Plataformas e Serviços X-Ops
(16233)

Software Testing and AI-Augmented Testing

(adapted from *Engineering Software Products: An Introduction to Modern Software Engineering*, Ian Sommerville, Pearson, 2020)

Topics covered

✧ Foundations of software testing

✧ Test design techniques

✧ Testing in Agile

✧ Automation and AI-augmented testing

Introduction to software testing

Software testing

- ✧ Software testing is a process in which you execute your program using data that simulates user inputs.
- ✧ You observe its behaviour to see whether or not your program is doing what it is supposed to do.
  - ▪ Tests pass if the behaviour is what you expect. Tests fail if the behaviour differs from that expected.
  - ▪ If your program does what you expect, this shows that for the inputs used, the program behaves correctly.
- ✧ If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.

Purpose of software testing

- ✧ **Ensure Quality and Reliability** – Verify that the software meets its requirements and performs consistently under expected conditions.
- ✧ **Detect and Prevent Defects Early** – Identify issues as soon as possible in the development lifecycle, reducing the cost and impact of fixing them.
- ✧ **Improve User Satisfaction** – Deliver a product that is functional, intuitive, and free from disruptive bugs, enhancing the user experience.
- ✧ **Support Maintainability and Scalability** – Ensure the system is easy to update, extend, and adapt to future needs without introducing new defects.

Bugs? What are Those?

- **Bug is an overloaded term.**
  - Does it refer to the bad behavior observed?
  - Is it the source code mistake that led to that behavior?
  - Is it both or either?

Program bugs (1 of 2)

- ✧ If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.

Program bugs (2 of 2)

- ✧ There are two causes of program bugs:
  - ▪ ***Programming errors*** You have accidentally included faults in your program code. For example, a common programming error is an ‘off-by-1’ error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.
  - ▪ ***Understanding errors*** You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn’t include code to handle this.

Faults and Failures

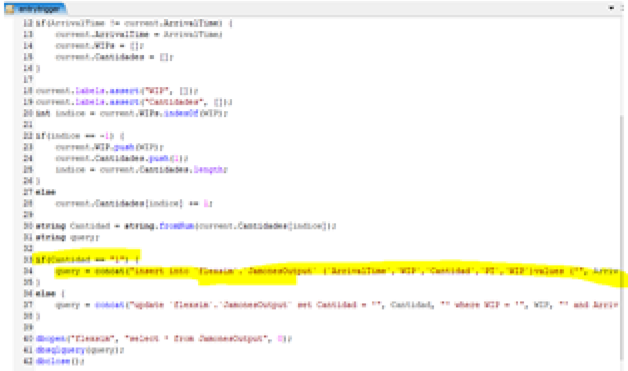
✧ **Failure**

- An execution that yields an incorrect result.

✧ **Fault**

- The problem that caused a failure.
- Mistake in the code, omission from the code, misuse.

✧ **When we *observe a failure*, we try to *find the fault*.**



```
if(ArrivalTime != current.ArrivalTime) {
    current.ArrivalTime = ArrivalTime;
    current.WIPs = [];
    current.Cantidades = [];
}

current.labels.assert("WIP", []);
current.labels.assert("Cantidades", []);
int indice = current.WIPs.indexOf(WIP);

if(indice == -1) {
    current.WIP.push(WIP);
    current.Cantidades.push(1);
    indice = current.Cantidades.length;
}
else
    current.Cantidades[indice] += 1;

string Cantidad = string.fromNum(current.Cantidades[indice]);
string query;

if(Cantidad == "1") {
    query = concat("insert into `flexsim`.`JamonesOutput` (`ArrivalTime`,`WIP`,`Cantidad`,`PT`,`WIP`)values ('", Arriv
}
else {
    query = concat("update `flexsim`.`JamonesOutput` set Cantidad = '", Cantidad, "' where WIP = '", WIP, "' and Arriv
}

dbopen("flexsim", "select * from JamonesOutput", 0);
dbsqlquery(query);
dbclose();
```

When is Software Ready for Release?

Software is ready for release when you can argue that it is ***dependable***.

✧ Correct, reliable, safe, and robust.

✧ Shown through **Verification and Validation (V&V)**.

Verification and Validation

Barry Boehm, inventor of the term “software engineering”, describes them as:

✧ **Verification:**

- “Are we building the product right?”

✧ **Validation:**

- “Are we building the right product?”

Verification and Validation

Verification

Validation

Verification and Validation

Activities that must be performed to consider the software “done.”

- ✧ **Verification:** The process of proving that the software conforms to its specified functional and non-functional requirements.

- ✧ **Validation:** The process of proving that the software meets the customer’s true requirements, needs, and expectations.

Verification

- ✧ Is the implementation consistent with its specification?
  - Does the software work under conditions we set?
  - (usually based on requirements)
- ✧ **Verification is an experiment.**
  - Perform trials, evaluate results, gather evidence.

Verification

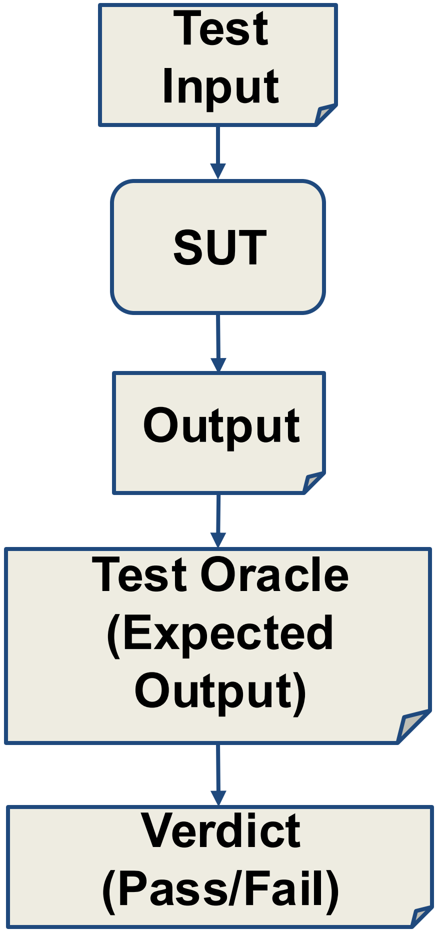
- Is a implementation consistent with a specification?
- “Specification” and “implementation” are roles.
  - Usually source code and requirement specification.
  - But also…
    - Detailed design and high-level architecture.
    - Design and requirements.
    - Test cases and requirements.
    - Source code and user manuals.

Software Testing

✧ An investigation into system quality.

✧ Based on sequences of **stimuli** and **observations**.

- **Stimuli** that the system must react to.
- **Observations** of system reactions.
- **Verdicts** on correctness.



