

Modern Application Architecture & Quality Engineering Practices

A comprehensive training for QA, QE, and SDET professionals to master architectural understanding and modern quality practices

MODULE 1

Application Architecture Basics

Understanding application architecture is no longer optional for quality engineers. This module establishes the foundational knowledge required to design effective test strategies, identify risk areas, and communicate confidently with development teams.

Quality engineering begins with architectural awareness—knowing how components interact, where data flows, and which integration points are vulnerable to defects.

What is Application Architecture?

Application architecture defines the structural design of software systems—how components are organised, how they communicate, and how they fulfil business requirements. It encompasses technology choices, integration patterns, data flow, and deployment models.

For quality engineers, architecture knowledge directly impacts test coverage, automation strategy, and defect prevention. Without understanding architecture, testing becomes reactive rather than strategic.

Components

Modules, services, databases

Integration

APIs, messaging, events

Deployment

Environments, infrastructure

Why QA/QE Must Understand Architecture

1

Strategic Test Planning

Architecture knowledge enables risk-based testing. Understanding component dependencies helps prioritise critical integration points and allocate testing effort effectively.

2

Efficient Defect Analysis

When defects occur, architectural awareness accelerates root cause analysis. You can quickly identify whether issues stem from frontend logic, API contracts, data layer problems, or integration failures.

3

Automation Strategy

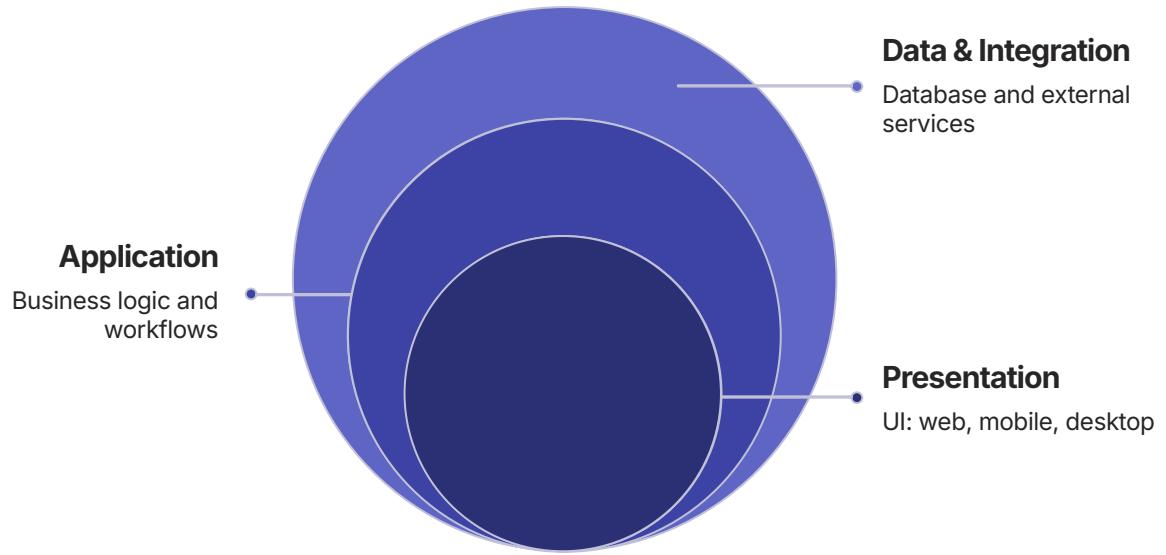
Effective automation requires understanding where to test. API-level testing, service virtualisation, and component isolation all depend on architectural knowledge.

4

Technical Credibility

Discussing architecture confidently with developers, architects, and stakeholders establishes your credibility and positions QE as a strategic partner rather than a gatekeeper.

High-Level Application Components



Modern applications consist of distinct layers, each with specific responsibilities and testing requirements.

Frontend / Presentation Layer

User interfaces (web, mobile, desktop) that handle user interaction, input validation, and display logic.

Backend / Application Layer

Business logic, APIs, service orchestration, authentication, and authorisation mechanisms.

Data Layer

Databases, data warehouses, caching systems, and data access logic that manage persistence and retrieval.

Integration Layer

Third-party services, external APIs, messaging queues, and event-driven communication channels.



