Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Theoretical Frameworks for API-First and Shift-Left Quality Engineering in Microservices Architectures

Navya Reddy Kunta

Independent Researcher, USA

ARTICLE INFO

Received: 11 Feb 2026

Revised: Feb 2026

ABSTRACT

Microservices architectures have shifted the model of how quality is assured. Specifically: end-to-end screening strategies used in monolithic architectures are not viable because UI centric approaches have delays in defect discovery; and resolving architectural defects post integration requires orders of magnitude more effort than resolving defects found at the level of service contracts. API first testing provides mechanisms for coping with the unmanageable exponential growth in possible test scenarios that arise from service dependencies in microservice architectures and other distributed systems. Contract-driven development specifies service interfaces in an executable way as provider-consumer contracts and generates testable contracts that enforce contract-test-first as an independent way to evolve and deploy services. Consumer-driven contract testing breaks up validating a contract into discrete steps, where consumer tests generate compatibility specifications as contract files and provider tests replay requests and responses against APIs. Defect seeding experiments show that an important percentage of integration defects can be detected in the consumer, provider, and HTTP-structural categories, although value range changes are a limitation of this method. The underlying economic rationale for shift-left quality activities is an exponential relationship between defect discovery delay and defect cost. Early-discovered defects are cheaper to fix than those found in a late phase of the software life cycle (after product deployment). Empirical studies suggest that shift-left quality activities can considerably lower defect levels, cost of rework, and lead-time, if a design-time validation philosophy is substituted for system test-time verification.

Keywords: API-First Testing, Microservices Architecture, Contract-Driven Development, Shift-Left Quality Engineering, Test Automation

1. Introduction

1.1 The Quality Assurance Challenge in Distributed Systems

The shift from monolithic packages to microservices based totally software program application architectures together with a set of independently deployable and attainable services poses a assignment to regular software program software amazing warranty and a fundamental trouble for software program software checking out and validation due to the widely numerous lifecycles, launch frequency & resiliency homes. Classic testing pyramids, which advocate for thorough end-to-end testing of the entire GUI application, fail to work properly when the services are not isolated and the number of possible test scenarios grows exponentially [1].

Microservices also introduce challenges not found in monolithic applications. Service decomposition leads to distributed state management, network message overhead, and failure propagation patterns that are not easily detected with UI testing. However, if the contracts, versions or behaviors of the services are incompatible, then they have the potential to fail at every service boundary. User interface (UI) testing cannot easily reach all combinations of services in integration tests. Because microservices are independently deployable, they are consumed by other services through a more varied set of communication styles, from synchronous REST calls, through asynchronous message queues to event streams. This leads to many more integration points and, since microservices are

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

distributed, the economics of testing change, making it expensive in compute and time to do full end-to-end testing [1].

The fundamental issue is that defect introduction and defect detection are separated in time and space. Inconsistent architecture, violations of the contract, and integration problems are detected later (usually during UI testing) and require rework that is orders of magnitude greater than rework done earlier. More generally, when a defect traverses multiple integration levels, root cause analysis is much more difficult as the fault conditions must be recreated across potentially distributed elements, and the cost of debugging is only part of the problem. There are also the costs of coordinating across multiple teams, regression testing, and rolling back the deployment of dependent elements [2].

Experience has demonstrated that defects are cheaper to fix if they are discovered earlier, and reliability engineering techniques such as Failure Mode Effects and Criticality Analysis (FMECA) and Reliability Block Diagrams (RBDs) show a clear cost advantage to discovering defects as soon as they occur. Static modeling, for example FMECA, can expose architectural weaknesses for component failure modes and system availability, prior to production. The aerospace and defense sectors apply the FMECA model with particular stringency, as these applications have high reliability requirements. It is also used to model failure propagation and recovery patterns. Despite evidence that these systems reduce critical failures and make recovery more predictable [2], such techniques remain uncommon in modern software engineering practice.

The economic trade-offs require verifying architectural assumptions and service contracts in advance, before costly system integration. Reliability block diagram modeling of the distributed architecture shows that the overall system availability is dominated by the weakest component, and each tier in the architecture must have higher availability requirements than the end-to-end requirements. For web-enabled applications designed using a three-tier server architecture, in order to provide four-nines (0.9999) availability of the target system, the subsystems must each provide five nines (0.99999) availability. Hence, there must be redundancy on all levels of the software architecture. Quantitative availability modeling proved that node availability is a dominating factor limiting system availability, and that redundancy is the correlate to component unavailability [2].

