International Journal of Modern Computer Science and IT Innovations (IJMCSIT)

eISSN: 3087-4289

Volume. 02, Issue. 10, pp. 85-98, October 2025"

A Consumer-Driven Contract-Based Approach to Verifying User Interface Integration in Microservices Architectures

Ngozi Okafor,
Department of Computer Science, University of Ghana, Accra, Ghana

Article Submitted: 20/09/2025Article received: 28/09/2025, Article Accepted: 15/10/2025, Article Published: 22/10/2025

ABSTRACT

Context: Microservices Architectures (MSA) enhance deployment velocity and service autonomy, but traditional End-to-End (E2E) User Interface (UI) testing often reintroduces systemic coupling, leading to slow feedback cycles and high-test flakiness. This friction undermines the core benefits of MSA.

Objective: This research proposes and evaluates a novel, Consumer-Driven Contract (CDC) testing framework—the **CDC-UI Hybrid Model**—to strategically replace brittle E2E tests for verifying UI-to-backend integration in MSA environments. The goal is to harmonize testing across service boundaries and the presentation layer, accelerating the feedback loop.

Methodology: The study introduces a two-layer contract strategy: standard service-to-service CDC, complemented by a dedicated **UI-Consumer Contract** where the UI layer defines its expectations of the Backend-for-Frontend (BFF)/API Gateway. A simulated MSA case study was used to compare a baseline E2E-heavy approach against the proposed CDC-UI hybrid model, measuring key indicators such as test execution time, flakiness rate, and defect detection efficacy.

Results: The implementation of the CDC-UI Hybrid Model yielded a notable reduction in overall integration test execution time and a significant decrease in the test suite's non-deterministic flakiness. The approach successfully shifted the detection of UI-backend integration faults earlier in the development pipeline, correlating with a lower Defect Escape Rate.

Conclusion: The CDC-UI Hybrid Model provides a highly effective and pragmatic solution for validating UI integration in microservices. It aligns strategically with modern testing practices, dramatically improving test stability and accelerating the feedback loop, thereby preserving team autonomy and realizing the velocity potential of distributed architectures.

KEYWORDS

Microservices Architecture, Contract Testing, Consumer-Driven Contracts, User Interface Testing, Integration Testing, Test Automation, Software Quality Assurance

1. Introduction

1.1. Contextualization of Distributed System Architectures

The landscape of enterprise software development has undergone a significant transformation, moving away from monolithic architectural paradigms to more distributed and decoupled structures. Traditional **monolithic systems**, while offering simplicity in deployment and unified data management, are inherently susceptible to coupling across internal modules. This tight linkage often results in prolonged development cycles, complex maintenance, and substantial friction in the deployment pipeline, a phenomenon documented widely in the literature. As business requirements necessitate continuous delivery and rapid, independent iteration, the **Microservices Architecture (MSA)** has emerged as a prevailing solution.

MSA posits a structure where a single application is composed of small, independently deployable services, each running its own process and possessing decentralized data management capabilities. This architectural style, championed by early adopters in large-scale internet operations, fundamentally shifts the complexity of the system. In a monolith, complexity resides within the service; in MSA, it is displaced to the **network interactions** and communication protocols between services. This externalized complexity introduces a new class of quality assurance challenges that traditional testing methodologies struggle to address effectively. The core benefit of MSA—**autonomy and velocity**—is intrinsically tied to the ability of development teams to deploy their services without needing to coordinate synchronized releases across the entire system.

1.2. The Testing Imperative in Microservices

In conventional software development, the **Testing Pyramid** provides a conceptual framework for quality assurance, advocating for a high volume of fast, inexpensive **Unit Tests** at the base, a moderate number of more comprehensive **Integration Tests** in the middle, and a small number of slow, expensive **End-to-End (E2E) Tests** at the apex. This model emphasizes fast feedback and test isolation.

However, the geometric structure of the pyramid encounters considerable resistance when applied naively to an MSA environment. The nature of a distributed system means that an E2E test, typically conducted via the **User Interface (UI)**, must orchestrate and validate the correct interaction of multiple independent services, databases, and network components. This orchestration introduces considerable **challenges of End-to-End testing** in MSA:

- **Environment Complexity:** E2E testing demands a fully operational, integrated test environment that replicates the production topology, which is notoriously difficult and resource-intensive to provision and maintain.
- **High Cost and Slow Feedback:** Each E2E test execution is slow, bound by network latency and the cumulative startup time of all dependent services, dramatically increasing the time required for a full regression suite.
- **Flakiness and Non-Determinism:** Tests that rely on the correct state and timing of numerous independent services are highly susceptible to non-deterministic failures (flakiness) stemming from environmental variability, data management issues, or subtle race conditions.

When E2E tests become slow and unstable, they effectively nullify the core speed and autonomy benefits of microservices, transforming the continuous delivery pipeline into a frustrating bottleneck. The consensus in contemporary literature suggests a critical need to **de-emphasize top-of-the-pyramid tests** (E2E/UI) and find more efficient, isolated ways to gain confidence in inter-service compatibility.