Validation

- ✧ Does the product work in the real world?
  - Does the software fulfill the users’ **actual needs**?
- ✧ Not the same as conforming to a specification.
  - If we specify **two buttons** and implement all behaviors related to those buttons, we can achieve verification.
  - If the user expected **a third button**, we have not achieved validation.

Verification and Validation

- Verification
  - Does the software work as intended?
  - Shows that software is dependable.
- Validation
  - Does the software meet the needs of your users?
  - Shows that software is useful.
  - **This is much harder.**

Verification and Validation

- ✧ Both are important.
  - A well-verified system might not meet the user’s needs.
  - A system can’t meet the user’s needs unless it is well-constructed.
- ✧ This class largely focuses on verification.
  - **Testing is the primary activity of verification**.

Foundations of software testing

Types of testing

Test type	Testing goals
Functional testing	Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.
User testing	Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software’s features and can use these features effectively.
Performance and load testing	Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.
Security testing	Test that the software maintains its integrity and can protect user information from theft and damage.

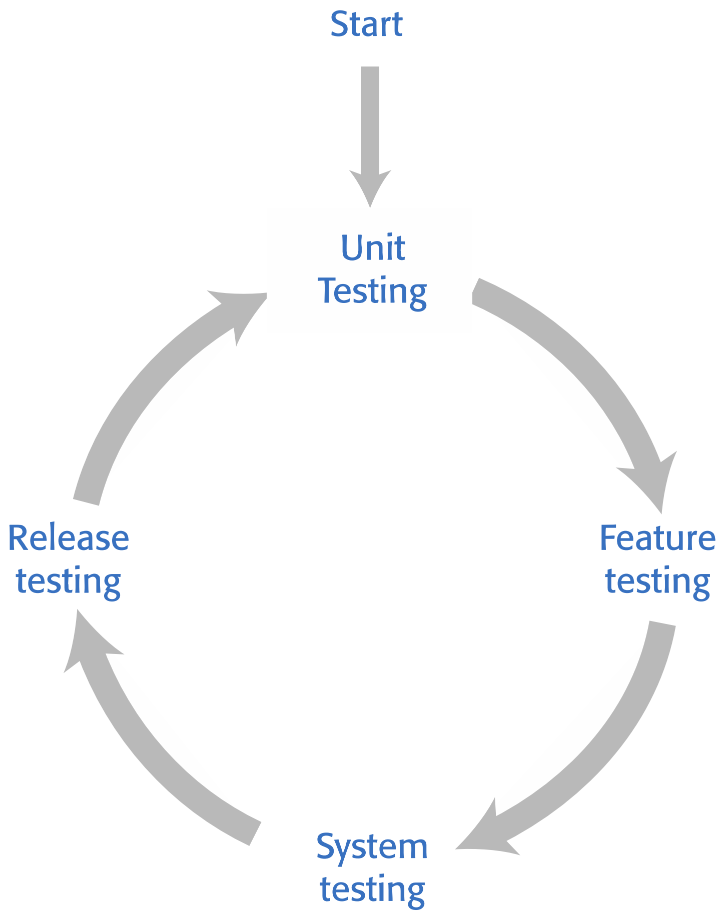
Functional testing (1 of 2)

- ✧ Functional testing involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once.
- ✧ The number of tests needed obviously depends on the size and the functionality of the application.
- ✧ For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.

Functional testing (2 of 2)

- Functional testing is a staged activity in which you initially test individual units of code. You integrate code units with other units to create larger units then do more testing.
- The process continues until you have created a complete system ready for release.

Functional testing



Start

Unit Testing

Feature testing

System testing

Release testing

Functional testing processes (1 of 2)

Testing process	Description
Unit testing	The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.
Feature testing	Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.

Functional testing processes (2 of 2)

Testing process	Description
System testing	Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability, and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.
Release testing	The system is package and release for customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If Devops is used, then the development team is responsible for release testing; otherwise, a separate team has that responsibility.

Unit testing (1 of 2)

- ✧ As you develop a code unit, you should also develop tests for that code.
- ✧ A code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions.
- ✧ Unit testing is based on a simple general principle:
  - If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

Unit testing guidelines (1 of 2)

Guideline	Explanation
Test edge cases	If your partition has upper and lower bounds (e.g., length of strings, numbers, etc.), choose inputs at the edges of the range.
Force errors	Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.
Fill buffers	Choose test inputs that cause all input buffers to overflow.
Repeat yourself	Repeat the same test input or series of inputs several times.
Overflow and underflow	If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.
Don’t forget null and zero	If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.

Unit testing guidelines (2 of 2)

Guideline	Explanation
Keep count	When dealing with lists and list transformations, keep count of the number of elements in each list and check that these are consistent after each transformation.
One is different	If your program deals with sequences, always test with sequences that have a single value.

Feature testing (1 of 2)

- ✧ Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.
  - ▪ For example, if your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.
  - ▪ You may want to check that it gives users the option to sign up for email information about your product.

Feature testing (2 of 2)

- Normally, a feature that does several things is implemented by multiple, interacting, program units.
- These units may be implemented by different developers and all of these developers should be involved in the feature testing process.

Types of feature test (1 of 2)

✧ Interaction tests

- These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.
- These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.
- The integration may also reveal bugs in program units, which were not exposed by unit testing.

Types of feature test (2 of 2)

✧ Usefulness tests

- These test that the feature implements what users are likely to want.
- For example, the developers of a login with Google feature may have implemented an opt-out default on registration so that users receive all emails from a company. They must expressly choose what type of emails that they don't want.
- What might be preferred is an opt-in default so that users choose what types of email they do want to receive.

User stories for the sign-in with Google feature

Story title	User story
User registration	As a user, I want to be able to log in without creating a new account so that I don’t have to remember another login ID and password.
Information sharing	As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don’t want to share this information.
Email choice	As a user, I want to be able to choose the types of email that I’ll get from you when I register for an account.

Feature tests for sign-in with Google

Test	Description
Initial login screen	Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the “Sign-in with Google” link. Test that the login is completed if the user is already logged in to Google.
Incorrect credentials	Test that the error message and retry screen are displayed if the user inputs incorrect Google credentials.
Shared information	Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is canceled if the cancel option is chosen.
Email opt-in	Test that the user is offered a menu of options for email information and can choose multiple items to opt in to emails. Test that the user is not registered for any emails if no options are selected.

System and release testing

- ✧ System testing involves testing the system as a whole, rather than the individual system features.
- ✧ System testing should focus on four things:
  - ▪ Testing to discover if there are unexpected and unwanted interactions between the features in a system.
  - ▪ Testing to discover if the system features work together effectively to support what users really want to do with the system.
  - ▪ Testing the system to make sure it operates in the expected way in the different environments where it will be used.
  - ▪ Testing the responsiveness, throughput, security and other quality attributes of the system.

