universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

Software Engineering

Advance Software Testing

2025/2026

Outline

- Different Types of Tests (Review)
  - Static
  - Unit
  - Integration
  - E2E
- Integrate with CI-CD pipelines
- Test in production

Definitions & Test Process

Problems for software without tests

Product

- Complete solution
- Increasing features, user base and clients
- Many users online
- …

Bugs found in production

- **Puts projects at risk**
- **High chances of regressions**

`cost(init) << cost(change)` (Kent Beck)

Lack of confidence in change

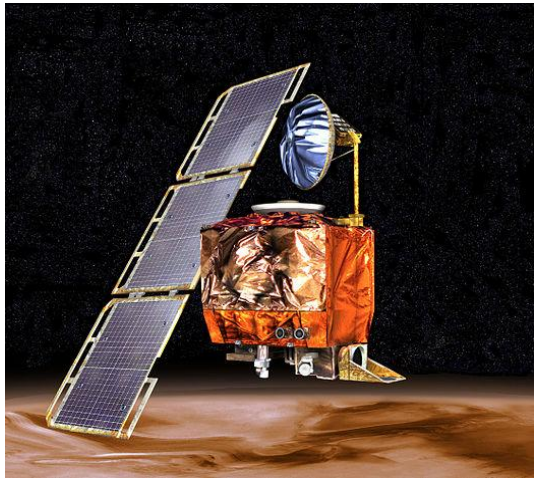
- **Feature requests are always coming**

Manual testing

- **Very slow and error prone**
- **Not affordable**

Let’s start by a real story

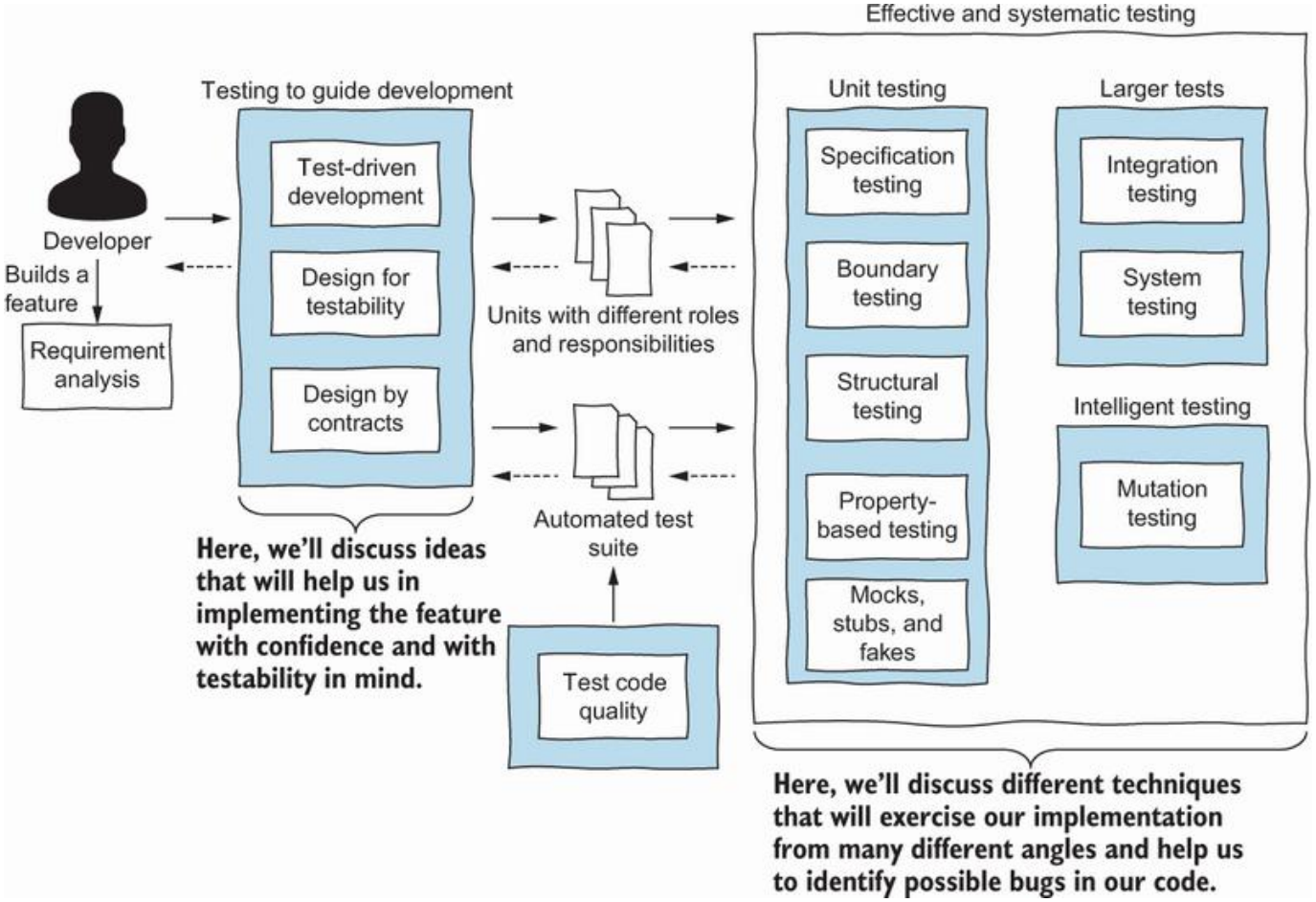
The ***Mars Climate Orbiter*** (formerly the **Mars Surveyor '98 Orbiter**) was a 638-kilogram (1,407 lb)[1] robotic space probe launched by NASA on December 11, 1998 to study the Martian climate, Martian atmosphere, and surface changes and to act as the communications relay in the Mars Surveyor '98 program for Mars Polar Lander. However, on September 23, 1999, communication with the spacecraft was permanently lost as it went into orbital insertion. The spacecraft encountered Mars on a trajectory that brought it too close to the planet, and it was either destroyed in the atmosphere or escaped the planet's vicinity and entered an orbit around the sun.[2] An investigation attributed the failure to a measurement mismatch between two software systems: metric units by NASA and non-metric ("English") units by spacecraft builder Lockheed Martin.[3]

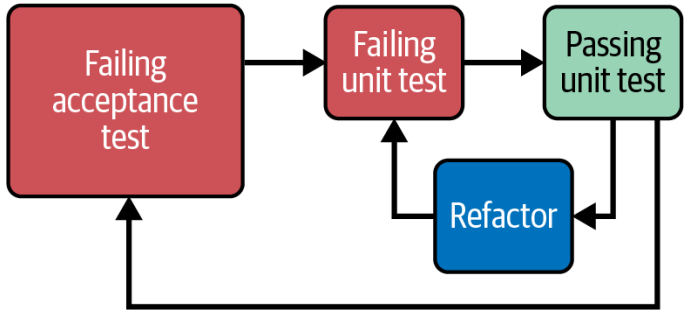


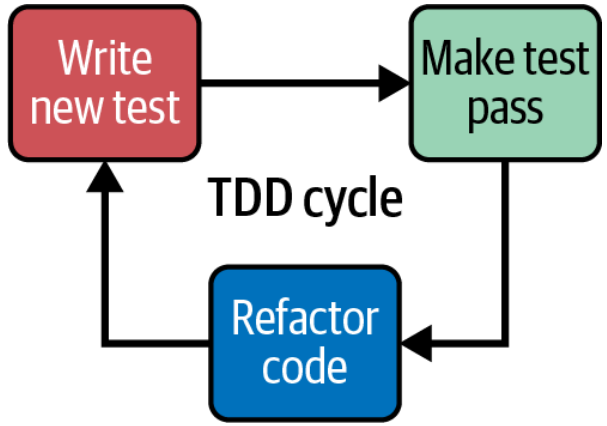
An investigation attributed the failure to a **measurement mismatch between two software systems**: metric units by NASA and non-metric ("English") units

https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

Test Process







Test Process

- Test programs on different levels, ranging from testing small methods to testing entire systems with databases and web services. Each level has advantages and disadvantages.
- Exhaustive testing is impossible (*). The life of a tester involves making trade-offs about how much testing is needed.
- **It is important to define the exit criteria:** The set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task that have not been finished. Exit criteria are used to report against and to plan when to stop testing.