Monolithic Architecture

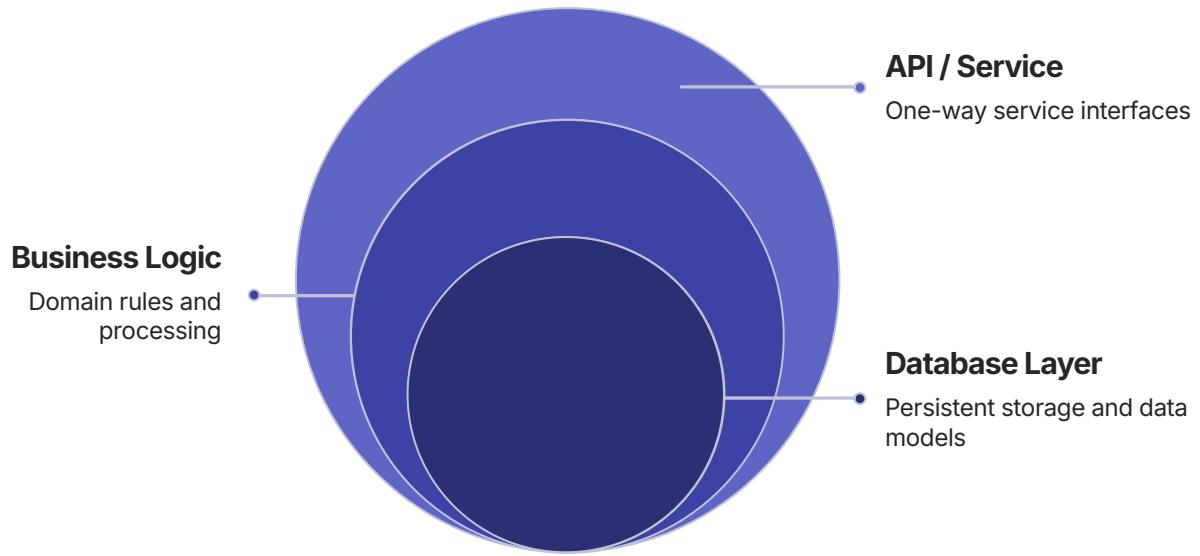
A monolithic application is built as a single, unified unit where all functionality resides in one codebase and typically shares a single database. The entire application is deployed as one package.

Characteristics: Tightly coupled components, single deployment unit, shared database, easier to develop initially but harder to scale and maintain as complexity grows.

Testing implications: Full regression required for any change, longer build and deployment cycles, but simpler end-to-end testing without complex service orchestration.

Examples: Traditional enterprise applications, legacy systems, many early web applications.

Layered Architecture



Layered architecture organises code into horizontal layers, each with a specific responsibility. Each layer depends only on the layer directly below it, creating clear separation of concerns.

Common layers: Presentation (UI), API/Service (endpoints), Business Logic (rules and workflows), Data Access (database operations).

Testing strategy: Each layer can be tested independently. Unit tests verify business logic, API tests validate service contracts, UI tests confirm user workflows, and integration tests ensure layers communicate correctly.

Benefits: Clear separation enables focused testing, easier mocking and stubbing, and parallel test development across layers.

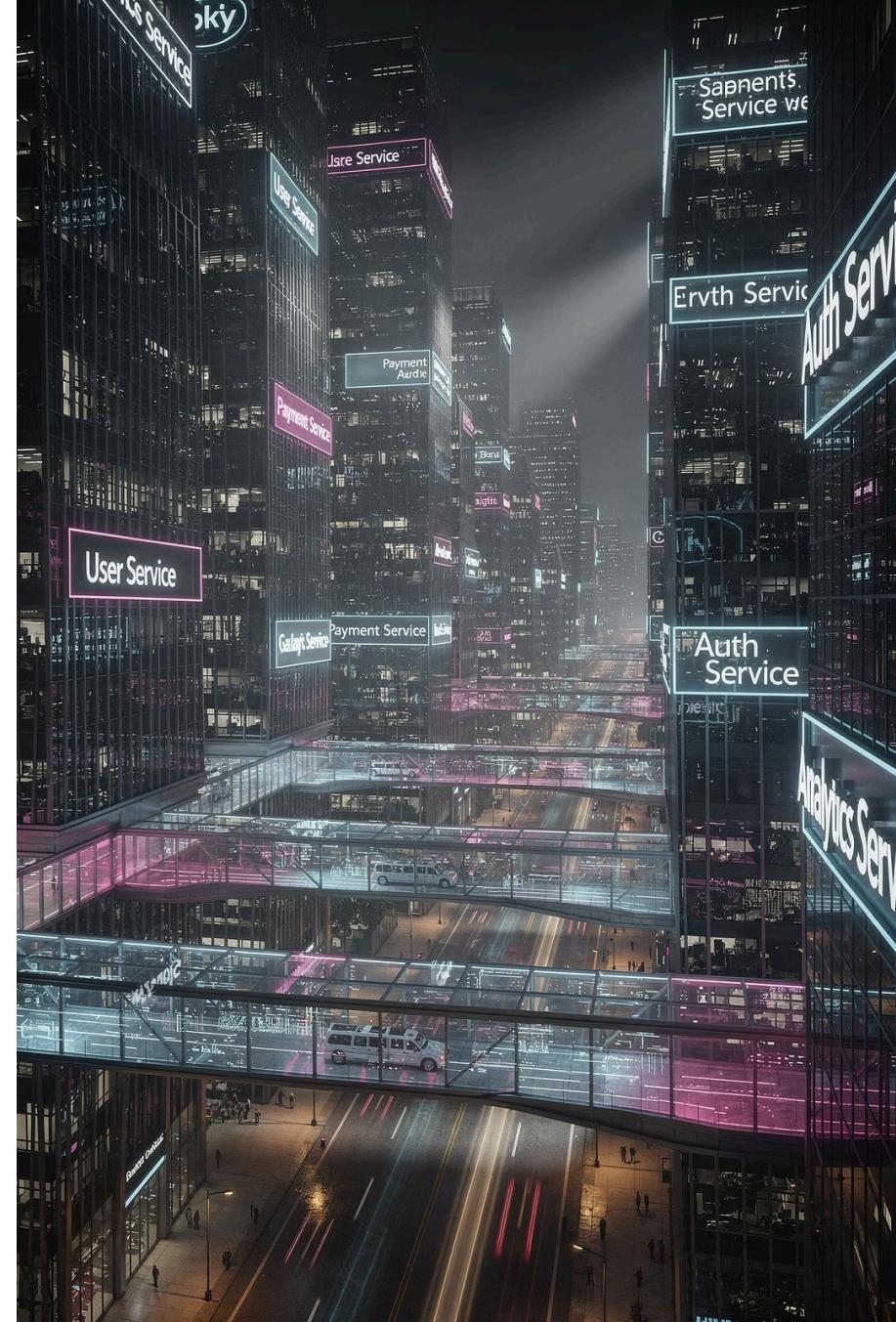
Microservices Architecture (Introduction)

Microservices architecture decomposes applications into small, independent services, each responsible for a specific business capability. Services communicate via APIs or message queues and can be developed, deployed, and scaled independently.