Scenario-based testing

- ✧ The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.
- ✧ Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system.
- ✧ An end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

Choosing a holiday destination

Andrew and Maria have a two-year-old son and a four-month-old daughter. They live in Scotland and they want to have a holiday in the sunshine. However, they are concerned about the hassle of flying with young children. They decide to try a family holiday planning product to help them choose a destination that is easy to get to and that fits in with their children's routines.

Maria navigates to the holiday planner website and selects the "find a destination" page. This presents a screen with a number of options. She can choose a specific destination or a departure airport and find all destinations that have direct flights from that airport. She can also input the time band that she'd prefer for flights, holiday dates, and a maximum cost per person.

Edinburgh is their closest departure airport. She chooses "find direct flights." The system then presents a list of countries that have direct flights from Edinburgh and the days when these flights operate. She selects France, Italy, Portugal, and Spain and requests further information about these flights. She then sets a filter to display flights that leave on a Saturday or Sunday after 7.30 am and arrive before 6 pm. She also sets the maximum acceptable cost for a flight. The list of flights is pruned according to the filter and is redisplayed. Maria then clicks on the flight she wants. This opens a tab in her browser showing a booking form for this flight on the airline's website.

End-to-end pathways

1. User inputs departure airport and chooses to see only direct flights. User quits.
2. User inputs departure airport and chooses to see all flights. User quits.
3. User chooses destination country and chooses to see all flights. User quits.
4. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User quits.
5. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User selects a displayed flight and clicks through to airline website. User returns to holiday planner after booking flight.

Release testing (1 of 2)

- Release testing is a type of system testing where a system that’s intended for release to customers is tested.
- The fundamental differences between release testing and system testing are:
  - Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.
  - The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that ‘fail’ may be ignored if these have minimal consequences for most users.

Release testing (2 of 2)

- ✧Preparing a system for release involves packaging that system for deployment (e.g. in a container if it is a cloud service) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

Test design techniques

- ✧ Black-box techniques
  - Equivalence partitioning, boundary value analysis, decision tables, state transition testing.
- ✧ White-box techniques
  - Statement, branch, and path coverage.
- ✧ Experience-based techniques
- ✧ Model-based testing

Black-box techniques (1 of 2)

✧ Equivalence Partitioning

- Divides input data into groups (partitions) where system behavior is expected to be the same.
- Reduces the number of test cases while maintaining coverage.

✧ Boundary Value Analysis

- Tests at the edges of input ranges, where errors are most likely to occur.
- Often combined with EP for better coverage.

✧ Decision Table Testing

- Represents complex business rules and input combinations in tabular form.
- Ensures coverage of all possible input/output combinations.

Black-box techniques (2 of 2)

- ✧ State Transition Testing
  - ▪ Validates that the system behaves correctly during changes from one state to another.
  - ▪ Particularly useful for systems with finite states or workflows.
- ✧ Use Case Testing
  - ▪ Derives test cases directly from user scenarios and system use cases.
  - ▪ Ensures the system supports real-world user workflows.

White-box techniques (1 of 2)

✧ Statement Coverage

- Ensures that each executable statement in the code is run at least once during testing.
- Helps identify unused or unreachable code segments.

✧ Branch Coverage

- Verifies that every possible branch (true/false) of each decision point is executed.
- Detects missing logic for certain decision outcomes.

White-box techniques (2 of 2)

✧ Path Coverage

- Ensures that all possible execution paths through the program are tested.
- Provides the highest thoroughness but can be impractical for large systems due to path explosion.

✧ Condition Coverage

- Tests each individual logical condition in a decision for both true and false outcomes.
- Can uncover subtle bugs hidden in compound conditions.

Experience-based techniques

- Error Guessing
  - Relies on tester experience to predict where defects are likely to occur.
  - Often guided by past defect patterns, common pitfalls, and knowledge of system complexity.
- Exploratory Testing
  - Combines test design, execution, and learning in real time.
  - Highly adaptive, allowing testers to respond to findings immediately.
  - Useful in early development stages or when documentation is incomplete.
- Checklist-Based Testing
  - Uses predefined checklists of common issues, risks, or compliance requirements.
  - Helps ensure coverage of typical defect areas without relying solely on memory.

Model-based techniques

- ✧ Models as the Single Source of Truth
  - ▪ All test cases stem from the same system model, ensuring consistency.
- ✧ Direct Test Derivation from Models
  - ▪ Reduces manual effort and guarantees complete traceability to requirements.
- ✧ Automation-Friendly Process
  - ▪ Facilitates automatic test generation and execution using Model-based tools.
- ✧ Requirements Alignment
  - ▪ Ensures tests always reflect the latest system design and specifications.

Types of models

- Structure diagrams:
  - class diagram.
- Behaviour diagrams:
  - use cases, activity, sequence, state.

Opportunities of model-based techniques (1 of 2)

✧ High Coverage with Reduced Manual Effort

- Automatically generating test cases from models allows coverage of more scenarios, including edge and corner cases, that manual testing might miss.
- Reduces repetitive manual work, enabling testers to focus on higher-level analysis and exploratory testing.

✧ Early Defect Detection Before Coding Begins

- Since tests are derived from system models, inconsistencies, ambiguities, or missing requirements can be detected during the design stage.
- This “shift-left” approach minimizes the cost of fixing defects later in the development lifecycle.

Opportunities of model-based techniques (2 of 2)

- ✧ Improved Stakeholder Communication Through Shared Models
  - ▪ Models provide a visual, formalized representation of system behaviour that is easier for non-technical stakeholders to understand compared to raw code or complex test scripts.
  - ▪ Encourages collaboration between business analysts, developers, and testers.
- ✧ Easy Adaptation to Requirement Changes
  - ▪ When the model changes, the test cases can be automatically regenerated, ensuring they always align with the latest requirements.
  - ▪ This reduces the risk of outdated or irrelevant test cases in evolving Agile/DevOps projects.
- ✧ Supports Scalability for Complex Systems
  - ▪ Can handle large-scale systems with multiple modules, dependencies, and cross-functional requirements.
  - ▪ Facilitates integration testing and system-level validation in distributed or multi-domain environments.

Challenges of model-based techniques (1 of 2)

- ✧ Requires Accurate and Complete Models
  - ▪ If the system model is incomplete, incorrect, or outdated, the generated tests will be misleading or irrelevant.
  - ▪ This makes the quality of the model a critical factor for model-driven success.
- ✧ Initial Learning Curve for Teams
  - ▪ Teams need to learn modelling techniques, model-driven tools, and integration workflows.
  - ▪ Requires training for testers who are used to traditional approaches.

