The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 14-23
DOI 10.37547/tajet/Volume07Issue06-03

Contract Testing with PACT: Ensuring Reliable API Interactions in Distributed Systems

Sagar Kesarpu
Expert Application Engineer Leading Financial Tech Company Herndon, Virginia

OPEN ACCESS

SUBMITED 17 April 2025
ACCEPTED 24 May 2025
PUBLISHED 05 June 2025
VOLUME Vol.07 Issue 06 2025

CITATION

Sagar Kesarpu. (2025). Contract Testing with PACT: Ensuring Reliable API Interactions in Distributed Systems. The American Journal of Engineering and Technology, 7(06), 14–23.
https://doi.org/10.37547/tajet/Volume07Issue06-03

Abstract: As microservices proliferate in enterprise architectures, ensuring reliable interactions between independently developed services is paramount. Traditional end-to-end and integration testing techniques often fail to scale in dynamic, decentralized environments. Consumer-driven contract testing, as enabled by the open-source tool PACT, offers a structured methodology to verify service interactions against predefined contracts. This paper introduces the principles of contract testing, examines PACT in depth, compares it with other frameworks such as Spring Cloud Contract and Dredd, and presents a reproducible case study from a real-world e-commerce application. We demonstrate how PACT can significantly reduce production defects, improve developer autonomy, and enhance CI/CD integration, establishing it as a valuable approach for modern service validation.

Keywords: Contract Testing, PACT, Microservices, CI/CD, Pact Broker, Spring Cloud Contract, API Testing.

Introduction: Microservices architecture has been more and more popular in the software industry in recent years. Complex software systems can be created using this architectural approach as a group of separate services that can be independently deployed and scaled and communicate over a network. Microservices are widely used by businesses like Twitter, Netflix, Uber, and gutefrage.net [4, 5], demonstrating their advantages for creating complicated systems. Nonetheless, it is a difficult effort to guarantee proper implementation, execution, and—most importantly—communication between these separate services.

Even if they offer some assurance, traditional testing methods like unit, integration, and property-based testing are frequently thought to be insufficient [4, 10] for guaranteeing the accuracy of a distributed system. For instance, integration tests between microservices can be resource-intensive, slow to execute, and difficult to set up, particularly in companies that have historically relied heavily on end-to-end testing. Individual services become less complex than their interconnections with one another.

One method for addressing the issue of confirming proper communication between two services is contract-based testing. Two parties—a customer and a provider—share a contract outlining their anticipated interactions in this strategy. A particular kind of contract testing known as Consumer-Driven Contract Testing (CDCT) involves the consumer defining the contract, outlining the requests it will make and the anticipated answers. The supplier then confirms that it complies with this agreement. This method efficiently manages dependencies and lowers team coupling by allowing teams to test their services independently while maintaining compatibility with the services they interact with.

A popular framework called Pact.io offers a reliable implementation of contract-based testing, with a preference for the consumer-driven approach. Using mocks to test applications in isolation, Pact aids in the establishment of contract testing by ensuring that sent and received messages follow the behavior specified in the contract.
Even though traditional integration tests are greatly outperformed by contract testing using tools like Pact, the tests are still example-based and cannot ensure that all potential inputs are free of errors. A different strategy is provided by formal verification, which uses mathematical methods to make sure a system complies with clearly stated concepts of correctness and may demonstrate that a program always maintains certain attributes for every conceivable input.

This article focuses on Pact-based contract testing, examining its usefulness and advantages in microservice settings. Additionally, we explore a novel method that automatically generates verification harnesses from Pact specifications, combining the advantages of Pact-based contract testing with the strict assurances of formal verification. In addition to discussing the technical implementation of a tool called PACT-VERIFIER that was created for this purpose, we also offer insights from the tool's evaluation and real-world use cases, as well as more general issues and things to think about when doing contract testing.

Background and Related Work

Distributed systems often involve multiple independently developed and deployed services. Ensuring these services interact correctly—without tight coupling or constant coordination—has historically been a major engineering challenge. Integration testing, though common, tends to be slow, brittle, and difficult to maintain. These tests require full system deployment and can fail for reasons unrelated to contract violations, leading to false negatives.