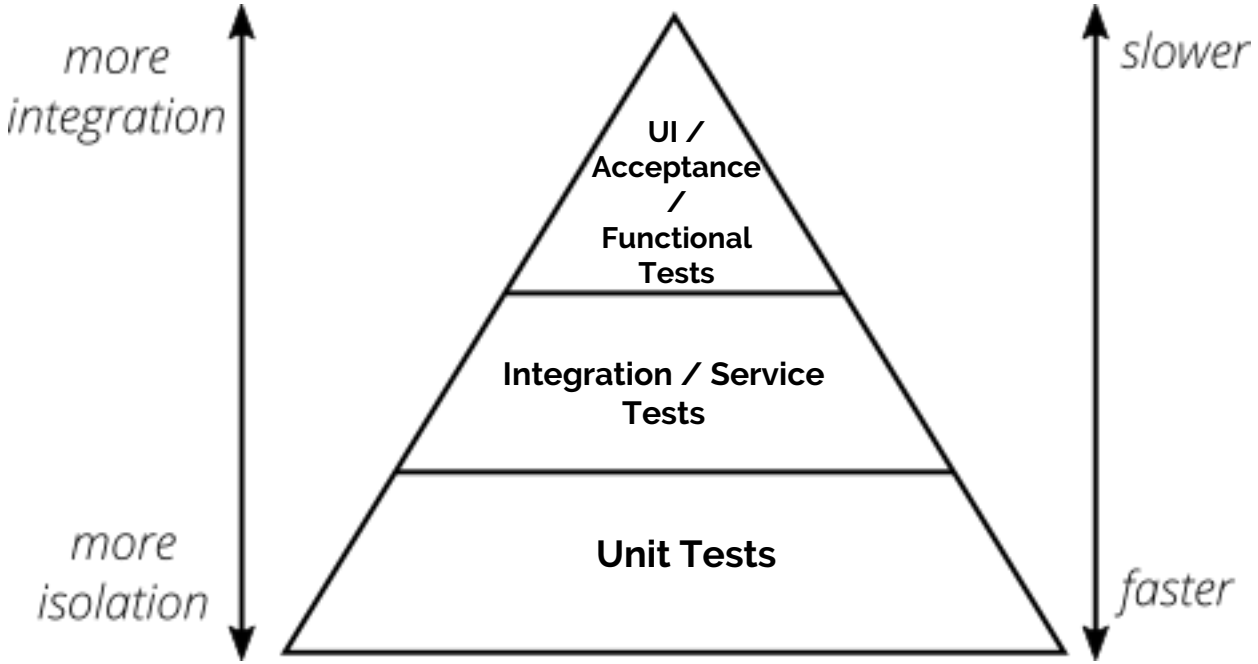
  From ISTQB (International Software Testing Qualifications Board's) Glossary

(*) in the LLMs era it seems that machine can cover almost everything, but dimension of the software should be considered

Properties of Tests

- Testability
  - The ability to support acceptance criteria
  - e.g. assert expected behavior
- Portability
  - Can run on different machines without changes (CI)
  - Do not depend on specific machine resources
  - e.g. files in external paths
- Understandability
  - The purpose of the test is clear
- Reliability
  - The ability to expect it to perform its function reliably

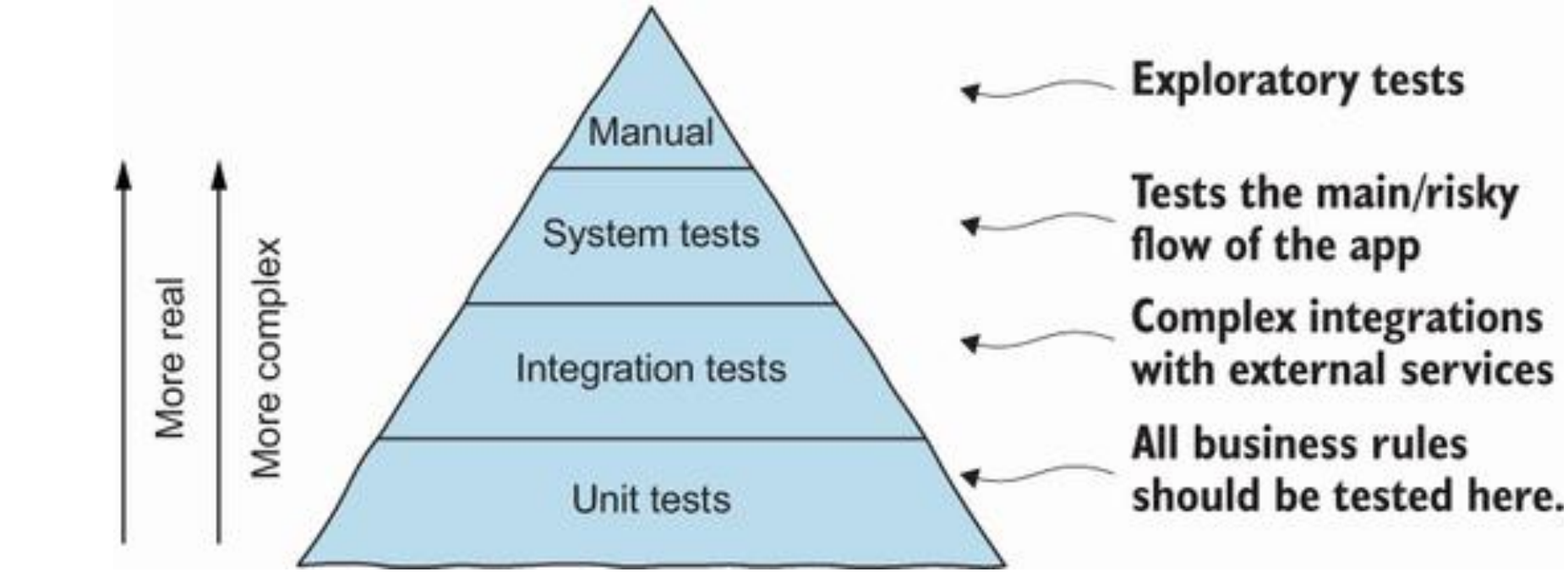
Test Pyramid



"Due to its simplicity the essence of the test pyramid serves as a good rule of thumb"