Challenges of model-based techniques (2 of 2)

- ✧ Tooling and Model Maintenance Overhead
  - ▪ Model-driven tools can be expensive, and integrating them into existing CI/CD pipelines can be time-consuming.
  - ▪ Models must be continuously updated alongside evolving system requirements, adding to project overhead.
- ✧ Less Effective if Requirements Are Unstable
  - ▪ If requirements change frequently and dramatically, the effort spent on keeping models aligned may outweigh the benefits.
  - ▪ In highly volatile projects, model-driven may need to be combined with more flexible, exploratory approaches.

Testing in Agile

Agile testing principles (1 of 2)

- ✧ Test Early, Test Often
  - ▪ Testing is integrated into every iteration rather than postponed until the end.
  - ▪ The goal is to detect and fix defects as soon as possible, reducing rework costs.
  - ▪ Continuous integration and automated testing support this principle.
- ✧ Collaborative Approach
  - ▪ Testers, developers, and business stakeholders work together to clarify requirements and expectations.
  - ▪ Knowledge sharing ensures that everyone understands the definition of "done" and the quality criteria.

Agile testing principles (2 of 2)

✧ Agile Testing Quadrants

The Agile Testing Quadrants framework helps teams balance different test types:

- Quadrant 1 – *Unit & Component Tests* → Support the team, technology-facing, automated.
- Quadrant 2 – *Functional & Story Tests* → Support the team, business-facing, automated or manual.
- Quadrant 3 – *Exploratory & Usability Testing* → Critique the product, business-facing, manual.
- Quadrant 4 – *Performance, Security, & Other Non-Functional Tests* → Critique the product, technology-facing, often automated.

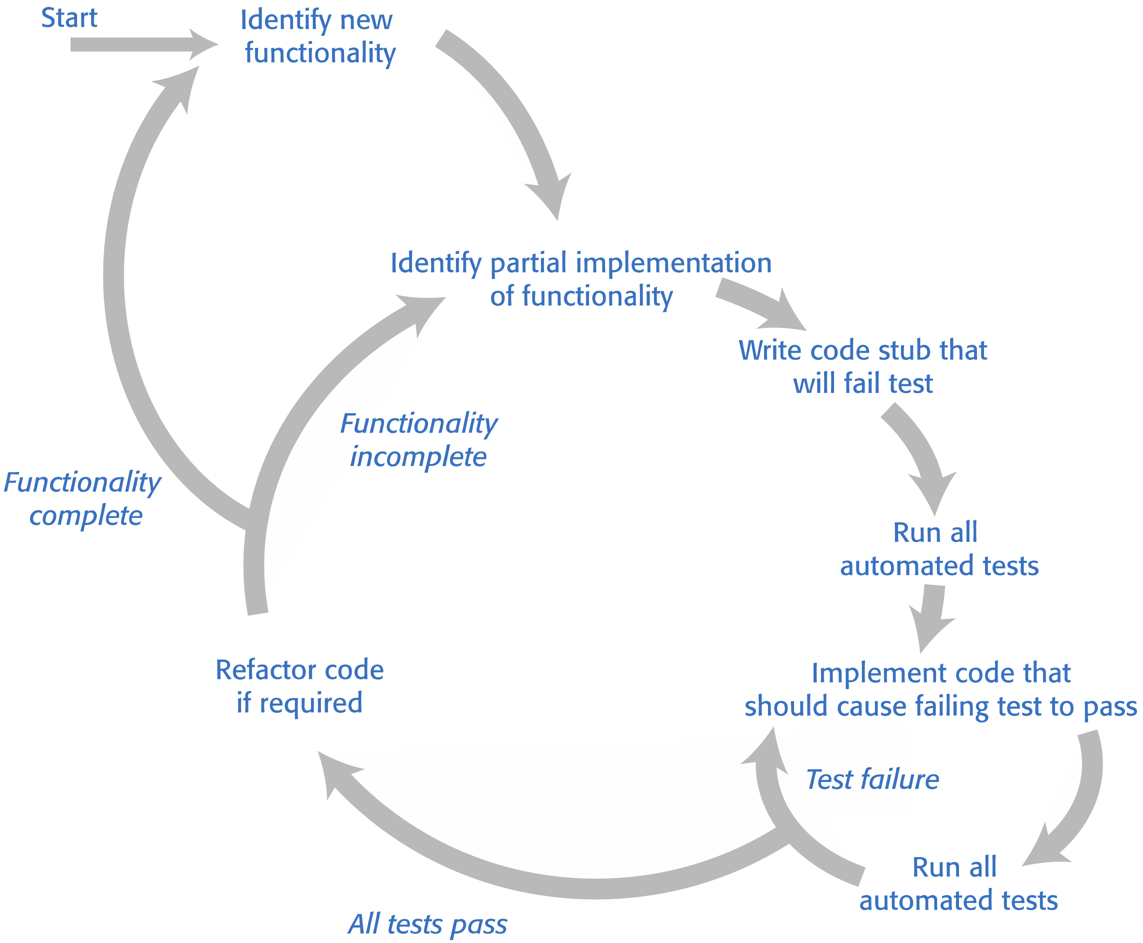
Test-driven development (1 of 2)

- ✧ Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.
- ✧ It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.

Test-driven development (2 of 2)

- ✧ Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- ✧ Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

Test-driven development


Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Run all automated tests
Test failure
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete

"

Stages of test-driven development (1 of 2)

Activity	**Description**
Identify partial implementation	Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
Write mini-unit tests	Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
Write a code stub that will fail test	Write incomplete code that will be called to implement the mini-unit. You know this will fail.
Run all automated tests	Run all existing automated tests. All previous tests should pass. The test for the incomplete code should fail.
Implement code that should cause the failing test to pass	Write code to implement the mini-unit, which should cause it to operate correctly.

Stages of test-driven development (2 of 2)

Activity	Description
Rerun all automated tests	If any tests fail, your code is incorrect. Keep working on it until all tests pass.
Refactor code if required	If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Benefits of test-driven development (1 of 2)

- ✧ It is a systematic approach to testing in which tests are clearly linked to sections of the program code.
  - ▪ This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code. In my view, this is the most significant benefit of TDD.
- ✧ The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.

Benefits of test-driven development (2 of 2)

- ✧ Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
- ✧ It is argued that TDD leads to simpler code as programmers only write code that’s necessary to pass tests. They don’t over-engineer their code with complex features that aren’t needed.

Behaviour-driven development (BDD)

- ✧ Extends TDD with natural language specifications
  - BDD builds on top of TDD, but instead of only developer-focused unit tests, it expresses acceptance criteria in plain language so all stakeholders understand.
- ✧ Promotes collaboration between technical and non-technical stakeholders
  - Business analysts, product owners, QA, and developers work together to define “what good looks like” before coding starts.
- ✧ Uses Gherkin syntax: Given – When – Then
  - Scenarios are written in a structured format:
    - **Given** — the starting context (preconditions)
    - **When** — an action or event that occurs
    - **Then** — the expected, observable outcome.

