**AMERICAN ACADEMIC PUBLISHER**

INTERNATIONAL JOURNAL OF NETWORKS AND SECURITY (ISSN: 2693-387X)

Volume 05, Issue 02, 2025, pages 59-66
Published Date: - 22-10-2025

# A FRAMEWORK FOR ASSURING INTER-SERVICE RELIABILITY IN DISTRIBUTED ARCHITECTURES: CONSUMER-DRIVEN CONTRACT TESTING WITH PACT

**Ayu Puspita Sari**
Faculty of Computer Science, Universitas Indonesia, Depok, Indonesia

**Nguyen Van Hoang**
Department of Software Engineering, FPT University, Hanoi, Vietnam

## Abstract

Purpose: The migration to microservices has substantially enhanced agility but simultaneously introduced critical complexities in managing inter-service communication and maintaining system-wide reliability. Traditional integration and End-to-End (E2E) testing are often too slow, brittle, and resource-intensive for continuous delivery pipelines. This article systematically examines Consumer-Driven Contract (CDC) testing, specifically through the implementation of the PACT framework, as a paradigm-shifting solution to assure inter-service reliability in distributed architectures.
Methodology: This paper develops a comprehensive theoretical framework for CDC and analyzes its technical workflow, focusing on artifact generation, mock service behavior, and provider verification. A conceptual comparative analysis is employed to contrast the PACT-based CDC approach against conventional testing strategies, utilizing key performance indicators such as pipeline execution time, integration defect leakage, and deployment frequency as metrics for success. We conduct a deep dive into the systemic role of the Pact Broker in governing contract evolution and enabling organizational autonomy.
Findings: CDC testing, particularly when formalized with PACT, is associated with a significant shift-left of integration defect discovery, leading to demonstrably faster and more stable CI/CD pipelines. The mechanism facilitates the independent evolution of services by establishing a machine-readable, shared understanding of the API interface. Crucially, the approach mitigates the coordination overhead and infrastructural cost associated with E2E environments.
Originality: This work contributes an academic synthesis of PACT's integration into the full development lifecycle, highlighting its capacity to enable high-velocity, high-assurance distributed systems, and identifying key future research avenues, including the integration of AI-driven tools.

## Keywords

Consumer-Driven Contracts, Contract Testing, Microservices, Distributed Systems, PACT, API Testing, Continuous Delivery.

## INTRODUCTION

### 1.1. Contextualizing Distributed Systems and Microservices

The evolution of enterprise software architecture has favored modularity and independence, culminating in the widespread adoption of distributed systems, particularly the microservices architecture. This shift facilitates independent development, deployment, and scaling of individual services, conferring significant advantages in terms of organizational agility and technical throughput. The autonomy granted to development teams allows for technology heterogeneity and rapid iteration, which are paramount in competitive markets.
However, the decomposition of a monolithic application into a network of loosely coupled services introduces a

fundamental challenge: maintaining reliable inter-service communication. As the number of services ($N$) grows, the number of potential integration points can approach $O(N^2)$, creating an exponential increase in potential failure vectors. The core problem in distributed systems is not the internal logic of any single service, but the assurance that the messages exchanged across service boundaries—the "contracts"—are consistently honored. A breaking change in a service's API structure, such as renaming a field or altering a data type, can propagate failures across the entire system, undermining the very stability that the architecture is intended to enhance.

### 1.2. The Integration Testing Dilemma

Traditionally, system-wide reliability in complex architectures has been assured through large-scale End-to-End (E2E) or full integration tests. These tests involve deploying a significant subset of the actual services, often in a dedicated staging environment, and executing workflows that span multiple service boundaries. While E2E testing provides high confidence that the complete system functions as intended, its drawbacks in a continuous delivery context are substantial:

- Slowness and Latency: The requirement to provision, start, and coordinate multiple services makes E2E tests inherently slow, leading to prolonged CI/CD pipeline times and delayed feedback to developers.
- Brittle and Flaky Nature: E2E tests are prone to intermittent failures ("flakiness") due to environmental inconsistencies, network latency, or dependencies on external systems, making it difficult to pinpoint the root cause of an issue.
- High Maintenance Overhead: Maintaining a complex, production-like integration environment for dozens or hundreds of microservices becomes a significant operational cost and a development bottleneck.
- Violation of Autonomy: Teams are forced to coordinate deployments and synchronize changes, which negates the primary benefit of the microservices paradigm—independent deployability.

