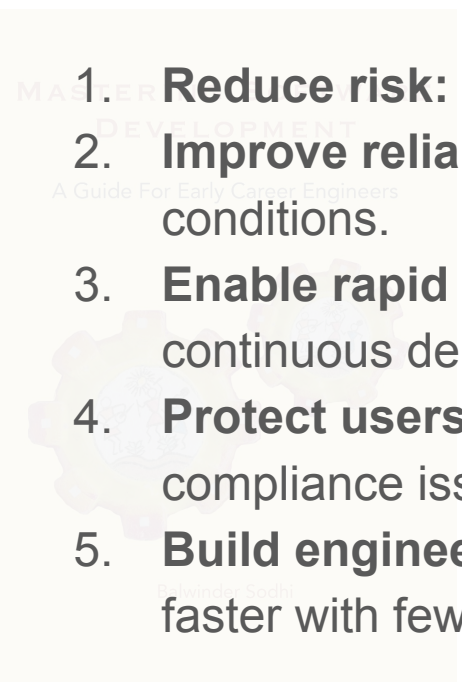
- **Quality Assurance (QA)**

- Broader process-oriented discipline ensuring that quality is built into every phase of development.
- Includes standards, best practices, process audits, documentation, and continuous improvement.

- **Key Difference**

- Testing finds defects. QA prevents them.

Why Testing & QA Matter

- 
1. **Reduce risk:** Catch defects early before they reach users or production.
 2. **Improve reliability:** Ensure predictable and stable behavior under real-world conditions.
 3. **Enable rapid development:** Automated tests allow safe refactoring and continuous delivery.
 4. **Protect users & business:** Avoid failures, downtime, security breaches, and compliance issues.
 5. **Build engineering confidence:** Good testing enables engineers to ship faster with fewer regressions.

Where Testing Fits in the SDLC

Testing is not a final phase — it spans the entire lifecycle.

- Requirements: Review for clarity, testability, ambiguity
- Design: Architecture reviews, threat modeling
- Development: Unit tests, static analysis, component tests
- Integration: API, DB, service-level testing
- System: Functional, UX, security, performance testing
- Post-release: Monitoring, observability, incident analysis (shift-right)
- Modern practice: Shift-left testing — test early and test continuously.

Testing vs. Quality Assurance

- QA (process-level)
 - Activities: process audits, release gating, metrics (MTTR, escaped defects), standards (code review checklist), supplier/vendor QA.
 - Deliverables: QA policy, release readiness report, root-cause analyses.
- Testing (product-level)
 - Activities: designing/ executing tests, defect logging, test automation suites.
 - Deliverables: test plans, test cases, test reports, regression suites.
- Practical tip: QA owns a definition of done that mandates what tests must pass before a release (unit coverage thresholds, passing security scan, smoke tests).

Goals and Objectives

- Primary measurable goals:
 - defect density
 - code coverage (but not blindly)
 - requirements coverage
 - mean time to detect.
- Secondary but important:
 - maintainability of tests
 - flakiness rate (target <2%)
 - execution time budget (e.g., < 10 min for fast pipeline).

Balwinder Sodhi

Shift-Left Testing

- Concrete practices:

- TDD/BDD, contract-first APIs
- pre-commit hooks (linting, small tests)
- feature-branch smoke tests.

- Tooling examples:

- pre-commit, Danger, git hooks
- ESLint, black/ruff/bandit, SonarQube.

- Metric: measure defects found per phase; aim to find >70% of defects before QA handoff.

Balwinder Sodhi

MASTERING SOFTWARE
DEVELOPMENT

A Guide For Early Career Engineers



Balwinder Sodhi

Different Types of Testing

Overview of Testing Techniques & Methodologies

- Based on Knowledge of Internal Implementation
 - Black Box Testing
 - White Box Testing
 - Gray Box Testing
- Based on Code Execution
 - Static Testing (no execution)
 - Dynamic Testing (executed code)
- Each technique is suited for different phases of development and different types of tests.

Balwinder Sodhi

Black Box Testing (Functional Testing)

- Testing the system based on inputs and expected outputs with no knowledge of internal logic or code structure.
- What engineers validate
 - Functional correctness
 - Requirements coverage
 - User workflows
 - API behavior
 - Error handling
- Common techniques
 - Equivalence Partitioning
 - Boundary Value Analysis
 - Decision Table Testing
 - State Transition Testing
- Where used
 - System testing
 - UAT (User Acceptance Testing)
 - API and UI testing
- Example:
Function accepts age 18–65.
Test cases: 17, 18, 65, 66 (boundaries)

White Box Testing (Structural Testing)

- Testing based on knowledge of internal implementation—logic, branches, data flows, loops, and code paths.

- Focus areas

- Branch, Path, Condition coverage
- Error handling paths
- Loop boundaries

```
def safe_div(a, b):  
    if b == 0:  
        return None  
    return a / b
```

- White-box test ensures:

- $b == 0$ branch is hit
- $b != 0$ path is hit
- Result is correct for multiple inputs

- Where used

- Unit testing
- Security testing (white-box pen tests)
- Code-level validation

- Tools

- Coverage.py, JaCoCo, gcov

Gray Box Testing

- Testing with partial knowledge of internal architecture (DB schemas, API contracts, high-level code flow).
- Why it's useful
 - Combines strengths of black-box + white-box
 - Helps design better integration tests
 - Useful in microservices where testers understand service boundaries

- Examples

- Knowing API schema and validating error conditions
- Knowing DB structure and asserting side effects
- Testing caching behavior with internal knowledge of how cache keys are built

```
# After API call, gray-box test validates DB row update
db_user = db.fetch("SELECT last_login FROM users WHERE
id=1")
assert db_user.last_login is not None
```

Static Testing (Without Executing Code)

- Reviewing code, design documents, configurations, and requirements *without running* the application.