1.2 Research Gap and Contribution

While there is broad acceptance of the shift-left principles, little research has contributed practical, repeatable metrics to investigate the efficiency of an API-first testing approach in a production microservices environment. Existing research focused on the idea or on small laboratory testbeds, but did not target large microservices systems with real world complexity. The relationships between test coverage in contract testing, escape rate of defects, and cost of quality, under various architectural styles and organizational contexts, are not well understood. Approaches for availability modeling are mathematically rigorous, but are seldom used in practice by software development organizations due to their mathematical sophistication and the assumptions they use, which do not hold in all system configurations [2].

Beyond quantitative aspects, academic research has addressed practical aspects of contract-driven development, including change control management, criteria for tool selection, and harmonizing new practices with existing development workflows. However, little guidance has been given to organizations with established development workflows and legacy testing frameworks on how to address the adoption of contract-driven development and shift-left quality principles. Although many methods from reliability engineering have been applied to software systems, because of the lack of failure data from software parts, the predictive availability modeling techniques that have been shown to be useful for hardware reliability modeling are difficult to apply to software reliability modeling because these models require data about the failure modes of the components in the system. While

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

not accurate in absolute terms, comparative studies of alternative design choices and sensitivity analyzes are useful [2].

This article addresses these shortcomings with theoretical models for API-first quality engineering and quantitative evidence from several microservices case studies on how applying validation at the service contract layer at early development stages can detect design and integration flaws early in the development lifecycle before investing heavily in the implementation. Our contribution is a contract testing framework for stabilizing the architecture, in order to support the independent evolution of services while maintaining the overall system behavior. The framework takes existing successful practices from aerospace and hardware reliability engineering and adapts them for software-intensive distributed systems. The approach also stresses that the systematic creation of reliability prediction models helps developers to better understand dependencies between components and architectural weaknesses and that the effort itself is helpful [2].

2. API-First Testing Methodology and Architectural Foundations

2.1 Contract-Driven Development Principles

An interface is pre-defined as part of contract-driven development. These contracts are agreements between the providing service and using service that describe behavior and are executable as tests before implementation begins. This is the opposite of the customary programming life cycle where the interface is created from the implementation. As a result, often expectations about behavior are not expressible, except by tests. Unfortunately, with no compiler or other syntactic solution, there is currently no way to prove that two networked microservices are interoperable at compile time, or to prove that they conform to the same contract at run time. Consumer-driven contract testing removes the need for that by splitting validation into two separate stages [3]: consumer tests produce compatibility specifications as contract files, which are replayed against the API within provider tests.

Beyond identifying corner-case issues, a second major benefit of contract tests is autonomous service evolution. Loose coupling and independent lifecycles are key tenets of production microservice architectures, and consumer-driven contracts enable provider teams to safely perform implementation refactoring, performance optimization, and technology replacement on their services with no coordination overhead, provided the contract is fulfilled. The decoupling mechanism relies on the use of contract files to remove the need to run consumer and provider services simultaneously. Research on distributed development teams has shown that contract testing enables parallel development, with service teams able to work independently from each other through the use of contract stubs and modifies the way they coordinate. This decoupling of cross-team synchronization dependencies is particularly useful for organizations that develop software with many autonomous teams across geographic boundaries, where synchronization latency normally constrains development speed [3].

Action research of microservices systems found that contract testing can be used both to validate systems and to enable inter-team communication. The research applied contract testing in production systems, and reported four integration issues that existed in initial microservices systems. Two of the issues were due to differing assumptions about the applicable value ranges of JSON attributes, while the other two were about the rules for whether certain JSON attributes were optional or mandatory. Both had happened during contract test development and not during production use, showing the preventative utility of the practice. Developers interviewed after being exposed to contract testing have lauded benefits including improved API design by forcing them to think in terms of the consumer perspective, greater consistency in API design, and implementation of the robustness principle by only parsing the properties of a payload they required [3].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

2.2 REST and SOAP Protocol Validation Strategies

Unlike SOAP, REST is not a protocol but an architectural style and therefore the test will differ from one to the other. For example, REST is resource-oriented and stateless with a uniform interface, thus validation may be focused on HTTP method semantics, status codes, and resource representations. The representational state transfer (REST) architecture, which is used to improve performance, scalability, simplicity, modifiability, visibility of communication, portability, and reliability, restricts its APIs to HTTP methods such as DELETE, GET, POST and PUT. API description formats such as OpenAPI Specification can aid automating REST conformance validation with the help of automated testing frameworks. The fact that REST implementations can be quite flexible makes it challenging to test them since there are no standard descriptive documents to rely on [4].

