Global Journal of Engineering and Technology Advances

eISSN: 2582-5003

Cross Ref DOI: 10.30574/gjeta

Journal homepage: https://gjeta.com/

(REVIEW ARTICLE)

Automated testing in microservices environments: A comprehensive approach

Aswinkumar Dhandapani *

Akraya Inc., USA.

Global Journal of Engineering and Technology Advances, 2025, 23(02), 207-214

Publication history: Received on 04 April 2025; revised on 24 May 2025; accepted on 26 May 2025

Article DOI: https://doi.org/10.30574/gjeta.2025.23.2.0162

Abstract

This article examines automated testing practices in microservices architectures, addressing the unique challenges of validating distributed systems. Beginning with an exploration of fundamental testing challenges in microservices environments, the discussion progresses through layered testing strategies, including unit, integration, contract, and end-to-end testing methodologies. The article evaluates specialized testing tools and technologies, emphasizing API testing frameworks, container orchestration platforms, contract testing solutions, and observability tools that support comprehensive validation. Continuous integration and delivery practices are examined with a focus on pipeline design, test execution strategies, data management approaches, and optimization techniques. The organizational implications of microservices testing include team structures, skills development requirements, cost-benefit considerations, and emerging testing practices. Through a detailed examination of each dimension, the article provides a holistic view of effective testing strategies for ensuring reliability in microservices ecosystems.

Keywords: Microservices testing; API validation; Contract Testing; Continuous Integration; Distributed systems reliability

1. Introduction to Microservices Testing Challenges

Modern software development has witnessed a significant paradigm shift from traditional monolithic architectures to distributed microservices-based systems. This architectural evolution represents a fundamental transformation in how applications are built, deployed, and maintained. While monolithic applications encapsulate all functionality within a single deployable unit, microservices architecture divides these capabilities into smaller, independently deployable services that communicate through well-defined interfaces. The decomposition follows domain-driven design principles, with each service typically owning its data and business logic, enabling greater flexibility and targeted scaling [1]. This architectural transition has gained widespread adoption across industries seeking to enhance development velocity and system resilience.

The transition to microservices brings numerous advantages, including improved scalability, enhanced fault isolation, and greater development team autonomy. However, this architectural approach introduces distinct testing challenges that traditional testing methodologies cannot adequately address. Testing distributed systems requires fundamentally different approaches compared to monolithic applications due to the increased complexity of service interactions and potential failure points. The independent deployability of services, while beneficial for development speed, creates additional complexity in ensuring end-to-end system integrity. As services evolve independently, maintaining compatibility between them becomes a significant challenge that demands specialized testing approaches [1].

Distributed systems present unique testing challenges stemming from their inherent complexity. These include managing inter-service dependencies, addressing asynchronous communication patterns, handling distributed data consistency, and navigating complex deployment topologies. Network partitions, latency issues, and partial system

* Corresponding author: Aswinkumar Dhandapani.

failures can occur unpredictably in production environments, necessitating robust testing strategies that simulate these conditions. Furthermore, microservices typically operate in containerized environments with orchestration systems that add another layer of complexity to testing scenarios. Ensuring consistent test environments that accurately reflect production conditions becomes increasingly difficult as the number of services grows [2].

Automated testing plays a critical role in ensuring microservices reliability, as manual testing approaches cannot scale to address the complexity and rapid evolution of microservices ecosystems. Comprehensive test automation across all layers becomes essential for maintaining quality while supporting fast-paced development cycles. Microservices architectures demand a multi-layered testing approach that includes unit tests for individual service components, integration tests for service interactions, and end-to-end tests that validate system behavior from the user's perspective. Contract testing emerges as particularly important in this context, as it verifies that service interfaces adhere to expected behaviors, helping prevent integration issues before they reach production environments [2].

2. Testing Strategies for Microservices Architecture

A comprehensive testing strategy for microservices architecture requires multiple testing layers to ensure both individual service reliability and proper system integration. Unit testing forms the foundation of this strategy, focusing on validating the behavior of individual components within each microservice. These tests verify that isolated pieces of code function correctly in controlled conditions, typically by examining a single function or method. In microservices environments, effective unit testing becomes more critical as system complexity increases. The goal is to catch defects early in the development lifecycle when they are least expensive to fix. Unit tests should be fast, reliable, and independent, allowing developers to receive rapid feedback during implementation. They require careful consideration of testing boundaries, with an emphasis on isolating the code under test from external dependencies through techniques like mocking, stubbing, and dependency injection. This isolation ensures that failures in unit tests accurately pinpoint issues in the specific code being tested rather than in the dependencies or testing infrastructure [3].