- Purpose

- Detect issues early (shift-left)
- Prevent defects before implementation or during coding
- Improve maintainability and design quality

- Static Testing Activities

- Code reviews
- Design reviews
- Linting and SAST (Static Application Security Testing)
- Requirements inspections
- Threat modeling

- Examples

- ESLint flagging unused variables
- Bandit detecting `subprocess(shell=True)`
- Threat model identifying missing authentication paths

Dynamic Testing (Executing Code)

- Testing software by running it and observing behavior.
- Used to validate
 - Runtime errors
 - Functional correctness
 - Performance & security behavior
 - Integration between components
- Example:

```
resp = client.get("/users")  
assert resp.status_code == 200
```

- Dynamic Testing Activities
 - Unit testing
 - Integration testing
 - E2E testing
 - Performance (load, stress) tests
 - DAST (Dynamic Application Security Testing)

Comparing Static vs. Dynamic Testing

<i>Aspect</i>	<i>Static Testing</i>	<i>Dynamic Testing</i>
Execution	No	Yes
Stage	Early (requirements, design, coding)	Later (integration, staging)
Detects	Logic errors, code smells, vulnerabilities	Functional issues, runtime defects
Examples	Code review, linting, SAST	Unit tests, DAST, E2E tests
Cost	Low	Medium/High
Accuracy	Can produce false positives	Highly accurate but may miss corner cases

When to Use Each Technique

- Black Box

- Testing functionality
- End-to-end validation
- API + UI testing

- White Box

- Unit tests
- Code coverage targets
- Security-sensitive components

- Gray Box

- Integration testing
- Microservices and API workflows

- Static

- Code review
- CI linting/security scanning
- Early defect prevention

- Dynamic

- Runtime testing
- Performance testing
- Full-stack system tests

- Best practice:

- Combine these techniques for maximal coverage and confidence.

MASTERING SOFTWARE
DEVELOPMENT

A Guide For Early Career Engineers



Balwinder Sodhi

Test Planning & Documentation

Purpose of Test Planning

- Ensures testing efforts are aligned with product goals and risks.
- Defines what will be tested, how it will be tested, and what resources are needed.
- Provides structure and predictability during development and releases.

Why Documentation Matters

- Improves communication across engineering, QA, and product.
- Enables repeatable, consistent testing processes.
- Supports compliance, audits, and knowledge transfer.

Test Plans: What They Contain

- A Test Plan outlines the testing strategy for a feature, release, or product.
- Key Components
 - Scope: what is in/out of testing (features, APIs, browsers/devices).
 - Objectives: what the test effort aims to validate (security, performance, correctness).
 - Test Strategy: types of tests used (unit, integration, performance, usability).
 - Environment Setup: staging URLs, test data, mock services.
 - Resources & Responsibilities: who designs, automates, and executes tests.
 - Risks & Mitigation: vague requirements, flaky dependencies, time constraints.

Example:

Scope: Test checkout flow including payment gateway.

Out of Scope: Fraud engine and loyalty program backend.

Risks: Third-party sandbox downtime → mitigate by adding mock gateway.

Test Scenarios

High-level user or system workflows to be validated.

Examples:

- “User signs up and completes onboarding.”
- “Admin updates user permissions.”
- “Payment API returns failure.”

Used to confirm complete business flows, not specific input/output pairs.

Balwinder Sodhi

Test Cases

- They include specific inputs, steps, and expected results.

- Typical Fields

- ID, Title
- Preconditions
- Steps to Execute
- Test Data
- Expected Result
- Actual Result & Status

Balwinder Sodhi

ID: TC-LOGIN-002

Title: Login fails with incorrect password

Steps:

1. Navigate to /login
2. Enter `email:` user@example.com
3. Enter `password:` wrongpass
4. Click "Login"

Expected:

Error message "Invalid credentials" displayed

E
X
A
M
P
L
E

Traceability Matrix: Ensuring Coverage


- A Traceability Matrix (RTM) maps requirements → test scenarios → test cases → defects.
- Purpose
 - Ensures full requirements coverage.
 - Shows which requirements have incomplete or failing tests.
 - Helps during audits and compliance checks.
- Typical Columns
 - Requirement ID
 - Requirement Description
 - Associated Test Scenarios
 - Test Cases
 - Status (Pass/Fail/Not Tested)

Example RTM

Requirement	Scenario	Test Case	Status
R1: User Login	S1: Login Flow	TC-LOGIN-001 (valid)	Pass
R1: User Login	S1: Login Flow	TC-LOGIN-002 (invalid)	Pass
R2: Add to Cart	S2: Cart Flow	TC-CART-004	Fail

- How Engineers Use It
 - Identify missing tests before release
 - Track regressions
 - Validate that all acceptance criteria are tested

Overview of Test Levels

- 
- Unit Testing
 - Integration Testing
 - System Testing
 - Specialized Testing Types (security, performance, usability, accessibility)

Unit Testing

Context: tests at function/class level. Fast and isolated.

● Principles

- Single responsibility per test.
- Arrange-Act-Assert structure.
- Use test doubles (mocks/stubs) only for external dependencies.
- Test-first (TDD) where beneficial.


● Testing tips

- Keep unit tests isolated (no DB/network). Use mocking libraries: unittest.mock, Mockito, sinon, jest.mock.
- Keep them fast — aim for milliseconds.

Python example

MAS
A Gu

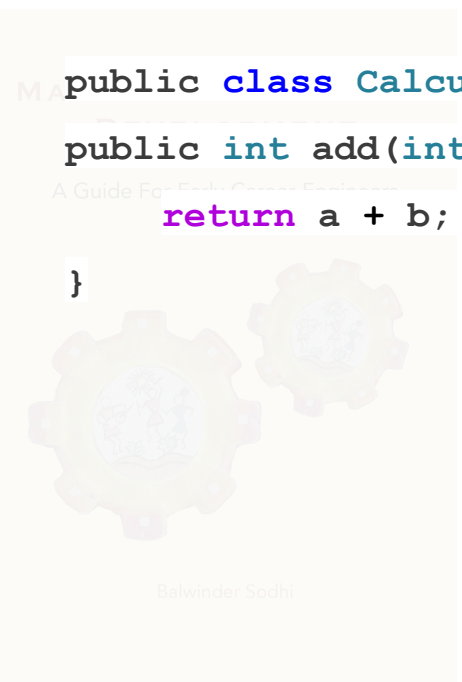
```
# math_utils.py  
def add(a, b):  
    return a + b
```