Benefits of behaviour-driven development (1 of 2)

- ✧ Shared Understanding
  - Creates a **common language** between business, QA, and developers.
  - Reduces ambiguity in requirements by using concrete examples.
- ✧ Better Requirements Quality
  - Flushes out edge cases early through collaborative discussion.
  - Makes acceptance criteria **testable** from the start.
- ✧ Living Documentation
  - Scenarios remain **up-to-date** because they run in CI.
  - Serves as a **single source of truth** for how the system behaves.
- ✧ Faster Feedback & Fewer Defects
  - Detects mismatches between expectations and implementation early.
  - Encourages **outside-in** design, focusing on business outcomes.

Benefits of behaviour-driven development (2 of 2)

- ✧ Improves Test Coverage Where It Matters
  - Targets **critical business rules** rather than just code paths.
  - Encourages covering both **happy paths** and **edge cases**.
- ✧ Supports Agile & DevOps
  - Integrates into CI/CD pipelines with automated acceptance tests.
  - Facilitates incremental delivery with confidence.
- ✧ Encourages Maintainable Tests
  - Uses **domain language**, making tests easier to read and update.
  - Reduces brittle, UI-driven test cases by focusing on service-level checks.

Challenges of behaviour-driven development

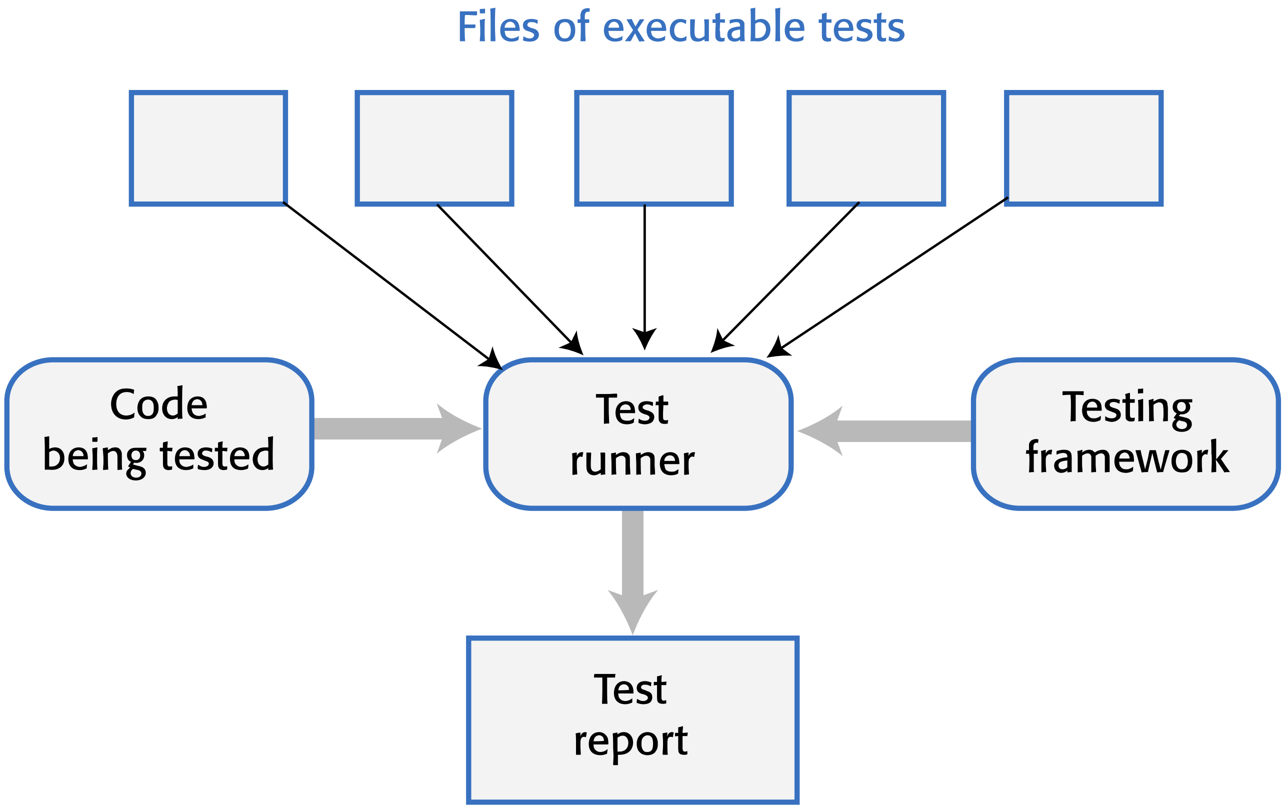
- ✧ **Cultural Shift Required**, teams must adopt collaborative habits.
- ✧ **Upfront Time Investment**, scenario discovery and example mapping take time before coding starts.
- ✧ **Risk of Over-Specification**, too many low-value scenarios can slow feedback and add maintenance cost.
- ✧ **Poor Scenario Quality**, if written in technical terms or without domain clarity, they lose value.
- ✧ **Flaky Tests,** reliance on unstable data, UI automation, or external dependencies can cause failures unrelated to real defects.
- ✧ **Tooling Learning Curve**, teams need to learn Gherkin syntax, step definitions, and CI integration.
- ✧ **Maintenance Overhead**, keeping scenarios, step definitions, and documentation in sync with evolving features requires discipline.

Automation and AI-augmented testing

Test automation

- ✧ Automated testing is based on the idea that tests should be executable.
- ✧ An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- ✧ You run the test and the test passes if the unit returns the expected result.
- ✧ Normally, you should develop hundreds or thousands of executable tests for a software product.