Efforts to systematically review REST API testing approaches have identified many issues that can complicate REST API testing. Security mechanisms can make automated testing challenging. The requirements for authentication and authorization are often not specified in API description documents. In practice, token-based, cookie-based, and API key-based authentication types are frequently used. In token-based authentication, an encrypted identifier is created on the client side and is valid only for a limited period of time after verifying the user's credentials. Cookie-based authentication works differently depending on the framework in question, while API key-based authentication negates the use of credentials. However, since these secret keys are pre-obtained, they limit the ability to write automatic tests against the protected endpoints. Also, by not testing authenticated endpoints during automated tests, code coverage decreases, as protected API usage is usually a majority of the production API surface area [4].

Inconsistency with description document best practices and maintenance of API specifications complicates validation. Organizations can maintain REST API documentation in XML, raw JSON or OpenAPI specifications without a universal standard for the method. Fast-paced development and the demand for fast rollout of production APIs may mean that test suites are either incomplete or incompatible if they fall out of sync with the implementation. APIs that utilize complex or composite data types such as file uploads, or requests with nested properties, can have difficulty being automatically generated as a test suite, as the range of possible inputs is indeterminable from structure alone. Evidence for this is that these constraints reduce the amount of code covered and authenticated endpoints are the major obstacle [4].

In assessing contract testing as a form of integration testing, defect seeding experiments quantify the effectiveness of contract testing at detecting failures under various conditions. Contract tests detected 41 of the 53 seeded integration defects (77%). The faults were grouped by the consumer or provider, to request, reply, or event, and whether they were due to removal, renaming, addition, or changes in value ranges. The omitted faults were as follows: 11 of them were value range changes to attributes, query_parameters, or path_parameters. As contract tests sample values in the range, rather than checking boundaries, they systematically suffer from blind spots for defects that involve expansion or contraction of ranges [3].

These limitations expose a key tradeoff between testing contracts for syntactic compatibility only, and testing all of the possible inputs to a contract. Our empirical results show contract testing is highly effective at detecting incompatibilities, especially those based on structure and type, but that other validation techniques complement contract testing to provide full quality assurance. Contract testing can be used in conjunction with functional testing, property-based testing, and boundary value analysis to allow for diverse testing techniques to identify different types of bugs without increasing testing time [3].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Testing Aspect	Contract-Driven Development	Protocol Validation
Interface Definition	Pre-defined contracts before implementation	REST resource-oriented, SOAP protocol-based
Service Independence	Parallel development with contract stubs	Dependent on API description formats
Validation Focus	Consumer-provider behavioral agreements	HTTP methods, status codes, authentication
Defect Detection	Structural and type incompatibilities	Security, data types, endpoint protection
Team Coordination	Decoupled cross-team synchronization	Specification maintenance challenges
Testing Limitations	Value range boundary blind spots	Authentication barriers, complex data types
Quality Assurance	Effective for syntactic compatibility	Requires complementary testing techniques

Table 1. API-First Testing Methodology: Contract-Driven Development and Protocol Validation [3, 4].

3. Shift-Left Quality Engineering: Temporal Defect Detection Optimization

3.1 Economic Analysis of Defect Detection Timing

An economic justification for shift-left quality practices is that defect remediation costs have an exponential relationship with the time required to detect defects. Studies show defect remediation costs exponentially increase at each phase of the software development lifecycle, so fixing a defect in the early phases is cheaper than after deployment. Defect economics studies show that defects in requirements and design are orders of magnitude cheaper to fix than defects found in production. Costs for fixing defects can grow exponentially as the defect propagates through the subsequent project phases, with integration defects being 10 to 100 times more expensive to fix than design defects [6].

There are costs associated with other services. Correcting late-stage defects may involve changes to multiple dependent services, regression testing, and coordination of deployment across multiple teams thereby increasing the cost and time required to fix the error. If design errors arise in architecture during system testing or production operation, identifying the cause of the failure is often complicated by the underlying failure propagation patterns in a microservice ecosystem. Cascade failures along dependencies often have a high noise-to-signal ratio, making it difficult to identify the faulty component. If one service fails, then all parent services also fail, concealing the root cause. A specific type of failure propagation, it is central to microservices fault localization, since the root cause can only be identified among the many affected services [5].