Balwinder Sodhi

```
# tests/test_math_utils.py  
import pytest  
from math_utils import add  
  
def test_add_positive():  
    assert add(2,3) == 5  
  
@pytest.mark.parametrize("a,b,expected", [  
    (0,0,0), (-1,1,0), (1.5, 2.5, 4.0)  
])  
def test_add_various(a,b,expected):  
    assert add(a,b) == expected
```

Example: Java + JUnit 5 (class under test)



```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b; }  
}
```

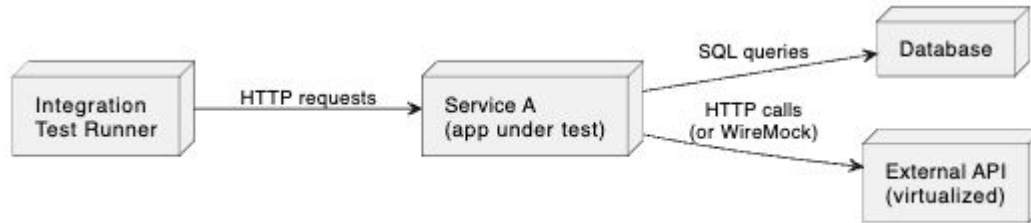
```
import org.junit.jupiter.api.Test;  
import static  
    org.junit.jupiter.api.Assertions.*;  
  
class CalculatorTest {  
    @Test void addTwoNumbers() {  
        Calculator c = new Calculator();  
        assertEquals(5, c.add(2,3));  
    }  
}
```

Integration Testing

Context: test interactions between components (APIs, DB, message queues).
May be slower and use real or containerized dependencies.

Approaches

- Top-down / bottom-up: use stubs/drivers accordingly.
- Service virtualization: mock external services (e.g., WireMock, MockServer).
- Contract testing: consumer-driven contracts (Pact) to verify API contracts.



Example: Python — requests-based API integration test

```
def start_app(pg_conn_str):  
    # Example: launch app in test mode pointing to pg_conn_str  
    # Could use subprocess to start, or app in same process  
    pass  
  
def test_create_user():  
    with PostgresContainer("postgres:15") as postgres:  
        db_url = postgres.get_connection_url()  
        # Start app bound to db_url (example skipped)  
        # wait for app to be ready...  
        resp = requests.post("http://localhost:5000/users",  
                             json={"name": "Alice"})  
        assert resp.status_code == 201  
        data = resp.json()  
        assert data["name"] == "Alice"
```

Example: Java + Spring Boot integration with Testcontainers

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Testcontainers
public class UserIntegrationTest {
    @Container
    public static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:15")
        .withDatabaseName("testdb").withUsername("sa").withPassword("sa");

    @LocalServerPort
    private int port;

    @Test
    void createUserViaApi() {
        RestTemplate rt = new RestTemplate();
        Map<String,String> body = Map.of("name", "Bob");
        ResponseEntity<User> r = rt.postForEntity("http://localhost:"
            + port + "/users", body, User.class);
        assertEquals(HttpStatus.CREATED, r.getStatusCode());
        assertEquals("Bob", r.getBody().getName());
    }
}
```

System Testing

- **Context:** full E2E in production-like environment.
- **Focus:** functional correctness, full data flow, integrations, user journeys.
- **Environments:** staging with production-like config, separate test accounts, production-like data or obfuscated copies.
- **E2E tools:** Playwright, Selenium, Cypress, Robot Framework.

Balwinder Sodhi

Example (Playwright JS)

```
// tests/e2e.spec.js
const { test, expect } = require('@playwright/test');

test('user signup flow', async ({ page }) => {
  await page.goto('https://staging.example.com/');
  await page.click('text=Sign up');
  await page.fill('#email', 'qa+1@example.com');
  await page.fill('#password', 'P@ssword123');
  await page.click('button[type="submit"]');
  await expect(page.locator('text=Welcome')).toBeVisible();
});
```


Performance Testing: Why It Matters

- Objective: Validate how the system behaves under expected and extreme conditions.
- Why engineers care
 - Ensures the system can handle real-world usage without failures.
 - Identifies performance bottlenecks early (CPU hotspots, DB locks, slow endpoints).
 - Provides confidence before scaling or deployment.
 - Supports SLO/SLAs: latency, throughput, error rates.
- Key dimensions measured
 - Latency: p50, p90, p95, p99
 - Throughput: requests/sec, jobs/sec
 - Resource usage: CPU, memory, disk, network
 - Error behaviour: 4xx/5xx rates
 - Scalability indicators: linearity of throughput vs. load

Types of Performance Testing

1. Load Testing: Evaluate performance under expected peak user or transaction load.
2. Stress Testing: Determine system behaviour beyond peak load—until the system breaks.
3. Endurance (Soak) Testing: Validate long-duration stability (memory leaks, resource starvation, slow degradation).

Balwinder Sodhi

Load Testing 1/2

Purpose: Understand whether the system meets performance expectations under normal and peak load.

A Guide For Early Career Engineers

A good load test includes:

- Traffic model based on analytics (e.g., 300 req/sec peak).
- Mix of operations (login 10%, checkout 2%, browse 70%, etc.).
- Realistic concurrency patterns (spikes, ramp-ups).
- Proper test data design (avoid caching effects skewing results).

Balwinder Sodhi

Load Testing 2/2

- What you validate:

- Response time stays below SLO (e.g., $p95 < 250\text{ms}$).
- Throughput scales predictably.
- No unexpected errors (timeouts, 5xx).
- System auto-scales correctly (if cloud based).

- Common bottlenecks found with load tests:

- DB connection pool saturation.
- Thread starvation in web server.
- Slow external service dependencies.
- Inefficient caching or high GC pause time.

Stress Testing 1/2

Purpose: Find the system's breaking point and behaviour under extreme pressure.

Stress test goals:

- Identify exactly where the system fails (e.g., 1800 req/sec).
- Observe failure mode:
 - graceful degradation?
 - crashing?
 - cascading failures?
- Verify system recovery after stress is removed.

Stress Testing 2/2

- Why stress testing is important:

- Helps prepare for unexpected traffic bursts (Black Friday, viral traffic).
- Validates resilience patterns: load shedding, circuit breakers, auto-scaling.
- Ensures protection against cascading failures to downstream systems.

- What you measure:

- Maximum sustainable throughput.
- Error patterns during overload.
- Latency slope as concurrency increases.
- Recovery time after overload.

Balwinder Sodhi

Endurance / Soak Testing 1/2

Purpose: Expose issues that happen only after long durations (hours/days).

Typical issues detected only by soak tests:

- Memory leaks (small but continuous growth).
- Resource handle leaks (file descriptors, DB connections).
- Slow but growing latency.
- Data corruption due to long-lived threads/sessions.
- Disk I/O saturation due to log file buildup.
- Message queue backlogs.

Endurance / Soak Testing 2/2

- Execution characteristics:

- Run at medium or near-peak expected load.
- Duration: 6–48 hours depending on system complexity.
- Monitor system metrics continuously.

- Success Criteria:

- Flat memory usage profile.
- Stable throughput and latency.
- No increase in error rates over time.
- No degradation in internal metrics (queues, worker pools, GC times).

Balwinder Sodhi

Example Load Test Using Locust (Python)