Key characteristics: Service independence, decentralised data management, polyglot technology stacks, API-based communication, independent deployment pipelines.

Testing challenges: Complex integration testing, service dependency management, distributed transactions, network latency, and the need for comprehensive contract testing between services.

Testing approach: Unit tests for each service, contract tests between services, integration tests for workflows spanning multiple services, and end-to-end tests for critical business scenarios.



Client–Server Architecture

Client–server architecture separates systems into two roles: clients that request services and servers that provide them. This model forms the foundation of web applications, mobile apps, and enterprise systems.

Client responsibilities: User interface, input validation, presentation logic, initiating requests.

Server responsibilities: Business logic, data processing, authentication, database operations, response generation.

Testing focus: Validate request/response contracts, verify server-side validation (never trust client), test error handling, ensure security mechanisms, and confirm performance under concurrent client load.

01

Client Sends Request

User action triggers HTTP request

02

Server Processes

Validates, executes logic, queries database

03

Server Responds

Returns data or error message

04

Client Renders

Displays result to user

Frontend vs Backend

Frontend

Everything users see and interact with directly

- HTML, CSS, JavaScript frameworks (React, Angular, Vue)
- User interface design and responsiveness
- Client-side validation and interactivity
- Browser compatibility

Testing focus: UI functionality, cross-browser testing, responsive design, accessibility, performance

Backend

Server-side logic users never see directly

- APIs, databases, business logic
- Authentication and authorisation
- Data processing and validation
- Integration with external systems

Testing focus: API contracts, business rules, security, data integrity, performance, integration points

APIs and Services

Application Programming Interfaces (APIs) define how software components communicate. They expose functionality as services that other components or applications can consume without understanding internal implementation details.

REST APIs

Use HTTP methods (GET, POST, PUT, DELETE) and return data in JSON or XML format. Stateless and widely adopted.

SOAP Services

XML-based protocol with strict contracts (WSDL). Common in enterprise systems requiring formal specifications.

GraphQL APIs

Query language allowing clients to request exactly the data they need. Single endpoint with flexible data retrieval.

Testing requirements: Validate request/response schemas, verify HTTP status codes, test authentication and authorisation, check error handling, validate data formats, and ensure performance under load.

Databases and Data Management

Relational Databases

SQL-based (MySQL, PostgreSQL, Oracle). Structured data with defined schemas and ACID transactions.

NoSQL Databases

Document, key-value, graph stores (MongoDB, Redis, Neo4j). Flexible schemas for unstructured data.

Data Warehouses

Analytical databases (Snowflake, Redshift) optimised for reporting and business intelligence.

Testing Considerations

Database testing extends beyond simple CRUD operations. Quality engineers must verify data integrity, referential constraints, transaction isolation, performance under concurrent access, and backup/recovery procedures.

Key focus areas: Data validation rules, query performance, index effectiveness, stored procedure logic, trigger behaviour, and data migration scripts.