1.3. Contract Testing as a Mitigation Strategy

The industry’s response to the E2E challenge is the strategic adoption of **Contract Testing**. Contract testing is a rigorous technique for verifying that the communication interfaces between two services (a **consumer** and a **provider**) adhere to an explicitly defined "contract." Crucially, this verification is performed by testing each service in isolation, mitigating the need for a complex, fully integrated test environment.

The most widely accepted and adopted pattern is the **Consumer-Driven Contract (CDC) Testing**. In CDC, the consuming service specifies the exact requests it intends to make and the minimal structure of the response it requires. This specification, the **contract**, is then shared with the provider. The provider executes an automated verification against the contract as part of its own build pipeline, ensuring that its API changes do not inadvertently break any existing consumer's expectations. This mechanism provides **fast feedback on breaking changes** and guarantees compatibility between services independently of their deployment schedules. CDC testing has proven robust for validating **service-to-service API boundaries**.

1.4. Problem Statement and Research Gap

While CDC testing is a powerful tool for internal service compatibility, a significant oversight exists at the system periphery. The ultimate consumer of the application's functionality is the end-user interacting with the **User Interface (UI)**. The UI layer, which typically interacts with an **API Gateway** or a **Backend-for-Frontend (BFF)**, is itself a critical **consumer** in the distributed architecture. Traditional CDC literature and practice often focus only on the immediate downstream service-to-service communication, neglecting the systematic application of CDC principles to the **UI-to-Backend integration point**.

This research identifies a clear **gap**: a lack of a comprehensive, prescriptive framework for **harmonizing CDC strategies with the UI testing layer**. Reliance on isolated UI component testing, coupled with the existing service-to-service CDC, still leaves a major vulnerability: a breaking change in the BFF/API Gateway's public-facing contract could still deploy and immediately fail the UI integration without a slow, high-flakiness E2E test to catch it. To fully realize the MSA promise of velocity, the integration check between the UI and its immediate backend must also be shifted-left into a fast, isolated, and deterministic test artifact.

1.5. Research Objectives and Paper Structure

This paper addresses the identified gap by proposing and evaluating the **CDC-UI Hybrid Model**, a strategic extension of CDC principles to include the UI layer as an explicit contract consumer.

The **primary objectives** of this research are to:

1. **Propose** a novel, two-layer CDC harmonization strategy that specifically integrates UI-driven contract generation.
2. **Evaluate** the proposed model's impact on key quality metrics: test execution speed and test stability (flakiness).
3. **Provide** a prescriptive methodological framework for implementing UI-as-Consumer Contract Testing to enable independent deployment of the UI and the backend services.

The paper is structured according to the IMRaD format: Section 2 details the **Methodological Framework** for the CDC-UI Hybrid Model. Section 3 presents the **Results** of the comparative analysis between the baseline and the proposed model. Section 4 offers a comprehensive **Discussion** of the findings, implications for modern testing strategies, and acknowledgments of the research limitations.

2. Methodological Framework: Integrating CDC into UI Verification

2.1. Theoretical Foundation: Design by Contract and Consumer-Driven Principle

The foundation of Contract Testing can be traced back to the concept of **Design by Contract (DbC)**. DbC specifies that a component (the provider) should formally define its precise interface constraints in the form of preconditions, postconditions, and invariants. Extending DbC to distributed systems, where communication occurs over a network, requires formalizing the inter-service communication as a set of agreed-upon interactions.

In the context of MSA, the **Consumer-Driven principle** is paramount. Instead of the provider imposing a broad, potentially over-specified interface, the consumer dictates its exact requirements. This minimizes coupling and facilitates independent evolution. When considering the UI as a **Consumer**, it is the front-end application that ultimately dictates the minimal data fields, response formats, and error codes it needs to render a component or complete a user action. The API Gateway or BFF—the immediate backend responsible for aggregating or orchestrating the downstream microservices—then becomes the **Provider** of this UI-Consumer Contract.

2.2. Proposed Harmonization Strategy (The Hybrid Model)

To systematically address the complexity of MSA and the volatility of E2E tests, this research proposes a **hybrid testing model** that significantly reduces reliance on full-stack orchestration by focusing on two distinct, yet complementary, layers of contract verification:

1. **Layer 1: Service-to-Service CDC (Internal Compatibility):** This layer addresses the

compatibility between the internal, foundational microservices (e.g., Inventory $\rightarrow$ Order Service). This is the standard CDC implementation, often using open-source tools to generate and verify contracts. This layer ensures that the underlying business logic remains compatible.

2. **Layer 2: UI-to-Backend CDC (Presentation Compatibility):** This is the novel element, introducing a dedicated **UI-Consumer Contract** between the **UI application (Consumer)** and its immediate aggregation point, typically a **Backend-for-Frontend (BFF) or API Gateway (Provider)**. The UI team writes isolated tests that, instead of hitting a live deployed BFF, interact with a **mock** of the BFF. These tests are designed to capture the **exact** expected HTTP requests (path, headers, body structure) and required responses (status code, body fields). This interaction is then serialized into the formal contract artifact (e.g., a "pact" file). This layer specifically verifies that the public-facing API signature that the UI depends on remains stable.