```
from locust import HttpUser, task, between

class EcommerceUser(HttpUser):
    wait_time = between(1, 3)
    @task(3)
    def browse_products(self):
        self.client.get("/products")
    @task
    def view_cart(self):
        self.client.get("/cart")
    @task
    def add_to_cart(self):
        self.client.post("/cart/add", json={"item_id": 42})
```

Run it as:

```
$ locust -f locustfile.py --headless \
-u 500 -r 50 -t 10m \
--host=https://staging.example.com
```

What this measures:

- Whether main browsing flow supports 500 concurrent users.
- Latency distribution under mix of read/write operations.

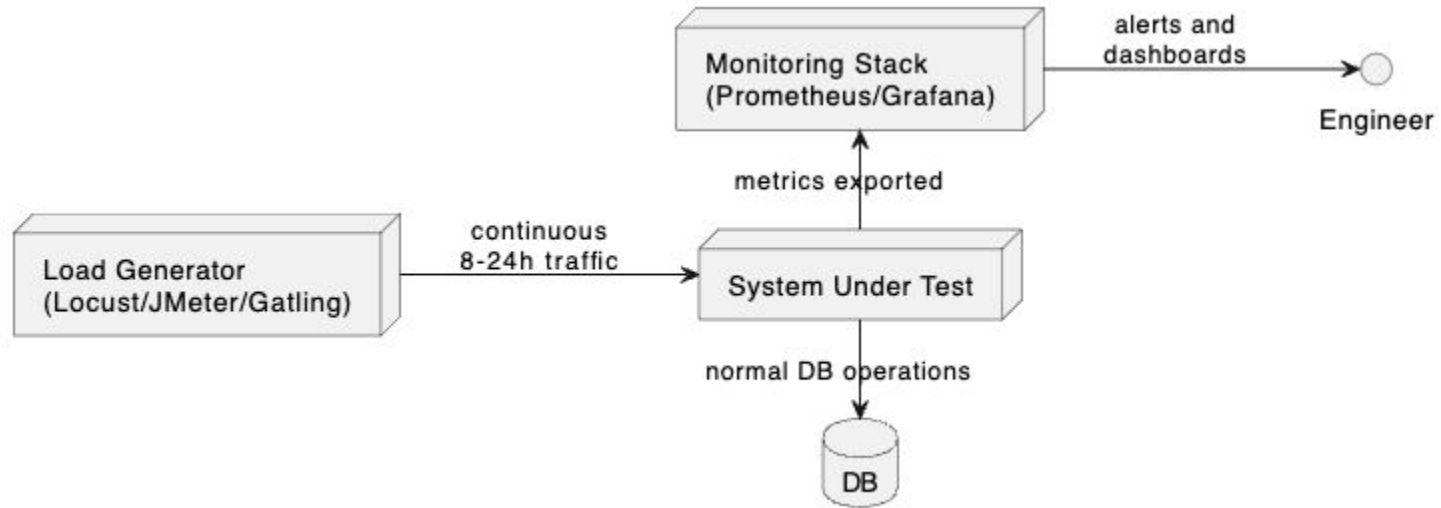
Stress Test Script Example (Locust Spike Pattern)