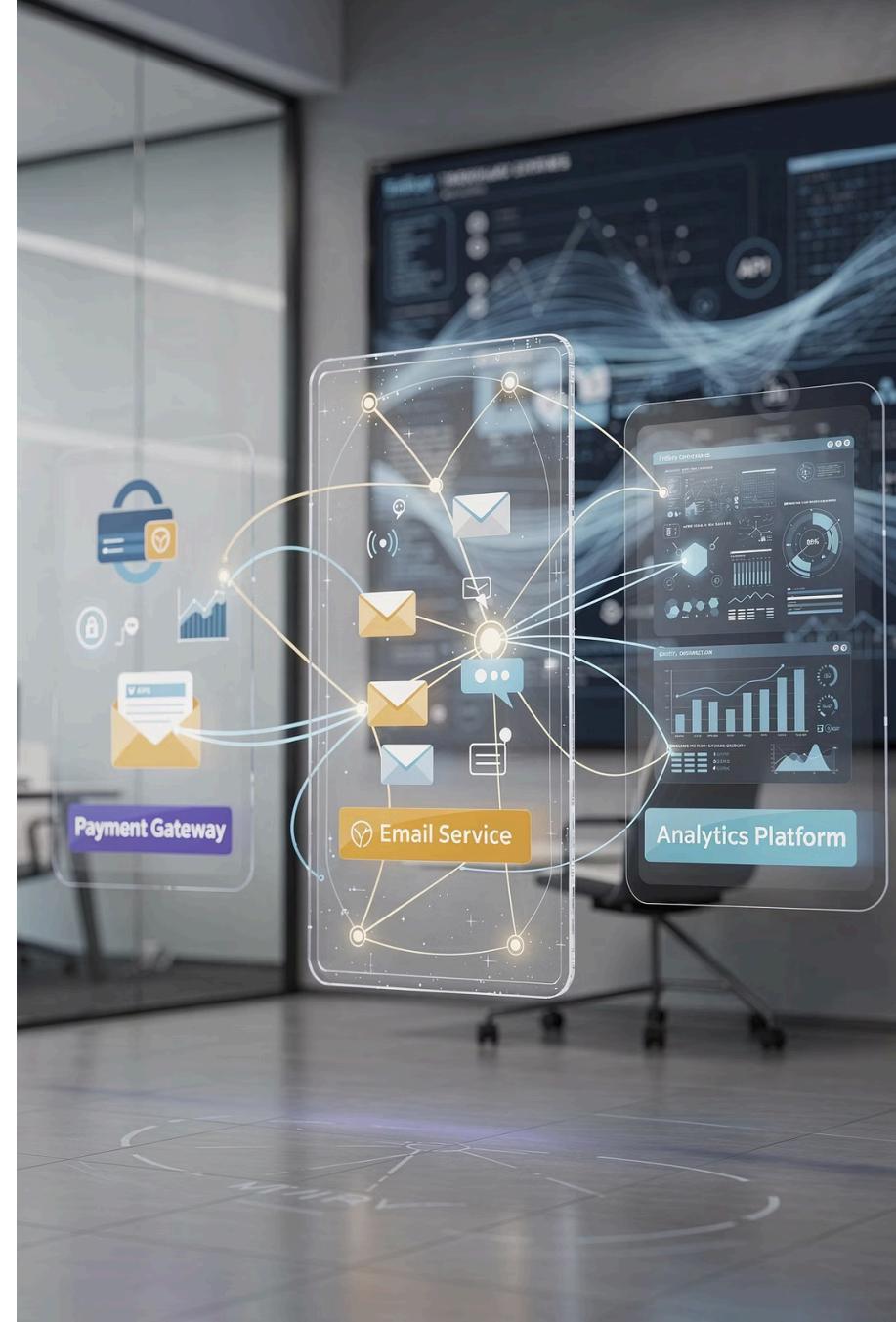
Third-Party Integrations

Modern applications rarely exist in isolation. They integrate with external services for payment processing, email delivery, SMS notifications, analytics, authentication, and countless other capabilities.

Common integrations: Payment gateways (Stripe, PayPal), email services (SendGrid, Mailchimp), cloud storage (AWS S3, Azure Blob), authentication providers (OAuth, SAML), analytics platforms (Google Analytics, Mixpanel).

Testing challenges: External services may be unavailable during testing, costly to invoke repeatedly, rate-limited, or return non-deterministic data. Service virtualisation becomes essential.

Testing strategy: Use contract tests to verify integration contracts, implement service virtualisation for stable testing, create fallback mechanisms, and monitor integration health in production.



Environment Separation

Professional software development uses multiple isolated environments to safely develop, test, and deploy applications without impacting production users.



DEV

Development environment where engineers write and test code locally or in shared development servers.

QA / TEST

Dedicated environment for quality assurance activities, automated testing, and exploratory testing.



UAT / STAGING

Pre-production environment mirroring production for user acceptance testing and final validation.

PRODUCTION

Live environment serving real users with real data, requiring maximum stability and security.

Quality engineering consideration: Test data management, environment configuration differences, deployment pipelines, and production-like testing conditions become critical.

Real-World Example: E-Commerce Application

The screenshot displays a user interface for an e-commerce platform. At the top, there's a navigation bar with a search icon, a user profile icon, and a shopping cart icon showing a count of 2. Below the navigation is a header "Online Store". The main content area shows four products arranged in a grid:

Image	Product Name	Price
	Product Name	\$20.00
	Product Name	\$50.00
	Product Name	\$90.00
	Product Name	\$80.00

Each product has an "Add to Cart" button below it. To the right, a "Shopping Cart" box is shown with the following items:

Image	Product Name	Price
	Product Name	\$20.00
	Product Name	\$50.00

Total \$70.00

Checkout

At the bottom left is a watermark "alamy". At the bottom right, there's a footer with "Image ID: 3BMM5PM" and the website "www.alamy.com".

An e-commerce platform demonstrates layered architecture with microservices:

Frontend: React-based web application and mobile apps presenting products and managing user interactions.

Services: Product catalogue service, user authentication service, shopping cart service, order management service, payment processing service, inventory service.

Data stores: Product database (PostgreSQL), user database (MySQL), session cache (Redis), search index (Elasticsearch).

Integrations: Payment gateway (Stripe), shipping providers (FedEx API), email service (SendGrid), analytics (Google Analytics).

Testing complexity: End-to-end checkout flows span multiple services, requiring comprehensive integration testing and careful service virtualisation.

Real-World Example: Banking Application

Banking applications exemplify complex, secure, multi-layered architectures with stringent compliance requirements:

Architecture: Multi-tier with strict security boundaries, mainframe integration for core banking, modern microservices for digital channels.

Components: Mobile banking app, internet banking portal, ATM interface, core banking system, fraud detection service, transaction processing engine, reporting system.

Data: Customer accounts database, transaction logs, audit trails, regulatory reporting data warehouse.

Integrations: Credit bureaus, payment networks (SWIFT, ACH), identity verification services, regulatory reporting systems.