Integration testing builds upon unit tests by verifying that different services can communicate effectively with each other. These tests validate that the interfaces between services operate correctly, ensuring that services can exchange data properly across network boundaries. While unit tests focus on individual components, integration tests examine the interactions between multiple components, services, or systems. In microservices architectures, integration testing faces unique challenges due to the distributed nature of the system. These tests need to account for network latency, serialization issues, and various failure modes that might occur in distributed systems. Integration tests typically involve running multiple services simultaneously and observing their interactions. They may utilize test doubles for external dependencies while testing real service-to-service communication. Integration testing provides confidence that independently developed services will work together as expected, catching issues that might not be apparent when testing services in isolation [3].

End-to-end testing represents the highest level of testing abstraction, validating complete business processes from the user's perspective. These tests traverse multiple services and verify that the system as a whole meets business requirements. End-to-end tests typically interact with the system in the same way users would, often through user interfaces or public APIs. In microservices architectures, end-to-end tests require carefully orchestrated environments that closely mirror production configurations. While these tests provide valuable validation of complete workflows, they present significant challenges in microservices contexts. They tend to be slower, more brittle, and more difficult to maintain than lower-level tests. Additionally, identifying the root cause of failures can be challenging as an end-to-end test might traverse numerous services. Despite these challenges, a carefully selected set of critical end-to-end tests remains essential for validating system-level behavior [4].

Contract testing has emerged as a particularly valuable approach for microservices ecosystems, focusing on validating the agreements between service providers and consumers. This methodology verifies that services adhere to the contracts they have established with other services, ensuring that changes to one service do not break dependent services. Unlike traditional integration testing, contract testing can be performed in isolation, with each service verifying that it meets its contractual obligations independently. This approach is particularly well-suited to microservices architectures, where services are developed and deployed independently by different teams. Contract tests help prevent integration issues by catching incompatible changes before they reach shared environments. They support the independent deployability that makes microservices valuable while providing confidence that the system will function correctly when integrated. Contract testing complements other testing approaches by focusing specifically on the boundaries between services, where many integration issues typically occur [4].

3. Tools and Technologies for Microservices Testing

The complexity of microservices architectures necessitates specialized tools for effective testing across various layers. API testing frameworks have become essential components in the microservices testing toolkit, enabling developers to validate service interfaces and behaviors systematically. In microservices environments, APIs serve as the primary communication channels between services, making their reliability critical to overall system functionality. Modern API testing tools offer capabilities ranging from simple request-response validation to complex scenario testing with dynamic data generation. These frameworks support various authentication mechanisms, request parameterization, and response validation strategies tailored to microservices needs. They enable teams to create test suites that verify functional correctness, performance characteristics, security properties, and error handling behaviors of service APIs. Many tools provide domain-specific languages for test specification, allowing testers to express expected behaviors clearly and concisely. The most effective API testing solutions for microservices support both black-box testing, which validates services from an external perspective, and white-box testing, which examines internal service behaviors. They integrate seamlessly with continuous integration pipelines, enabling teams to execute comprehensive API test suites automatically with each build or deployment cycle, thereby catching integration issues early in the development process [5].

Table 1 API Testing Tool Comparison. [5]

Tool Type	**Primary Use Cases**	**Benefits**	**Limitations**
HTTP Clients	Manual API exploration, simple tests	Easy to use, visual interface	Limited automation capabilities
Automation Frameworks	Comprehensive API testing	Powerful scripting, CI/CD integration	Steeper learning curve
Service Virtualization	Simulating dependencies	Testing in isolation, controlled responses	Maintenance overhead
Security Scanners	API vulnerability testing	Identifies security weaknesses	Focused scope, separate from functional testing