The strategic harmonization occurs because the UI team can generate the contract without needing the BFF to be deployed, and the BFF team can verify the contract without needing the entire downstream microservice ecosystem to be running. This decouples the development and testing pipelines, accelerating the feedback for the most critical integration point: the user experience.

2.3. Operationalizing the UI-Consumer Contract (The Implementation Pipeline)

The operationalization of the CDC-UI Hybrid Model relies on a four-step, automated pipeline integrated into the Continuous Integration/Continuous Delivery (CI/CD) process:

2.3.1. Contract Definition and Generation

The UI application's component and functional tests are leveraged to define the contract.

- **The UI Component Test:** A dedicated test suite for a specific UI component (e.g., a product detail view) is executed. In this test, the part of the code that normally makes an API call is configured to interact with a contract-testing client instead of an actual HTTP client.
- **The Interaction Definition:** Within the test, the UI explicitly states the *expected* interaction: *“When I make a GET request to /api/v1/product/123, I expect a 200 OK response with a JSON body containing at least the fields 'id', 'name', and 'price'.”*
- **Contract Artifact Creation:** The contract-testing framework intercepts this definition and, upon a successful UI test run (i.e., the UI successfully processes the mocked response), it generates a structured JSON file—the **Contract Artifact**. This artifact formally records the consumer’s expectations.

2.3.2. Contract Broker and Publishing

The contract artifact must be centrally managed to be accessible by the Provider.

- **Centralized Broker:** The generated contract is immediately published to a centralized repository, a **Contract Broker**. The Broker's essential functions are versioning and establishing the relationships between consumers and providers.
- **Version Control:** The contract is tagged with the current version of the UI application. The ability to manage multiple versions (e.g., UI v1.2 requiring Contract v3) is critical for supporting *safely* rolling out non-breaking changes and identifying *explicit* breaking changes.

2.3.3. Provider Verification in CI/CD

The BFF/API Gateway team (the Provider) is now responsible for validating its code against the UI’s expectations.

- **Automatic Retrieval:** As part of the Provider’s CI pipeline, a **verification task** is automatically triggered. This task queries the Contract Broker to retrieve all contracts published by all Consumers (including the UI) that are designated to integrate with the Provider's current version.
- **Mock Consumer-Side Requests:** The verification tool simulates the exact requests specified in the retrieved contracts (e.g., a GET to /api/v1/product/123).
- **Provider-Side Validation:** The Provider’s actual

deployed code handles the simulated request. The resulting response (status code, headers, and body) is then compared against the **expected** response defined in the contract.

- **Pipeline Gate:** If the Provider's response deviates from *any* consumer's expectation (e.g., the Provider removes the 'price' field that the UI requires), the verification task **fails the Provider's build pipeline**. This mechanism shifts the fault detection entirely to the Provider's side, preventing the deployment of a service that would break the UI.

2.4. Metrics for Evaluation

The efficacy of the CDC-UI Hybrid Model is measured by quantifying the improvements in a simulated MSA deployment pipeline against a traditional, E2E-heavy baseline. The key metrics used for quantitative analysis are:

1. **Test Execution Time (Feedback Loop Speed):** The total time required to execute the full integration/E2E test suite. A significant reduction is expected due to the shift from slow network I/O to rapid, in-memory contract verification.
2. **Test Stability/Flakiness Rate:** The percentage of test runs that fail non-deterministically without any corresponding code change. Flakiness is a major drain on developer confidence and pipeline efficiency, and it is intrinsically linked to the inherent instability of shared, multi-service test environments.
3. **Defect Escape Rate (Fault Isolation):** The percentage of integration-related defects that are discovered post-deployment (i.e., in a testing or production environment beyond the developer's local machine). A successful CDC-UI implementation is predicted to detect a higher proportion of integration faults earlier, leading to a lower escape rate.

3. Results and Comparative Analysis

3.1. Case Study Environment and Setup

To empirically evaluate the proposed strategy, a simulated e-commerce Microservices Architecture was established. The environment consisted of four core components:

- **Backend Services (Providers):** Inventory, Order, and Payment Services (written in polyglot languages for realistic complexity).
- **API Gateway/BFF (Provider):** A dedicated service responsible for composing responses for the front-end. This service acts as the **Provider** for the Layer 2 UI-Consumer Contracts.
- **User Interface (Consumer):** A web application that interacts exclusively with the API Gateway/BFF. This application is the **Consumer** for the Layer 2 contract.
- **Contract Broker:** A centralized repository for publishing and retrieving all Layer 1 and Layer 2 contracts.

The evaluation was conducted over two distinct phases, each running for four development sprints:

- **Phase 1 (Baseline):** The testing strategy relied on extensive **UI-driven E2E tests** (70% of integration coverage) which required the full orchestration of all four services and their respective databases. Layer 1 (Service-to-Service) CDC was used only for critical, non-UI-related backend paths.
- **Phase 2 (Experimental):** The majority of the E2E test suite was systematically decommissioned and replaced by the **CDC-UI Hybrid Model**. UI-Consumer Contracts (Layer 2) covered all primary user journeys, effectively mock-substituting the API Gateway/BFF in UI tests and enabling the API Gateway/BFF to verify its own code in isolation.