Quantitative data on shift-left testing shows a 40% decrease in defects and a 30% decrease in rework cost when organizations move testing activities left to earlier stages of the requirement and design stages, compared to coding stage testing. The economics of shift-left testing also have opportunity costs, as late-stage defects slow down the pace of releases, and take engineering time away from developing new features. Examination of release schedules shows that when testing is done in a proactive manner in design review instead of a reactive manner in system testing, the time taken by the release schedule is reduced by 25%. The maximum savings come through when the costs of

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

correcting a defect in the later phases are compared with the costs of preventing the defect in earlier phases [6].

Detecting issues sooner can also have an impact on overall cost, the organization, and software correctness. Mean times to resolution depend heavily on when an issue is detected, but it can be easier to debug issues and more obvious on who is responsible. This isolates the failure domain, so that environmental differences and service dependencies that would confound debugging after the fact do not play a role. It is also important to factor in the costs of incident response, customer communication and penalties on service level agreements from production bugs. Organizations using a full shift-left approach also see a reduction in mean time to detect defects by 50%, since automated testing provides instant feedback during development, rather than during integration and production [6].

3.2 Architectural Flaw Detection at Service Boundaries

Service boundary validation prevents architectural failures, when passed through integration layers, from cascading and causing debugging problems. Contract testing and causal analysis applied to service boundaries enable structural failures to be detected earlier. These include incorrect runtime dependencies, circular service reference violations, and behavioral incompatibilities, which may cause transient and non-deterministic failures to occur in production. The boundary validation occurring early in a microservices system is quite natural concerning the task of root cause analysis. However, working with a production failure is more complicated and the noise of failure propagation causes difficulties in analyzing the failure. More generally, the cascade of secondary failures resulting from the primary failure complicates fault localization in distributed systems [5].

Boundary testing separates out the failure domains and isolates environment issues that would otherwise obfuscate root cause discovery. Contract-based validation of service boundaries provides an obvious failure attribution, either the failure is due to a provider implementation error or due to a consumer expectation error, without the complexity of running the entirety of services in unison. This provides an important reduction in the diagnostic burden compared to integration tests where distributed traces, inter-service communication, and temporal dependencies need to be interpreted. Applying recent developments in causal inference to the systematic application of causal discovery algorithms to perform fault diagnosis of microservice systems thus amounts to the problem of interventional target identification using causally-inferred interventions in a faulty target node [5].

In quantitative studies, the effect of shift-left testing is an increase in defect detection and development speed. Companies that employed exhaustive unit testing and achieved 90% code coverage saw 60% fewer regression bugs than the baseline testing strategy. The implementation of CI/CD pipelines, and the attached quality gates, reduced the mean time to detection by half, with continuous feedback providing the ability to detect defects in the development stage rather than integration stage. The cycle time for deploying changes was reduced by 40%, allowing for faster feature delivery while maintaining quality control. This was done through early testing, automated static code analysis, and continuous testing, resulting in a 35% defect reduction after deployment [6].

Boundary validation encourages parallel development by allowing teams to decouple their development cycles using contract stubs. In distributed systems, we often have to deal with coordination problems. For instance, customary microservices development is often limited by sync bottlenecks between services, which only have a contract-based interface. We show that hierarchical and local learning root-cause analysis algorithms can be made to scale to production microservices systems, achieving high recall while keeping relatively good performance. Evaluation on synthetic data with increasingly complex clusters of nodes shows that more advanced causal discovery methods that can identify interventional candidates outperform simple correlation-based or expert knowledge based baselines, with a top-1 recall of 98%. The method is shown to be practical in experiments on

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

real-world production microservices systems under different types of failures: resource exhaustion, configuration and cascading failures [5].

Quality Dimension	Early Detection Approach	Late Detection Challenge	Strategic Benefit
Defect Remediation Cost	Design and planning phase corrections	Post-deployment fixes with coordinated changes	Exponential cost reduction
Root Cause Analysis	Clear failure attribution at service boundaries	Complex analysis due to cascade failures	Simplified diagnostics
Failure Propagation	Isolated failure domains with boundary testing	High noise-to-signal ratio from dependent services	Prevention of cascade effects
Rework Effort	Minimal changes in early phases	Multiple service modifications with regression testing	Resource optimization
Testing Integration	Requirements and design phase validation	System testing and production operation	Overall defect rate reduction
Release Timeline	Proactive design-phase testing	Reactive system testing with delays	Accelerated deployment cycles
Mean Time to Resolution	Immediate feedback during development	Delayed discovery with debugging complexity	Faster defect remediation
Incident Management	Eliminated production incident response	Customer communication and SLA penalties	Operational overhead avoidance
Architectural Validation	Service boundary contract testing	Integration layer failure analysis	Structural defect prevention
Development Workflow	Parallel development with contract stubs	Synchronization bottlenecks in coordination	Independent team velocity
Failure Attribution	Unambiguous interface-level validation	Distributed trace analysis across services	Reduced diagnostic effort
Quality Gate Automation	Continuous integration with immediate validation	Manual post-integration testing	Real-time issue identification