These limitations necessitate an alternative testing paradigm that provides high confidence in inter-service compatibility while supporting the high-velocity, low-latency requirements of modern continuous deployment practices.

### 1.3. Introduction to Consumer-Driven Contract (CDC) Testing

Consumer-Driven Contract (CDC) testing emerges as a strategic intermediate testing layer that resolves the dichotomy between fast unit tests and slow E2E tests. The central principle of CDC is that the consumer of an API dictates the contract—the minimal set of requests and responses it requires from the provider service. This is a fundamental reversal from traditional API testing where the provider often defines the specification unilaterally.

The CDC process involves two main components:

1. The Consumer defines its expectations (the contract) and runs a test against a mock version of the provider, ensuring its own code correctly makes the request and handles the expected response.
2. The generated contract is then shared with the Provider, which runs a test against its real API, verifying that it can, in fact, fulfill every expectation specified by the consumer.

This methodology provides immediate, isolated feedback to both teams: the consumer knows their client code is correct, and the provider knows their latest changes will not break any known consumer.

### 1.4. The Role of PACT in Standardization

The PACT framework is the most widely adopted open-source tool for formalizing the CDC methodology. PACT provides the necessary toolchain for generating, sharing, and verifying contracts across diverse technology stacks (supporting languages from Java to JavaScript). PACT contracts are standardized JSON files that record the specific interactions, including the HTTP method, path, headers, request body structure, and the minimum required response structure. The framework's significance lies in its ability to translate the conceptual agreement of the contract into a machine-executable, highly reliable test artifact.

### 1.5. Research Gap and Contribution

While PACT is established in industry best practices, there remains a gap in comprehensive academic literature that

systematically analyzes its full lifecycle integration and its specific quantitative and qualitative impact on organizational metrics within large-scale distributed systems. Existing studies often focus on the architectural migration or the abstract concept of contract testing, but lack a deep, comparative synthesis of the PACT toolchain's role in the entire CI/CD pipeline.

This article addresses this gap by providing a systematic examination of the PACT framework as a robust and scalable solution to the core reliability challenge in modern distributed systems. Our contribution is a detailed theoretical and conceptual analysis of the PACT-based workflow, its measurable impact on CI/CD velocity and quality, and a deep exploration of the systemic governance provided by the Pact Broker.

## 2. Theoretical Framework and Methodology (Methods)

### 2.1. Theoretical Foundations of CDC

The theoretical underpinning of CDC is rooted in the principle of Postel's Law—Be conservative in what you send, be liberal in what you accept—applied to service interactions. The contract defined by the consumer should be the minimal necessary expectation rather than a mirror of the provider's entire API surface. This "loosely coupled" approach ensures that providers retain the flexibility to evolve their implementation details and add non-breaking features without requiring every consumer to update and re-verify their contracts.

CDC directly addresses the shortcomings of two common, yet flawed, approaches to integration assurance:

- End-to-End Testing (E2E): Slow, resource-intensive, and violates team autonomy.
- Provider-Driven Contracts: While useful for external APIs (e.g., OpenAPI/Swagger specifications), they often suffer from the "golden consumer" problem, where the provider over-specifies the contract or only considers its idealized consumer, leading to brittle tests when new, diverse consumers emerge.

The PACT implementation operationalizes the CDC concept by ensuring that the contract is driven by the actual code of the consumer client, not merely a static document that may drift out of sync with reality.

### 2.2. The PACT Workflow and Artifact Generation

The PACT workflow is a cyclical, two-stage process that centers on the Pact file (the contract artifact) and the Pact Broker (the central repository).

1. Consumer Side (Contract Generation):
   - The consumer service team writes an integration test that uses a Pact Mock Provider (a simulated service managed by the PACT framework).
   - The test defines an Interaction—a specific request/response pair (e.g., "Given state X, when I send a GET to /users, I expect a 200 response with data structure Y").
   - When the consumer test runs successfully, the PACT framework records all defined interactions into a JSON-formatted Pact file.
2. Contract Sharing:
   - The generated Pact file is published to the Pact Broker, acting as a source of truth for all service contracts.
3. Provider Side (Contract Verification):
   - The provider service team retrieves the relevant Pact file(s) from the Broker.
   - The provider runs a verification test using the Pact Verifier against its actual running service instance (often locally or in an isolated unit test environment).
   - The Verifier replays each request from the Pact file against the provider and asserts that the real response meets the minimum expectations recorded in the contract.