Security

Multi-factor authentication, encryption, fraud prevention



Compliance

Regulatory requirements, audit trails, data privacy



Performance

High transaction volumes, real-time processing

Knowledge Check: Architecture Basics

Test your understanding of application architecture fundamentals:

1

Question 1

Which architecture pattern allows independent deployment and scaling of individual business capabilities?

- A) Monolithic
- B) Layered
- C) Microservices
- D) Client-Server

Correct answer: C) Microservices

2

Question 2

In a layered architecture, which layer should never be bypassed when accessing data?

- A) Presentation layer can directly access database
- B) Business logic layer should mediate data access
- C) Any layer can access database for efficiency
- D) Only API layer accesses database

Correct answer: B) Business logic layer should mediate data access

3

Question 3

What is the primary purpose of environment separation in software development?

- A) Reduce infrastructure costs
- B) Isolate changes and enable safe testing
- C) Improve application performance
- D) Simplify deployment processes

Correct answer: B) Isolate changes and enable safe testing

Case Study: Testing Failure Due to Architecture Ignorance

Scenario

A QA team tested an e-commerce checkout flow exclusively through the UI, achieving 95% functional test pass rate before production deployment.

Production Incident

Within 24 hours of release, customers reported failed payments despite successful order confirmation emails. The QA team had not understood that payment processing occurred asynchronously through a separate microservice.

Root Cause

UI tests only verified the order submission response, not the actual payment service integration or eventual consistency handling between order and payment services.

Lessons Learnt

- Architectural understanding prevents blind spots in test coverage
- Asynchronous processes require specific testing strategies
- API-level testing would have caught the payment service failure
- Service dependency mapping is essential for comprehensive testing

Impact: £45,000 in failed transactions, damaged customer trust, emergency rollback, and week-long remediation effort.

Interview Keywords: Architecture

Master these terms to demonstrate architectural knowledge in interviews and daily work:



API Contract

Formal specification of request/response structure, endpoints, methods, and data formats between services.



Service Dependency

Relationship where one service relies on another to fulfil its functionality or complete operations.



Environment Parity

Maintaining consistency in configuration and infrastructure across development, testing, and production environments.



Data Flow

Path that data takes through system components from input to storage to output.



Integration Point

Location where two systems or components connect and exchange data, often a source of defects.



Eventual Consistency

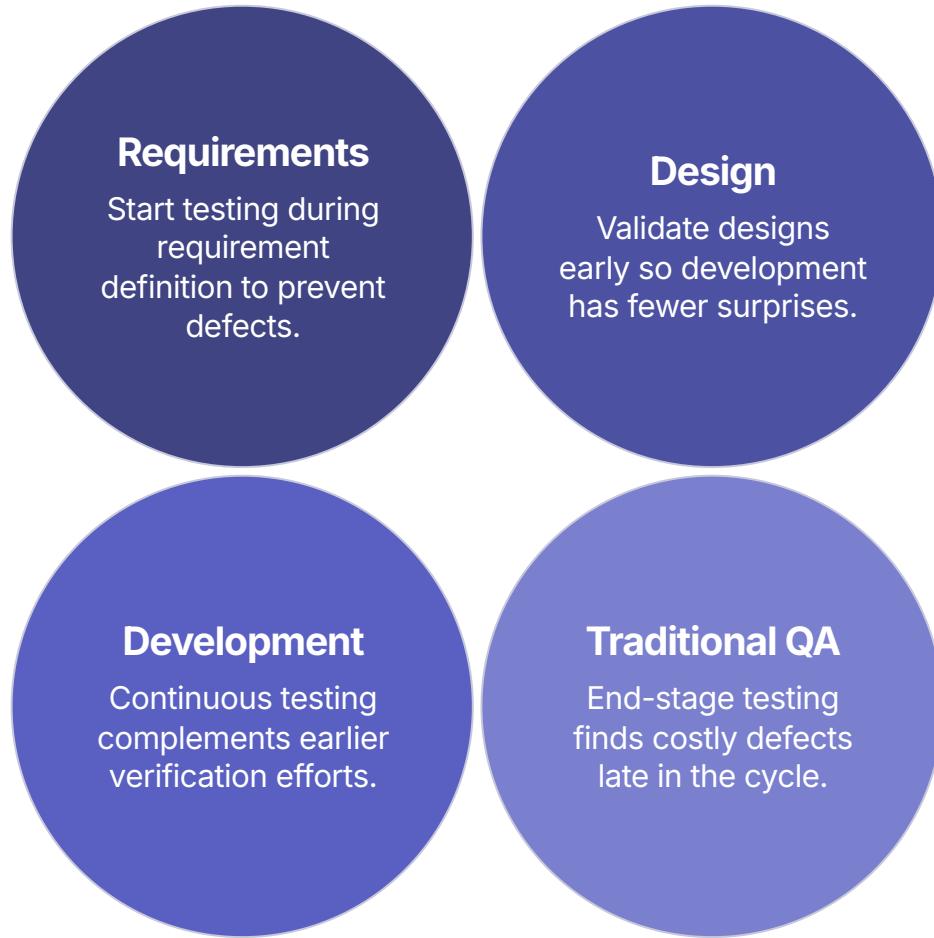
Distributed systems pattern where data becomes consistent across services over time rather than immediately.

Shift-Left Testing

Shift-left testing represents a fundamental paradigm shift in quality engineering—moving testing activities earlier in the software development lifecycle to prevent defects rather than merely detecting them.