Automated testing



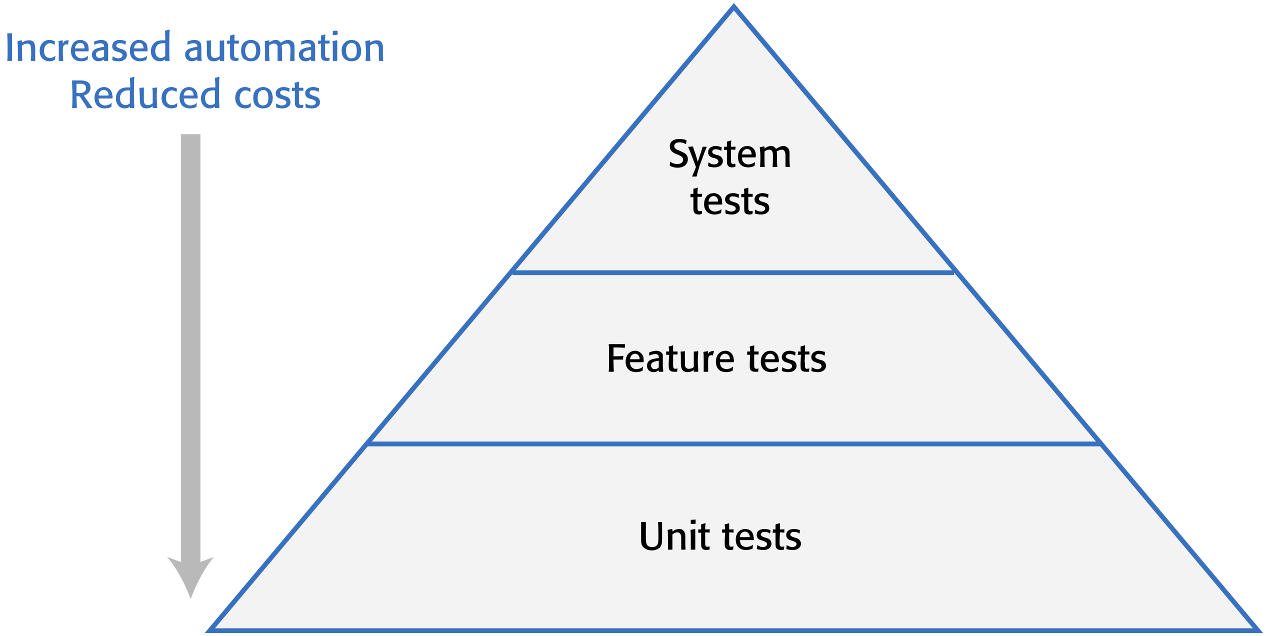
Automated tests (1 of 2)

- ✧ It is good practice to structure automated tests into three parts:
    - **Arrange** You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
    - **Action** You call the unit that is being tested with the test parameters.
    - **Assert** You make an assertion about what should hold if the unit being tested has executed successfully.

Automated tests (2 of 2)

- ✧ If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

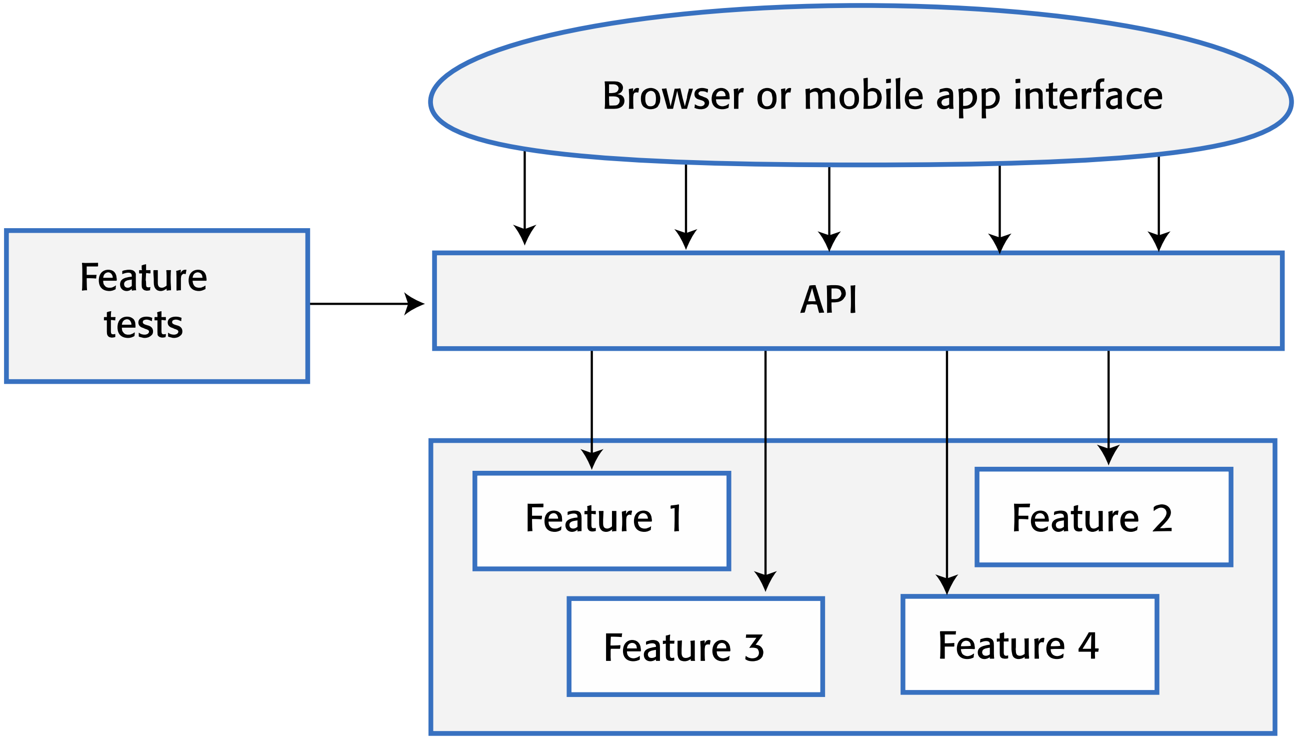
The test pyramid



Automated feature testing

- ✧ Generally, users access features through the product’s graphical user interface (GUI).
- ✧ However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- ✧ The feature tests can then access features directly through the API without the need for direct user interaction through the system’s GUI.
- ✧ Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

