Quality Assurance and Validation Framework

Generating tests is only half the battle; ensuring their quality is paramount. This section provides checklists and guidelines for validating AI-generated tests.

Generated Test Quality Checklist

Use this comprehensive checklist to evaluate every AI-generated test.

**Structural Quality Assessment**

**Class and Method Organization**

* + [ ] Test class name clearly indicates what is being tested (e.g., UserServiceTest, UserControllerIT).
  + [ ] Test methods have descriptive, behavior-focused names (e.g., shouldReturnUserWhenValidId(), shouldThrowExceptionWhenInvalidInput()).
  + [ ] Test methods follow consistent naming convention: should[ExpectedBehavior]When[Condition] or test[Scenario].
  + [ ] Tests are logically grouped (e.g., all happy paths together, all error cases together).
  + [ ] Setup and teardown methods (@BeforeEach, @AfterEach) are used appropriately for common setup/cleanup.

**Annotation and Configuration**

* + [ ] Correct test framework annotations are used (@Test, @DisplayName, @ExtendWith, @ParameterizedTest).
  + [ ] Appropriate Spring test annotations are used (@SpringBootTest, @WebMvcTest, @DataJpaTest, etc.).
  + [ ] Mock annotations are used correctly (@Mock, @MockBean, @InjectMocks, @SpyBean).
  + [ ] Test configuration (e.g., @TestPropertySource, profiles) is minimal and focused on the test's needs.
  + [ ] No unnecessary annotations or imports are present.

**Dependencies and Mocking**

* + [ ] All external dependencies are mocked appropriately for unit tests.
  + [ ] Mock setup (when().thenReturn(), doNothing().when()) is clear, concise, and relevant to the test scenario.
  + [ ] No over-mocking of internal implementation details or private methods.
  + [ ] Real objects are used where appropriate (e.g., value objects, utility classes, DTOs).
  + [ ] Mock verification (Mockito.verify()) is meaningful and not excessive; it verifies interactions relevant to the tested behavior.

**Behavioral Quality Assessment**

**Test Coverage and Scenarios**

* + [ ] All primary happy path scenarios are covered.
  + [ ] Error conditions are tested with proper exception verification (type and message).
  + [ ] Edge cases and boundary conditions are included (e.g., null, empty, max/min values).
  + [ ] All public methods have corresponding tests covering their main behaviors.
  + [ ] Key business logic branches (e.g., if/else statements, loops) are covered by at least one test.

**Assertions and Verification**

* + [ ] Assertions are specific, meaningful, and clearly state the expected outcome (e.g., assertThat(result.getName()).isEqualTo("John");).
  + [ ] Multiple assertions test different aspects of the result or state change.
  + [ ] Error scenarios verify the correct exception type and message.
  + [ ] Side effects are verified where important (e.g., verify(emailService).sendEmail(...), database state changes).
  + [ ] No weak assertions are used (assertTrue(true), assertNotNull(result) without further checks).

**Test Data and Input**

* + [ ] Test data is realistic and representative of real-world scenarios.
  + [ ] Input variations cover different scenarios (e.g., valid, invalid, empty, large).
  + [ ] No hardcoded magic numbers or strings without clear explanation or use of constants.
  + [ ] Test data builders or factories are used for complex object creation to improve readability and maintainability.
  + [ ] Edge case inputs are specifically included and tested (e.g., null collections, zero values).