3.2. Quantitative Results of Test Execution Time

The most immediate and profound impact of adopting the CDC-UI Hybrid Model was observed in the overall test execution time, which directly correlates with the **feedback loop speed** for developers.

Test Phase	Test Type	Total Test Count	Average Execution Time (Minutes)

Phase 1 (Baseline)	UI E2E (Full-Stack)	150	48.7
	L1 CDC (Internal)	80	3.5
Phase 2 (Experimental)	UI Component/Contract Generation	150	6.1
	L1 CDC (Internal)	80	3.5
	L2 CDC Verification (BFF)	150	4.2

- The total time required for the *critical integration validation* in the Baseline phase (E2E) averaged 48.7 minutes. This lengthy duration created a bottleneck, often forcing developers to context-switch or defer integration checking until an overnight build.
- In the Experimental phase, the E2E test suite was replaced by the combined execution of the UI-side contract generation and the BFF-side contract verification. The total time for this equivalent integration check was reduced to $6.1 + 4.2 = 10.3$ minutes.
- This represents a **78.8% reduction** in the time required to achieve integration confidence. The resulting sub-10-minute feedback loop allows developers to execute the full integration check locally or in a rapid CI pipeline, aligning closely with industry best practices for continuous integration.

3.3. Analysis of Test Stability (Flakiness)

Flakiness, defined as non-deterministic test failures, was measured using a rolling average over the test phases, tracking instances where a test failed but passed on immediate re-run without any corresponding code change.

Test Phase	**Test Type**	**Observed Flakiness Rate (%)**	**Primary Root Cause of Flakiness**
Phase 1 (Baseline)	UI E2E (Full-Stack)	7.8%	Environment startup timing, database eventual consistency, network latency.
Phase 2 (Experimental)	UI Component/Contract Generation	0.2%	Minor dependency initialization/mock setup issues.
	L2 CDC Verification (BFF)	0.0%	N/A (Isolated, in-memory verification).

The data indicates a dramatic **reduction in flakiness from 7.8% to near 0%** for the UI integration verification path. The significant flakiness in the Baseline E2E tests was largely attributed to the distributed nature of the test environment. Specifically, the test was vulnerable to:

- The precise **timing** of service startup across containers.

- Transient network latency variations between services.
- Complex, multi-database **test data management** setup and tear-down that could lead to stale data being present in a shared environment.

By replacing the E2E test with two isolated, deterministic checks (UI test against a local mock; BFF test against a contract artifact), the sources of non-determinism were almost entirely eliminated. The 0.2% flakiness in the UI contract generation was traced to minor, manageable issues within the UI component testing framework itself, completely independent of backend infrastructure.

3.4. Defect Detection Efficacy

The primary goal of any testing strategy is to catch defects as early as possible. The concept of **Shift-Left** is essential in MSA. An analysis of the integration-related defects discovered during the four sprints of each phase revealed a clear shift in fault isolation:

Defect Discovery Location	**Phase 1 (Baseline - E2E Heavy)**	**Phase 2 (Experimental - CDC-UI)**
Developer Local Machine (Unit/Contract)	12%	45%
CI/CD Pipeline (Verification/Integration)	65% (E2E Failure)	50% (CDC Verification Failure)
Staging/QA Environment (Defect Escape)	23%	5%

The **Defect Escape Rate** (faults discovered in Staging/QA) saw an 18% absolute reduction (from 23% to 5%). In Phase 1, the long, flaky E2E suite often delayed fault detection; a significant number of integration faults would only be found after a full, often overnight, deployment to the Staging environment.

In Phase 2, the high rate of fault detection on the **Developer Local Machine** (45%) is highly significant. This suggests that the mandatory process of **defining and generating the UI-Consumer Contract** acts as a powerful design tool, forcing the UI developer to think about and validate the *contractual* interaction before pushing code. Furthermore, the L2 CDC Verification in the CI pipeline (50%) now isolated the error to the specific Provider (the BFF), making debugging instantaneous, unlike the hours-long triage required for a full E2E failure.

4. Discussion and Implications

4.1. Core Achievement: Accelerated Feedback and Enhanced Autonomy

The quantitative results unequivocally support the efficacy of the **CDC-UI Hybrid Model** in addressing the primary testing challenges of microservices: slow feedback and system coupling. The reduction in test execution time by almost 80% is not merely an optimization; it represents a fundamental **acceleration of the entire software development lifecycle**. By moving the core integration confidence check from a 48-minute E2E run to a ~ 10-minute combined CDC cycle, the pipeline enables developers to maintain a state of flow and receive near-instantaneous feedback on changes, a critical factor for productivity.

The strategic isolation achieved by the Layer 2 UI-Consumer Contract directly **preserves team autonomy**. The UI team can evolve and deploy its front-end application with confidence, knowing its contract is defined and validated against a mock. Simultaneously, the BFF team can update and deploy its backend service, knowing its changes are verified against the UI's explicit expectations via the contract broker. The system is designed to break the Provider's pipeline, not the integrated environment, allowing the UI and backend services to deploy independently while maintaining a verifiable guarantee of compatibility.

