

QA Manifesto

[QA Manifesto](#)

[Introduction and Purpose](#)

[Testing Pyramid](#)

[Functional Test Types](#)

[Non-Functional Test Types](#)

[Other Test Types \(both Functional and Non-Functional\)](#)

[Testing Life Cycle](#)

[Continuous Integration/Continuous Development \(CI/CD\)](#)

[Approach to Performance Testing](#)

[Test Driven Development \(TDD\)](#)

[Behavior Driven Development \(BDD\)](#)

[Shift Left Testing](#)

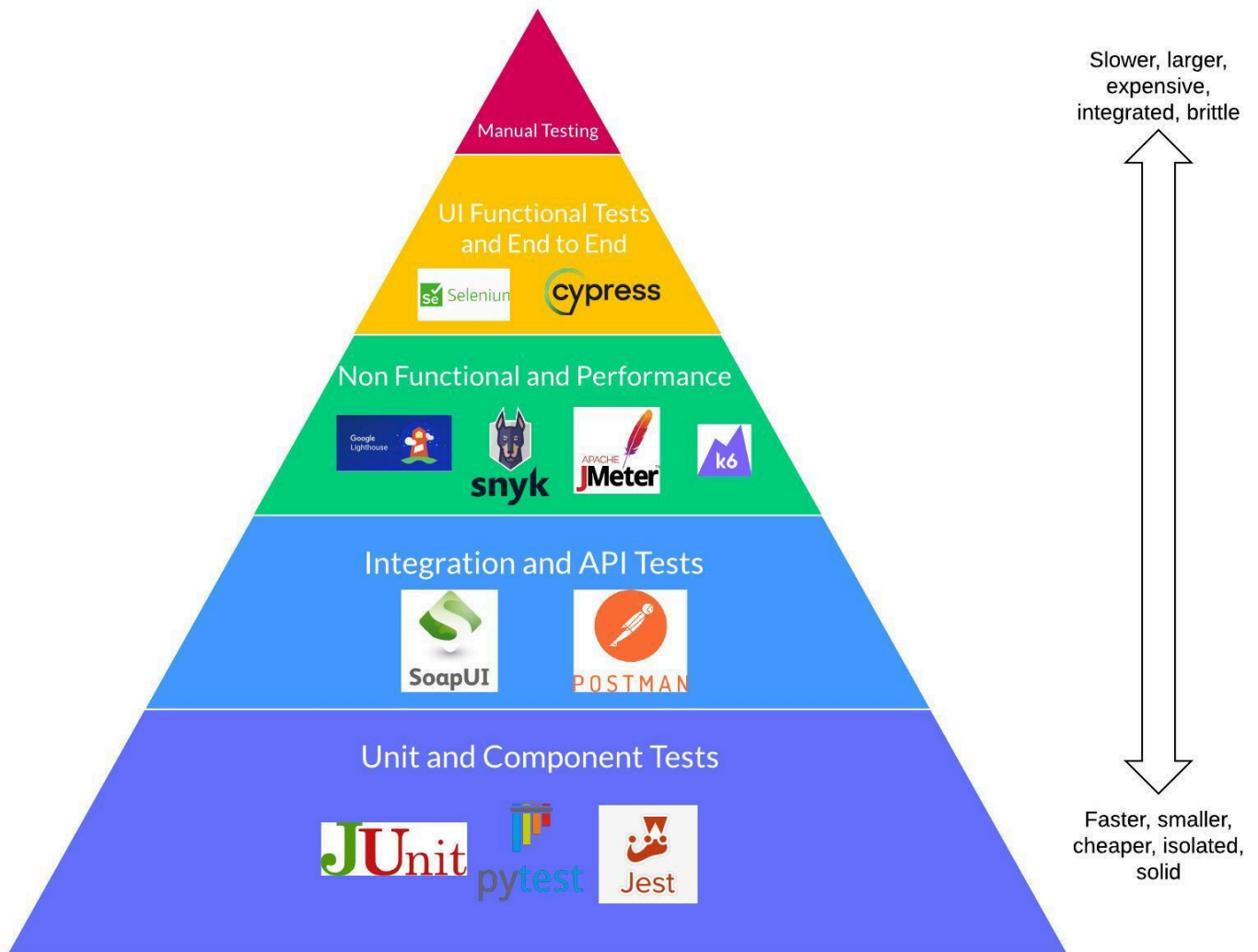
[Questions to ask yourself when approaching testing](#)

Introduction and Purpose

This manifesto outlines the core principles and practices that guide us, the QA professionals, in our mission to deliver exceptional software products.