This approach transforms QE from a post-development verification function into a proactive quality partner embedded throughout the entire development process.

What is Shift-Left Testing?



Shift-left testing moves quality activities earlier in the development lifecycle—beginning at requirements and design phases rather than waiting for code completion.

Core principle: Prevention over detection. By involving QE from project inception, teams identify ambiguities, design flaws, and testability issues before code is written.

Economic impact: Defects found in requirements cost 10–100 times less to fix than defects found in production. Shift-left directly reduces development costs and time-to-market.

Cultural shift: Quality becomes everyone's responsibility, not just the QA team's. Developers write unit tests, business analysts create testable requirements, and QE guides the entire team toward quality outcomes.

The earlier defects are found, the exponentially cheaper they are to fix.

Why Traditional Testing Fails

Late Discovery

Defects found in testing or production are expensive to fix, requiring code changes, retesting, and potential rework across multiple components.



Bottleneck Effect

When testing occurs only after development completes, QE becomes a bottleneck. Rushed testing leads to missed defects and production incidents.

Requirements Gaps

Ambiguous requirements aren't questioned until testing begins, leading to rework, missed deadlines, and features that don't meet user needs.



Architectural Defects

Fundamental design flaws discovered during testing are prohibitively expensive to fix, forcing teams to accept technical debt or make compromises.

Traditional "testing at the end" creates a reactive quality culture focused on finding defects rather than preventing them.

Shift-Left Position in SDLC

Shift-left testing integrates quality activities across every SDLC phase:

01

Requirements

Review, clarify, identify testability issues

02

Design

Test case design, identify edge cases, review architecture

03

Development

Unit tests, API tests, static analysis, continuous integration

04

Testing

Integration tests, system tests, exploratory testing

05

Deployment

Smoke tests, deployment validation, monitoring setup



Shift-Left Activities: Requirements Phase

Requirements Review

Active quality engineering participation in requirements definition

- Challenge ambiguous or contradictory requirements
- Identify missing acceptance criteria
- Clarify edge cases and error scenarios
- Ensure requirements are testable
- Define success metrics and quality criteria

Test Case Design (Early)

Begin designing test scenarios before development starts:

- Create test conditions from requirements
- Identify positive and negative test scenarios
- Define test data requirements
- Map test coverage to requirements
- Share test scenarios with developers to clarify expectations

Benefit: Early test design uncovers requirement gaps and drives conversations that prevent future defects.

Shift-Left Activities: Development Phase

Static Testing

Analyse code without executing it using static analysis tools (SonarQube, ESLint, CheckStyle). Identify code quality issues, security vulnerabilities, code smells, and standards violations before code reaches testing environments.

Early Automation

Develop automated tests in parallel with feature development. API tests can be written as soon as endpoints are defined. Unit tests guide implementation through test-driven development (TDD) practices.

API Testing Before UI

Validate business logic at the API layer before UI is complete. API tests execute faster, provide better diagnostics, and enable parallel UI and backend development without blocking test automation efforts.

Continuous Integration

Integrate automated tests into CI pipelines so every code commit triggers test execution. Immediate feedback prevents defects from accumulating and enables rapid issue resolution while code context is fresh.

Benefits of Shift-Left Testing

60%

Cost Reduction

Defects found in requirements cost 60–100x less to fix than production defects

40%

Faster Delivery

Teams report 40% reduction in time-to-market through early defect prevention

75%

Fewer Escaped Defects

Production incidents decrease by 75% when comprehensive shift-left practices are adopted

35%

Higher Productivity

Development teams spend 35% less time on rework and defect fixing

Practical Example: Requirement Gap Found Early

Scenario

During requirements review for a payment refund feature, the QE team asked clarifying questions about edge cases.

Questions Raised

- What happens if partial refund exceeds original payment?
- Can refunds be processed for cancelled orders?
- What if the customer's payment method has expired?
- How are refunds handled for promotional discounts?