4.2. Strategic Alignment with the Testing Pyramid (The "Testing Honeycomb")

The traditional image of the Testing Pyramid, with E2E at the peak, is largely incompatible with the high-velocity, distributed nature of MSA. The findings presented here advocate for a fundamental re-shaping of this paradigm, aligning with the concept of a **Testing Honeycomb** or a **Testing Trophy** as proposed in industry discussions.

In this re-conceptualized model, **Unit Tests** remain the foundation. **Integration Tests** (which, in MSA, largely equate to Layer 1 Service-to-Service CDC) occupy the middle. The critical change occurs at the top: the slow, flaky **E2E/UI Tests** are largely replaced by the **Layer 2 UI-Consumer Contract Verification**. The small number of remaining E2E tests are then reserved strictly for high-level business workflow validation, security checks, and cross-browser concerns, rather than acting as the primary guardrail against API integration breakage. The CDC-UI approach effectively pushes the critical integration validation *down* the testing stack, making it faster, more stable, and more deterministic.

4.3. Advantages over End-to-End Testing (Elaboration on Isolation and Data Management)

The inherent weaknesses of E2E testing are amplified in a microservices environment. The proposed CDC-UI Hybrid Model directly counteracts these weaknesses, providing a more robust and sustainable approach to integration confidence. The advantages center on three critical areas: **Fault Isolation**, **Data Management**, and the **Compounding Effect of Flakiness**.

Fault Isolation:

In a typical E2E failure scenario, the test fails at the UI level. The resulting error trace is an ambiguous indicator, requiring a lengthy process of distributed tracing and log correlation to pinpoint the offending service. The failure could be a front-end rendering bug, a network timeout, a malformed request from the UI, an error in the API Gateway's aggregation logic, or an internal error deep within a downstream service's business logic. This requires synchronous coordination and debugging across multiple independent teams, directly subverting the organizational autonomy that MSA is intended to deliver.

In contrast, the **CDC-UI Hybrid Model provides immediate, deterministic fault isolation**. If the UI-side contract *generation* fails, the fault is isolated to the UI team's understanding of the contract or its client-side implementation. If the BFF/API Gateway's **Provider Verification** fails, the fault is isolated immediately to the BFF team's code—it has violated a known, explicit contract. This failure is caught in the BFF's CI pipeline, before deployment, and provides an exact, machine-readable artifact (the contract) that specifies the violation (e.g., "Expected field 'X' but received 'Y'"). The time-to-fix is dramatically reduced because the team responsible for the breakage is notified first, and the nature of the fault is clearly defined.

Distributed Test Data Management:

Perhaps the most pernicious complexity of E2E testing in MSA is the challenge of Distributed Test Data Management. An E2E test often requires a precise, known state across potentially dozens of independent, bounded contexts (databases).

1. **Setup and Tear-down:** Setting up this complex, multi-state data (the "given" state) and guaranteeing its cleanup after the test runs is a fragile operation. This setup is susceptible to failure due to network issues, eventual consistency delays, or improper rollback mechanisms.
2. **State Contention:** In shared test environments, one E2E test's actions may pollute the data state required by a parallel or subsequent E2E test, leading to the unpredictable, non-deterministic failures characteristic of **flakiness**.

The CDC-UI approach elegantly side-steps this complexity.

- In the UI-side contract generation, all backend dependencies are **mocked locally**. The test data is a simple, deterministic JSON payload defined *within* the test, entirely eliminating the need for database setup.
- In the BFF/API Gateway's Provider Verification, the necessary state is defined by the contract's **Provider State** clause (e.g., *"Given a user with ID 123 exists"*). This state setup is executed as an isolated function *within the Provider's test environment* and is independent of all other services and their data stores. This isolation ensures that the test state is precisely controlled and local, completely removing the possibility of

cross-test state contention and the complexity of multi-database orchestration. The reduction in test complexity associated with decoupling the tests from external data stores is a major contributor to the observed stability.

The Compounding Effect of Flakiness:

As demonstrated in the results, even a small percentage of flakiness (e.g., 7.8% in the baseline) can have a compounding negative effect on the deployment process. Flaky tests erode developer trust; teams begin to ignore a "red" (failing) E2E pipeline, often re-running the tests multiple times until they pass, a phenomenon known as "passing the pipeline by lottery." This creates a culture of distrust in the automation and fundamentally slows down the delivery process, as time is wasted on diagnosing false-positive failures. The CDC-UI Hybrid Model's demonstrated ability to reduce flakiness to a negligible level ($<1\%$) is perhaps its most significant strategic advantage, restoring trust in the automated quality gates and empowering teams to proceed with high confidence.

Strategic Enhancement: The Critical Role of the Contract Broker and Deployment Safety Verification

The success of the proposed CDC-UI Hybrid Model hinges on a highly functional and centralized component: the **Contract Broker**. This component is more than a simple repository; it is the **system of record** for all inter-service and UI-to-backend expectations, effectively functioning as the decentralized governance layer for the entire distributed system. The broker's capabilities are essential for enabling the necessary velocity and independent deployment safety that MSA promises.

The Contract Broker as the Governance Center

In a polyglot microservices environment, services are often written in different languages (Java, Node.js, Python), run on various frameworks, and maintained by autonomous teams. The only commonality is the network contract. The Contract Broker ensures that this contract remains the single source of truth and provides the meta-data necessary for automated deployment decisions.