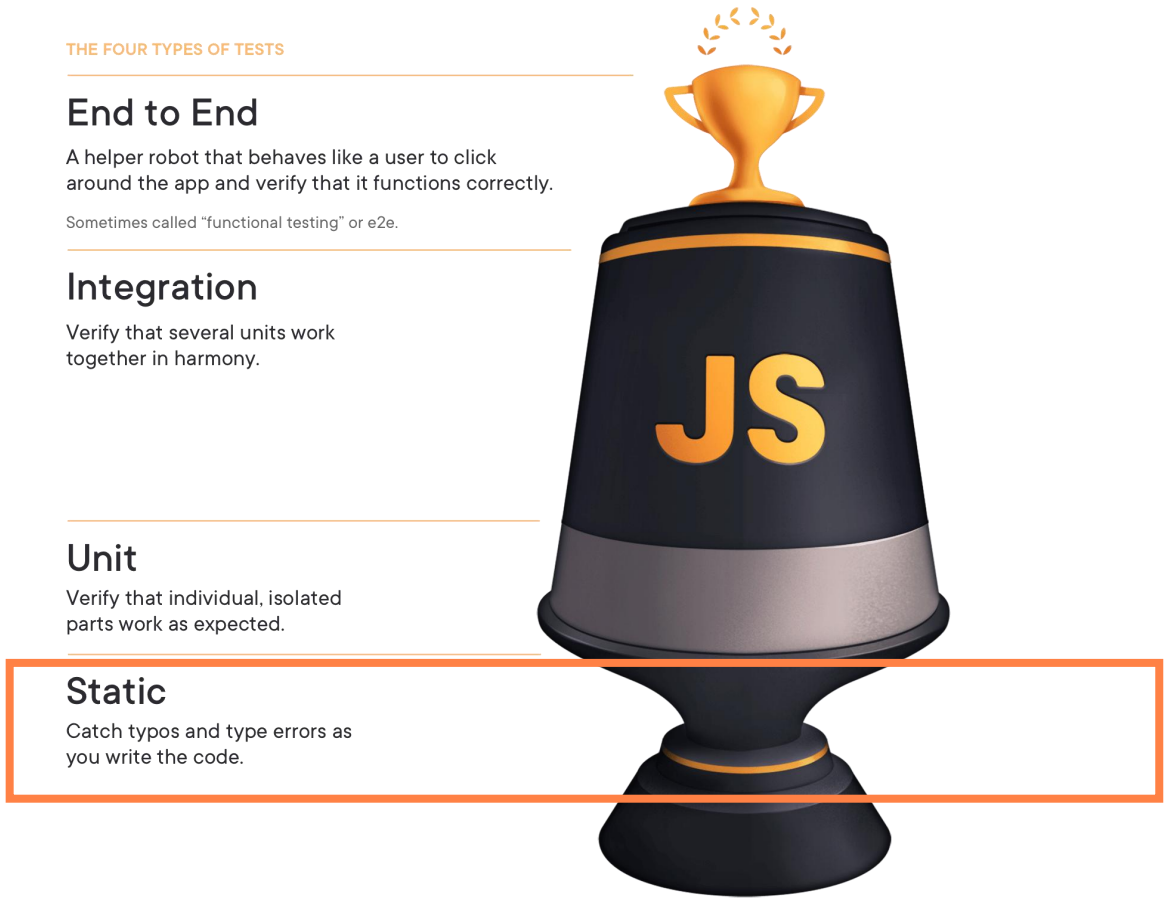
Martin Fowler: The Practical Test Pyramid

Test Pyramid



Another version of pyramid, by Mauricio Aniche

The four types of tests



Kent C. Dodds: The Test Trophy

Different type of testing

Static Testing

Catch errors as code is written.

Assistance in conveying the programmer's thoughts into code.

E.g. JavaScript:

Linter	ESLint
Type checker	TypeScript
Formatter	Prettier

(lint example: duplicate key names in object literals)

Good to set up as a pre-commit hook.

Static Testing in TypeScript

```
export function fibonacci(n) {
  if (n <= 1) {
    return 0;
  } else if (n === 2) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

noImplicitAny – will make this function returns error, due to the type checks.

Static Testing in TypeScript

```typescript
export function fibonacci(n: number): number {
  if (n <= 1) {
    return 0;
  } else if (n === 2) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Static testing detects issues such as missing types (e.g., implicit any in TypeScript) at compile time, before the code is executed, improving reliability and reducing runtime errors.

Static Testing in Java

Don’t use raw types!

- Legacy construct, doesn’t check types
- Add <> to constructors

```
Pair<String, Integer> pair1 = new Pair("score", 5);
Pair<String, Integer> pair2 = new Pair("score", "five"); // Compiles. Oops!
```

✗

```
Pair<String, Integer> pair1 = new Pair<>("score", 5);
Pair<String, Integer> pair2 = new Pair<>("score", "five"); // Does not compile
```

✓

Type mismatch: cannot convert from `Pair<String, String>` to `Pair<String, Integer>`

Static Testing in Java

Don’t use raw types!

- Legacy construct, doesn’t check types
- Add <> to constructors
- If don’t care, use wildcard ? in declarations

```
List list1 = Arrays.asList("one", "two", "five");
List<Integer> list2 = list1; // Compiles. Oops!
```

✗

```
List<?> list1 = Arrays.asList("one", "two", "five");
List<Integer> list2 = list1; // Type mismatch: cannot convert from List<capture#1-of ?> to List<Integer>
```

✓

Static Testing in Python

You can add type annotations (since Python 3.5)

- Use a type verification tool (mypy)

```
from typing import Literal

def process(tokenType: Literal[“website", “page"]):
    if (tokenType == “website"):
        processWebsite()
    else:
        processPge()
```

Static Testing

Limited to the type system

- It can’t describe complex semantic contracts
- Assumptions could be wrong

Some lints are not useful

- Can be opted out
- Strike the right balance

Could hinder quick prototyping

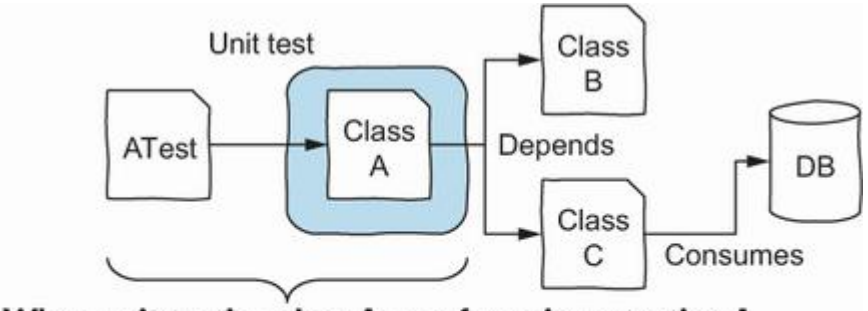
- E.g. TypeScript is still quite flexible
- Transpilation step is sometimes required

Unit Testing

- Testing software units in isolation (methods, classes, or small logical groups).
- Verify the behavior of independent, isolated components.
- Unit?
  - According to Roy Osherove (2009): “A unit test is an automated piece of code that invokes a unit of work in the system… to achieve one single logical purpose that can be verified.”
- Great case coverage and significance / Faster root cause analysis
- Fast and simple

Unit Testing

- Handing dependencies:
  - Units should avoid reliance on external systems (databases, services).
  - If unavoidable, use stubs/mocks to simulate dependencies and keep tests isolated.



When unit testing class A, our focus is on testing A, as isolated as possible from the rest! If A depends on other classes, we have to decide whether to simulate them or to make our unit test a bit bigger.

Unit Testing

Does not answer whether components work well together.

Expectations might be unaligned with reality

Not always feasible

- Use of global context
- Tightly dependent on other components
- Writing test doubles (mocks/stubs)

Integration Testing

- Tests interactions between multiple components.
- Focuses on whether components **communicate correctly** (e.g., DAO ↔ database).
- More realistic than unit tests but still not a full system test.
- **Characteristics**
  - Validates **contracts** between our code and external systems (databases, web services, files, etc.).
  - Example: submitting SQL queries to a real database instead of mocking it.