This decoupled process ensures that a consumer can be confidently deployed knowing that its expected contract is published, and a provider can be confidently deployed knowing that it satisfies all published contracts, all without requiring the simultaneous deployment of any other service.

### 2.3. Methodology for Empirical Comparison (Conceptual)

To quantify the value proposition of CDC, we conduct a conceptual comparative study contrasting the PACT-based approach with two traditional methods—E2E and full integration testing—based on four key performance indicators

(KPIs) relevant to a high-velocity microservices organization.

### 2.3.1. Defining Key Performance Indicators (KPIs)

- Test Execution Time ($T_{exec}$): The time required to execute the full test suite that validates an integration point. CDC tests, being isolated and mock-based, are expected to execute in seconds/minutes (near-unit test speed), whereas E2E tests often take tens of minutes or hours.
- Environment Provisioning Cost ($C_{env}$): The resource and maintenance cost of the required testing environment. CDC requires minimal/zero shared environments (just the local provider/consumer instance), while E2E requires a dedicated, full-stack environment.
- Integration Defect Rate (IDR): The percentage of integration errors detected after the initial development stage (i.e., in staging or production). A successful CDC implementation "shifts left" the failure detection, predicting a near-zero IDR for contract violations in later stages.
- Deployment Frequency ($D_{freq}$): The rate at which code is deployed to production. Reduced $T_{exec}$ and $C_{env}$, combined with lower IDR, are conceptually associated with a higher $D_{freq}$ due to increased pipeline velocity and safety.

### 2.3.2. Comparative Study Design (Conceptual)

The comparison is modeled on a representative distributed system with 10 interdependent microservices and a Continuous Deployment pipeline.

Testing Strategy Texec (Expected) Cenv (Expected) IDR Reduction Mechanism Organizational Impact

End-to-End (E2E) High (Hours) High (Dedicated Staging) Discovers issues late (in the full stack). Low $D_{freq}$, High Coordination Overhead.

Full Integration Medium (Minutes-Hours) Medium (Test-only environments) Discovers issues in a pre-prod environment. Medium $D_{freq}$, Medium Coordination Overhead.

CDC (PACT) Low (Seconds-Minutes) Low (Local/Unit Test Environment) Shift-Left: Discovers issues on the developer's machine during unit testing. High $D_{freq}$, High Team Autonomy.

This conceptual framework positions PACT not just as a tool, but as an architectural enabler of high-velocity delivery by fundamentally altering the risk profile of integration testing.

### 2.4. Technical Implementation Details (Focusing on the Provider Verification)

The technical rigor of the CDC approach is most evident in the provider verification stage. The provider’s verification process must not only confirm the API schema but also ensure that the actual business logic and state associated with the interaction are satisfied. PACT facilitates this through Provider States.

A Provider State is metadata embedded in the Pact file (e.g., providerState: "a user with id 123 exists"). Before replaying the consumer's request, the Pact Verifier invokes a specific setup hook on the provider service corresponding to that state. This ensures that the provider's test environment is correctly configured (e.g., a specific database entry is created or a required downstream service is mocked) to truthfully execute the interaction defined by the consumer. This mechanism is crucial as it isolates the verification to the API boundary while still ensuring the underlying system can correctly handle the prerequisites for the interaction.

## 3. Results and Impact Analysis

### 3.1. Quantifiable Benefits in CI/CD Metrics

The most immediate and measurable impact of a PACT implementation is the exponential reduction in test execution time ($T_{exec}$). By replacing multi-service integration tests with isolated contract verification runs, the system moves from testing against a slow network boundary to testing against a local mock or the provider's API in isolation. A typical E2E test spanning 5 services might take 30 minutes; a PACT verification for the same interaction can execute in milliseconds on the consumer side (against a mock) and a few seconds on the provider side (against a local

instance). This acceleration directly translates into a more rapid feedback loop for the development team—a critical success factor for Continuous Integration.