Feature testing through an API



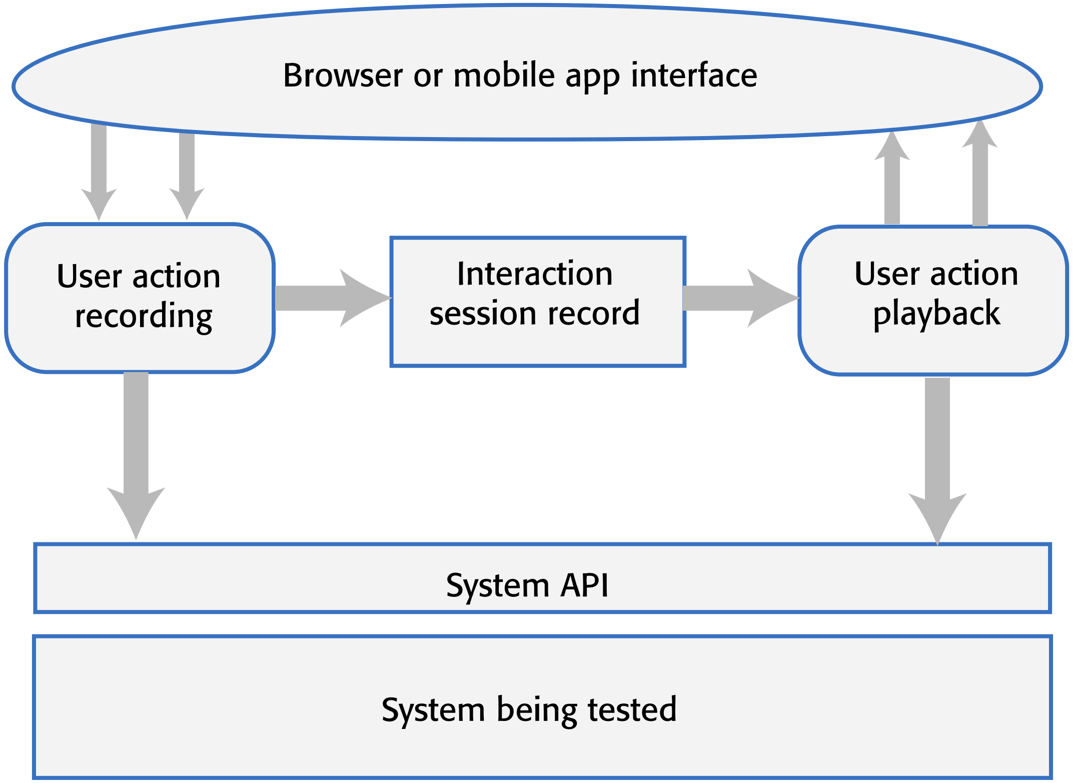
System testing (1 of 2)

- ✧ System testing, which should follow feature testing, involves testing the system as a surrogate user.
- ✧ As a system tester, you go through a process of selecting items from menus, making screen selections, inputting information from the keyboard and so on.
- ✧ You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.

System testing (2 of 2)

- Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
  - To avoid these problems, testing tools have been developed that can record a series of actions and automatically replay these when a system is retested

Interaction recording and playback



Browser or mobile app interface

User action recording

Interaction session record

User action playback

System API

System being tested

Testing in CI/CD pipelines

- **Automated tests executed at every code integration**. Means that as soon as new code is committed or merged (in CI), automated test suites run to verify functionality.
- **Types: Unit, Integration, UI, Performance.** A balanced test strategy includes:
  - **Unit tests**, for fast, isolated code checks
  - **Integration tests**, to ensure components work together
  - **UI tests**, for end-to-end user flows
  - **Performance tests**, to check speed, scalability, and stability
- **Ensures fast feedback and reduces integration risks.** Early detection of issues prevents defects from piling up, making integration smoother and safer.

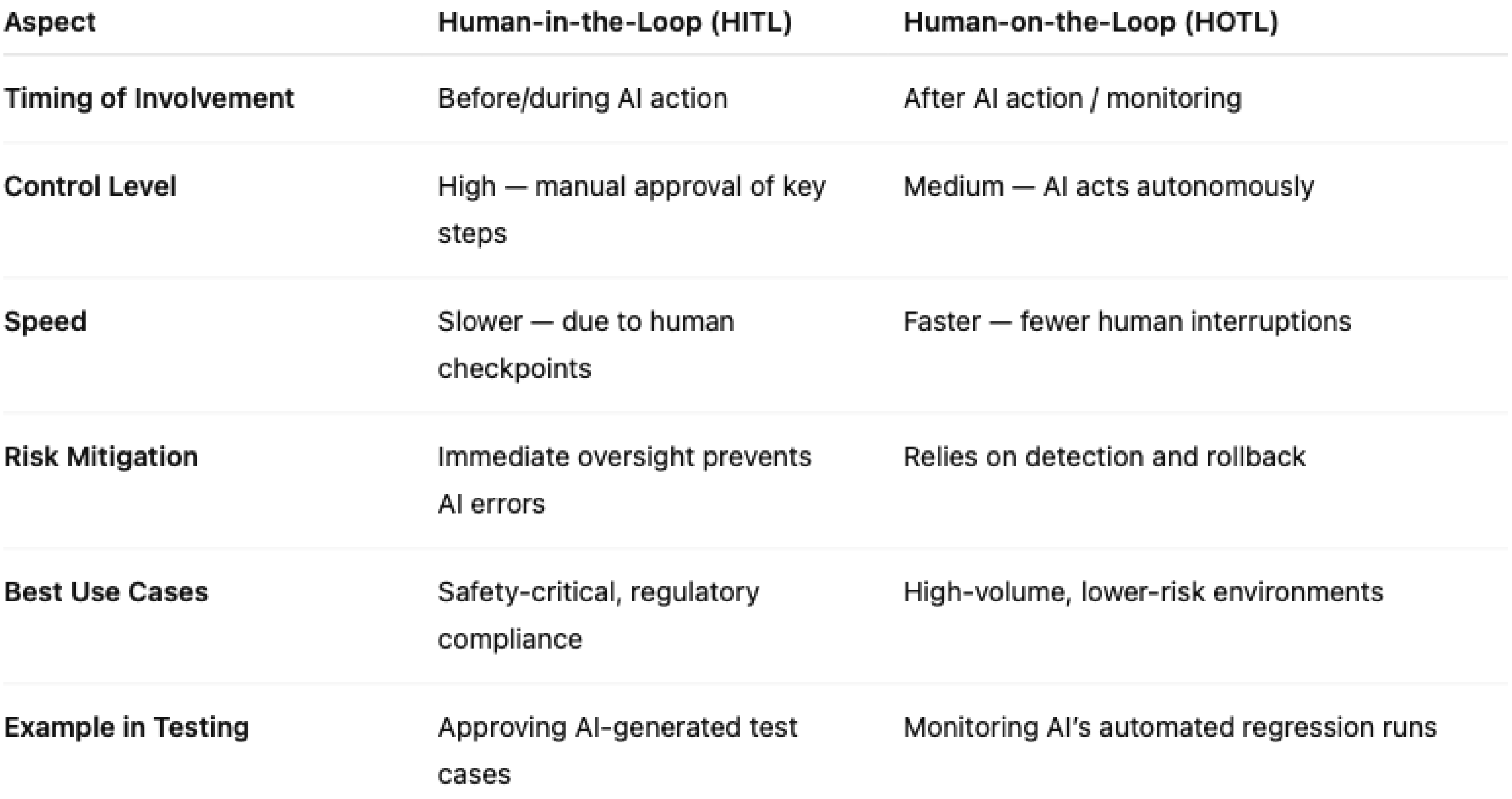
DevOps test automation strategy

- ✧ **Use Infrastructure as Code (IaC) for consistent environments.** Provision and configure test environments automatically (e.g., Terraform, Ansible, CloudFormation) to avoid “works on my machine” issues.
- ✧ **Manage test data for reproducibility.** Seed known datasets, use data snapshots, or synthetic data to ensure consistent test results.
- ✧ **Integrate monitoring and rollback mechanisms.** Detect failures quickly in test or production, and have automated rollback procedures to revert to a stable state.