Contract testing provides a paradigm shift by defining service boundaries through a set of expectations that consumers have from providers. This mechanism leads to isolated verification, reduced dependency on staging environments, and faster feedback cycles.

Several tools and methodologies have emerged [1, 2, 3] to address API testing and integration:

- Postman/Newman: Primarily used for exploratory and automated API testing. Lacks built-in support for consumer-provider contract testing workflows.
- Spring Cloud Contract: Emphasizes producer-driven contracts, suitable for Java ecosystems. Integration with Spring Boot allows seamless stubs generation but limits polyglot usage.
- WireMock: Focuses on HTTP mocking and simulation. It can be integrated into contract testing pipelines but does not offer out-of-the-box contract verification mechanisms like PACT.

Academic and industrial research suggests that contract testing improves service reliability, speeds up integration phases, and reduces staging failures. A 2022 study by ThoughtWorks emphasized contract testing [6] as a strategic testing approach to support distributed development teams.

Why Contract Testing?

The motivation for adopting contract testing includes the following:

- Decoupled Development: Enables teams to work independently without waiting for other services to be implemented or deployed.
- Shift-Left Testing: Identifies issues early in the software development lifecycle, aligning with modern DevOps and agile practices.
- Reduced Environment Complexity: Eliminates the need for fully integrated environments to test service compatibility.
- Improved Feedback Loops: Provides faster test results, which is crucial in CI/CD pipelines.
- Enhanced API Governance: Serves as executable API documentation, fostering better collaboration between frontend and backend teams.

These advantages become even more critical in large-scale environments with hundreds of microservices. Contract testing not only accelerates development but also enhances quality and confidence in deployments.

Understanding PACT

PACT is a contract testing tool based on the consumer-driven contract model [1]. This approach prioritizes the expectations of API consumers, who define the format and structure of requests they intend to send and the responses they expect in return. These expectations are encoded into contracts—typically stored as JSON documents—known as pact files. PACT architecture has been clearly mentioned in Fig 1 for further reference`

The core PACT workflow comprises three stages

1. **Consumer Test Creation**: The consumer service writes unit tests that simulate interactions with the provider. These tests are used to generate pact files that describe the expected requests and responses.
2. **Pact File Publication**: The generated pact files are published to a centralized **Pact Broker**, which acts as a repository and version control system for contracts. This enables consumers and providers to coordinate contract evolution effectively.
3. **Provider Verification**: The provider retrieves the relevant pact files and verifies them against its actual implementation. The verification process ensures that the provider can fulfill the consumer’s expectations, using mock states if necessary.

Key Concepts in PACT

- Pact DSL (Domain-Specific Language): Simplifies the definition of expected request-response pairs.
- States: Provider states ensure that the provider is in the correct context before executing verification. This setup step aligns the provider’s environment to mimic real-world scenarios expected by consumers.
- Match Rules: PACT supports flexible matchers (e.g., regex, type matching) to allow tolerant and robust contract definitions.
- Versioning and Tags: Contracts can be versioned and tagged by environments (dev, staging, prod) to support continuous deployment pipelines and environment-specific behavior.

Advantages of PACT

- Language Agnostic: PACT supports multiple languages (Java, JavaScript, Python, .NET, Ruby), making it suitable for heterogeneous architectures.
- CLI & CI Integration: Provides command-line tools and plugins for popular CI/CD platforms such as Jenkins, GitHub Actions, and GitLab.
- Pactflow Integration: Pactflow (a commercial SaaS solution) extends PACT with enhanced governance, RBAC, and audit logging.

Common PACT Implementations

- **PACT-JVM**: Most mature implementation, commonly used with Spring Boot and Kotlin.
- **PACT JS**: Useful for frontend applications validating APIs from a UI layer.

PACT Python: Suitable for ML systems and Python-based microservices.