Furthermore, the decrease in the requirement for shared, resource-heavy testing environments significantly lowers the environment provisioning cost ($C_{env}$) and associated maintenance overhead. Teams can work entirely independently until the final deployment stage, minimizing the need for complex, cross-functional coordination meetings and shared infrastructure. The conceptual result is a substantial improvement in Deployment Frequency ($D_{freq}$), as the confidence barrier to deployment is reduced.

### 3.2. Reduction in Integration Defect Leakage

The "shift-left" phenomenon in PACT testing is arguably its most valuable contribution. Integration defects—bugs arising from a mismatch in expectations between two services—are typically discovered late in the development cycle, such as during E2E testing or, worse, in production. PACT moves the detection of these defects to the moment a developer attempts to integrate a new change.

- Consumer-Side Detection: If a consumer-side change breaks the local contract test (e.g., the client code attempts to parse a field that no longer exists in its mock), the developer receives instant feedback.
- Provider-Side Detection: If a provider-side change breaks the contract verification (e.g., the provider accidentally renames an API field), the CI/CD pipeline fails immediately upon the provider's build, well before deployment.

This early detection capability substantially reduces the Integration Defect Rate (IDR) in later stages. The cost of fixing a bug is known to increase non-linearly with the stage of detection. By catching contract violations at the unit/build stage, PACT minimizes the impact, cost, and risk associated with integration failures. This systemic robustness is not associated with traditional unit testing, which only validates internal service logic.

### 3.3. Deep Dive: The Role of the Pact Broker in Distributed Governance

The Pact Broker is not merely a repository; it serves as the crucial systemic component that provides governance, orchestration, and intelligence across the entire distributed system. Its role extends the PACT framework from a simple testing tool to a distributed system compatibility manager.

#### 3.3.1. Managing Contract Evolution and Versioning

In a system of dozens of microservices, contracts must evolve as features are added or deprecated. The Broker manages this complexity through contract versioning and the intelligent use of tags (e.g., main, production, feature-x). When a consumer updates a contract, the Broker can immediately signal to the provider that a verification is pending.

Crucially, the Broker supports the can-i-deploy verification tool. This command queries the Broker's graph of service dependencies and published contract verification results to answer a highly impactful business question: "Is it safe to deploy this version of service A to environment X?" The Broker is able to determine, with high confidence, if the provider has successfully verified all contracts from the current production versions of its consumers. This capability transforms deployment from a risky, cross-team coordination effort into a safe, automated check, directly increasing the system's deployment velocity. The mechanism ensures that service A's deployment will not introduce breaking changes to its known consumers, which is a powerful enabler of team autonomy.

#### 3.3.2. Organizational Autonomy and Decoupling

The Pact Broker decouples the development and deployment pipelines of individual services. Teams can work on their service roadmaps entirely independently because the Broker acts as the central communication channel for compatibility requirements. A consumer team defining a new feature only needs to update its contract and publish it to the Broker; it does not need to interrupt or synchronize with the provider team's development schedule. Similarly, a provider team knows that as long as their new version successfully verifies against all contracts in the Broker, their changes are non-breaking. This separation of concerns significantly enhances organizational autonomy, a key objective of adopting microservices.

### 3.3.3. Systemic Challenges in Broker Adoption and Management

While the Pact Broker offers profound systemic benefits, its successful adoption is not without significant challenges that must be rigorously addressed. The complexity of a distributed system often mirrors the complexity of its governance tool, and the Broker, as the central authority for inter-service communication, introduces its own set of management overheads.

A critical challenge arises from the sheer volume of contracts and the potential for the N-squared problem of integrations to manifest in the Broker's complexity. For a system with $N$ services, the number of consumer-provider relationships can be substantial. Maintaining and reviewing hundreds or even thousands of individual Pact files becomes a significant operational task. Teams may struggle with contract sprawl, where contracts for deprecated features or services are not cleaned up, leading to unnecessary verification overhead and clutter in the Broker's database. Without diligent governance and automation, the Broker, intended to reduce coordination, can inadvertently become a new source of bureaucratic drag.