Container orchestration platforms have revolutionized test environment management for microservices, addressing the challenge of maintaining consistent and isolated test environments. These platforms leverage containerization technology to package services along with their dependencies, ensuring that testing environments accurately represent production configurations. Orchestration solutions provide sophisticated scheduling capabilities that place containers optimally across infrastructure resources, facilitating efficient resource utilization during test execution. They implement service discovery mechanisms that allow dynamically deployed services to locate and communicate with each other, mimicking real-world deployment scenarios during testing. Advanced container orchestration frameworks support configuration management through declarative specifications, enabling teams to version-control their environment definitions alongside application code. This capability ensures that test environments remain consistent across different stages of the development pipeline and between team members. Additionally, these platforms offer networking abstractions that simulate production network topologies, allowing teams to test service communication patterns under realistic conditions. By providing infrastructure-as-code capabilities, container orchestration tools enable testers to provision complete microservices ecosystems on-demand, execute tests against them, and then tear them down, supporting efficient and repeatable testing workflows [5].

Contract testing tools have emerged as specialized solutions for validating service interaction patterns in microservices architectures. These tools implement a shift-left testing approach by formalizing and validating the contracts between service providers and consumers early in the development cycle. Contract testing frameworks operate on the principle that if each service adheres to its contracts, the integrated system should function correctly when deployed. They typically support a workflow where consumer services define expectations about provider behavior, creating executable specifications that both parties can verify independently. This approach reduces the need for complex integration environments while still providing confidence in service compatibility. Modern contract testing solutions support various communication protocols including REST, GraphQL, and message-based systems, making them applicable across diverse microservices implementations. They provide capabilities for schema validation, ensuring that service responses conform to expected data structures and types. Advanced contract testing frameworks offer matchers and rules that allow contracts to specify expected patterns rather than exact values, providing flexibility while maintaining meaningful validation. These tools integrate with continuous integration systems to verify contract compliance automatically during the build process, preventing incompatible changes from progressing through the deployment pipeline. By providing early feedback on potential integration issues, contract testing tools help teams maintain service compatibility while preserving the independence that makes microservices valuable [6].

Monitoring and observability solutions play a crucial role in microservices testing by providing insights into system behavior during test execution. These tools collect telemetry data across the distributed system, enabling testers to verify correct behavior beyond simple response validation. Modern observability platforms implement distributed tracing capabilities that track requests as they propagate through multiple services, revealing the execution path and timing information for each transaction. This functionality is particularly valuable for identifying performance bottlenecks and understanding failure propagation patterns during testing. These solutions aggregate and correlate logs from multiple services, providing contextual information that helps diagnose test failures in complex scenarios. They collect runtime metrics that reveal system health indicators such as resource utilization, request rates, and error frequencies during test execution. Advanced observability tools support anomaly detection, automatically identifying unusual patterns in telemetry data that might indicate potential issues not explicitly checked by test assertions. By integrating with testing frameworks, these platforms enable teams to incorporate observability signals directly into test assertions, creating more comprehensive validation criteria. The insights provided by monitoring and observability tools extend beyond basic functionality testing, supporting performance testing, chaos engineering experiments, and resilience testing in microservices environments. This comprehensive approach to validation helps teams identify subtle interaction issues that might not be apparent through traditional testing methods [6].

4. CI/CD Integration and Continuous Testing

Effective Continuous Integration and Continuous Delivery (CI/CD) pipeline design is foundational to successful microservices testing strategies. In microservices architectures, pipelines must accommodate the distributed nature of services while ensuring comprehensive validation across the entire system. This requires thoughtful structuring of pipeline stages that support both independent service verification and system-level integration testing. Modern CI/CD implementations for microservices typically feature multi-branch strategies that enable feature development in isolation before merging changes to main branches. These pipelines implement sophisticated orchestration that triggers appropriate test suites based on the nature and scope of code changes. Effective pipeline design incorporates environment management strategies that provision isolated testing environments with the necessary infrastructure components, including databases, message queues, and external service simulators. Security scanning and compliance validation are integrated directly into the pipeline workflow, ensuring that non-functional requirements are verified alongside functional correctness. Pipeline configurations are typically managed as code, enabling version control, peer review, and automated validation of pipeline changes themselves. Monitoring and analytics components provide visibility into pipeline performance, helping teams identify bottlenecks and optimization opportunities. These pipeline designs balance the need for thorough testing with delivery speed by implementing quality gates that verify critical aspects of service behavior before allowing progression to subsequent stages [7].