Table 2. Comparative Analysis of Early vs. Late-Stage Defect Detection Strategies [5, 6]

4. Quantitative Impact Assessment of API-First Practices

4.1 Defect Detection Metrics and Coverage Analysis

API-first implementations have shown promising improvements to standard quality measures, and API test generation tools have successfully discovered integration bugs as well as architectural flaws. Furthermore, search-based test generation methods have reached the ability to automate complete software verification of REST APIs by exploring service boundaries, endpoint payloads, and response structures. Empirical studies of software test generation have shown that systematic generation yields test suites with a statement coverage between 32% and 65% in different implementations of

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

microservices, which is lower than that of developer-written test suites containing unit tests, integration tests and system tests [8].

In contrast to defect detection distribution of end-to-end testing related to the phase of testing, API-first testing generated REST API tests that discovered 80 buggy API endpoints across five microservices systems by systematically exploring HTTP method, parameters, and states. Most discovered bugs were related to the handling of erroneous input, e.g., HTTP error code response handling, uncaught exceptions due to boundary cases, and invalid assumptions about the state of the application. The main causes of these 5xx statuses by the web services were programming errors, resource depletion, and violations of preconditions on the server side. The number of microservice endpoints for which a server error occurred during the automated tests ranged from 0 to 33 [8].

Contract testing techniques, which operate at service boundaries, give deterministic verification of interface contracts and enable rapid detection of breaking changes before they reach consumer services. Based on defect seeding analysis of microservices systems, contract tests captured 77% of the systematic integration defects in the consumer-side, provider-side and HTTP-structural defect categories. Out of 53 manually injected incompatibilities (configuration changes, end-point changes, message format alterations, and schema changes), only 41 were detected by contracted testing of the automatic test runs (the rest were not detected). Twelve non-detected incompatibilities involved only the range of attribute and parameter values; this shows limits of contract testing that checks for differences between individual values rather than boundaries [3].

The main challenges with microservices API evolution are communication and integration. Practitioners have reported a high overhead in coordinating breaking changes between distributed development teams. The semi-structured interviews with 17 practitioners from 11 companies revealed that manual change impact analysis is error-prone and developers could miss breaking API changes in the implementation. 66.8% of the organizations use relaxation apis to talk among microservices while 22.6% use occasion-pushed messaging styles. This dual maintenance problem and the lack of automated methods to identify changes in behavior are exacerbated by the fact that static analysis methods only identify structural changes (changes in the microservices' construction) and not semantic ones (changes in what the computation performs or the range of values in its answer) [7].

4.2 Test Execution Performance and Maintenance Efficiency

API-level tests can run much faster and have lower maintenance costs than using UI testing, which requires managing infrastructure and orchestrating a number of service calls. A benefit of contract testing over other integration testing approaches is that tests are exercised in isolation at service boundaries with no dependencies on a distributed runtime or other external services. For experiments on RESTful microservices, produced test suites range from 24 to 214 test cases for each microservice system. Each test case contains only sequences of HTTP calls intended to establish required preconditions for performing tests on the target endpoints. Final test suites contain test cases that cover at least one of the target coverage criteria: code statements, branches, method calls, and HTTP return statuses per endpoint [8].

The test maintenance effort can be reduced by reducing the number of UI test dependencies and the complexity of the test environments. Smart sampling strategies for REST API test generation can achieve 12.5% to 15.1% more average coverage of testing targets over baseline random sampling strategies for REST API test generation, and over 70% improvement for systems with hierarchical resource structures. Schema properties play a key role in determining the efficacy of smart sampling strategies. The biggest impact arises when services expose hierarchically structured resources that need to be created in a coordinated manner. A statistical analysis using Vargha-Delaney effect sizes and Mann-Whitney-Wilcoxon U-tests confirmed that at a sampling probability of 0.6, smart sampling improves performance on the systems under test [8].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Evolution of microservices APIs requires the effective management of backwards compatibility and technical debt. Microservices developers have reported that avoiding breaking changes using API extensions and optional parameters leads to erosion of the original API design and increased complexity of implementing new features and regression testing to verify that existing features are not broken. Multiple equivalent endpoints addressing typos or supporting different languages, while a workaround addressing backwards compatibility constraints, increases the complexity for a consumer as well as the cost of maintaining the system. Backwards compatibility of APIs may take between 2-5% of the development effort, and may slow down optimizations of implementations or adding more end-user friendly endpoints [7].