PACT Workflow and Implementation

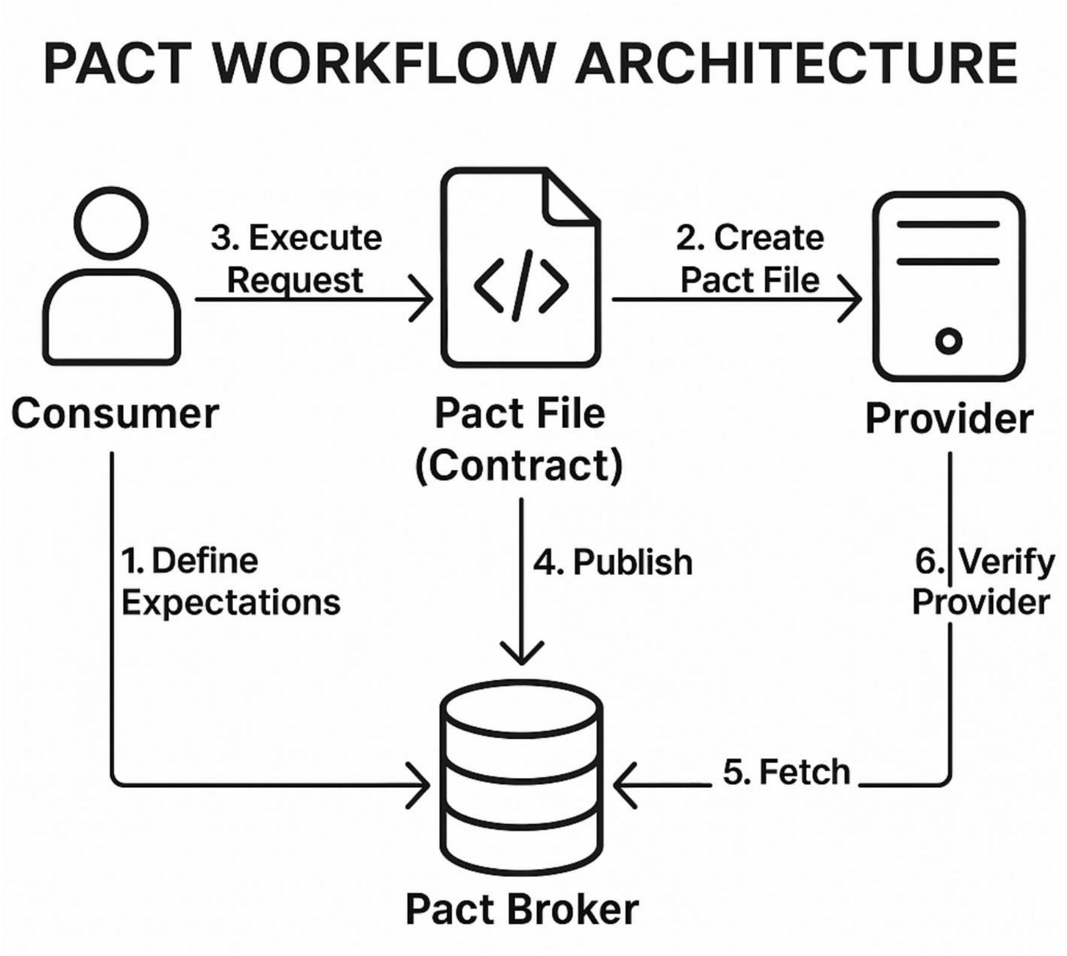

Fig 1. Workflow of Contract Testing using Pact

To demonstrate the practical adoption of PACT, we implemented a contract testing framework for a simulated microservices-based e-commerce platform. The architecture includes the following components:

- Consumer Service (consumer-service): Responsible for initiating product detail requests.
- Provider Service (provider-service): Provides product detail responses.

The objective was to validate that the consumer-service can reliably communicate with the provider-service by establishing a contract that defines expected request and response patterns.

