Comprehensive Test Strategy — Fintech Microservices Platform

Author: Amit Patil
Date: Sep 15, 2025

Quality Vision & Objectives

Quality Vision: Deliver certification of a secure, consistent and highly available transaction processing system verified against measurable KPIs for correctness, latency, resilience with complete coverage and minimal customer impact.

Quality Objectives:

- Consistency: Transaction comply to ACID properties along with idempotency
- Availability: System should honor a defined SLA (service level agreement) of service uptime.
- Low Latency: API response time for each service with strict low latency targets.
- Security: Verify access controls, encryption at-rest and in-transit and other compliance/vulnerabilities.
- Resiliency: Verify Observability and fast recovery mechanism.

Quality metrics and KPIs

- Test pass rate: 99.9% for P90 test runs per release.
- Availability: 99.9% availability and operational.
- Throughput: System should be able to handle 500 transaction per second
- Latency: Latency of each api call P90 of < 200ms
- Mean time to detect outage: < 5 mins
- Mean time to recover : < 20 mins for P0 (Critical) issues.

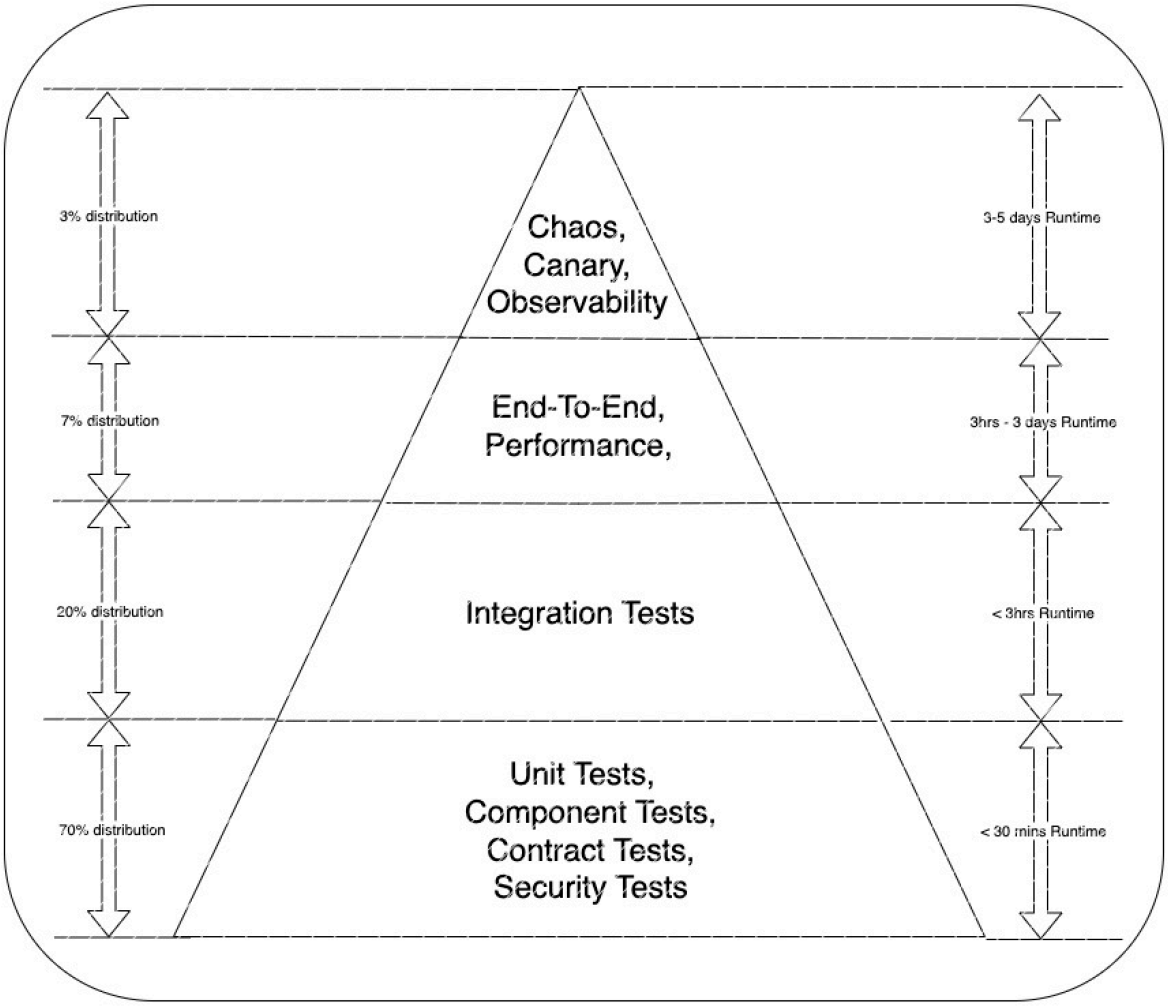
- Escape Defect/release: Aim for not more that 1-2 escape defects per release
- Test coverage: Test coverage cannot go down below 90% for any commit.
- CI Pipeline pass rate: The CI test pipeline should have a pass rate of >98%
- SLA for bugfix: Bugs coming out of testing should have a < 24 hrs SLA
- SLA for test code fix: Bugs in test code should have a <24 hrs SLA
- Test Feedback: Test runs should provide the right signal (red/green) within 3 hrs.
- Flaky Test: Flaky tests < 0.1% with auto-quarantine mechanism in place.