Unwillingness of consumers to upgrade away from a version of the API leads teams to support a deprecated API indefinitely, contradicting the microservices principle of independent development. Based on interviews, 12 of the 17 practitioners experienced difficulty in convincing their consumers to migrate after breaking changes. Consumer teams cited a lack of resources or a lack of motivation to change working integrations. As a result, the situation becomes worse when consumers belong to other organizations or generate revenue for the organization. In these cases, provider teams cannot deprecate versions, even though their maintenance burden increases. This consumer lock-in scenario can lead to services exposing 2 to 8 API versions at a time. For business critical interfaces in which key customers are involved, contract lifetimes can be indefinite [7].

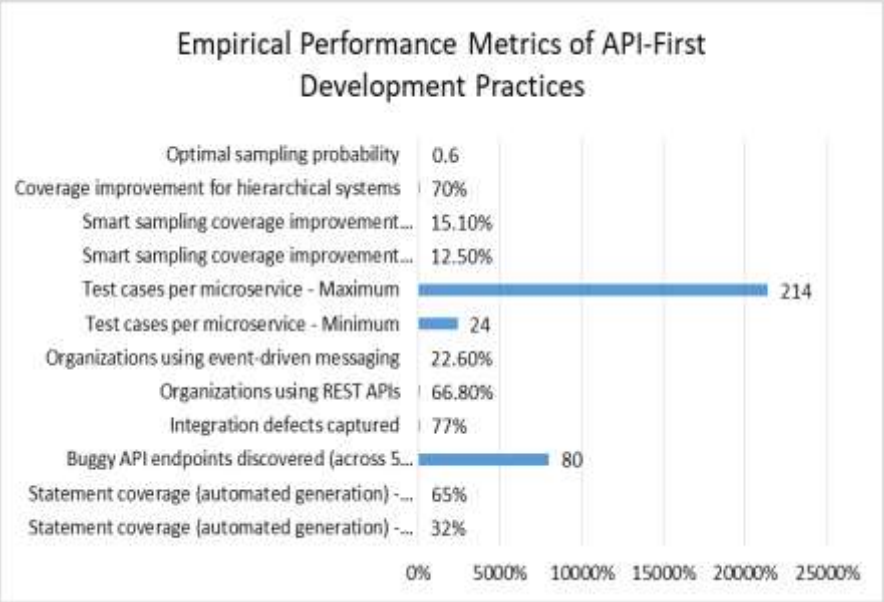

Figure 1: Empirical Performance Metrics of API-First Development Practices [7, 8].

5. Implementation Framework for Contract Testing Adoption

5.1 Technological Ecosystem and Tool Selection

Successful contract testing requires choosing validation frameworks that match the technology stack and architectural style of the organization. Consumer driven contract testing tools allow the consumers of services to express their expectations as executable specifications which are validated against the service implementation during the separate test execution stages. These artifacts can then be used in provider tests to replay these requests against the API providers and verify that the consumers' expectations are met. Pact and Spring Cloud Contracts are the two most commonly used tools for consumer driven contract testing and are the most widely adopted within organizations that are adopting contract testing [9].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

In terms of architectural consequences of consumer-driven approaches, the most important difference is that testing is reversed compared to the classic approach since interfaces are created on the basis of the implementation rather than vice versa. Systematic literature reviews have been conducted over electronic databases such as Google Scholar, IEEE Xplore, ACM Digital Library and Scopus, identifying a total of 11 papers. The 11 publications were published between 2018 and 2022, and include 7 conference papers, 3 journal articles, and 1 paper from workshop proceedings. This classification shows that contract testing is widely accepted, presented in many conferences and journals [9].

To prove the benefits of consumer-driven contract testing, action research has been executed using open source microservices projects. The JValue Open Data Service project, which is a set of microservices consisting of the data extraction, transformation, loading, and notification services, is the project used in order to prove the findings. With 3 core developers available for 5, 10 and 30 hours a week, a 3-phase continuous integration environment is adopted: phase 1 consists of linting, unit tests and isolated black-box tests on every microservice to validate functional requirements. Phase 2 runs functional tests against deployed system on full backend with API facade. Phase 3 published Docker container image for each microservice and used Pact library, which allowed putting the consumer tests in the build stage and the provider tests in system test stage, so they could run in parallel [9].