Table 2 CI/CD Pipeline Stages for Microservices. [7]

Pipeline Stage	Testing Activities	Environments	Gate Criteria
Build	Static analysis, unit tests	Developer workstation	No code quality issues, unit tests pass
Component	Service-level validation	Isolated containers	Component tests pass, code coverage thresholds
Integration	Service interaction validation	Test environment	Contract tests pass, integration tests pass
System	End-to-end validation	Staging environment	End-to-end tests pass, performance metrics
Production	Synthetic monitoring, canary testing	Production	Health checks, user impact metrics

Automated test execution strategies in microservices environments must balance comprehensiveness with efficiency to support rapid delivery cycles. Continuous testing practices integrate test automation throughout the development lifecycle rather than treating it as a separate phase. This approach requires close collaboration between development and quality assurance teams to design testable services and implement appropriate test coverage at multiple levels. Effective test automation in microservices environments implements the testing pyramid concept, with numerous fast-executing unit tests at the base, service-level component tests in the middle, and a smaller number of end-to-end tests at the top. Test instrumentation plays a crucial role in these strategies, providing the hooks and extensions necessary to exercise services in isolation and validate their behavior. Automated test execution also requires sophisticated reporting mechanisms that aggregate test results across multiple services and test types, providing a holistic view of system quality. Many organizations implement progressive testing approaches that execute different test suites at different pipeline stages, with fast-running tests providing early feedback and more comprehensive tests running later. These execution strategies often incorporate retry mechanisms for intermittent failures, quarantine processes for flaky tests, and dynamic test selection based on risk assessment. By thoughtfully designing test execution strategies, teams can maintain high confidence in system quality while preserving the rapid feedback cycles that enable continuous delivery [7].

Table 3 Microservices Testing Pyramid.

Testing Level	**Characteristics**	**Scope**	**Execution Speed**
Unit Tests	Isolated, no external dependencies	Single component/function	Very fast
Component Tests	Service-level, stubbed dependencies	Individual service	Fast
Contract Tests	Validates service interfaces	Service boundaries	Moderate
Integration Tests	Multiple real services	Service interactions	Slow
End-to-End Tests	Complete system validation	User workflows	Very slow

Test data management presents unique challenges in distributed microservices environments, where data consistency across services is essential for meaningful test results. Effective test data strategies consider the entire data lifecycle, from generation and provisioning to cleanup and archiving. Many organizations implement dedicated test data management services that provide APIs for creating and manipulating test data across multiple databases and storage systems. These services typically support data templating capabilities that enable the generation of diverse test datasets while maintaining referential integrity across service boundaries. Containerization approaches to test data management have gained popularity, allowing teams to create isolated database instances with pre-populated data that can be rapidly provisioned and disposed of after test execution. Data virtualization techniques enable efficient sharing of read-only reference data across test environments while maintaining isolation for transactional data. Masking and anonymization processes ensure that sensitive production data can be safely used in test environments without exposing protected information. Effective test data management also includes monitoring and cleanup mechanisms that prevent test data accumulation over time, avoiding performance degradation in test environments. These approaches collectively enable teams to create realistic test scenarios while maintaining the isolation necessary for reliable and repeatable test execution [8].

Optimizing test execution time and resource utilization requires both technical solutions and strategic testing approaches to prevent testing from becoming a bottleneck in delivery pipelines. Pipeline optimization begins with performance profiling to identify execution bottlenecks, followed by targeted improvements to address the most significant delays. Caching strategies accelerate pipeline execution by preserving build artifacts, dependencies, and test environments between runs when appropriate. Infrastructure optimization ensures that test environments have sufficient computational resources while implementing efficient resource allocation to avoid waste. Many organizations implement dynamic scaling of test infrastructure based on demand, expanding capacity during peak usage periods and contracting during low-demand periods. Test optimization techniques include refactoring long-running tests into smaller units, eliminating redundant test coverage, and implementing risk-based testing approaches that focus resources on the most critical areas. Parallel execution strategies distribute tests across multiple environments simultaneously, dramatically reducing total execution time for large test suites. Incremental testing approaches execute only those tests affected by specific code changes, providing efficient verification without running the entire test suite for every change. These optimization strategies are complemented by monitoring systems that track test execution metrics over time, helping teams identify trends and continuously improve pipeline efficiency. By implementing these approaches, organizations can maintain comprehensive test coverage while supporting the rapid iteration cycles that microservices architectures are designed to enable [8].

