8 Fallacies of Testing Microservice Systems

1 End-to-end testing is the only way to verify functionality

Effects: Engineers over-invest in creating end-to-end tests that become brittle and more costly to maintain as the software ages.

Solution: Learn about the Test Pyramid and invest in a range of loosely-coupled (modular) tests to support both continual business and technical verification.

2 Contract testing is too expensive to maintain

Effects: Developers under-invest in creating contracts due to maintenance concerns, leading to unverified interactions within specific areas of the system.

Solution: Ensure that core APIs between system and service boundaries are continually verified with up-to-date contracts. Don't use contract testing for every API, as this can result in overly-tight coupling.

3 Mocks, stubs, and doubles and the only way to simulate dependencies

Effects: Engineers rely on custom mocks that have implicit assumptions encoded within them. As the systems changes, the assumptions may not keep pace.

Solution: In addition to mocks, use "local-to-remote" development tools like Telepresence to test against actual dependencies running in a production-like environment.

4 Properties of production infrastructure do not impact component tests

Effects: Running tests on a non-production-like platform results in poor quality verification e.g. the use of container and cloud technology impacts network performance and memory allocation.

Solution: Ensure the local dev environment is as production-like as possible, e.g. run local tests in containers. Run component tests in a production-like environment within the build pipeline.

5 It's impossible to run fast and accurate integration tests

Effects: Compromises are made with integration tests either providing a high level of confidence but running slow or providing low confidence but quick execution.

Solution: Prioritize accuracy with integration tests. For speed, use TestContainers to run databases with pre-canned data, and use build pipelines to scale verification with shared staging environments.

6 Testing only takes place during pre-production

Effects: Customers find bugs in production, and unless reported, the engineering team may be unaware of the issues.

Solution: Invest early in observability throughout your applications, API gateway, and service mesh (and other infrastructure). Run semantic monitoring for key business journeys in production.

7 Test data is homogenous and easily generated

Effects: The use of poor quality test data leads to incorrect assumptions being made about functionality and performance.

Solution: Work with data and ops teams to understand the quantity and shape of core data. Ensure build pipeline tests again production-like databases.

8 Cross-functional tests (performance, security, etc) are ops responsibility

Effects: Cross-functional requirements are either neglected or poorly implemented as a product nears the go-live stage.

Solution: Developers should be encouraged to "shift left" the design and implementation of cross-functional requirements.

ambassador