```
# Simulate extreme bursts by ramping up aggressively
locust -f locustfile.py --headless \
    -u 5000 -r 500 \
    -t 5m --host=https://staging.example.com
```

Interpreting results:

- Watch for sudden rise in p99 latency before system starts failing.
- Identify saturation points (DB pool exhaustion, CPU 100%, thread exhaustion).
- Verify whether the system recovers automatically once load is removed.

Endurance Test Architecture



What to Monitor During Performance Tests

- Application Metrics

- Response times (p50, p95, p99)
- Throughput (req/sec)
- Error rates
- Thread pools, worker queues
- Cache hit/miss ratio

- System Metrics

- CPU usage per core
- Memory, GC time
- Disk I/O
- Network throughput
- File descriptor count

- Distributed System Metrics

- DB connection pools
- Redis/cache saturation
- Message broker queue depth
- API/Service dependency latency

- Tools: Prometheus, Grafana, New Relic, Datadog, CloudWatch.

How to Read Performance Test Results 1/2

- Latency Patterns

- Sudden upward curve after a concurrency level = saturation point.
- p99 degrading faster than p50 = contention.

- Throughput Curves

- Linearly increasing throughput = good scaling.
- Throughput flattening early = bottleneck.
- Throughput collapse = catastrophic failure.

Balwinder Sodhi

How to Read Performance Test Results 2/2

- CPU & Memory Patterns

- High CPU + stable latency = OK.
- High CPU + rising latency = CPU bottleneck.
- Increasing memory over hours = leak.

- Error Patterns

- Too many read timeouts = downstream dependency issues.
- Lots of 500s = application logic errors.
- 429/503 = server actively shedding load (expected under stress).

Balwinder Sodhi

Summary of Performance Testing Types

Type	Goal	When to Use	Duration
Load Testing	Validate performance under peak expected load	Every release or major feature	10–30 minutes
Stress Testing	Find breaking point + recovery behavior	Before big launches, resilience checks	5–20 minutes
Endurance Testing	Detect long-term stability issues	Before major releases, infra changes	6–48 hours

Security Testing: What It Is & Why It Matters

- Purpose: Ensure software is resistant to attacks, protects sensitive data, and adheres to security principles.
- Why engineers need strong security testing
 - Prevent breaches, financial loss, and reputational damage.
 - Satisfy compliance requirements (GDPR, PCI-DSS, HIPAA).
 - Proactively detect vulnerabilities before attackers do.
 - Reduce technical debt and future remediation costs.
- Security testing is NOT just scanning tools
 - It includes threat modeling, secure SDLC practices, and continuous monitoring.

Balwinder Sodhi

Core Security Concepts (CIA Triad)

- Confidentiality

- Protect data from unauthorized access.
- Examples: encryption at rest, strong authentication, proper access controls.

- Integrity

- Ensure data is accurate and unmodified.
- Examples: hashing, digital signatures, database constraints, audit logs.

- Availability

- Ensure the system and data remain accessible.
- Examples: rate limiting, auto-scaling, DDoS protection, redundancy.

Balwinder Sodhi

Additional Core Concepts Engineers Must Know

- **Authentication**

- Verifying user identity (passwords, tokens, MFA).

- **Authorization**

- Controlling what a user can access (RBAC, ABAC, least privilege).

- **Non-Repudiation**

- Proof that a user performed an action (signed logs, signatures).

- **Input Validation & Sanitization**

- Prevent injection attacks (SQLi, XSS, command injection).

- **Secure Session Management**

- Session rotation, timeout, cookie flags (Secure, HttpOnly, SameSite).

- **Threat Modeling**

- Identify assets, attack vectors, and mitigation strategies.

Threat Modeling

- Purpose is to identify and prioritize threats before testing.

- Process (STRIDE model):

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

- Apply threat modeling results to:
 - Security test cases
 - Unit tests for security logic
 - Abuse-case scenarios
 - Access control tests
 - API-specific tests

OWASP Top Ten (A Must-Know for All Testers)

Most common and impactful categories (<https://owasp.org/Top10/>):

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable & Outdated Components
7. Identification & Authentication Failures
8. Software/Data Integrity Failures
9. Logging & Monitoring Failures
10. Server-Side Request Forgery (SSRF)

Use case: Build test cases for each category.

Overview of Security Testing Types

- Security testing validates that a system can withstand malicious activity and protect sensitive data.
- Major Types:
 - SAST — Static Application Security Testing
 - DAST — Dynamic Application Security Testing
 - IAST — Interactive Application Security Testing
 - SCA — Software Composition Analysis
 - Penetration Testing
 - Fuzz Testing
 - Security Regression Testing
 - API Security Testing
- Each technique finds different classes of vulnerabilities.

Static Application Security Testing (SAST)

- It involves analyzing source code or bytecode without execution to detect insecure coding patterns.
- Finds vulnerabilities like:
 - SQL injection
 - Hardcoded credentials
 - Unsafe deserialization
 - Broken access control logic
 - Weak cryptographic usage
 - Command injection
- When to use: Early in development (shift-left). Ideal for PR checks and CI.
- Tools:
 - Semgrep
 - SonarQube
 - Bandit (Python)
 - ESLint security plugins

Dynamic Application Security Testing (DAST)

- It is testing a **running application** by simulating real-world attack techniques from outside ("black-box" testing).
- Finds vulnerabilities like:
 - Cross-Site Scripting (XSS)
 - SQL injection
 - Broken authentication/authorization
 - Missing security headers
 - Server misconfiguration
- When to use: After deployment to test/staging environments.
- Tools:
 - OWASP ZAP
 - BurpSuite
 - Nessus
 - Acunetix

Interactive Application Security Testing (IAST)

- It is an instrumentation-based approach — analyzes the application **from inside** while it runs.
- Provides:
 - Code-level insight + runtime data
 - Accurate detection with fewer false positives
 - Good for modern microservices and APIs
- Finds vulnerabilities like:
 - Runtime injection issues
 - Logic flaws
 - Authentication/authorization errors
- When to use: During automated integration tests or staging deployments.
- Tools:
 - Contrast Security
 - Hdiv IAST
 - Seeker (Synopsys)

Software Composition Analysis (SCA)

- It involves scanning your dependencies (libraries, containers) for known vulnerabilities (CVEs).
- Why it matters:
 - Most breaches today come from vulnerable libraries — not custom code.
- Finds:
 - Known CVEs
 - Outdated dependencies
 - License violations

- Tools:
 - Snyk
 - OWASP Dependency-Check
 - GitHub Dependabot
 - Trivy (container scanning)
- Example Dependabot alert:
 - “Vulnerability in lodash 4.17.19 — upgrade to 4.17.21”

Penetration Testing (Pen Testing)

- Manual, human-driven security assessment where experts attempt to exploit vulnerabilities.
- Types:
 - Black-box: no system knowledge
 - Gray-box: partial knowledge
 - White-box: full knowledge + credentials

- Focus areas:
 - Business logic flaws
 - Authentication bypass
 - Privilege escalation
 - Data exfiltration paths
 - Chaining low-risk issues into a high-risk exploit
- Tools:
 - Kali Linux
 - Metasploit
 - BurpSuite Pro
 - Nmap

Fuzz Testing (Fuzzing)

- It involves sending malformed, random, or unexpected inputs to uncover crashes and edge-case vulnerabilities.
- Used for:
 - Parsers
 - File handlers
 - Network protocols
 - Microservices APIs

- Finds vulnerabilities like:
 - Buffer overflows
 - Input parsing bugs
 - Memory corruption
 - Logic inconsistencies
- Tools:
 - AFL (American Fuzzy Lop)
 - libFuzzer
 - Burp Intruder (payload fuzzing)
 - Peach Fuzzer

API Security Testing

- API testing is critical because modern applications rely heavily on microservices and REST/GraphQL APIs.
- Common vulnerabilities tested:
 - Broken object-level authorization (BOLA)
 - Broken authentication
 - Rate-limit bypass
 - Mass assignment
 - Excessive data exposure
 - Input injection

- Tools:
 - OWASP ZAP
 - Postman + security collections
 - BurpSuite
 - ReadyAPI
 - APIsec
- Example Test Case:

```
# User shouldn't access another user's orders
# Logged in userId=14
resp = client.get("/orders?userId=52",
                  headers=auth_token_user1)
assert resp.status_code == 403
```

Security Testing in CI/CD Pipelines

Pipeline Sequence Example:

- Pre-commit: linting (eslint, bandit, semgrep)
- CI — Stage 1: SAST + SCA
- CI — Stage 2: DAST against ephemeral/staging env
- CD: Infrastructure scanning (Terraform, Docker images)
- Post-deploy: Continuous monitoring (SIEM, cloud logs)

Balwinder Sodhi

Test Automation

- Execution of tests using automated scripts, tools, and frameworks to validate application behavior reliably and repeatedly.
- Why automation is essential
 - Reduces manual testing effort and human error.
 - Enables fast feedback loops in CI/CD.
 - Improves regression reliability.
 - Supports shift-left and shift-right testing.
 - Ensures repeatability and scalability.
- When NOT to automate
 - UI tests for rapidly changing UIs.
 - Subjective tests (usability).
 - One-off exploratory tests.

Automation Tools: Categories Overview

The automation ecosystem spans multiple layers:

- Functional Test Automation Tools
- Bug Tracking & Test Management Systems
- Performance Testing Tools
- Code Analysis (Static & Dynamic) Tools
- Mobile Testing Tools

Each category supports different levels of the testing pyramid.

Functional Test Automation Tools

- Purpose: Automate user flows, API tests, integration tests, and UI interactions.
- Popular frameworks
 - Selenium WebDriver — browser automation standard.
 - Playwright — modern, fast, auto-waiting, cross-browser.
 - Cypress — fast JS-native UI and API testing.
 - Robot Framework — keyword-driven tests for enterprise QA teams.
 - Appium — mobile automation using WebDriver protocol.
 - Pytest/JUnit/Jest — foundational unit + integration test frameworks.

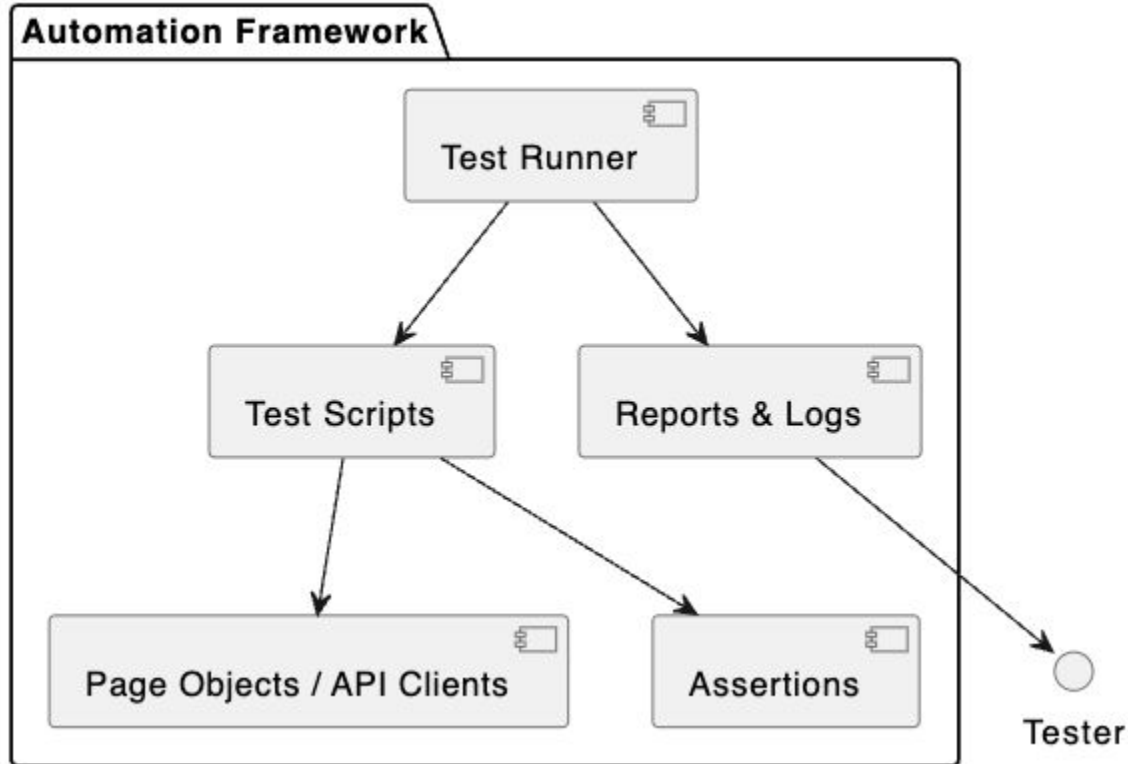
Balwinder Sodhi

Example — Playwright JS

```
const { test, expect } = require('@playwright/test');

