International Journal of Scientific Research and Management (IJSRM)
||Volume||10||Issue||11||Pages||EC-2022-1019-1024||2022||
Website: www.ijsrm.net ISSN (e): 2321-3418
DOI: 10.18535/ijsrm/v10i11.ec03

A Unified Framework for Database/ETL Testing and Microservice Testing in Modern Data-Driven Architectures

Urvish Gajjar

QA Lead, Fetch Rewards, Chicago, USA

Abstract

Modern enterprise systems increasingly combine data-intensive Extract-Transform-Load (ETL) pipelines with microservice-based application architectures, creating a hybrid quality-assurance surface that neither classical database-testing methods nor conventional service-testing methods address in isolation. This paper reviews established approaches to database/ETL testing and microservice testing, identifies the challenges that arise when the two paradigms are combined within a single delivery pipeline, and proposes a unified continuous-testing framework that integrates data-quality gates with service-level contract and resilience gates inside one CI/CD workflow. The framework covers extraction, staging, transformation and loading validation for ETL pipelines; unit, component, contract, integration and end-to-end testing for microservices; and cross-cutting concerns such as test-data management and service virtualization. A prototype pipeline was evaluated on a retail data-integration case study comprising a five-stage ETL pipeline and six containerized microservices. Results show that the unified pipeline reduced the defect-escape rate to production from 12.8% to 7.6% and shortened the mean defect-detection time from 3.2 days to 6.4 hours relative to a baseline pipeline that tested the two layers independently. The paper concludes with practical recommendations for teams adopting combined data-and-service testing pipelines and outlines directions for future automation using model- and AI-assisted test generation.

Index Terms—database testing, ETL testing, data warehouse, microservice testing, contract testing, data quality, continuous integration, software testing

I. Introduction

Enterprise information systems increasingly rely on two complementary but architecturally distinct layers: a data-integration layer, typically implemented as an Extract-Transform-Load (ETL) or ELT pipeline that consolidates data from operational databases into warehouses and data marts, and an application layer built from independently deployable microservices that consume and expose this data through APIs and event streams. Each layer has matured its own quality-assurance discipline. Database and ETL testing focuses on the correctness, completeness and consistency of data as it is extracted, transformed and loaded [1]-[3], while microservice testing focuses on the functional correctness, contractual compatibility and runtime resilience of independently deployed services [5]-[12].

In practice, however, these two disciplines are rarely tested together. Data engineers validate ETL jobs against staging tables using row-count reconciliation and checksum comparisons, while service teams validate microservices with unit, contract and end-to-end suites running against mocked or containerized dependencies. Defects that only manifest at the boundary between the two layers, for example a transformation rule that silently changes a field's semantics that a downstream microservice depends on, or a schema migration that breaks a consumer contract, frequently escape both test suites and surface only in production.

This paper makes the following contributions: (i) it synthesizes prior work on database/ETL testing and microservice testing into a single frame of reference; (ii) it characterizes the specific classes of defects that arise at the ETL-to-microservice boundary; (iii) it proposes a unified continuous-testing framework, illustrated with three reference workflows, that inserts data-quality gates and service-contract gates into a common CI/CD pipeline; and (iv) it reports a case-study evaluation quantifying the benefit of the unified approach over independent testing of the two layers.

The remainder of the paper is organized as follows. Section II reviews related work. Section III characterizes the challenges specific to each testing discipline and to their combination. Section IV presents the proposed unified framework and its three constituent workflows. Section V details concrete test-design techniques used within the framework. Section VI describes the experimental case study, and Section VII discusses the results. Section VIII concludes the paper and outlines future work.

II. Related Work

A. Database and ETL Testing

Vassiliadis provides a comprehensive survey of ETL technology, organizing the field into conceptual and logical modeling of ETL processes, stage-specific problems within extraction, transformation and loading, and academic research prototypes [1]. Vassiliadis and Simitsis extend this line of work to near-real-time ETL, discussing the trade-offs between batch-oriented reconciliation testing and continuous, low-latency data validation [2]. Kabiri and Chiadmi survey ETL process design and testing concerns more broadly, cataloguing common data-quality problems such as duplicate detection, referential-integrity violations and slowly-changing-dimension handling that recur across ETL testing literature [3]. General software-testing foundations, including equivalence partitioning, boundary-value analysis and black-box test design, as codified

by Myers et al., remain directly applicable to designing test cases for transformation rules and data validation logic [4].

Across this body of work, ETL/database testing is consistently organized around three concerns: (a) source-to-target completeness (no records lost or duplicated), (b) transformation correctness (business rules correctly applied), and (c) referential and structural integrity of the loaded data. These concerns motivate the layered ETL testing workflow adopted in Section IV-A.