Furthermore, the management of the Broker itself introduces a security and infrastructural concern. As the single source of truth for all API interactions, the Broker becomes a high-value target in terms of security. Access control, auditing, and secure transmission of Pact files are paramount. Organizations must ensure that the Broker is deployed with appropriate authentication and authorization mechanisms to prevent unauthorized publication or retrieval of contract information. The decision to use a self-hosted open-source Broker versus a commercial managed service (e.g., Pactflow) is a strategic one, balancing cost and control against the overhead of operational maintenance, patching, and scaling. In large enterprises with high transaction volumes, the scalability and performance of the Broker under continuous contract publishing load must be rigorously stress-tested.

Another challenge is the 'golden contract' anti-pattern, where the contract is made overly specific or strict. If a consumer specifies an exact value for a field (e.g., expecting an array of precisely three items) rather than a pattern or type (e.g., expecting an array of at least one item), it places unnecessary constraints on the provider. When the provider attempts to evolve its API in a non-breaking way (e.g., by returning an array of four items), the verification fails, creating a false positive failure. This necessitates a culture shift and a rigorous training program for developers to leverage PACT's powerful matchers (e.g., like, eachLike, somethingUnexpected) to write contracts that are both robust and flexible. The failure to adopt a permissive matching strategy can undermine the entire goal of independent deployability, transforming the Broker from an enabler of autonomy into an enforcer of stagnation.

Finally, the transparency of state dependency presents a subtle but significant issue. Provider States are necessary for verification, but they obscure the true system dependencies. The Broker knows that Provider A must be in State X for verification to pass, but the underlying complexity of how State X is achieved (e.g., three downstream systems must be called successfully) remains hidden. This can lead to brittle provider verification tests if the state setup is not meticulously managed and isolated. Teams must invest heavily in mock service creation and database sanitization for verification, adding to the initial setup cost of the PACT adoption. The solution involves rigorous adherence to isolated test principles, ensuring that the Provider State setup is fast, reliable, and does not require a complex, multi-service environment, thus preserving the core benefit of fast feedback. The commitment to maintaining a robust, scalable, and secure Broker, alongside the necessary cultural shift toward writing minimalist, flexible contracts, represents a major, yet necessary, organizational investment that is associated with successful, long-term distributed system stability.

## 3.4. Extending PACT to Non-HTTP Interactions

While PACT is most commonly associated with RESTful API interactions, its underlying philosophy—a consumer defining its minimum expectations—is equally applicable to asynchronous and event-driven architectures. PACT has evolved to support message pacts.

In a message-based system (e.g., Kafka, RabbitMQ), the consumer defines the contract as the expected structure and content of a message it will consume, independent of the transport layer. The provider then verifies that the messages it produces conform to this structural contract. This extension is crucial for systems that utilize event sourcing or asynchronous messaging for communication, expanding the scope of CDC from mere Request/Response integrity to full inter-service data flow assurance. This application is associated with a reduction in data-mismatch errors in decoupled, eventual consistency-based architectures.

## 4. Discussion and Conclusion

### 4.1. Interpreting the Impact on Development Velocity and Quality

The adoption of Consumer-Driven Contract testing with the PACT framework is associated with a powerful dual-benefit to software organizations: a measurable increase in development velocity and a significant enhancement in system quality. The velocity gain stems directly from the shift-left of testing and the isolation of service teams. By replacing slow, monolithic E2E tests with rapid, isolated contract verification, the feedback cycle shortens dramatically. This enables developers to commit and deploy more frequently, directly improving the conceptual Deployment Frequency ($D_{freq}$) metric.

The quality enhancement is derived from the formalized agreement inherent in the contract. The process forces a crucial communication—the consumer must explicitly articulate its needs, and the provider must explicitly confirm its capability to meet them. This structured, machine-readable negotiation minimizes the potential for miscommunication, which is a leading cause of integration failures. The use of the Pact Broker further formalizes this communication, ensuring that compatibility is continuously and programmatically verified, which is a powerful predictor of production stability.

### 4.2. Addressing Implementation Challenges and Anti-Patterns

Despite its clear advantages, the successful implementation of PACT is highly dependent on addressing key cultural and technical challenges. As discussed, the risk of overly strict contracts—where developers fail to use flexible matchers—can lead to brittle tests and slow down service evolution. This is a behavioral anti-pattern requiring a shift in mindset: the contract should not test the provider's logic, only the structural interface.