This document serves as a comprehensive guide to testing methodologies, encompassing various functional and non-functional testing types. Delving into the software testing life cycle, we explore each stage and the testing activities associated with it. Furthermore, we explore the integration of QA practices within the modern development workflow, specifically focusing on Continuous Integration and Continuous Delivery (CI/CD) pipelines.

Testing Pyramid








Functional Test Types

Test Type	Description	Tools
Unit	Isolates and tests the smallest testable unit of code, typically individual functions, modules, or classes.	Jest, pytest, JUnit

Component	Validating the functionality and interactions of a group of collaborating units (functions, modules) that form a cohesive component within the larger system.	  
Integration	Validating how different software components interact and function together to form a cohesive system.	 
System	Comprehensive evaluation of the entire system, ensuring it meets all requirements and functions.	 
End to End	Validating user journeys and core functionalities working together seamlessly from start to finish.	 

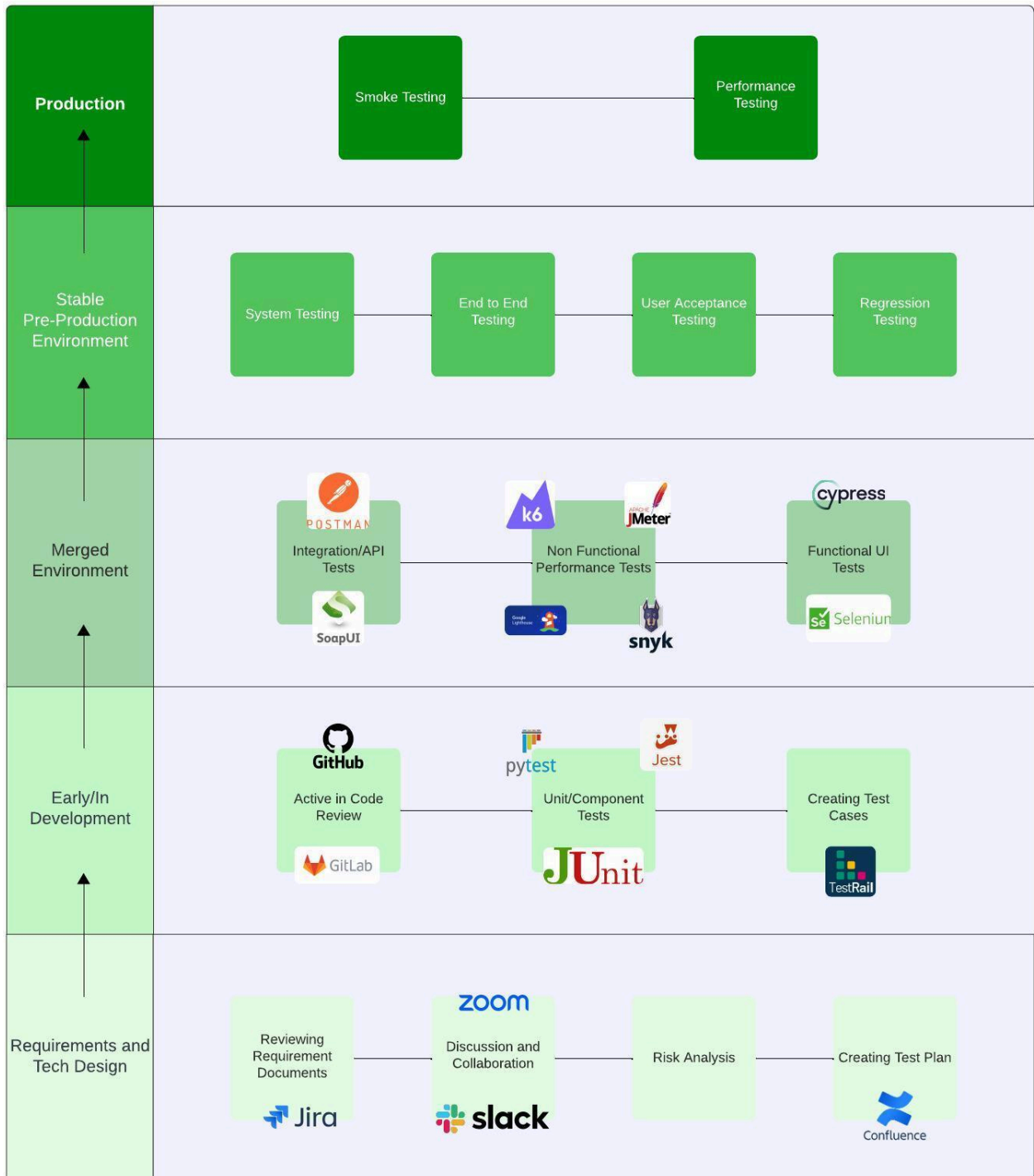
Non-Functional Test Types

Test Type	Description	Tools
Performance	Evaluating the performance and scalability of a system under a simulated workload. It aims to identify bottlenecks, measure response times, and ensure the system can handle expected usage patterns without degradation in performance.	 
Security	Identifying vulnerabilities, weaknesses, and potential security risks in an application, system, or infrastructure	
Usability/Accessibility	Assess how easy and intuitive it is for users to interact with a product, system, or website. Focuses on identifying any pain points, areas of confusion, or obstacles that users might encounter while trying to achieve their goals.	 

Other Test Types (both Functional and Non-Functional)

Test Type	Description
Smoke	A quick and shallow assessment designed to identify major issues or defects early on, determining if the software is even stable enough for further, more in-depth testing.
User Acceptance (UAT)	Testing from the end-user's perspective to ensure its functionality, usability, and compatibility with real-world scenarios. It essentially puts the power in the hands of the actual users to validate if the software meets their needs and expectations.
Exploratory	Emphasizes learning, test design, and test execution happening simultaneously. The tester delves into the software with a curious mind, experimenting and adapting your testing strategy as he/she goes based on their observations and discoveries.
Regression	Systematically re-testing existing functionalities after code changes, ensuring software stability, preventing regressions, and delivering a high-quality product to users. Ensures changes you make to the code don't negatively impact existing features.

Testing Life Cycle



Continuous Integration/Continuous Development (CI/CD)

- CI/CD is a powerful approach in software development that automates many aspects of the development lifecycle. It plays a crucial role in software testing and quality assurance (QA) by enabling faster feedback loops, improved quality control, and a more efficient testing process.