test('user login flow', async ({ page }) => {
  await page.goto('https://staging.example.com/login');
  await page.fill('#email', 'qa@example.com');
  await page.fill('#password', 'P@ssw0rd!');
  await page.click('button[type=submit]');
  await expect(page.locator('text=Dashboard')).toBeVisible();
});
```

Automation Framework Architecture



Bug Tracking Tools

- Purpose: Central system to log, triage, track, and resolve issues from automation and manual testing.
- Popular Bug Tracking Systems
 - Jira — industry-standard; integrates with CI and test management.
 - Azure DevOps Boards — enterprise DevOps integration.
 - YouTrack — fast search, Agile-friendly.
 - GitHub Issues — lightweight, integrates well with automation.
 - Bugzilla — long-standing OSS system.
- Automating test ↔ bug integration
 - Auto-create bugs on failure (with logs/screenshots).
 - Auto-assign labels (performance, regression, security).
 - Link test cases to bugs for traceability.
 - Generate dashboards with defect trends.

Performance Testing Tools (Already seen earlier)

- Purpose: Evaluate application behavior under load, stress, and endurance.
- Common Tools
 - Locust (Python) — programmable load tests.
 - JMeter — XML-based, widely used for enterprise load testing.
 - Gatling (Scala/Java) — high-performance, scenario-driven.
 - k6 — modern JavaScript-based load testing tool.
 - Artillery — Node.js-based, great for API load testing.
- Modern teams seem to prefer k6/Locust:
 - Code-based test scripts version-controlled with application code.
 - Integrates cleanly with GitHub Actions / GitLab CI.

Code Analysis Tools (Static & Dynamic)

- Purpose: Catch bugs, vulnerabilities, code smells, and complexity issues early.
- SAST Tools
 - SonarQube
 - Semgrep
 - Bandit (Python)
 - ESLint / Pylint / Rubocop
- DAST & Runtime Analysis
 - OWASP ZAP
 - Burp Suite
 - Valgrind
 - Perf (Linux) for profiling
 - Dynatrace/New Relic for runtime behavior in production

Mobile Testing Tools

Purpose: Automate tests for iOS, Android, hybrid apps, and responsive UIs.

● Key Tools

- Appium — cross-platform, WebDriver-based
- Espresso (Android) — fast, native instrumentation tests
- XCUITest (iOS) — official Apple framework
- Detox — end-to-end tests for React Native apps
- BrowserStack / Sauce Labs — device farm testing

● When to choose what

- Espresso/XCUITest: fastest, stable → test core flows
- Appium: cross-platform → well-suited for UI regression suites
- Detox: RN testing with parallel execution

Appium python example

```
from appium import webdriver

caps = {
    "platformName": "Android",
    "deviceName": "Pixel_5",
    "app": "/path/app.apk"
}