AI-augmented testing in Agile/DevOps

AI-augmented testing
AI-augmented testing is the application of **artificial intelligence** and **machine learning** to **assist or enhance** software testing activities, from **planning** to **execution and analysis**. It’s **human-in-the-loop**: testers still control the process, but AI accelerates repetitive tasks, finds patterns, and suggests optimizations.

Human-in-the-Loop *vs* Human-on-the-Loop



Aspect	Human-in-the-Loop (HITL)	Human-on-the-Loop (HOTL)
Timing of Involvement	Before/during AI action	After AI action / monitoring
Control Level	High — manual approval of key steps	Medium — AI acts autonomously
Speed	Slower — due to human checkpoints	Faster — fewer human interruptions
Risk Mitigation	Immediate oversight prevents AI errors	Relies on detection and rollback
Best Use Cases	Safety-critical, regulatory compliance	High-volume, lower-risk environments
Example in Testing	Approving AI-generated test cases	Monitoring AI's automated regression runs

When to use HITL vs HOTL in testing

✧ Choose HITL if:

- AI is new to the team and trust is low.
- Application under test has **safety-critical** or **compliance-heavy** requirements.
- Tests require deep domain expertise to validate.

✧ Choose HOTL if:

- AI has proven consistent reliability in prior cycles.
- You need **maximum execution speed** for large regression suites.
- Failures can be **quickly rolled back** without severe consequences.

Lifecycle stages and AI capabilities (1 of 5)

✧ Test Creation

- **Natural language to test cases**: Large Language Models (LLMs) convert user stories, requirements, or acceptance criteria into structured test scripts (e.g., Gherkin, Cypress scripts).
- **Model-based generation**: AI builds state models from system interactions to generate complete test coverage.
- **Exploratory suggestion**: AI proposes additional scenarios based on defect patterns or unusual workflows.
- **Example tools**:
  - GitHub Copilot generates unit tests from function code.
  - Testim & Functionize for UI test flows.

Lifecycle stages and AI capabilities (2 of 5)

✧ Test Data Management

- **Synthetic data generation**: AI creates realistic but anonymized datasets to preserve privacy.
- **Data pattern analysis**: Ensures coverage across all value ranges, boundary conditions, and common production patterns.
- **Self-updating datasets**: ML updates data sets as application domains evolve.
- **Example tools:** Tonic.ai, GenRocket for synthetic test data.

Lifecycle stages and AI capabilities (3 of 5)

✧ Test Execution & Maintenance

- **Self-healing locators**: AI dynamically adapts test scripts when UI element identifiers change.
- **Impact-based test selection**: ML predicts which tests to run based on code diffs and commit history.
- **Dynamic environment provisioning**: AI optimizes resource allocation for faster runs.
- **Example Tools:**
  - Mabl, Testim for self-healing UI tests.
  - Launchable for ML-driven test selection.

Lifecycle stages and AI capabilities (4 of 5)

✧ Defect Analysis

- **Failure clustering**: Groups related failures, reducing triage noise.
- **Root cause prediction**: Correlates logs, commits, and failures to suggest likely causes.
- **Flake detection**: Identifies tests failing intermittently without code changes.
- **Example tools:**
  - ReportPortal.io for AI-based failure clustering.
  - BuildPulse for flake detection.

Lifecycle stages and AI capabilities (5 of 5)

✧ Continuous Improvement

- **Coverage gap detection**: AI identifies untested code paths or user journeys.
- **Suite optimization**: Flags redundant tests and ranks tests by historical defect yield.
- **Adaptive regression suites**: Dynamically adjust scope to meet execution time limits.

Benefits of AI-augmented testing

- ✧ **Speed**: Cuts time for writing, running, and maintaining tests.
- ✧ **Resilience**: Self-healing reduces breakages from minor UI changes.
- ✧ **Smarter prioritization**: Test selection based on historical defect patterns focuses effort where risk is highest.
- ✧ **Better coverage**: AI spots scenarios humans might miss.

Challenges of AI-augmented testing

- ✧ **Trust**: AI suggestions may be incorrect, requiring review.
- ✧ **Bias**: If training data is skewed, generated tests may miss critical cases.
- ✧ **Integration complexity**: AI tools must work smoothly with CI/CD pipelines.
- ✧ **Cost & compute**: ML models, especially large-scale analytics, can require significant resources.
- ✧ **Data security**: Using production-like data in AI pipelines must comply with privacy laws.

Key points (1 of 3)

- The aim of software testing is to **find defects** and **demonstrate expected behaviour**.
- Common types of testing:
  - **Functional testing** — unit, feature, and system tests to verify behavior.
  - **User testing** — validates real-world usability and expectations.
  - **Load & performance testing** — measures speed, scalability, and stability.
  - **Security testing** — identifies vulnerabilities and mitigates risks.
  - **Unit testing** focuses on small, isolated units (functions, methods) with clear responsibilities.
  - **Feature testing** ensures individual system features behave as intended.
  - **System testing** checks for unwanted interactions and environment compatibility.

Key points (2 of 3)

- ✧ **User stories** and acceptance criteria can guide test design, especially in Agile environments.
- ✧ **Test automation** ensures consistent, repeatable verification with faster feedback loops.
- ✧ **Test-driven development (TDD)** — write executable tests first, then develop code to pass them.
- ✧ **Behaviour-driven development (BDD)** — extends TDD with **natural language specifications** and **Gherkin syntax** to align business and technical perspectives.
- ✧ **Testing in CI/CD** — automated tests at every integration detect issues early, reduce integration risks, and support rapid delivery.

Key points (3 of 3)

- ✧ **AI-Augmented Testing** integrates ML/AI into the testing lifecycle:
  - ▪ Generates tests from natural language requirements.
  - ▪ Creates synthetic test data and self-heals broken scripts.
  - ▪ Prioritizes tests using defect prediction and coverage gap analysis.
- ✧ **Human-in-the-Loop (HITL)** vs **Human-on-the-Loop (HOTL)**:
  - ▪ HITL — human reviews/approves AI outputs **before execution** (best for high-risk or early adoption).
  - ▪ HOTL — human monitors autonomous AI testing and intervenes **only if needed** (best for mature, large-scale pipelines).
- ✧ Benefits: speed, smarter prioritization, better coverage.
- ✧ Challenges: trust, bias, integration complexity, cost, and data security.

HOW
WHY
HOW
WHEN
WHERE
WHO
WHERE
WHERE
WHY
WHO
WHO
WHEN
WHAT
WHAT
WHY
WHAT
WHEN
HOW
WHAT
WHEN
WHO
WHERE
WHY
HOW