 - How QA Integrates with CI/CD:
 - Designing Automated Tests: QA plays a vital role in designing and developing automated tests that can be integrated into the CI/CD pipeline. These tests can cover various aspects of the software, including unit testing, functional testing, and API testing.
 - Defining Test Criteria: QA helps define the testing criteria and success metrics that will be used to evaluate the automated tests within the pipeline. This ensures the tests are effective in identifying defects.
 - Monitoring and Analyzing Results: QA actively monitors the results of the automated tests within the CI/CD pipeline. They analyze failures and work with developers to identify and fix bugs promptly.
 - Benefits of CI/CD for QA:
 - Reduced Testing Time: Automation helps save time and resources by eliminating the need for repetitive manual testing.
 - Improved Test Coverage: CI/CD allows for more frequent testing, potentially leading to more comprehensive test coverage.
 - Increased Confidence: By having a robust CI/CD pipeline with clear testing procedures, QA can gain greater confidence in the quality of the software being delivered.
-

Approach to Performance Testing

Performance testing is a crucial aspect of software development, ensuring your application can handle expected user loads without compromising responsiveness or stability. Here's a breakdown of key steps to take when approaching performance testing:

1. Define Performance Goals and Constraints
 - a. Identify Performance Objectives: Clearly define what you want to achieve with performance testing. Typical goals include measuring response times, identifying bottlenecks, and determining scalability under load.
 - b. Set User Load Expectations: Estimate the number of concurrent users and usage patterns your application is expected to handle. Factors like peak traffic hours and user behavior need to be considered.

- c. Establish Performance Benchmarks: Define acceptable response times and throughput thresholds based on user experience and business needs. These benchmarks will serve as success criteria for your tests.
 - d. All of this can be achieved by reviewing current production usage for similar services and/or collaborating with Product Managers.
2. Design and Develop Test Scenarios
 - a. User Actions and Workflows: Identify key user actions and workflows that will be simulated during the test. This could involve logins, searches, product purchases, etc.
 - b. Test Script Development: Develop test scripts within your chosen performance testing tool to simulate user behavior and load conditions.
 - c. Data Considerations: Prepare realistic test data to populate the test environment and ensure accurate performance evaluation.
 3. Configure and Execute Load Tests
 - a. Load Test Configuration: Configure the test parameters within your chosen tool, specifying the number of virtual users, ramp-up times, and duration of the test.
 - b. Test Execution and Monitoring: Run the load tests and monitor key metrics like response times, throughput, CPU utilization, and memory usage. Tools typically provide various dashboards and reports for analysis.
-