Contract Versioning and Matrix

The broker manages a complete **compatibility matrix**. It records every published contract (Layer 1 and Layer 2) and, for each contract, records every verification result from the corresponding Provider. This forms a living, self-maintaining relationship graph. For instance, the broker can instantly verify: *Consumer_A (UI v3.1)* is compatible with *Provider_B (BFF v4.5)*, and *Provider_B (BFF v4.5)* is compatible with *Provider_C (Order Service v2.0)*. This matrix replaces the manual effort of maintaining documentation or the fragile reliance on shared integration environments.

The power of this versioning capability is particularly critical for the UI-Consumer Contract (Layer 2). As the UI team develops a new feature that requires a new field from the BFF, they define a new contract (e.g., Contract v5). They then deploy the new UI version (UI v3.2) against a mock of BFF that honors Contract v5. When the BFF team prepares to deploy its next version (BFF v4.6), its CI pipeline retrieves and verifies *both* Contract v4 (from older UI versions) and Contract v5 (from the new UI version).

- If BFF v4.6 passes verification for both contracts, it is safe to deploy, as it will not break either the old or the new UI.
- If BFF v4.6 fails verification for Contract v4, it indicates a **breaking change** for the older, potentially still deployed, UI. The deployment is blocked, and the BFF team is immediately notified that their change is incompatible with a living consumer.

This mechanism formalizes the communication between the UI team and the backend team, substituting an explicit, machine-enforced agreement for informal communication or late-stage E2E test failures.

Living Documentation and API Evolution

The collection of published contracts within the broker serves as the most accurate and up-to-date **living documentation** of the system's external interfaces. Unlike static API specification documents (which tend to drift from the actual implementation), the contracts in the broker are generated **by the running code** (the UI's test suite) and verified **against the running code** (the

BFF's verification suite).

Furthermore, the CDC-UI approach fosters responsible **API evolution**. The contracts reveal which specific fields or endpoints are actually being consumed by the UI. If a field exists in the BFF's internal model but is *never* requested in any UI-Consumer Contract, the BFF team has a clear mandate to deprecate or remove that field without risk of breaking the front-end. This minimizes API surface area, simplifies the BFF's maintenance, and counteracts the tendency of large APIs to accumulate cruft over time.

Deployment Safety Verification: The "Can I Deploy?" Query

The ultimate value proposition of the Contract Broker is its ability to facilitate automated **Deployment Safety Verification**. In a microservices ecosystem, the decision to deploy a service is inherently risky because it might break a consumer that is currently in production or is in the queue to be deployed. The CDC-UI model resolves this risk through the broker's meta-data.

The most advanced implementation of this is the **"Can I Deploy?"** feature. This is a crucial query made by the CI/CD pipeline of a service immediately before a deployment or release candidate is packaged.

$$\text{CanDeploy}(P_{version}, C_{versions}, E_{target}) \rightarrow \text{Boolean}$$

Where:

- $P_{version}$ is the version of the Provider (e.g., BFF v4.6) that is intended for deployment.
- $C_{versions}$ is the set of all *known* consumer versions (e.g., UI v3.1, UI v3.2) that are currently deployed to, or intended for, the target environment.
- E_{target} is the target environment (e.g., Staging, Production).

The broker evaluates this query by checking two critical conditions:

1. **Consumer Compatibility Check:** Has the new Provider version ($P_{version}$) successfully verified against *all* contracts from *all* Consumers ($C_{versions}$) that are currently deployed to, or are designated for, E_{target}? This confirms that the new Provider will not break any existing services.
2. **Provider Compliance Check:** For any Consumer that is *also* new and intended for E_{target} (e.g., UI v3.2), has the Provider successfully verified that Consumer's contract? This ensures that the new UI and new BFF are compatible with each other.

If both conditions are met, the query returns True, and the pipeline is allowed to proceed to deployment. If the check fails, the deployment is blocked, and the pipeline receives a highly specific error message indicating which Consumer/Contract relationship failed, e.g., "BFF v4.6 cannot be deployed to Production because it has failed verification for UI v3.1's Contract."

This single, automated gate transforms the deployment process from a high-risk, cross-team coordination effort into a low-risk, deterministic decision. It is the architectural mechanism that finally fulfills the promise of truly **independent deployability** in an MSA setting, particularly as it relates to the front-facing UI.

Integration with Asynchronous Communication and the Need for Extended Contracts While the CDC-UI Hybrid Model is highly effective for synchronous HTTP-based interactions, modern distributed systems increasingly incorporate **asynchronous communication** via message queues (e.g., Kafka, RabbitMQ) and event streaming platforms. The UI often initiates a synchronous request to the BFF that then triggers a chain of asynchronous events in the background (e.g., an Order Service publishes an OrderPlaced event). The UI may eventually subscribe to a different, synchronous endpoint to check the final status of the event chain.

The current model's **limitation** is that the Layer 2 UI-Consumer Contract is typically defined only for the synchronous request-response of the API Gateway. To fully harmonize the testing strategy, a deeper layer of contract definition is required for the asynchronous domain.