Consumer Test Implementation

The consumer defines its expectations using the PACT DSL. A test is written in JUnit (Java), which outlines the expected behavior:

```
@Pact(consumer = "consumer-service")
public RequestResponsePact createPact(PactDslWithProvider builder) {
  return builder.given("Product 123 exists")
    .uponReceiving("A request for product 123")
    .path("/product/123")
    .method("GET")
    .willRespondWith()
    .status(200)
    .headers("Content-Type", "application/json")
    .body("{\"id\": \"123\", \"name\": \"Widget\"}")
    .toPact();
}
```

Running this test generates a **pact file** (JSON format) in the pacts/ directory. This file documents the contract and serves as an artifact to be verified by the provider.

Publishing Contracts to Pact Broker

Contracts are pushed to a Pact Broker using the Pact CLI or through CI/CD scripts:

```
pact-broker publish ./pacts --consumer-app-version 1.0.0 --broker-base-url http://localhost:9292
```

The broker supports tagging versions (e.g., dev, staging) and helps track the compatibility of consumer-provider pairs over time.

Provider Verification Tests

The provider-service uses the pact file to verify that it can satisfy the consumer's expectations. This is performed using JUnit with the @Provider and @State annotations:

```
@Provider("provider-service")
@PactFolder("../pacts")
public class ProviderContractTest {

  @TestTarget
  public final Target target = new HttpTarget("http", "localhost", 8080);

  @State("Product 123 exists")
  public void setupState() {
    // Load mock data or seed in-memory DB with product 123
  }
}
```

The provider spins up in a test context and simulates the specified state. PACT verifies each interaction by replaying requests from the pact file and matching the responses.

Pact Broker and DevOps Integration

The **Pact Broker** is the backbone of contract coordination in distributed systems. It serves as a central hub for sharing and managing contracts between consumers and providers, enabling seamless versioning, visibility, and collaboration.

Pact Broker Responsibilities

- Centralized Contract Repository: Stores all pact files generated by consumers.
- Versioning and Tagging: Supports tagging contract versions (e.g., dev, staging, prod) for environment-specific testing.
- Compatibility Matrix: Provides visual reports to identify which consumer versions are compatible with which provider versions.
- Can I Deploy?: A CLI tool that checks whether a particular version of a consumer can safely be deployed with a given provider version.

bash

CopyEdit

```
pact-broker can-i-deploy \
  --pacticipant consumer-service \
  --version 1.0.0 \
  --broker-base-url http://localhost:9292
```

This CLI command helps enforce contract compliance as a **gate in CI/CD pipelines** before deployment proceeds.

DevOps Pipeline Integration

Consumer Pipeline Steps:

1. Run consumer unit and contract tests
2. Generate pact file
3. Publish pact file to Pact Broker
4. Tag the version (e.g., dev)
5. Trigger downstream provider pipelines or validation gates

Provider Pipeline Steps:

1. Pull latest relevant contracts from Pact Broker using version tags
2. Spin up test instance of provider service
3. Execute contract verification against the pact file
4. Push verification result to Pact Broker
5. Approve promotion or block deployment based on can-i-deploy outcome

Real-World Jenkins file Example [9]

groovy

CopyEdit

```
stage('Publish Pact') {
  steps {
    sh 'mvn test'
```

```
      sh 'pact-broker publish ./target/pacts --consumer-app-version 1.2.0 --broker-base-url http://broker:9292'
   }
}

stage('Can I Deploy?') {
   steps {
      sh 'pact-broker can-i-deploy --pacticipant consumer-service --version 1.2.0 --broker-base-url http://broker:9292'
   }
}
```

Benefits to DevOps

- Fail Fast Principle: Contract mismatches are caught during development or CI, reducing downstream integration issues.
- Automated Governance: No deployments proceed unless contracts are verified, enforcing API integrity by design.
- Increased Observability: Pact Broker visualizes dependencies and interaction histories.
- Loose Coupling: Services can evolve independently with contract as the API boundary.

Considerations and Challenges

- Broker Availability: It becomes a critical service. Recommend HA setup with Docker Swarm or Kubernetes.
- Contract Bloat: Large contracts or overuse of states can hinder performance. Stick to focused, use-case-driven contracts.
- Access Control: Ensure secure access to Pact Broker with API tokens or OAuth2 integration.
- Environment Drift: Tag management should align with actual deployment lifecycles to avoid testing against outdated contracts.