Test Driven Development (TDD)

- TDD is a methodology that flips the traditional development process on its head. Instead of writing tests after the code is complete, TDD emphasizes writing the tests first. This approach ensures the code is designed with testability in mind and helps developers write clean, functional code from the get-go.
 - Benefits of TDD:
 - Early Defect Detection: By writing tests first, TDD helps identify potential issues early in the development lifecycle, reducing the cost of fixing bugs later.
 - Improved Code Quality: The focus on small, well-tested units of code leads to more robust and maintainable software overall.
 - Increased Confidence: Developers gain confidence in their code knowing it has been thoroughly tested through the TDD process.
 - Better Design: TDD can promote better code design as developers consider testability from the outset.
-

Behavior Driven Development (BDD)

- BDD is an agile software testing approach that emphasizes collaboration between developers, testers, and other stakeholders. It focuses on using clear, natural language to describe the expected behavior of the software from the perspective of the end user.
 - BDD utilizes a specific vocabulary, often including terms like "Given," "When," and "Then," to structure user stories and acceptance criteria.
 - Given: This describes the initial context or setup for the scenario.
 - When: This describes the action taken by the user.
 - Then: This describes the expected outcome or behavior of the system.
 - Benefits of BDD:
 - Improved Communication: BDD promotes clear communication between testers, developers, and other stakeholders by using a common language.
 - Early Defect Detection: By focusing on user stories and acceptance criteria, BDD helps identify potential issues early in the development process.
 - Increased Test Coverage: BDD encourages testers to think about the software from a user's perspective, leading to more comprehensive test coverage.
 - Living Documentation: The BDD specifications serve as living documentation that evolves with the software, promoting better understanding throughout the lifecycle.
-

Shift Left Testing

- Shift Left is a philosophy that emphasizes moving testing activities earlier in the Software Development Lifecycle (SDLC). Traditionally, testing might have been a separate phase towards the end of development. Shift left testing aims to integrate testing throughout the entire development process.
- How Does it Work?
 - Shift left testing involves incorporating various testing activities throughout the SDLC. This can include:
 - Unit testing by developers as they write code.
 - Integration testing to ensure different code modules work together seamlessly.
 - User Acceptance Testing (UAT) earlier in the development cycle to get user feedback on prototypes or early builds.
 - Techniques like exploratory testing and BDD (Behavior-Driven Development) can be valuable tools for early-stage testing.
- Benefits of Shift Left Testing:
 - Reduced Costs: Catching defects earlier lowers the overall cost of fixing them compared to waiting until later stages of development.
 - Improved Communication and Collaboration: Shift left testing encourages better communication and collaboration between developers, testers, and other stakeholders throughout the development process.

- Increased Confidence: By proactively testing and addressing issues early on, developers and stakeholders gain more confidence in the final product.
-

Questions to ask yourself when approaching testing

1. What happens if I enter invalid data here?
2. Can I break the expected workflow by performing these actions out of order?
3. Are there any error messages that seem unclear or misleading?
4. Does the system behave consistently across different browsers or devices (if applicable)?
5. How does the system handle unexpected inputs or edge cases?
6. Is the interface intuitive and easy to navigate?
7. Can users easily find the information or features they need?
8. Are there any confusing or cluttered elements on the screen?
9. Is the user feedback (e.g., error messages, loading indicators) clear and helpful?
10. How accessible is the software for users with disabilities?
11. What areas haven't I explored yet?
12. Are there any high-risk functionalities that I should prioritize testing?
13. Based on my observations, what additional test cases can I create?
14. Should I adjust my testing approach based on the discoveries I've made so far?
15. How can I leverage these findings to improve the overall test coverage?
16. What would a new user encounter for the first time using this feature?
17. How might someone with malicious intent try to exploit this system?
18. Can I think of any unusual scenarios or workflows that might expose weaknesses?
19. How can I push the boundaries of the system to see how it reacts under stress?
20. Can I leverage these findings to identify potential improvements for future features?
21. What bugs or defects have I encountered during this exploration session?
22. Are there any usability issues or areas for improvement that I've identified?
23. Can I capture screenshots or recordings to illustrate my observations?
24. How can I clearly document my findings to be communicated effectively to developers or other stakeholders?