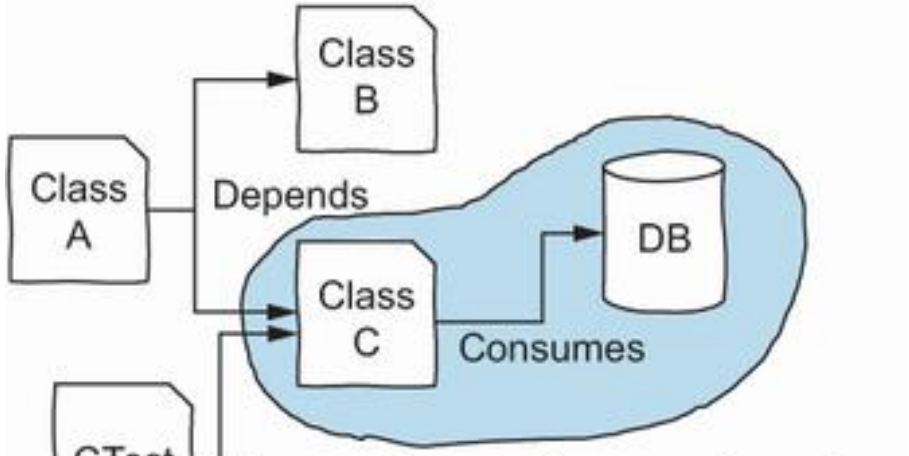
Integration Testing

- **Advantages**
  - Ensures correctness of component communication.
  - Targets critical integration points without requiring full system execution.
- **Disadvantages**
  - **Harder to write** than unit tests (requires setup, teardown, and data preparation).
  - **Slower execution** due to reliance on external components.
- **Scope**
- Tests involve:
  - **Our component** under test.
  - **External component** it integrates with.

Integration Testing

Verify that several units work together in harmony.

- Many kinds of tests
- Mocking may still be important
- A trade-off
  - Coverage/Focus
  - Feasibility



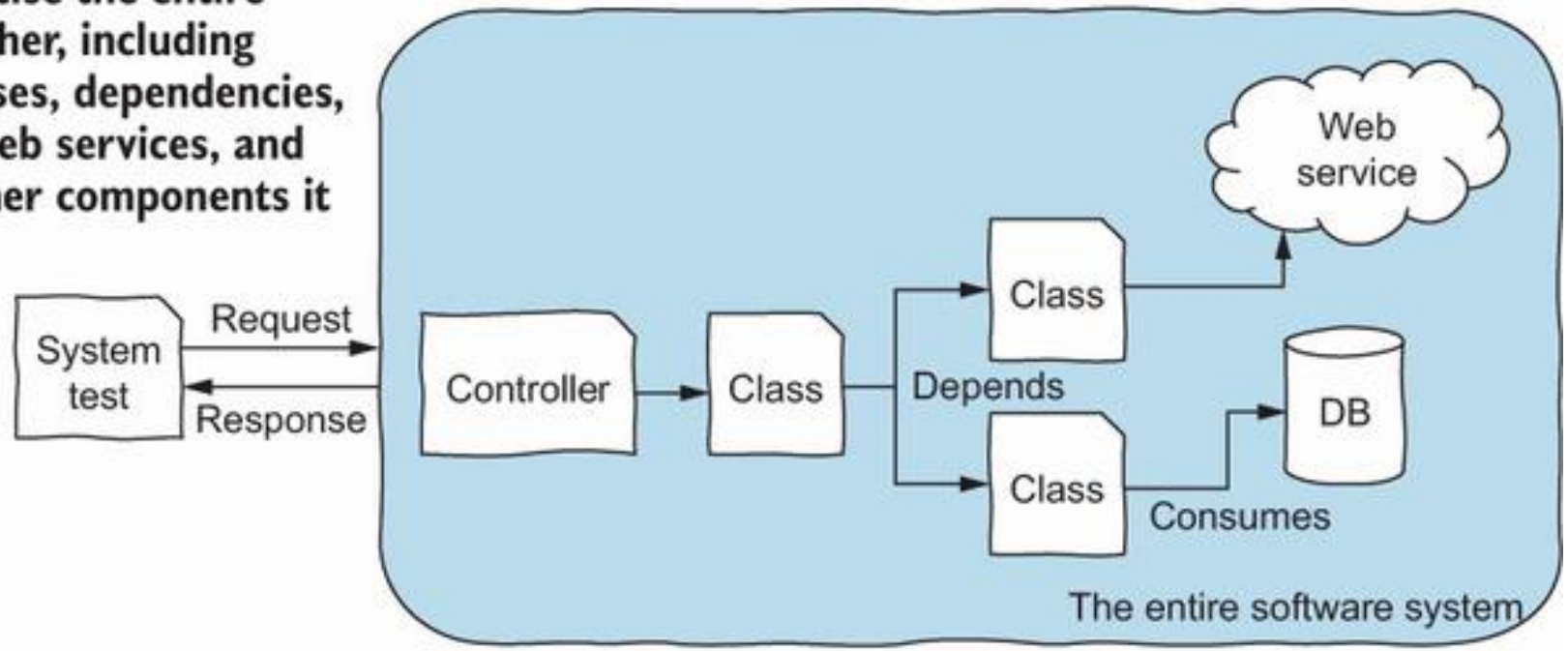
Integration testing exercises the integration between a component of your system and some external component (e.g., a database).

System testing

- Runs the **entire software system** (databases, frontend, backend, etc.) as a whole.
- **Black-box approach**: focus on input/output, not internal implementation.
- **Advantages**
  - Realistic: simulates how end users interact (web page, form, API call).
  - Builds higher confidence in overall system behavior.
- **Disadvantages**
  - **Slow**: starting full system, containers, and services takes time.
  - **Complex setup**: databases, authentication, and test data preparation required.
  - **Flaky tests**: timing/network variations may cause inconsistent results.

System testing

When system testing, you want to exercise the entire system together, including all of its classes, dependencies, databases, web services, and whatever other components it may have.



Server Mocking/Stubbing

- Surrounding the tested component with stubs
  - Better isolation
  - Less components to set up and worry about
  - No external resources
- In JavaScript:
  - msw: A modern REST/GraphQL API mocking tool
  - nock: An HTTP server mock
- In Java:
  - WireMock & others

E2E Testing in JavaScript

Goal: Ensure the full application flow behaves as expected in real usage scenarios.

- **Tests the system from the user's perspective, end-to-end.**
- **Verifies that frontend, backend, database, and external services work together correctly.**

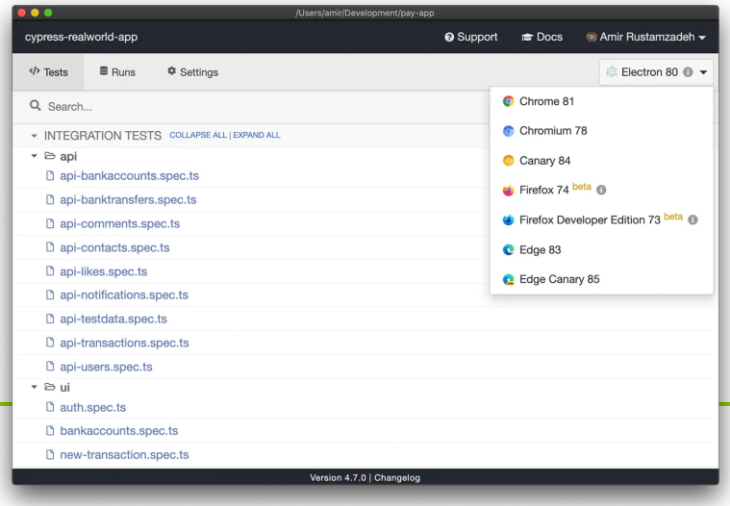
Used by several frameworks, eg nighwatch.js:

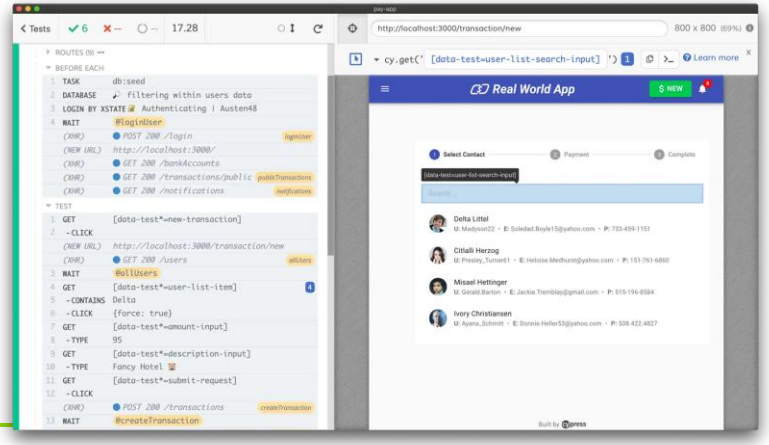
Selenium

- API to a browser driver (chromedriver.exe) / WebDriver Protocol
- Not a test framework (add test framework, driver, ...)
- Complicated API
  - Each operation requires wiring data to/from the driver

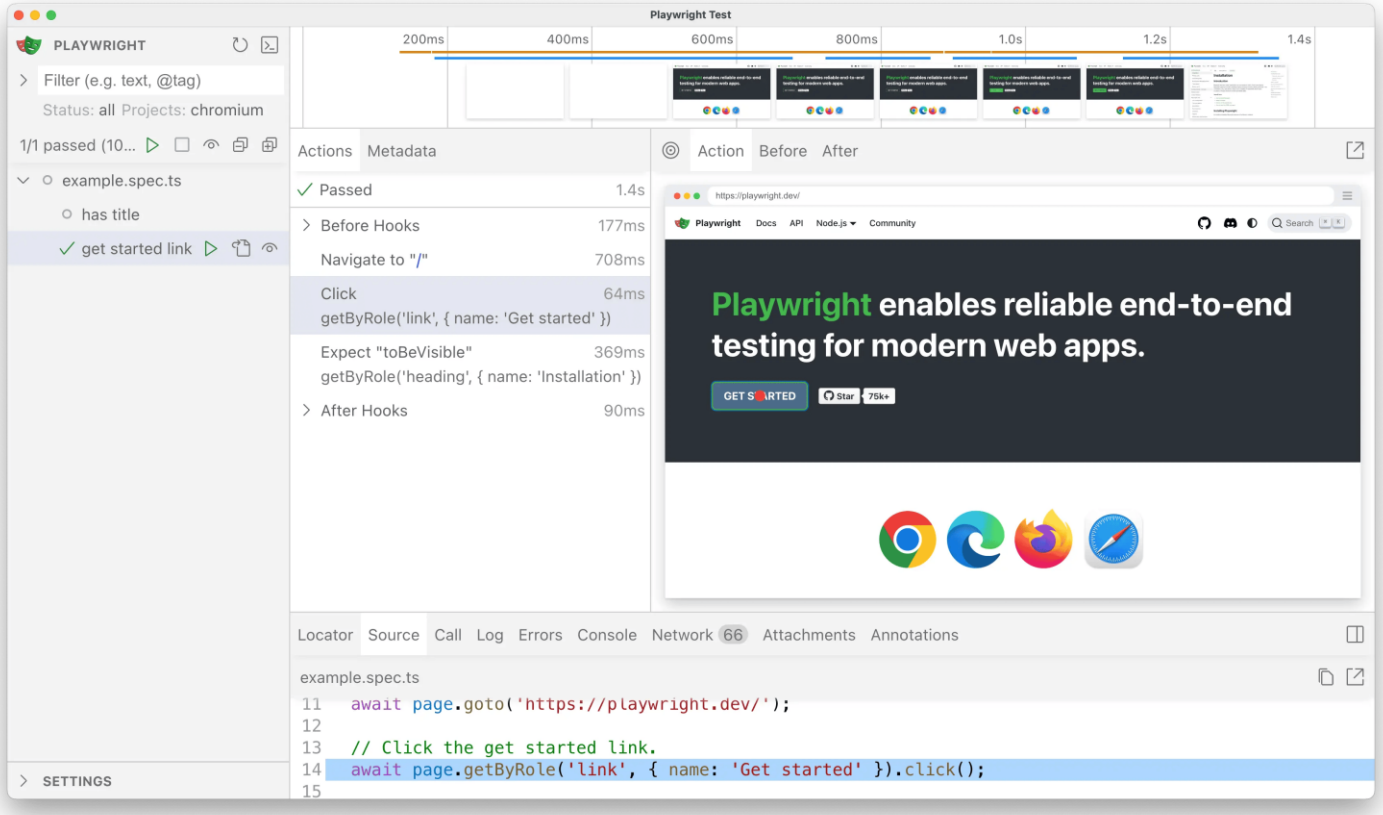
Cypress

- **Cypress**
  - Modern solution to web page E2E testing, all-in-one
  - Easier API and tooling
    - Runs directly on the browser / inject in the DOM
  - Less browser support; no multi-window