5. Organizational impact and strategic considerations

The transition to microservices architecture significantly impacts organizational structures and team responsibilities, particularly regarding testing practices. Successful microservices implementations recognize that architectural decisions cannot be separated from organizational structures, as Conway's Law demonstrates the strong correlation between communication patterns and system design. The distributed nature of microservices requires reconsideration of traditional team boundaries and quality assurance approaches. Organizations typically evolve toward product-oriented team structures that take full ownership of their services from development through testing and operations. This transition often follows a maturity model, beginning with centralized testing expertise that gradually distributes as teams build capability. Effective implementations establish a success triangle that balances technology choices, organizational structure, and business domain understanding. Team boundaries are ideally aligned with business capabilities rather than technical specialties, creating clear ownership and reducing coordination overhead. Testing responsibilities in these structures typically reside primarily with the service-owning teams, with platform teams providing testing infrastructure and frameworks as internal products. Cross-cutting concerns like security testing and performance validation often require specialized expertise shared across multiple teams through enablement models. Organizations frequently establish internal developer platforms that provide standardized testing tools and environments, reducing the cognitive load on product teams while ensuring consistent quality practices. These transformations require significant leadership support during transition periods, as teams develop new skills and adapt to increased autonomy and responsibility for quality outcomes [9].

Table 4 Organizational Models for Testing in Microservices. [9]

Organizational Model	**Testing Responsibility**	**Coordination Mechanism**	**Best For**
Centralized QA	Dedicated testing team	Handoffs to QA	Organizations transitioning to microservices
Embedded Testers	QA specialists in each team	Testing community of practice	Balanced approach with specialized expertise
Fully Cross-functional	Developers perform all testing	Test infrastructure teams	Mature DevOps organizations
Testing Center of Excellence	The platform team provides frameworks	Enablement model	Organizations scaling microservices adoption

Skills development for microservices testing requires significant investment as teams adapt to the unique challenges of distributed systems testing. The complexity of microservices environments demands a multidisciplinary skill set that spans traditional testing expertise, infrastructure knowledge, and software development capabilities. Organizations typically implement tiered learning approaches that begin with foundational concepts like API testing and test automation before progressing to more complex topics such as distributed tracing and chaos engineering. Effective learning programs combine theoretical knowledge with practical application, often through progressive projects that allow teams to apply new skills to real-world challenges. Many organizations establish technical learning paths specific to quality engineering in distributed systems, defining clear competency models and growth trajectories. Peer learning mechanisms prove particularly effective, including communities of practice, tech guilds, and internal knowledge exchanges focused on testing practices. Mentoring programs pair experienced testers with developers to accelerate skill development and cross-pollinate perspectives across traditional role boundaries. External knowledge sources become increasingly important as microservices testing practices continue to evolve rapidly, with organizations encouraging participation in industry conferences, open source communities, and professional networks. Simulation exercises and game days provide safe environments to practice complex testing scenarios, particularly for chaos engineering and resilience testing. These comprehensive skill development approaches enable organizations to build the collective capabilities needed for effective quality assurance in distributed systems while supporting individual growth and career development [9].

Cost-benefit analysis of comprehensive test automation in microservices environments must consider both direct implementation costs and broader organizational impacts. The investment calculation for microservices testing includes immediate factors like automation tooling, infrastructure expenses, and engineering time, alongside longer-term considerations such as maintenance burden and opportunity costs. Effective organizations implement balanced testing strategies that distribute investment across multiple layers of the testing pyramid based on risk assessment and business impact analysis. The return on investment for automation typically increases over time as initial setup costs are amortized across multiple development cycles. Test maintenance represents a significant ongoing cost that organizations must account for through sustainable authoring practices and infrastructure investments. The business case for comprehensive testing typically considers multiple benefit dimensions, including accelerated delivery through increased deployment confidence, reduced incident response costs, improved customer satisfaction from higher service reliability, and enhanced developer productivity through faster feedback cycles. Many organizations implement cost-sharing models for testing infrastructure that distribute expenses across multiple teams while enabling economies of scale. Metrics-driven approaches to test investment help prioritize areas with highest business impact, focusing initial automation efforts on critical services and high-risk components. The true cost-benefit analysis extends beyond traditional ROI calculations to consider organizational learning, reduced technical debt, and improved business agility, all of which contribute significantly to competitive advantage in digital markets [10].