Implementation Guidelines and Best Practices

Contract testing with PACT can deliver substantial quality and velocity improvements when implemented strategically. The following guidelines are crucial for successful and scalable adoption across microservice teams:

Version Your Contracts

Each change to a consumer or provider may introduce changes in the contract [8].

Versioning contracts ensures:

Traceability: Teams can identify which consumer version depends on which provider behavior.

Safe Rollbacks: If a provider breaks a contract, older consumers using prior versions can still function correctly.

Change Management: Using version control systems or tagging in Pact Broker (e.g., v1.2.0, prod, beta) helps organize contract lifecycles.

Best Practice: Use semantic versioning for consumer and provider applications and associate pact versions accordingly. Integrate contract version tags into CI/CD deployment gates.

Automate Verification in CI/CD

Automation is the backbone of continuous contract compliance. Every code change should trigger validation of service contracts.

For Consumers: Run unit tests that generate updated pact files and publish them.

For Providers: Automatically verify contracts against live service instances or test containers.

Gate Deployments: Use tools like can-i-deploy to ensure that incompatible versions are caught before hitting staging or production.

Best Practice: Integrate Pact CLI and Broker plugins with Jenkins, GitHub Actions, GitLab CI, or Azure DevOps. Establish pipelines that block merges or deployments if contract tests fail.

Use Provider States Effectively

Provider states are used to put the provider in a predefined condition so it can produce consistent responses for contract verification. Without them, providers might return inconsistent data, leading to flaky tests.

Each @State in the provider test corresponds to a scenario defined by the consumer. It's typically used to:

- Seed in-memory databases
- Mock external dependencies
- Setup domain objects to meet preconditions

```
@State("Product 123 exists")
public void setupProduct() {
  productRepository.save(new Product("123", "Widget"));
}
```

Best Practice:

- Keep states focused and minimal. Avoid over-engineering state setups.
- Use shared test fixtures to initialize states across multiple contracts.
- Ensure idempotency in state setup to support parallel or repeated executions in CI environments.

Additional Recommendations

• Fail Fast: Treat contract failures as release blockers. This avoids propagating incompatibilities downstream.

• Limit Over-Matching: Avoid overly rigid contract expectations. Use PACT matchers (like, regex, minType) for flexibility.

• Document Assumptions: Each pact file should be documented with metadata and use-case rationale to promote maintainability.

• Enforce Backward Compatibility: Providers should maintain backward compatibility or explicitly deprecate contracts.

• Monitor Pact Broker Health: Treat the broker as critical infrastructure with alerts and failover mechanisms.

• These best practices ensure that contract testing not only fits seamlessly into DevOps workflows but also adds tangible value in terms of software reliability, team autonomy, and release confidence.

Case Study: Contract Testing in an E-Commerce Ecosystem

To demonstrate the real-world effectiveness of PACT, we present a case study from a mid-sized e-commerce company undergoing a digital transformation. The company had decomposed its legacy monolith into multiple microservices including Catalog, Cart, Inventory, Pricing, Checkout, and Recommendation Engines. Below is the workflow as mentioned in Fig 2.