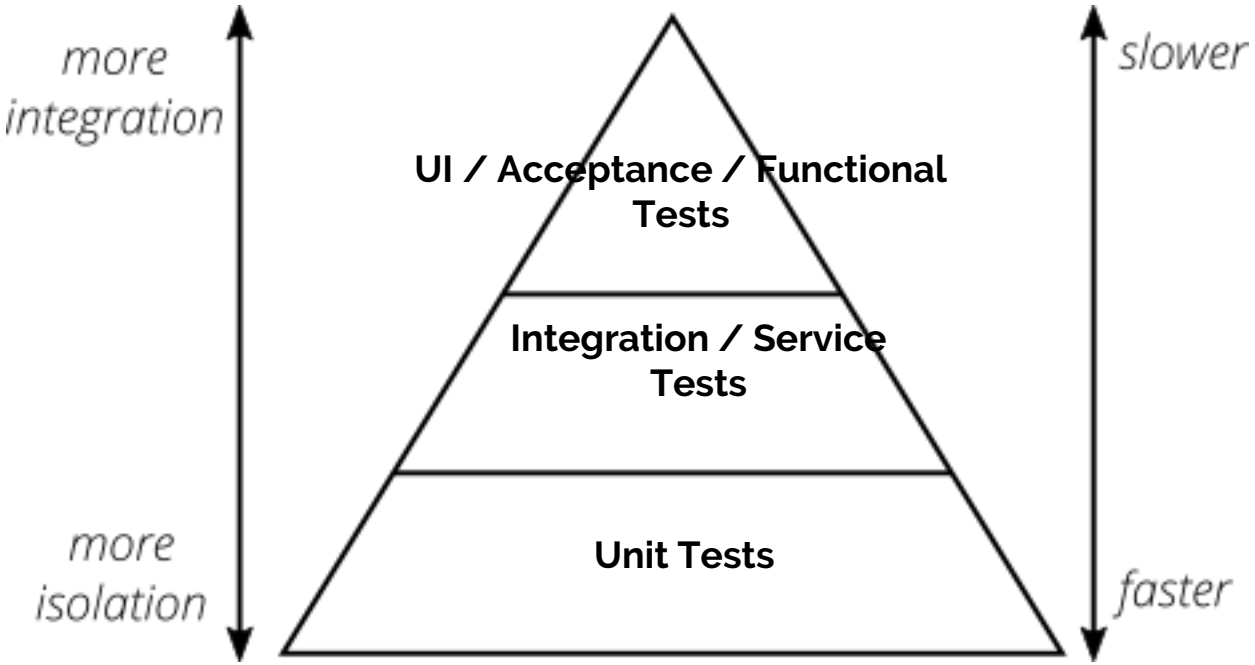




Playwright


Playwright Test
PLAYWRIGHT
Filter (e.g. text, @tag)
Status: all Projects: chromium
1/1 passed (10...
example.spec.ts
has title
get started link
SETTINGS
200ms
400ms
600ms
800ms
1.0s
1.2s
1.4s
Actions
Metadata
Passed 1.4s
Before Hooks 177ms
Navigate to "/" 708ms
Click 64ms
getByRole('link', { name: 'Get started' })
Expect "toBeVisible" 369ms
getByRole('heading', { name: 'Installation' })
After Hooks 90ms
Action
Before
After
Playwright enables reliable end-to-end testing for modern web apps.
Locator
Source
Call
Log
Errors
Console
Network 66
Attachments
Annotations
example.spec.ts
11 await page.goto('https://playwright.dev/');
12
13 // Click the get started link.
14 await page.getByRole('link', { name: 'Get started' }).click();
15

Wrap up



Write tests

Test Pyramid: Mostly unit

Conclusion

THE FOUR TYPES OF TESTS

End to End

A helper robot that behaves like a user to click around the app and verify that it functions correctly.

Sometimes called “functional testing” or e2e.

Integration

Verify that several units work together in harmony.

Unit

Verify that individual, isolated parts work as expected.

Static

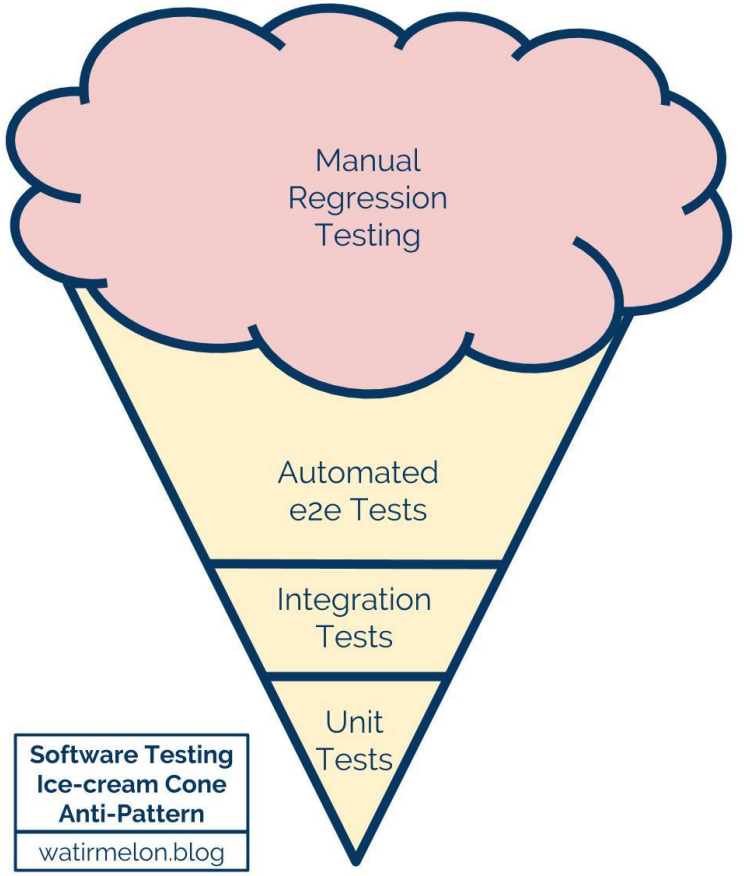
Catch typos and type errors as you write the code.

Write tests

Test Pyramid: Mostly unit

Test Trophy: Mostly integration

No test ice cream please



Test methodology

Write tests

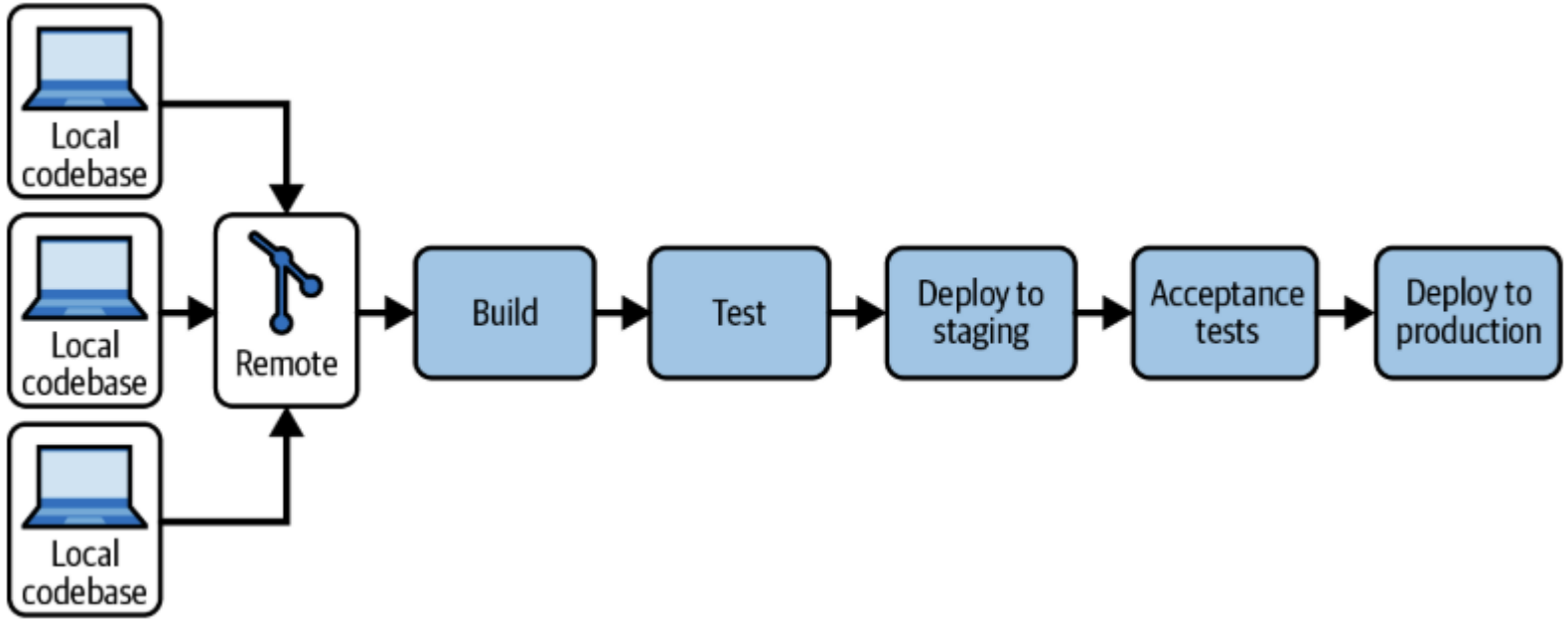
- Introduce tests alongside changes
  - Refactor may be required
- Write new tests on:
  - New implementations (design ahead)
  - Shift-left testing!
  - Critical components
- Don’t strive for 100% test coverage