Test Pyramid Strategy

Test Layers

	Tests	Includes	Tools	Environment	Frequency	Distribution
1	Unit	White box tests	C++/Python	Dev + Stg	Per commit	70%
2	Component	Functional with Mocks	Cypress	Dev + Stg	Per commit	
3	Contract	Consumer-driven contracts	Pact	Dev + Stg	Per commit	
4	Security	Vulnerabilities	SonarQube	Dev/Stg	Per commit	
5	Integration	Cross Functional	pytest	Stg	Nightly	20%
6	End-To-End	System Flow	Selemium/ Playwright	Stg	Nightly	7%
7	Performance	Scale and Load	Jmeter	Stg	Weekly	
8	Chaos	Fault Injection, Disaster Recovery	Pytest, Chaos Monkey	Stg	Per Release	3%
9	Canary	Exploratory	Argo	Stg + Prod	Per Release	
10	Observability	Metrics based	Prometheus , Grafana, Splunk	Prod	Per release	

Test Distribution



Unit, Component, Contract, and Security Tests
These should be fast-running tests executed within 30 minutes and must block any code merges if broken. They run on every commit and can be executed in either the development or staging environment. Since they rely on mock data and environments, they are both fast and cost-effective. Ideally, 70% of the test distribution should consist of these tests.

Integration Tests
These use test containers and mocks provided by third-party dependencies. They require a fully functional service running in test containers, which makes them slower but provides valuable integration coverage by validating how services interact with each other. These tests should make up 20% of the total test distribution and complete within 3 hours. While they can be executed on every commit, they are generally run post-merge or during nightly builds.

End-to-End (E2E) and Performance Tests
These tests require fully deployed services to be up and running. They are long-running, expensive, and sometimes flaky due to inter-service dependencies, but they provide the closest simulation to real-world scenarios. Limit these tests to critical paths only, and keep them to about 7% of the total test distribution.

Chaos, Canary, and Observability Tests
These are executed primarily in the Production environment to mimic real-world customer behavior. They help validate resilience, disaster detection, and recovery capabilities. These would contribute to only 3% of the total test distribution.

CI/CD Quality Gates

Quality Gates should be in place throughout SDLC of the system.