REST API validation framework implementations for OpenAPI specifications allow existing implementations to be served as source code for contracts lowering the adoption cost for organizations with existing service inventories. Validation of 3 HTTP-based integrations (between user interface and backend services) during the action research resulted in the discovery of 4 integration issues. Two defects targeted mismatched ranges of values on JSON attributes, and the other two targeted mismatched attributes marked as optional and required. The systematic defect seeding evaluation seeded 53 integration defects, composing consumer-side, provider-side, and HTTP-structural defect classes. Contract tests caught 41 defects, with a 77% detection effectiveness. Most undetected defects (11 out of 12) occurred in the area of changing the range of attribute values, query parameters, and path parameters, indicating the intrinsic weakness of contract testing approaches that focus solely on value and not on the entire boundary [9].

5.2 Organizational Change Management and Process Integration

Transitioning to API-first quality practice requires an organizational culture change, beyond just the tools that are used. This transformation involves common steps of participatory action research: diagnosis of the problem, planning the intervention, implementation, evaluation of the outcome and lessons learned. The 9 iterations of the JValue project in 3 months were based on a consensus strategy where each successful release was based on the criteria agreed by executing researchers and core developers during regularly scheduled subject matter meetings. Commonly raised issues included the lack of setting up contract testing, untested HTTP and AMQP communication, immature tooling configurations, and inconsistent status codes and response payloads [9].

Process integration helps with issues like contract versioning, breaking changes and provider-consumer coordination workflows. Another challenge for consumer-driven contract testing approaches that is mentioned in literature are rich contract exchange mechanisms, especially in complex multi-repository setups. In the implementation of the action research, commits were initially automated to send the contract, then continuous integration artifacts were used. Pact Broker was not used to keep the solution simple. Asynchronous message delivery is done using the transactional outbox pattern which introduces additional complexity as the database, the outbox service and the AMQP broker need to be started in order to create a contract [9].

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

A compatibility-driven version orchestration approach is a next step in the evolution of protocol description schema generation mechanisms provided with DevOps automation capabilities. For microservices systems the application APIs are generated automatically from versioned shared resources such as JSON schemas or Protocol Buffers definitions. Build systems store service names and API versions in a service catalog registered in external repositories. The service's orchestration mechanism checks whether the versions are compatible using protocol-specific compatibility validation tools. For gRPC services, the protolock utility in its containerized form detects backward-incompatible changes in interface definitions (e.g. the removal of fields or incompatible changes to field types). In JSON-Schema based protocols, subtype-based rules can be used with IBM's jsonsubschema. If the new schemas are subtypes of the previous schemas, then the schema change is non-breaking; otherwise, it is a major version increment (breaking change)[10].

The external orchestration application knows the versioning information, and the use case-based compatibility between consumers and providers, and builds an appropriate deployment scenario by selecting the most recent version of each one meeting the declared compatibility requirements. Whenever a service gets a new version of its API, the automation checks compatibility with its clients. Deployment of the new version proceeds when and only when all API consumers support it. In cases of incompatibilities, deployment must be delayed until consumers support the new version or a backward-compatible change can be made. This is most useful when stability is paramount, such as in air traffic control or power grid control. In situations where stability is not as important, such as on small web applications, internal corporate applications or experimental systems, the overhead may not be justified [10].

A previous experiment showed that the 3-component microservice chain with breaking change was successfully deployed in a Kubernetes cluster using MicroK8s running in multiple OrangePi 5 single board computers. Communication was achieved using gRPC and HTTP JSON Schema. Throughput load testing was conducted using Gatling for 10 minutes with 100 HTTP requests/second. Manually toggling breaking JSON schema changes (changing field names) automatically triggered an increment in the major version in the Jenkins pipelines, and load testing displayed HTTP errors due to JSON schema validation failing between services. When the incompatible schema versions were applied with the Blue/Green deployment strategy, the Kubernetes traffic throttling and circuit breaking safeguards were activated, while the compatible deployment coordinated with the Blue/Green deployment strategy allowed for a smooth transition to the stable service state without an error by switching the traffic only after the full service becomes healthy [10].