Other type of tests

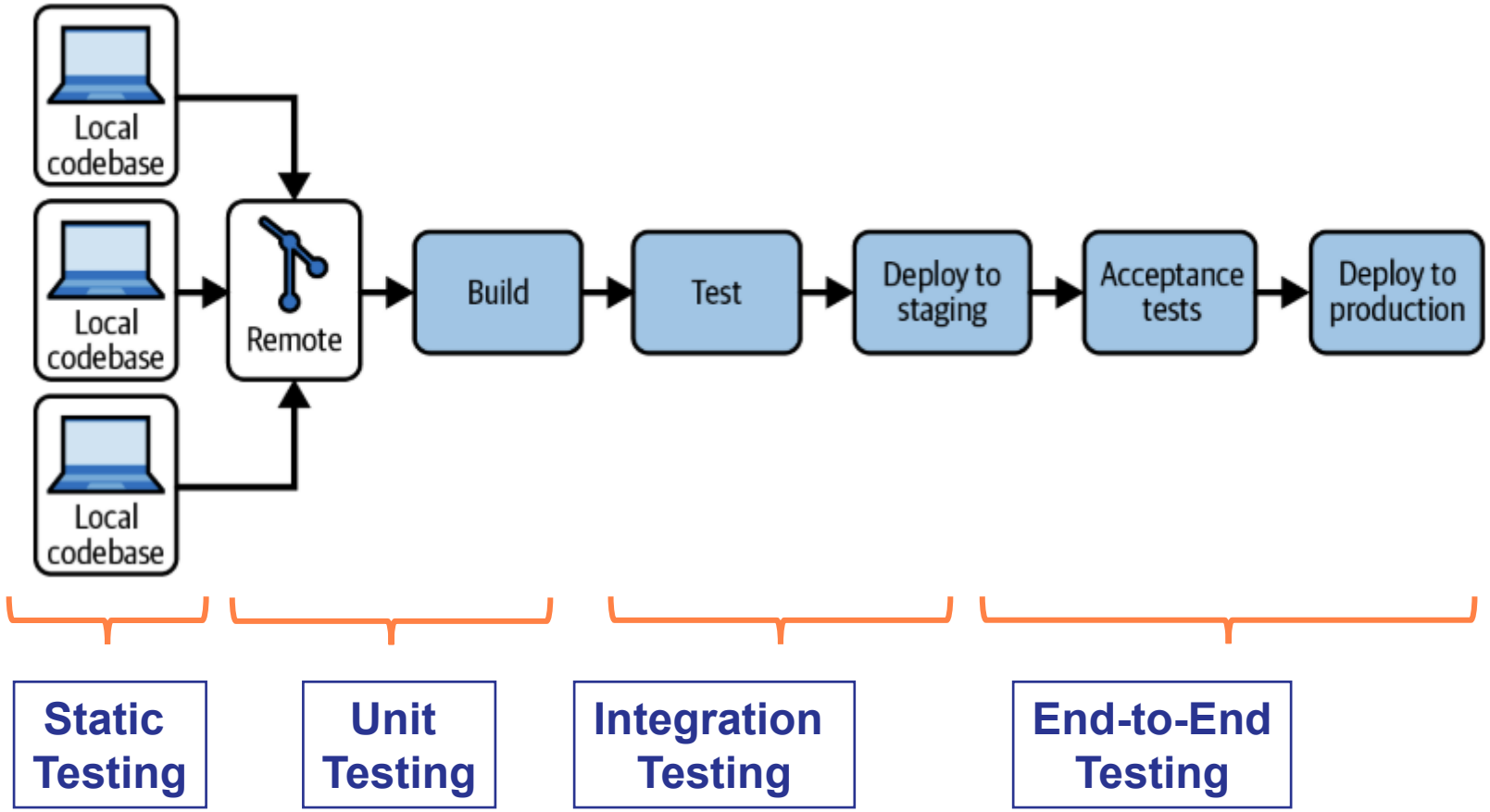
- Performance Tests
  - Validates system behavior under stress, peak traffic
  - Identifies bottlenecks (CPU / DB / network / locks)
  - Often integrated into CI for regression/performance budgets
  - Tools: Locust, k6, Gatling, Jmeter
- Chaos Testing
  - Tests fault tolerance and self-healing capabilities
  - Introduces controlled failure on infra/services
  - Common in cloud-native and microservices architectures
  - Tools: Chaos Mesh, Litmus, Gremlin
- Security Testing
  - Checks vulnerabilities, misconfigurations, and broken auth
  - Includes SAST, DAST, dependency scanning, secrets detection
  - Integrated into CI before deployment gates / release
  - Tools: OWASP ZAP, Trivy, Snyk, GitLab Security

Integrate tests in CI-CD

Where does each type of test fit?



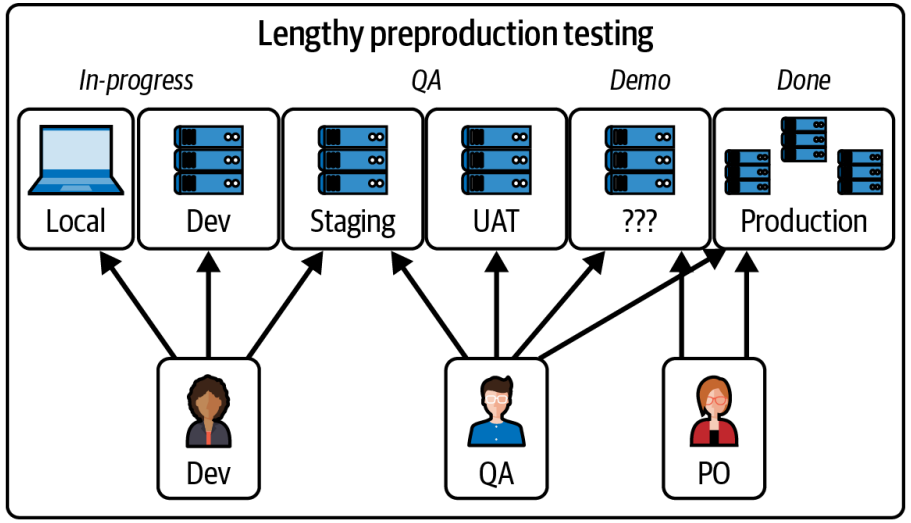
Where does each type of test fit?

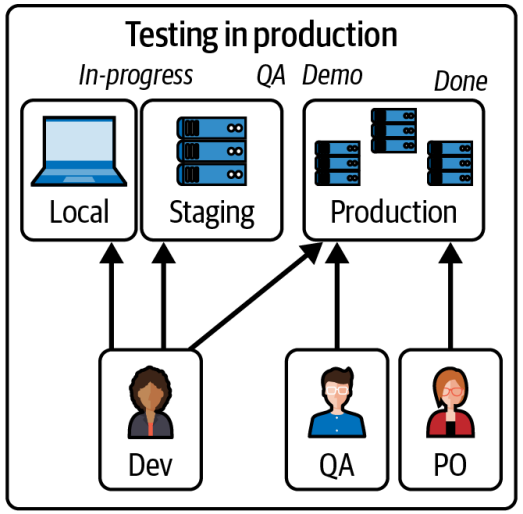


Note: this is where it should fit, but it may vary from team practices / products / etc.

Testing in production

- Higher confidence: Real environment reveals issues missed in staging; faster, safer releases.
- Accuracy of reality: True data volume and shape, historical quirks, and real request patterns emerge only in prod.
- Traffic realism: Incoming and outgoing traffic, scaling behavior, and instance counts match real usage.
- Config and network fidelity: Auth, pools, caches, CDNs, proxies, and third-party contracts differ from staging.
- Version alignment: Validates against the actual versions of internal and third-party services.
- Lower cost and better hygiene: Less staging load-testing, fewer environments, avoids risky prod-data copies.