Outcome

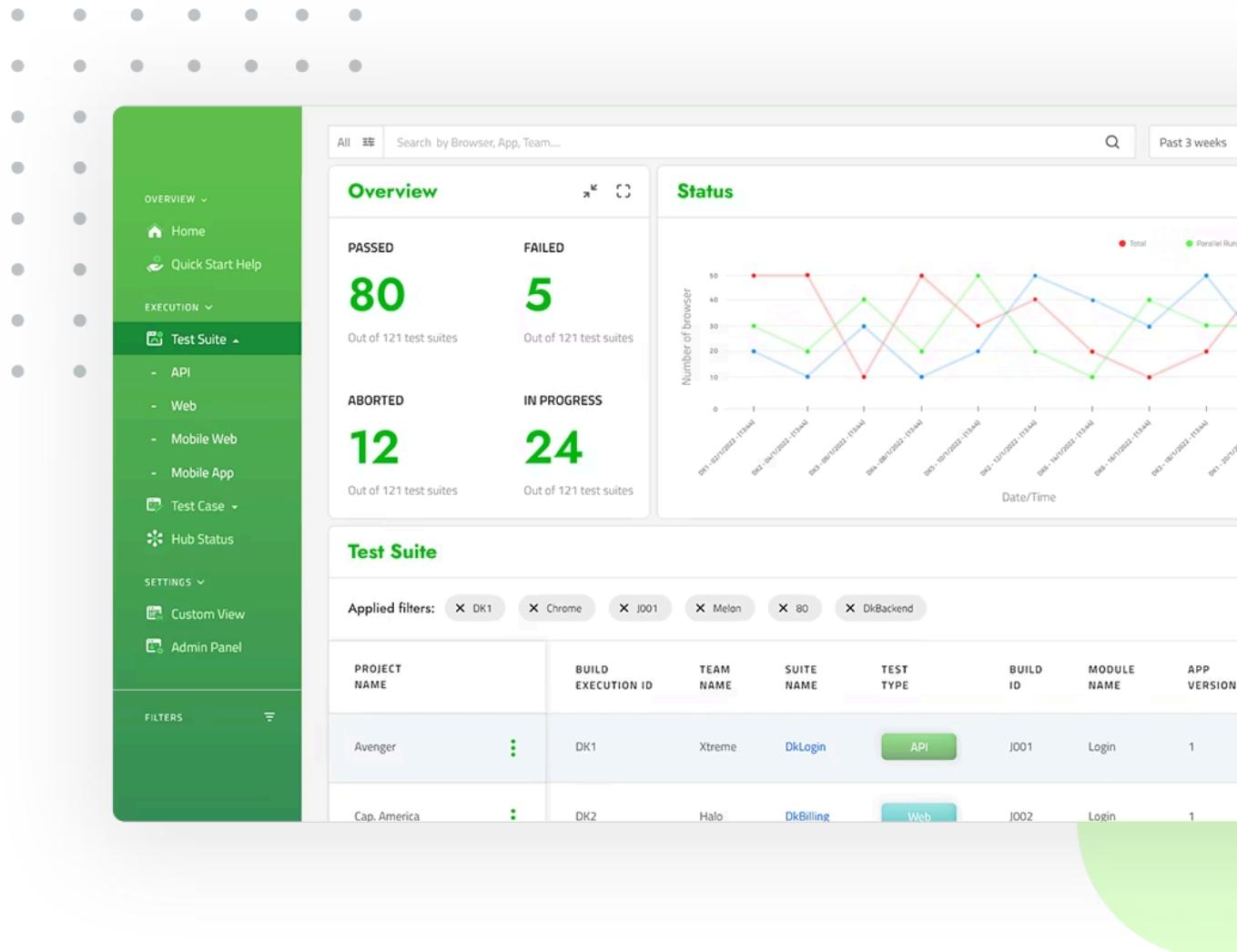
These questions revealed that requirements were incomplete. Business analysts clarified rules, preventing at least four production defects.

Impact

- Saved 3 weeks of rework that would have occurred if discovered during testing
- Prevented potential production incidents affecting customer trust
- Cost of fix: 2 hours requirements clarification vs estimated 40 hours post-development rework

Key lesson: Simple questions during requirements save exponentially more effort than late-stage defect fixes.

Practical Example: API Tested Before UI



Situation

A team building a customer management system needed to add search functionality. The API was developed first while UI design was still in progress.

Approach

QE created comprehensive API tests covering search by name, email, customer ID, and various filter combinations—all before the UI existed.

Results

- Discovered pagination defect when result sets exceeded 100 records
- Identified missing validation for special characters in search terms
- Found performance issue with concurrent searches
- All issues resolved before UI integration began

Benefit: When UI was complete, integration was seamless with zero defects related to search functionality.

Knowledge Check: Shift-Left Testing

1

Question 1

What is the primary economic benefit of shift-left testing?

- A) Faster test execution
- B) Fewer testers required
- C) Exponentially lower cost of defect fixes
- D) Reduced automation maintenance

Correct answer: C) Exponentially lower cost of defect fixes

2

Question 2

Which activity best exemplifies shift-left testing?

- A) Running regression tests after deployment
- B) Reviewing requirements for testability before development
- C) Writing test cases during system testing phase
- D) Conducting user acceptance testing

Correct answer: B) Reviewing requirements for testability before development

Case Study: Project Saved Through Early Testing

Background

A financial services company planned a 6-month project to rebuild their loan application system. Traditional approach would have delayed testing until month 5.

Shift-Left Approach

- QE participated in all requirements workshops
- Test scenarios created alongside user stories
- API automation began in sprint 1
- Static analysis integrated into CI from day one
- Weekly exploratory testing of completed features

Results

Defects found in requirements: 47 issues that would have required significant rework

Defects caught during development: 156 issues resolved immediately within same sprint

Defects found in formal testing: Only 12 minor issues

Production defects (first 3 months): 2 low-severity issues

Schedule impact: Project delivered 2 weeks early

Cost savings: Estimated £180,000 saved compared to traditional approach

Business impact: Loan processing time reduced 40%, customer satisfaction increased significantly.

Interview Tips: Shift-Left vs Traditional Testing

Articulate the difference confidently in interviews:

When	After development completes	From requirements onwards
Focus	Defect detection	Defect prevention
QE Role	Gatekeeper, verifier	Consultant, collaborator
Automation	After manual testing	Parallel with development
Cost	High rework and fix costs	Lower overall project costs
Speed	Testing bottleneck delays release	Faster delivery, fewer delays
Quality	Reactive, defects escape	Proactive, fewer escaped defects

Shift-Right Testing

Whilst shift-left testing focuses on prevention, shift-right testing extends quality practices into production environments. This approach recognises that some issues only manifest under real-world conditions with actual user behaviour, production data volumes, and infrastructure complexity.