Future directions in microservices testing demonstrate a clear trend toward increased intelligence and automation in testing practices. The emerging landscape of microservices testing shows convergence between traditional quality assurance approaches and operational concerns, with observability becoming a central component of comprehensive testing strategies. Testing practices increasingly shift both left and right in the development lifecycle, with earlier validation through practices like specification by example complemented by production testing through progressive delivery techniques like canary deployments and feature toggles. Advanced visualization tools for service dependencies and data flows help teams understand the implications of changes across distributed systems, enabling more targeted testing approaches. Testing increasingly incorporates real-world conditions through techniques like traffic replay and production simulation, creating more realistic validation scenarios without production risk. Continuous verification replaces point-in-time testing in mature implementations, with persistent test suites that constantly validate system behavior against specifications. The distinction between testing and monitoring continues to blur, with synthetic transaction monitoring providing ongoing validation of critical user journeys in production environments. Machine learning approaches show promise for anomaly detection and test optimization, analyzing patterns in system behavior to identify potential issues before they impact users. Container-based approaches to test environment management continue to evolve, with improved isolation, resource efficiency, and environment fidelity. These advances collectively enable more comprehensive testing with reduced overhead, helping organizations manage increasing system complexity while maintaining quality and delivery velocity [10].

6. Conclusion

Automated testing in microservices architectures represents a multifaceted challenge requiring balanced technical and organizational strategies. The transition from monolithic to distributed testing necessitates adopting layered testing approaches, specialized tooling, and mature CI/CD practices to ensure system reliability. Contract testing emerges as particularly valuable for maintaining interface compatibility while preserving service independence. Effective implementation requires thoughtful organizational structures that balance team autonomy with system-level quality concerns, alongside significant investment in skills development across multiple domains. The cost-benefit equation favors comprehensive automation despite increased complexity, particularly when considering long-term benefits to delivery speed, service reliability, and reduced incident costs. As microservices ecosystems continue to evolve, testing practices show convergence between quality assurance and operational concerns, with increasing emphasis on observability, production testing, and intelligent automation. Organizations that master these testing challenges position themselves to fully realize the benefits of microservices architecture while managing its inherent complexity.

References

[1] Daniel Lebrero, "Book notes: Building Microservices - Second edition," 2023. [Online]. Available: https://danlebrero.com/2023/01/24/building-microservices-second-edition-designing-fine-grained-systems-summary/

[2] Navdeep Singh Gill, "Microservices Testing | Strategies and Processes for Enterprises," XenonStack, 2024. [Online]. Available: https://www.xenonstack.com/blog/microservices-testing

[3] Toby Clemson, "Testing Strategies in a Microservice Architecture," ThoughtWorks, 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

[4] Shatanik Bhattacharjee, "Microservices testing: Strategies, tools, and best practices," vFunction, 2024. [Online]. Available: https://vfunction.com/blog/microservices-testing/

[5] Sandhya Karande, "API Testing in Microservices: A Comprehensive Guide," Techify Solutions, 2024. [Online]. Available: https://techifysolutions.com/blog/api-testing-in-microservices/

[6] Hypertest, "Top Contract Testing Tools Every Developer Should Know in 2024," 2023. [Online]. Available: https://www.hypertest.co/contract-testing/best-api-contract-testing-tools

[7] Hannah Son, "Continuous Testing in DevOps: A Comprehensive Guide from Strategy to Execution," TestRail, 2024. [Online]. Available: https://www.testrail.com/blog/continuous-testing-devops/

[8] Microtica, "How to Optimize Your CI/CD Pipeline for Faster Deployments," 2025. [Online]. Available: https://www.microtica.com/blog/optimize-your-ci-cd-pipeline-for-faster-deployments

[9] Chris Richardson, "The evolution of the success triangle: microservices as the enabler of DevOps and team topologies," microservices.io, 2024. [Online]. Available: https://microservices.io/post/architecture/2024/03/28/success-triangle-microservices-as-an-enabler.html

[10] Pratik Patel, "End-to-End Microservices Testing for Modern Applications," AlphaBin, 2025. [Online]. Available: https://www.alphabin.co/blog/end-to-end-microservices-testing