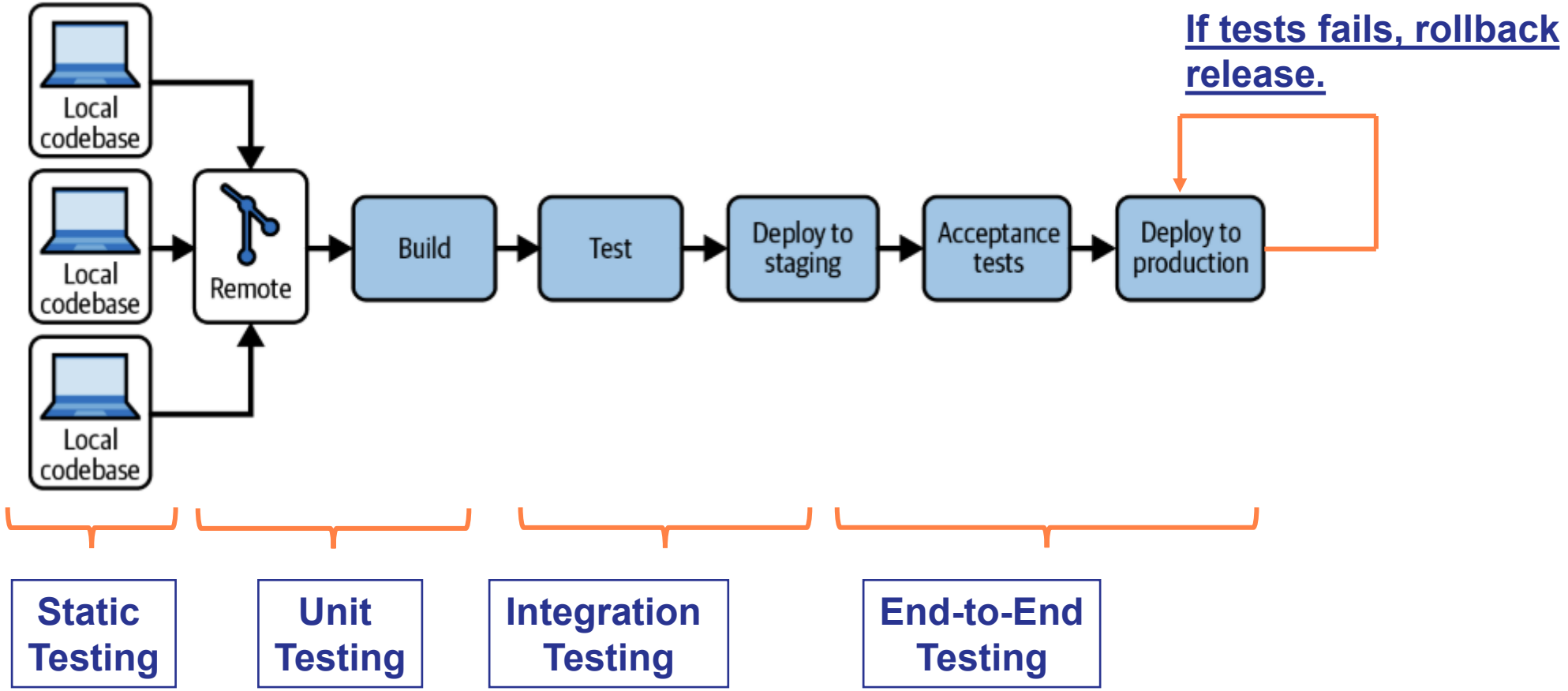




How to test in production (Safely)?

- Feature toggles: Precise, easy activation for exploratory tests via query params, headers, cookies, user IDs, or roles.
- Test data strategy: Create and flag synthetic data; hide by default; control visibility with the same toggle mechanisms.
- Prevent leakage: Define API policy for test data (local only, opt-in, or always flagged) and communicate contracts.
- Debugging in prod: Prefer enabling source maps (or controlled hosting) to make front-end issues diagnosable.
- Lean preprod: Keep minimal infra for automated checks; many teams reduce or retire staging as maturity grows.
- Maturity goal: With solid safety practices, production testing becomes routine in continuous delivery.

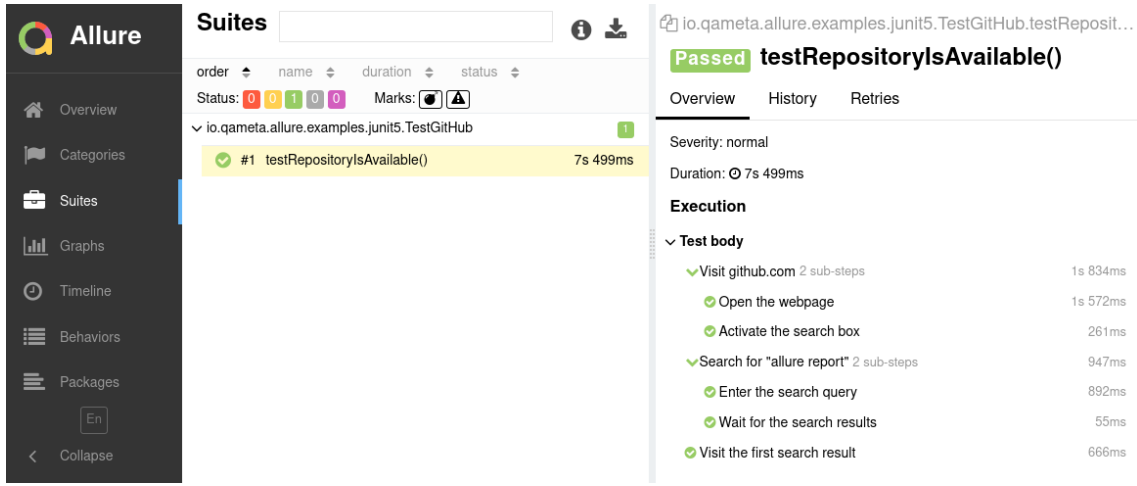
Where does each type of test fit?



Practice Guide

Allure Report

- A lightweight test reporting platform that aggregates results from multiple test frameworks
  - pytest, JUnit, Cypress, Playwright, etc.) into a rich, interactive dashboard.
  - Available here https://allurereport.org/
- Sample practical workflow:
  - Tests run (pytest/JUnit/Playwright/etc.)
  - Allure collects raw results
  - CI/CD generates an HTML dashboard
  - Report is published as an artifact or hosted statically



Integrate Tests in CI and CD

- Example of Github Action for Java

```
steps:
  - uses: actions/checkout@v5
  - uses: actions/setup-java@v4
    with:
      java-version: '17'
      distribution: 'temurin'
  - name: Run the Maven verify phase
    run: mvn --batch-mode --update-snapshots verify
```

Practical Guide

- In this lab:
  - Learn from this tutorial (Building and testing your code):
    - https://docs.github.com/en/actions/tutorials/build-and-test-code/java-with-maven
- In your project:
  - Make sure that you implement some different type of tests in your project (one of each at least)
  - They should be integrated in your CI-CD pipeline
  - You need to make sure that the tests will run before any deployment, in different steps.
    - It is acceptable simulate stage / production environment, as the same.
  - Allure Reports should be integrated
  - For the next iterations, you need to make sure that this pipeline is updated.

  - Other possibilities: regressions should create new tickets/github issues

Bibliography