Shift-right testing complements shift-left by validating assumptions, monitoring quality in production, and gathering insights that inform future development.

What is Shift-Right Testing?

Shift-right testing moves quality activities beyond deployment into production environments, monitoring real user experiences and system behaviour under actual operating conditions.

Key principle: Not all defects can be found in test environments. Production monitoring, observability, and controlled releases enable teams to detect issues quickly and respond before widespread impact.

Relationship to shift-left: These are complementary strategies, not competing approaches. Shift-left prevents defects; shift-right validates production quality and gathers intelligence to improve future development.

Production Monitoring

Real-time tracking of application health, performance, and errors

User Analytics

Understanding actual user behaviour and pain points

Controlled Releases

Gradual rollouts to limit risk and validate changes

Why Production Feedback Matters

Realistic Conditions

Test environments cannot perfectly replicate production infrastructure, data volumes, user behaviour patterns, or third-party service responses. Production is the ultimate test environment.

Performance at Scale

Load testing provides estimates, but actual production traffic patterns reveal performance bottlenecks, database contention, and infrastructure limitations that only appear under real-world conditions.

Unknown Unknowns

Despite thorough testing, unexpected user workflows, edge cases, and integration scenarios emerge only when real users interact with the system. Shift-right helps discover these "unknown unknowns".

Continuous Learning

Production data informs development priorities, reveals user pain points, and validates whether features deliver intended business value. This feedback loop improves product quality over time.

Shift-Right Practices: Monitoring and Observability

Metrics

Quantitative measurements: response times and error rates

Traces

Request journeys across distributed services



Logs

Event records and contextual diagnostic details

Dashboards & Alerts

Unified views and proactive notifications

Observability provides visibility into system internal state based on external outputs—the foundation of shift-right testing.

Metrics

Quantitative measurements: response times, error rates, throughput, resource utilisation, business KPIs. Visualised in dashboards for real-time health monitoring.

Logs

Timestamped records of discrete events: errors, warnings, transactions, user actions. Essential for debugging production issues and understanding system behaviour.

Traces

Request journey through distributed systems: tracking a single transaction across microservices to identify bottlenecks and failures in complex architectures.

Shift-Right Practices: Controlled Releases



Canary Releases

Deploy new versions to a small subset of users first (5–10%), monitor for issues, then gradually increase traffic. If problems occur, rollback affects minimal users.



A/B Testing

Route different user segments to different feature variants simultaneously. Compare metrics (conversion rates, engagement, performance) to determine which version performs better.



Feature Flags

Deploy code with features disabled, then enable selectively for specific users, regions, or customer segments. Enables testing in production without full rollout risk.



Blue-Green Deployment

Maintain two identical production environments. Deploy new version to inactive environment, verify thoroughly, then switch traffic. Instant rollback capability if issues arise.

Shift-Right Practices: User Behaviour Analysis

Understanding how users actually interact with your application provides invaluable quality insights:



01

Session Recording

Replay user sessions to understand workflows and identify friction points

02

Heatmaps

Visualise where users click, scroll, and focus attention

03

Funnel Analysis

Track user drop-off through critical workflows (checkout, signup)

04

Error Tracking

Capture client-side errors users encounter in real-time

Image ID: 2SA97MJ
www.alamy.com

Shift-Right Lifecycle

Shift-right testing operates as a continuous feedback loop, not a one-time activity.

Production insights drive continuous improvement.

Deploy: Release changes using controlled rollout strategies.

Monitor: Track system health, performance, and user experience metrics.

Detect: Identify anomalies, errors, and degraded performance quickly.

Analyse: Investigate issues, understand root causes, and assess business impact.

Learn: Extract insights about user needs, system behaviour, and improvement opportunities.

Improve: Feed learnings back into requirements, architecture, and testing strategies.

Practical Example: Production Monitoring

Scenario

An e-commerce platform deployed a new payment gateway integration. All pre-production tests passed successfully.

Production Observation

Within 2 hours of release, monitoring dashboards showed payment success rate dropped from 98% to 92%. Error logs revealed timeout issues affecting specific credit card types during peak traffic periods.

Response

- Automatic alert triggered within 5 minutes of threshold breach
- Engineering team analysed distributed traces
- Identified connection pool exhaustion under high load
- Applied configuration fix within 30 minutes
- Monitored recovery and confirmed normal operation

Key Outcomes

Impact limitation: Only 420 transactions affected due to rapid detection and response.

Customer communication: Proactive outreach to affected customers before complaints arrived.

Testing improvement: Insights fed back to performance testing—added realistic peak-load scenarios.

Lesson: No amount of pre-production testing could have replicated the exact load pattern that exposed this defect. Production monitoring was essential.

Practical Example: User Analytics Reveal Quality Issue



Shall we solve
those problems?

Discovery

User analytics showed 35% of users abandoned the registration form at a specific step.

Functional testing had validated all fields worked correctly, yet users were consistently dropping off.

Investigation

Session recordings revealed users struggled with password validation rules. Error messages appeared only after form submission, forcing users to guess requirements. Many gave up after 2-3 failed attempts.

Resolution

- Added real-time password strength indicator
- Displayed validation rules proactively
- Provided immediate feedback on each criterion

Result

Registration completion rate increased to 89%. This was a usability defect that passed functional testing but failed real users.