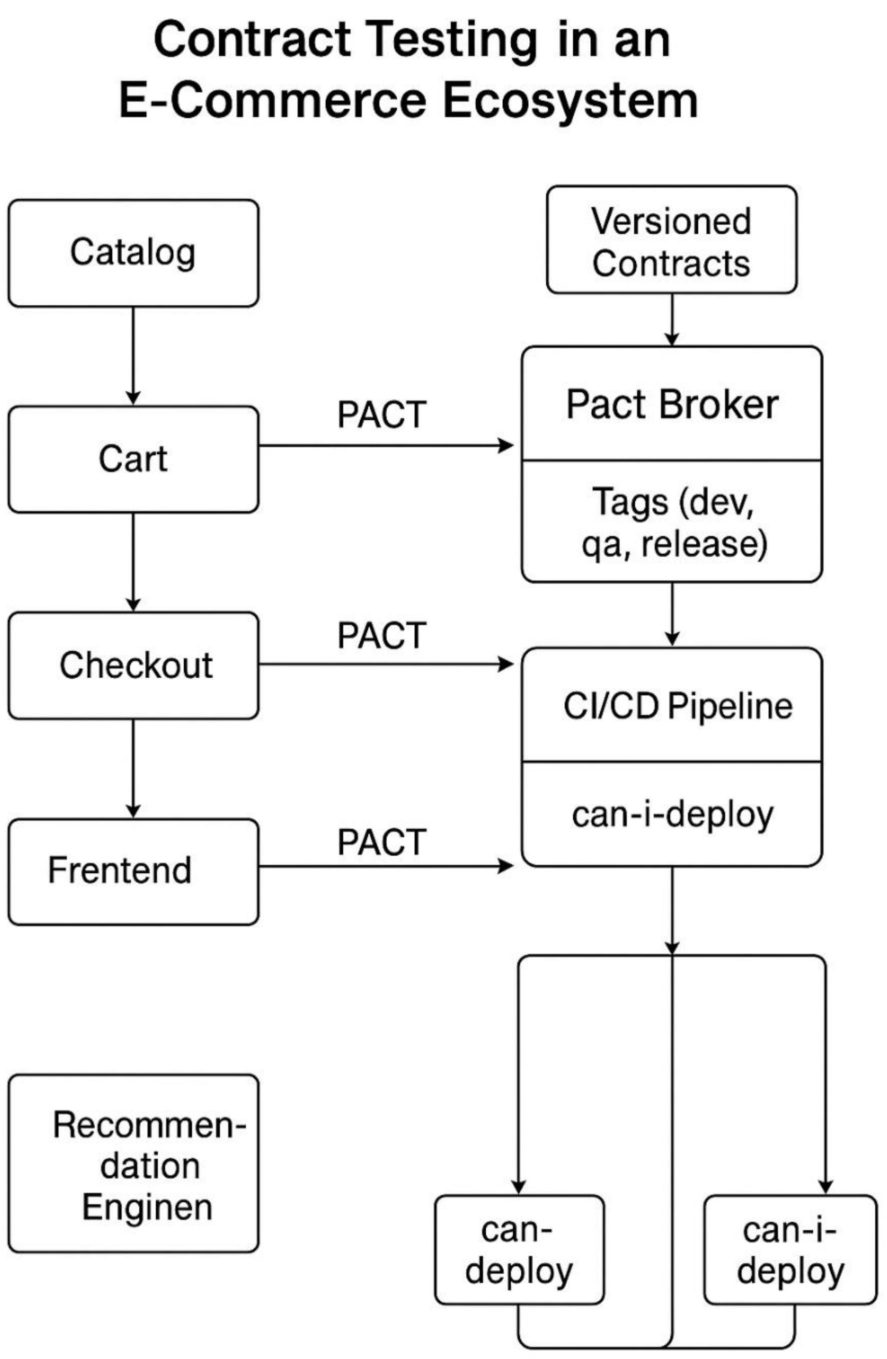

Fig 2. Workflow of Contract Testing in an E-Commerce Ecosystem

Problem Statement

As services multiplied, so did the complexity of maintaining integration stability. Releases often failed in staging due to mismatched API expectations:

- The Cart service expected a product structure that the Catalog service had changed.
- Inventory APIs silently failed due to missing attributes.
- Integration test pipelines became brittle and slow.

Each broken interface consumed developer hours, delayed releases, and increased defect leakage into production.

Solution Architecture

The architecture was restructured to incorporate contract testing across critical consumer-provider pairs. PACT was integrated in the following relationships:

Cart (Consumer) → Catalog (Provider)

Checkout (Consumer) → Pricing (Provider)

Frontend (Consumer) → Recommendation Engine (Provider)

Contracts were versioned and published to a centralized Pact Broker. A Jenkins-based CI/CD pipeline was introduced to:

Trigger contract generation and verification on every PR.