driver = webdriver.Remote("http://localhost:4723/wd/hub", caps)
element = driver.find_element_by_accessibility_id("login_button")
element.click()
```

How to Choose an Automation Tool

- Choose based on:
 - Tech stack (web, mobile, backend)
 - Skill set of team (Python/JS/Java)
 - Execution speed (UI tools vary widely)
 - Maintainability & ecosystem support
 - CI/CD integration
 - Recordability vs code-based scripting
 - Reporting & dashboarding needs
- General rule: Prefer code-based frameworks for maintainability + version control.

MASTERING SOFTWARE
DEVELOPMENT

A Guide For Early Career Engineers



Balwinder Sodhi

Compliance & Regulatory Testing

Compliance & Regulatory Testing: What It Is

- Purpose is to Ensure that software systems adhere to legal, industry, and organizational regulations designed to protect users, data, and critical systems.
- Why compliance matters
 - Avoid legal penalties, fines, and lawsuits
 - Protect user privacy and safety
 - Improve trust and credibility
 - Required for audits and certifications
 - Ensures secure handling of sensitive data
- Who performs compliance testing?
 - QA engineers
 - Security engineers
 - Legal/compliance teams
 - External auditors

Types of Compliance Testing

- Industry-specific compliance testing
 - Finance, healthcare, government, telecom.
- Data protection & privacy compliance
 - India specific: DPDP, RBI, UIDAI, SEBI/IRDAI, DoT/TRAI, etc.
 - Others: GDPR, HIPAA, CCPA, PCI-DSS.
- Accessibility compliance testing
 - WCAG 2.1/2.2 standards.
- Security & audit trail compliance
 - Logging, monitoring, access controls, encryption.

Compliance testing blends functional, non-functional, security, and UX testing.

Overview of Industry-Specific Regulations

Finance Sector

- RBI Compliance for Fintech, PSPs, NBFCs, Banks (India specific)
- PCI-DSS (Payment Card Industry Data Security Standard)
 - Protects cardholder data
 - Requires encryption, strict access control, network segmentation
 - Tests: secure transmission, tokenization, logging & monitoring
- SOX (Sarbanes–Oxley Act)
 - Focus on financial data accuracy
 - Requires audit trails and integrity checks
- FINRA regulations
 - Data retention rules (e.g., WORM storage)
 - Monitoring of financial transactions

Healthcare Sector Regulations 1/2

- **HL7 / FHIR Compliance**

- Standards for electronic healthcare data exchange.
- Test cases verify correct data formats & secure communication.

- **DISHA (Digital Information Security in Healthcare Act) — India**

- Role-based access to EHI (doctor vs. lab vs. admin)
- Audit trail validation for access to health data
- Automatic session timeout
- Encryption at rest and in transit

- **ABDM (Ayushman Bharat Digital Mission) Compliance — India**

- FHIR interoperability (HL7 FHIR standards for medical data exchange)
- Consent management system using ABDM-approved flow
- Health Information Exchange (HIE) workflows
- Secure data transmission using ABDM Encryption Guidelines
- Token-based access control (gateway-issued tokens)
- Privacy Sandbox compliance for viewing/sharing records

Healthcare Sector Regulations 2/2

- HIPAA — USA

- Protects Protected Health Information (PHI).
- Applies to healthcare providers, insurers, partners.

- Testing areas for HIPAA compliance

- Access control enforcement
- PHI encryption in transit & at rest
- Secure user authentication / MFA
- Audit logging of all access to PHI
- Timeout/session expiration
- Data minimization practices

Government & Public Sector Compliance

- **FISMA (US)**
 - Requires risk assessments, continuous monitoring, security auditing.
- **ISO 27001**
 - Broad information security framework used globally.
- **GDPR (EU) for public institutions**
 - Strict data privacy and consent rules.
- **Example test cases**
 - Validate consent-based access for personal data
 - Validate audit log integrity
 - Validate encryption and data retention policies

GDPR Compliance Testing

● Key GDPR Requirements For Engineers

- Lawful basis for collecting personal data
- Right to access, right to erase, right to portability
- Consent management with explicit opt-in
- Data minimization — collect only what is needed
- Data protection by design & default
- Breach notification within 72 hours
- Cross-border transfers with safeguards

● GDPR Test Cases

- Verify DSAR APIs: user can request/export/delete their data
- Verify “Delete account” actually wipes personal data
- Verify “Reject cookies/trackers” path functions correctly
- Confirm PII is encrypted at rest
- Confirm logs & DB backups also follow GDPR retention rules

Core HIPAA Security Rule Tests

- Access Control
 - RBAC, least privilege
 - Break-glass access events must be logged
- Audit Controls
 - All access to PHI must be logged
 - Logs must be immutable and time-synchronized
- Transmission Security
 - TLS 1.2 or higher
 - No PHI transmitted over insecure channels
- Integrity Controls
 - Hashing/signing sensitive medical documents
 - Prevent tampering

Accessibility Testing

- Objectives: Ensure applications are usable by people with disabilities (visual, auditory, motor, cognitive).
- **Standards**:
 - WCAG 2.1/2.2 (Web Content Accessibility Guidelines)
 - Section 508 (US Federal accessibility requirements)
 - EN 301 549 (EU accessibility standard)
- **Accessibility is not optional**:
 - Required for public sector
 - Required for e-commerce in many regions
 - Reduces legal risk (example: many ADA lawsuits)

WCAG 2.1 Guidelines: Key Principles

WCAG is organized under POUR:

- **P — Perceivable**
 - Content must be presented in ways users can perceive.
 - Examples: alt-text, proper color contrast, captions.
- **O — Operable**
 - Interface must be navigable.
 - Examples: keyboard navigation, focus states, skip links.
- **U — Understandable**
 - Content and UI must be clear.
 - Examples: descriptive labels, error messages.
- **R — Robust**
 - Compatible with assistive technologies.
 - Examples: ARIA attributes, semantic HTML.

MASTERING SOFTWARE
DEVELOPMENT

A Guide For Early Career Engineers

Emerging Trends in Software Testing

Balwinder Sodhi

Emerging Trends (as seen in 2025)

- **Shift-Right Testing**
 - Testing not just before release, but in production.
 - Uses observability, monitoring, feature flags, and canary releases.
 - Detects issues only visible under real traffic patterns.
- **Autonomous Test Generation**
 - AI systems generate tests from requirements, logs, code behavior.
 - Reduces manual test-writing cost and improves coverage.
- **Self-healing Test Automation**
 - Tests automatically adjust selectors, wait logic, and interaction flows.
 - Especially valuable for UI tests.
- **Continuous Quality Engineering**
 - Quality integrated into the entire delivery pipeline.
 - Test metrics tied to business KPIs (conversion, latency, crash rates).
- **Cloud-Native Testing**
 - Testing distributed, serverless, and container-based architectures.
 - Requires testing of autoscaling, networking, and resilience behaviors.

AI in Testing: Intelligent Automation 1/2

How AI Improves Testing

- Converts test execution logs → test cases automatically.
- Predicts which areas of code are most likely to break (risk-based testing).
- Clusters test failures to reduce noise.
- Reduces flaky tests by analyzing run patterns.
- Suggests missing tests by analyzing untested paths.

Balwinder Sodhi

AI in Testing: Intelligent Automation 2/2

- Practical Applications

- NLP-based test creation from requirements ("write test from user story").
- Auto-maintenance of test scripts (fixing broken locators).
- ML-driven anomaly detection for performance metrics.

- Example AI-Driven Flow

- AI inspects code changes
- Predicts impacted modules
- Selects or generates tests
- Executes them
- Produces reasoning and coverage map

AI for Prediction and Quality Insights

- Predictive Quality Metrics

- Predict probability of a defect appearing in a module.
- Predict which tests are most valuable for a given commit.
- Predict release readiness using multi-signal analysis.

- ML Signals Used

- Commit metadata (files changed, churn, ownership)
- Historical defect data
- Test failure patterns
- Static analysis + code complexity
- Production telemetry from observability systems

- Outcome

- Faster CI pipelines (intelligent test selection).
- Better risk-based decisions for releases.
- Reduced regression rates over time.

Testing for Emerging Technologies 1/2

- IoT (Internet of Things) Testing

- Device interoperability
- Network instability simulation
- Sensor accuracy validation
- Firmware update testing
- Security of constrained devices (memory/CPU limited)

- Edge Computing

- Latency-sensitive behavior
- Partition tolerance tests (network splits)
- Sync behavior between cloud + edge nodes
- Failover and fallback scenarios

Testing for Emerging Technologies 2/2

- **AI/ML Systems**

- Data quality validation
- Model drift detection
- Testing model explainability and fairness
- Evaluation of precision/recall and degradation over time

- **Blockchain & Distributed Ledgers**

- Consensus algorithm behavior under load
- Smart contract vulnerability tests
- Double-spend and transaction ordering tests

Balwinder Sodhi

Practical Tools for Emerging Tech Testing 1/2

● IoT Testing Tools

- IoT simulators (AWS IoT Device Simulator, Azure IoT SDK)
- MQTT fuzzers
- Network chaos tools (toxiproxy)

● AI/ML Testing Tools

- Great Expectations (data validation)
- Deepchecks
- Evidently AI (drift detection)
- MLflow + unit tests for models
- Adversarial attack frameworks (e.g., Foolbox)

Practical Tools for Emerging Tech Testing 2/2

- Blockchain Testing Tools

- Hardhat (Ethereum testing)
- Ganache (local blockchain sandbox)
- Mythril (smart contract security analysis)

- Edge/Distributed Testing Tools

- Chaos Mesh
- Gremlin
- k6 for edge load generation
- Kubernetes failure injection

Balwinder Sodhi

Overall Summary

- **Testing and QA play different but complementary roles**
 - Testing verifies functionality and finds defects; QA ensures the process prevents them.
 - QA defines standards and practices; testing validates outcomes against them.
 - Together they improve both product quality and the development workflow.
- **Modern testing is continuous, automated, and integrated early**
 - Testing starts at requirements and continues through deployment (shift-left + shift-right).
 - Automation covers unit, API, UI, security, and performance tests.
 - CI/CD pipelines provide fast feedback and reduce regression risk.
- **Engineers need both technical testing skills and process awareness**
 - Technical skills: writing good tests, understanding edge cases, using testing tools.
 - Process awareness: prioritizing tests by risk, structuring suites, maintaining coverage.
 - Collaboration with QA, product, and DevOps improves quality outcomes.
- **Emerging tech and AI are reshaping the testing landscape**
 - AI helps auto-generate tests, predict failure hotspots, and reduce flaky tests.
 - New technologies (IoT, ML, blockchain, edge systems) require new testing approaches.
 - Testing now includes data quality, model behavior, resilience, and security validation.