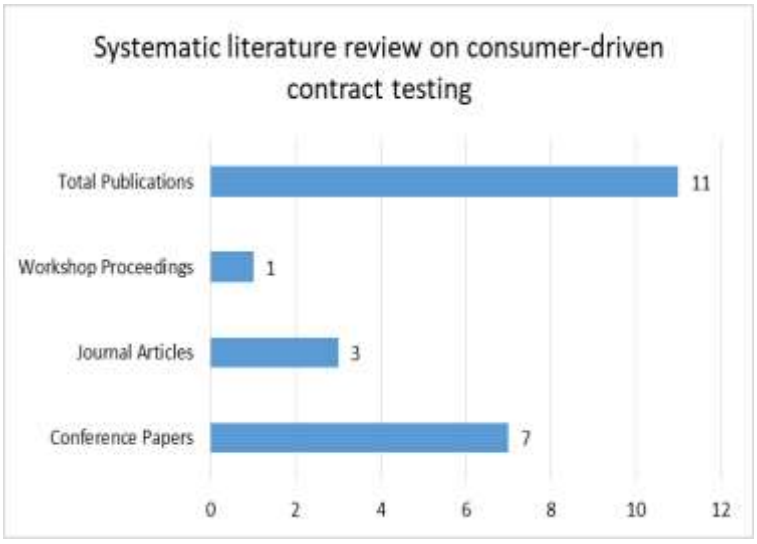

Figure 2: Systematic literature review on consumer-driven contract testing [9]

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Conclusion

There is empirical evidence that API-first quality engineering and shift-left are architectural patterns for the microservices ecosystem, and applied contract testing can considerably reduce defect density, remediation costs and improve mean time to detect (MTTD) compared to customary end to end validation approaches in all of the above categories. These benefits derive from the architectural aspects of interface validation. They allow failures to be segregated to single failure domains, an additional layer of integration environment can be omitted and assertions may be evaluated earlier during the lifecycle before substantial effort has been invested into implementing services. An API-first approach includes development workflows, quality measures and collaboration patterns. Success stories describe improvements in time-to-market, huge savings in test maintenance cost, and huge reductions in escape rates of defects to production compared to baseline strategies. For this reason, contract testing is the major technique for stabilization of distributed systems, allowing services to evolve independently while keeping the behavior of the overall distributed system coherent. Consumer-driven contract testing is a solution to the lack of syntactic interoperability (compiled type compatibility) between web APIs. The consumer specifies its requirements as executable specifications that the provider runs in isolation. The contract files decouple the provider and consumer so that the two services do not need to run concurrently when the tests are executed, changing the interaction between services in distributed development. However, achieving full coverage for semantic and value range changes is sometimes considered not feasible due to contract testing tools' analytical capabilities, and microservices APIs are especially susceptible to backward compatibility issues and technical debt, which can degrade an API's design over time due to the implementation of workarounds (in the form of additional request/response fields) or the deprecation of certain functionality, as industry practitioners have noted. Future work could include machine learning approaches to deduce contracts from the observed request and response traffic patterns, adapting contract testing to the event-driven and streaming architectures where request-to-response assertions are not sufficient and developing automated approaches to detect changes in behavior that are difficult to detect by static analysis techniques, such as implicit state changes.

References

[1] Pooyan Jamshidi et al., "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8354433

[2] K. Trivedi et al., "Availability modeling of SIP protocol on IBM WebSphere," ACM, 2008. [Online]. Available: https://dl.acm.org/doi/10.1109/PRDC.2008.50

[3] Georg-Daniel Schwarz et al., "Ensuring Syntactic Interoperability Using Consumer-Driven Contract Testing," Wiley, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.70006

[4] Adeel Ehsan et al., "RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions," MDPI, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/9/4369

[5] Azam Ikram et al., "Root Cause Analysis of Failures in Microservices through Causal Discovery," 36th Conference on Neural Information Processing Systems, 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/c9fcd02e6445c7dfbad6986abee53d0d-Paper-Conference.pdf

[6] Santosh Kumar Jawalkar, "Proactive Software Testing: The Shift-Left Approach to Early Defect Detection and Prevention," International Journal for Multidisciplinary Research, 2019. [Online]. Available: https://www.researchgate.net/profile/Santosh-Kumar-Jawalkar/publication/397880686

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

[7] Alexander Lercher et al., "Microservice API Evolution in Practice: A Study on Strategies and Challenges," arXiv, 2023. [Online]. Available: https://arxiv.org/pdf/2311.08175

[8] Andrea Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," ACM Transactions on Software Engineering and Methodology, 2019. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3293455

[9] Georg-Daniel Schwarz et al., "Ensuring Syntactic Interoperability Using Consumer-Driven Contract Testing," Wiley, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.70006

[10] Mykola Yaroshynskyi et al., "Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach," MDPI, 2025. [Online]. Available: https://www.mdpi.com/2673-6470/5/3/27