Another significant anti-pattern is the perception of PACT as a replacement for all other forms of testing. PACT is designed to test the seams between services; it does not eliminate the need for robust unit testing (internal logic) or a minimal set of E2E tests (system health and critical path functional flows). A robust testing strategy employs PACT as the critical layer between unit and system testing, optimizing the trade-off between coverage and speed.

### 4.3. Literature Gaps and Future Research Directions

While this work confirms the critical role of PACT in modern distributed systems, several avenues for future research exist to further advance the field:

1. Standardized Metrics: There is a need for standardized, cross-organizational metrics for CDC success beyond the conceptual KPIs used here. Research should focus on empirically correlating PACT adoption with tangible business outcomes like Mean Time To Repair (MTTR) and customer-reported defect rates.
2. AI/ML-Driven Contract Generation: The manual effort of writing and maintaining hundreds of contract tests, especially in very large systems, remains a significant overhead. Future research should explore the application of Artificial Intelligence and Machine Learning techniques to infer contract expectations by observing actual service traffic or analyzing Open API specifications, thereby reducing the friction and human error associated with contract authoring.
3. FaaS and Serverless Integration: The application of CDC to modern serverless and Function-as-a-Service (FaaS) architectures presents a unique challenge, as the service boundary is often less explicit. Research is needed to develop methodologies and tooling to adapt the CDC pattern to the ephemeral and often event-driven nature of serverless functions.
4. Beyond REST: While PACT supports message-based contracts, further work on standardizing contract testing for advanced communication protocols like gRPC and WebSockets is warranted.

### 4.4. Limitations of the Current Study

This study is subject to limitations inherent in synthesizing industry best practices and conceptual modeling. The conceptual comparative analysis (Section 2.3) relies on qualitative industry experience and theoretical performance modeling rather than a dedicated, controlled, empirical case study. While the presented conclusions are aligned with

documented industry success stories, a large-scale, controlled deployment study remains the ultimate validator of the quantifiable benefits. Furthermore, the analysis of PACT is concentrated on the open-source tooling, and may not fully capture the benefits or constraints of commercial solutions built upon the PACT ecosystem.

### 4.5. Concluding Remarks

The journey toward high-velocity, reliable software delivery in a distributed system architecture is fundamentally a journey to master inter-service communication. Consumer-Driven Contract testing, championed by the PACT framework and orchestrated by the Pact Broker, provides the necessary technological and organizational mechanism to achieve this mastery. By shifting the detection of integration flaws to the earliest possible point in the development cycle, PACT is associated with lower development friction, increased team autonomy, and greater overall system resilience. The framework is a pivotal enabler, transforming the complex network of a microservices architecture from a source of instability into a foundation for rapid, confident, and continuous deployment. Its full potential will be realized as organizations embrace the necessary cultural shifts and as academic research further formalizes its application to the next generation of computing paradigms.

### References


1. PACT Foundation, “Pact Documentation.” [Online]. Available: https://docs.pact.io
2. Spring Cloud Team, “Spring Cloud Contract Reference Documentation.” [Online]. Available: https://cloud.spring.io/spring-cloud-contract/
3. 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
4. Postman Inc., “Postman API Platform.” [Online]. Available: https://www.postman.com/
5. M. Fowler, “Microservice Testing Strategies,” MartinFowler.com, 2018. [Online]. Available: https://martinfowler.com/articles/microservice-testing/
6. S. Newman, Building Microservices, 2nd ed. O’Reilly Media, 2021.
7. ThoughtWorks, “Technology Radar Vol. 26,” 2022. [Online]. Available: https://www.thoughtworks.com/radar
8. Pactflow, “Secure, Scalable Contract Testing.” [Online]. Available: https://pactflow.io/
9. T. Richardson and B. Abbott, “Contract Testing: A Best Practice Guide,” InfoQ, 2022. [Online]. Available: https://www.infoq.com/articles/contract-testing-guide/
10. Nagaraj, V. (2025). Ensuring low-power design verification in semiconductor architectures. Journal of Information Systems Engineering and Management, 10(45s), 703–722. https://doi.org/10.52783/jisem.v10i45s.8903
11. GitHub, “Using the Pact CLI in GitHub CI.” [Online]. Available: https://github.com/pact-foundation/pact-js/blob/master/docs/ci/github.md
12. 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.
13. 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