- **Asynchronous Contract Definition:** This extension would require the UI team (or a dedicated BFF component) to also define a contract for the events it *produces* or the events it *consumes* via a WebSocket or similar mechanism. This involves defining the schema of the event payload and the expected channel name.

- **Specialized Tooling:** While tools like Pact are primarily focused on HTTP, extensions for **Asynchronous API Specification (AsyncAPI)** contracts are emerging. Integrating an AsyncAPI-based contract definition (Layer 3) into the testing framework would allow the UI to enforce the shape of the data it expects to receive *eventually* without waiting for a full, non-deterministic asynchronous chain to complete.

This necessity for a Layer 3 (Asynchronous Contract) underscores that the **CDC-UI Hybrid Model is a living framework**, adaptable to the polyglot and poly-protocol nature of contemporary MSA. The core principle remains consistent: *define the contract at the consumer's boundary and verify it in the provider's pipeline, in isolation*.

Practical Implementation Considerations and Tooling

The successful implementation of the CDC-UI Hybrid Model necessitates the selection of appropriate tools and the establishment of strict developer practices.

Component	Function	Tooling Considerations
Contract Generation	UI Component/Functional Tests	**Pact-JS**, Cypress/Playwright with custom contract extensions. **Crucial:** Must run locally and mock the provider.
Contract Broker	Centralized Artifact Store & Compatibility Matrix	**Pact Broker**, dedicated custom implementation for large enterprise scale. **Essential:** Webhook integration for CI/CD notification.
Provider Verification	BFF/API Gateway CI Pipeline	**Pact JVM**, **Pact-Net**, etc., corresponding to the provider's language. **Key:** Must block the build on verification failure.
Deployment Safety	CI/CD Gate	Broker's **"Can I Deploy?"** CLI/API call, integrated into the final deployment script (e.g., Jenkins, GitLab CI).

The cultural change associated with this implementation is equally critical. It requires a shift from a *Provider-first* mindset (where the backend dictates the API) to a **Consumer-first** collaboration. The UI team must be trained not only to write component tests but to explicitly define the minimal contract, and the BFF team must view a contract verification failure as a **critical bug** requiring immediate attention, a failure distinct from an environment-related E2E flakiness. The results suggest this investment in practice and tooling yields significant returns in stability and speed.

4.4. Limitations and Future Research

While the CDC-UI Hybrid Model demonstrates a powerful solution for synchronous HTTP integration testing, the research acknowledges several **limitations**. Firstly, the proposed method **does not replace UI-specific functional or usability testing**; it is an integration verification technique only. Browser-specific rendering issues or user experience bugs still necessitate targeted functional testing. Secondly, as elaborated, the current model requires extension to fully address **asynchronous messaging and event-driven architectures**. The verification of event schema compliance (i.e., a Layer 3 Asynchronous Contract) introduces additional complexity related to schema evolution and consumer registration. Finally, the case study, while representative of typical MSA complexity, is a controlled environment, and deployment in a massive-scale enterprise with hundreds of microservices may introduce unforeseen governance and performance challenges for the Contract Broker itself.

Future research should focus on three areas: **1) The full integration of AsyncAPI contract verification** for UI-initiated event chains. **2) Investigating the integration of UI-driven performance contracts**, where the UI contract defines not just the functional interaction but also the expected latency, allowing for performance checks to be shifted-left. **3) Empirical study on the correlation between contract coverage metrics** (i.e., percentage of public API covered by consumer contracts) and the resulting stability and defect escape rates in large-scale production environments.

4.5. Conclusion

The rapid adoption of Microservices Architecture has created a tension between the goals of independent deployment velocity and the need for system-wide quality assurance. Traditional reliance on End-to-End UI testing has proven to be an unsustainable bottleneck, reintroducing systemic coupling and flakiness. This research introduced and evaluated the **CDC-UI Hybrid Model**, a strategic framework that extends Consumer-Driven Contract testing to the User Interface-to-Backend integration boundary. The findings demonstrate that by shifting this critical validation to a deterministic, isolated contract verification check, the architecture realizes an approximately 78.8% acceleration in feedback speed and a near elimination of integration-related flakiness. This approach represents a fundamental evolution in microservices quality assurance, providing a highly effective, production-ready strategy for harmonizing testing efforts, preserving team autonomy, and ensuring the continued integrity of distributed systems. The CDC-UI Hybrid Model is, therefore, a crucial enabler for organizations seeking to maximize the velocity and resilience inherent in their microservices investment.

References

1. Harsh Bhasin, Esha Khanna, and Sudha. “Black Box Testing based on Requirement Analysis and Design Specifications”. In: International Journal of Computer Applications 87.18 (2014), pp. 36–40.
2. MDN contributors. HTTP response status codes. Feb. 2022.
3. Vahid Garousi and Junji Zhi. “A survey of software testing practices in Canada”. In: Journal of Systems and Software 86.5 (2013), pp. 1354–1376.
4. J.R. Horgan, S. London, and M.R. Lyu. “Achieving software quality with testing coverage measures”. In: Computer 27.9 (1994), pp. 60–69.
5. IBM. What is software testing?
6. Irena Jovanović. “Software testing methods and techniques”. In: The IPSI BgD Transactions on Internet Research 30 (2006).
7. Chandra Jha, A. (2025). VXLAN/BGP EVPN for Trading: Multicast Scaling Challenges for Trading Colocations. International Journal of Computational and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3478
8. Mohd Ehmer Khan. “Different forms of software testing techniques for finding errors”. In: International Journal of Computer Science Issues (IJCSI) 7.3 (2010), p. 24.
9. Mohd. Ehmer Khan and Farmeena Khan. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques”. In: (IJACSA) International Journal of Advanced Computer Science and Applications 3.6 (2012), pp. 12–15.