1. **Commit Stage**
    a. Purpose: Validate every developer change early.
    b. Execution: Developers run tests locally or via CI.
    c. Tests:
        i. Required: Unit, Component, Contract
        ii. Optional: Integration, End-to-End, Performance
    d. Expectation: Fast feedback, failures must be fixed before proceeding.
2. **Pull Request**
    a. Purpose: Ensure code quality before merging.
    b. Execution: Opening a PR auto-triggers test pipelines.
    c. Tests:
        i. Unit, Component, Contract
        ii. Smartly selected Integration tests (based on impact analysis)
    d. Optional (Developer-Initiated): Full Integration, End-to-End, and Performance tests
    e. Approval Criteria: 100% test pass rate required for PR approval.
3. **Merge Train**
    a. Purpose: Safely integrate changes into the main branch.
    b. Execution:
        i. A temporary build is created by merging the PR with the latest master.
        ii. Unit, Component, and Contract tests are executed.
    c. Failure Handling: Any failure halts the merge train, reverts the failing change, and restarts the train for other PRs without the broken commit.
4. **Post Merge**
    a. Purpose: Catch integration issues after changes land.
    b. Execution: Full Integration test suite (up to 3 hours) runs after each merge.
    c. Failure Handling:

        i. The PR owner, test owner, and feature owner may be summoned.
        ii. The failing PR can be reverted while triage (product, test, or infra issue) occurs.
5. **Nightly**
    a. Purpose: Validate service integration stability in a staging-like environment.
    b. Execution: Long-running tests using a full staging environment.
    c. Tests: Unit, Component, Contract, Integration, End-to-End
    d. Cost: Resource-intensive; failures are expensive and must be triaged promptly.
6. **Release Certification**
    a. Purpose: Certify readiness for production release.
    b. Execution:
        i. Runs on a release branch cut from master.
        ii. Full test suite executed in cycles until:
            1. No critical test failures
            2. Zero critical known bugs remain open
    c. Outcome: Approval for release deployment.
7. **Canary**
    a. Purpose: Safely roll out features to production with real users.
    b. Execution:
        i. Rollout strategy: 25% → 50% → 100% (incremental exposure)
        ii. Controlled by feature flags
        iii. Observability via Prometheus and Grafana
    c. Validation:
        i. Canary users act as early adopters
        ii. Requires soak time of 3–5 days before full rollout

Non-Functional Testing

Performance

- Capture and analyze CPU, memory, and I/O utilization while services are running.

Load Testing

- Measure throughput to determine system limits.
- Example: Can the system reliably handle 500 transactions per second?

Scalability

- Verify scale-out and scale-in strategies based on CPU and memory utilization.
- Ensure the system adapts smoothly to changing demand.

Capacity & Limits

- Identify the maximum transaction volume the system can process without failures or degraded performance.

Reliability

- Ensure compliance with ACID properties and idempotency guarantees.
- Validate consistent behavior under stress and failures.

Security

- Perform regular static analysis scans.
- Conduct penetration testing to uncover potential vulnerabilities.

Resilience

- Assess the system's ability to recover quickly from failures.
- Measure recovery time and effectiveness of failover mechanisms.

Chaos Engineering

- Use fault injection techniques to simulate outages or disruptions.
- Validate that the system maintains availability and recovers gracefully.

Summary

This test strategy establishes a comprehensive approach to ensuring product quality across the entire software development lifecycle (SDLC). It defines the types of tests, their distribution, and the quality gates at each stage, from commit through release and production validation.

The strategy emphasizes a balanced test pyramid:

- Unit, Component, Contract, and Security tests form the majority, enabling fast feedback and preventing regressions.
- Integration and End-to-End tests validate service interoperability and critical user flows, with controlled execution to balance speed and coverage.
- Performance, Scalability, Reliability, Security, and Resilience tests ensure the system meets non-functional requirements and production readiness.

Quality gates are applied consistently at commit, pull request, merge train, post-merge, nightly, release certification, and canary stages, ensuring issues are detected early and addressed before impacting customers.

Finally, the strategy incorporates production-focused validation through canary releases, chaos testing, and observability practices, ensuring real-world resilience and customer trust.
This layered and systematic approach not only reduces risk but also promotes faster releases, higher confidence, and long-term product stability.