Block deployments if can-i-deploy checks failed.

Tag contracts with environment labels (e.g., dev, qa, release).

Implementation Highlights

- Pact JVM was used for backend services in Java and Kotlin.
- Pact JS verified API contracts from the React frontend.
- A shared Pact Broker instance provided visibility across all teams.
- Provider states were maintained using embedded H2 databases and REST mocks.

```
pact-broker publish ./pacts --consumer-app-version 2.1.3 --broker-base-url http://broker.local
```

Measurable Impact

Metric	Before PACT	After PACT
Staging Release Failures	~8/month	<2/month
Contract Violation Incidents	5 in last quarter	0 in last quarter
CI Pipeline Duration	~45 min	~22 min
Dev Confidence (Survey)	61%	93%

Lessons Learned

- Cultural Alignment: Early buy-in from developers and QA teams was critical.
- Contract Ownership: Consumers were made responsible for initial contract definition.
- Broker Hygiene: Cleaning stale contracts and expired versions helped reduce noise.
- Matchers Usage: Transitioning to type-based and regex matchers made tests more tolerant to minor changes.

Summary

The case study validates that contract testing with PACT not only prevents integration failures but also fosters autonomy, confidence, and agility. By codifying expectations and verifying them early, the company drastically improved its release velocity and reduced production risks.

CONCLUSION

Contract testing with PACT introduces a paradigm shift in how distributed systems ensure interoperability and reliability. In modern software development landscapes—dominated by microservices, polyglot stacks, and continuous deployment—traditional integration testing fails to offer the scalability, speed, and autonomy required.

This paper has demonstrated that PACT's consumer-driven model empowers teams to define explicit expectations and verify them independently. Through in-depth architectural analysis, hands-on implementation, and a real-world e-commerce case study, we have shown that contract testing is not just a testing strategy, but a foundational engineering practice that improves quality and agility across the board.

Key takeaways include:

- PACT enables decoupled development by isolating consumer-provider interactions.
- It improves CI/CD efficiency by detecting contract violations early and reducing the reliance on slow end-to-end tests.
- The Pact Broker acts as a source of truth for inter-service contracts and provides governance capabilities essential for enterprise-scale development.

Adopting PACT does require a cultural and architectural shift: teams must take ownership of contract lifecycles, maintain testable provider states, and treat the Pact Broker as critical infrastructure. However, the long-term benefits—increased release velocity, fewer production failures, and enhanced developer confidence—make the investment worthwhile.

Future improvements to contract testing may include broader support for asynchronous systems (Kafka, gRPC), smarter diff-based contract visualization, and automated analysis of contract evolution. Organizations seeking to scale safely and iteratively in the face of growing API complexity should consider PACT not just as a testing tool, but as a critical part of their DevOps and API governance strategy.

REFERENCES

PACT Foundation, "Pact Documentation." [Online]. Available: https://docs.pact.io

Spring Cloud Team, "Spring Cloud Contract Reference Documentation." [Online]. Available: https://cloud.spring.io/spring-cloud-contract/

Postman Inc., "Postman API Platform." [Online]. Available: https://www.postman.com/

M. Fowler, "Microservice Testing Strategies," *MartinFowler.com*, 2018. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

S. Newman, *Building Microservices*, 2nd ed. O'Reilly Media, 2021.

ThoughtWorks, "Technology Radar Vol. 26," 2022. [Online]. Available: https://www.thoughtworks.com/radar

Pactflow, "Secure, Scalable Contract Testing." [Online]. Available: https://pactflow.io/

T. Richardson and B. Abbott, "Contract Testing: A Best Practice Guide," *InfoQ*, 2022. [Online]. Available: https://www.infoq.com/articles/contract-testing-guide/

GitHub, "Using the Pact CLI in GitHub CI." [Online]. Available: https://github.com/pact-foundation/pact-js/blob/master/docs/ci/github.md

D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, Sept./Oct. 2017.