10. Charles M. Kozierok. The TCP/IP-Guide: A comprehensive, illustrated internet protocols reference. No Starch Press, 2009.

11. Jyri Lehvä. Testing Integrations with Consumer-Driven Contract Tests. eng. 2019.

12. Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. “Consumer-Driven Contract Tests for Microservices: A Case Study”. In: Product-Focused Software Process Improvement. Ed. by Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández. Cham: Springer International Publishing, 2019.

13. James Lewis and Martin Fowler. Microservices a definition of this new architectural term. Mar. 2014.

14. Lu Luo. “Software testing techniques”. In: Institute for software research international Carnegie mellon university Pittsburgh, PA 15232.1-19 (2001), p. 19.

15. Qingzhou Luo et al. “An Empirical Analysis of Flaky Tests”. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014.

16. Y.K. Malaiya. “Antirandom testing: getting the most out of black-box testing”. In: Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE’95. 1995, pp. 86–95.

17. B. Meyer. “Applying ’design by contract’”. In: Computer 25.10 (1992), pp. 40–51.

18. Elliot Murray. Dockerized Pact Broker. Aug. 2020.

19. Elliot Murray. Gotchas. Aug. 2020.

20. Elliot Murray and joklek. Contract Tests vs Functional Tests. Jan. 2021.

21. Florian Nagel and Martin Leucker. “Analysis of Consumer-driven contract tests with asynchronous communication between microservices”. In: (Dec. 2019).

22. Srinivas Nidhra and Jagruthi Dondeti. “Black box and white box testing techniques-a literature review”. In: International Journal of Embedded Systems and Applications (IJESA) 2.2 (2012), pp. 29–50

23. Infrastructure as Code (IaC) Best Practices for Multi-Cloud Deployments in Enterprises. (2025). International Journal of Networks and Security, 5(01), 174-186. https://doi.org/10.55640/ijns-05-01-10

24. Charlene O’Hanlon. “A Conversation with Werner Vogels: Learning from the Amazon Technology Platform: Many Think of Amazon as ’That Hugely Successful Online Bookstore.’ You Would Expect Amazon CTO Werner Vogels to Embrace This Distinction, but in Fact It Causes Him Some Concern.” In: Queue 4.4 (May 2006), pp. 14–22.

25. C. Pautasso et al. “Microservices in Practice, Part 1: Reality Check and Service Design”. In: IEEE Software 34.01 (Jan. 2017), pp. 91–98.

26. G. Purna Sudhakar. “A model of critical success factors for software projects”. In: Journal of Enterprise Information Management 25.6 (2012), pp. 537–558.

27. SMK Quadri and Sheikh Umar Farooq. “Software testing–goals, principles, and limitations”. In: International Journal of Computer Applications 6.9 (2010), p. 1.

28. 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

29. Chris Richardson. Pattern: Decompose by business capability.

30. Ahmad Salman et al. “An Empirical Investigation of the Impact of the Communication and Employee Motivation on the Project Success Using Agile Framework and Its Effect on the Software Development Business”. In: Business Perspectives and Research 9.1 (2021), pp. 46–61.

31. Abhijit A Sawant, Pranit H Bari, and PM Chawan. “Software testing techniques and strategies”. In: International Journal of Engineering Research and Applications (IJERA) 2.3 (2012), pp. 980–986.

32. Dharmendra Shadija, Mo Rezai, and Richard Hill. “Towards an understanding of microservices”. In: 2017 23rd International Conference on Automation and Computing (ICAC). 2017, pp. 1–6.

33. Beth Skurrie. Matching. Aug. 2020.

34. Beth Skurrie, Matt Fellows, and Elliot Murray.

Getting started. Mar. 2021.

35. Beth Skurrie, Matt Fellows, and Elliot Murray. How Pact works. May 2021.

36. Beth Skurrie, Matt Fellows, and Elliot Murray. Webhooks. Oct. 2021.

37. Beth Skurrie and Elliot Murray. When to use Pact. July 2020.

38. Beth Skurrie, Elliot Murray, and obinna240. Can I Deploy. Nov. 2021.

39. Beth Skurrie et al. FAQ. Mar. 2022.

40. Beth Skurrie et al. Writing Consumer tests. Jan. 2021.

41. Johannes Thönes. “Microservices”. In: IEEE Software 32.1 (2015), pp. 113–116.

42. Unit testing best practices with .NET Core and .NET Standard. Nov. 2021.

43. Markos Viggiato et al. Microservices in Practice: A Survey Study. 2018. arXiv: 1808.04836 [cs.SE].

44. Ham Vocke. The Practical Test Pyramid. Feb. 2018.