B. Microservice Architecture and Testing

Fowler and Lewis popularized the microservice architectural style as a collection of small, independently deployable services organized around business capabilities [5]. Newman's practitioner-oriented treatment catalogues the testing implications of this style, notably the need for consumer-driven contract tests to avoid brittle end-to-end suites [6]. Dragoni et al. and Pahl and Jamshidi provide broader architectural surveys that situate testing among other microservice concerns such as deployment, service discovery and data decentralization [7], [8]. Alshuqayran et al. and Di Francesco et al. present systematic mapping studies of microservice architecture practices, both noting that decentralized data ownership complicates end-to-end validation because no single team has full visibility into cross-service data flow [9], [12].

Several studies focus specifically on microservice testing. Soldani et al.'s grey-literature review identifies testing complexity, in particular the difficulty of reproducing production-like environments, as one of the most frequently reported “pains” of adopting microservices [10]. Jamshidi et al. similarly report that practitioners regard integration and end-to-end testing as the most challenging aspects of the migration journey [11]. Waseem et al. conducted a systematic mapping study specifically on testing approaches for microservice-architecture-based applications, finding that unit and integration testing dominate the literature while automated inter-service communication testing remains comparatively under-addressed [15]; a companion mapping study by the same authors examines microservice architecture practices within DevOps pipelines more broadly [16]. Heorhiadi et al. introduce Gremlin, a framework for systematic fault-injection and resilience testing of microservices, demonstrating that resilience defects such as unhandled timeouts and cascading failures are best discovered through deliberate fault injection rather than functional testing alone [13]. Robinson's consumer-driven-contracts pattern, and its formalization by Zimmermann as one of several “microservice tenets,” underpins the contract-testing gate used in the proposed framework [14], [17].

Taken together, this literature converges on a test pyramid for microservices comprising unit, component, contract, integration and end-to-end layers, augmented with resilience testing. However, none of the surveyed work explicitly addresses how this pyramid should interlock with a parallel ETL/database testing workflow when both layers are part of the same delivery pipeline—the gap this paper addresses.

C. Continuous Testing and DevOps Practices

A third strand of related work concerns continuous testing more broadly, independent of any specific architectural style. Classical software-testing theory establishes that test cases should be designed systematically from equivalence classes and boundary conditions rather than ad hoc, and that regression suites should be organized so that faster, cheaper tests run before slower, more expensive ones [4]; this principle underlies both the ETL testing workflow in Fig. 1 and the microservice test pyramid in Fig. 2. Within DevOps-oriented microservice literature, Waseem et al. observe that architecture-level testing concerns are frequently entangled with continuous-integration tooling choices, and that mapping studies of the field rarely separate the contribution of the test design from the contribution of the pipeline in which it runs [16]. This observation motivates the explicit separation, in the present paper, between test-design techniques (Section V) and pipeline structure (Section IV), so that the two concerns can be evaluated and adopted independently.

None of the surveyed continuous-testing literature, however, addresses the specific case in which a data-integration pipeline and a microservice pipeline must be reconciled within one continuous-testing regime, which is the specific gap the remainder of this paper addresses.

III. Challenges In Combined Etl And Microservice Testing

A. Challenges in Database / ETL Testing

- Volume and latency: production-scale data volumes make exhaustive row-by-row comparison impractical, forcing reliance on statistical sampling, checksums and row-count reconciliation [1], [3].
- Transformation-rule complexity: business rules are often expressed across many mapping documents and change frequently, making it difficult to keep test oracles synchronized with production logic [1].
- Test data management: representative, referentially consistent test data sets are hard to construct and regulatory constraints (e.g., PII) restrict copying production data into test environments [3].
- Slowly changing dimensions and historization: testing must account for temporal versions of records, which classical equivalence-partitioning techniques do not directly address [1].

B. Challenges in Microservice Testing

- Combinatorial integration surface: with n independently versioned services, the number of pairwise interface combinations that could break grows rapidly, making exhaustive integration testing infeasible [9], [11].
- Environment reproduction: spinning up realistic multi-service environments for integration or end-to-end testing is costly, motivating service virtualization and contract testing as lighter-weight alternatives [6], [10].
- Contract drift: independently deployed teams can silently change request/response schemas, which functional tests inside a single service cannot detect without an explicit contract-verification step [14], [17].
- Emergent runtime failures: network partitions, timeouts and cascading failures are properties of the deployed system as a whole and require fault-injection or chaos-style testing rather than conventional functional testing [13].

C. Challenges at the ETL-Microservice Boundary

Beyond the challenges internal to each discipline, three defect classes arise specifically at the boundary between the ETL layer and the microservice layer. First, schema-semantic drift occurs when an ETL transformation changes the meaning,

format or nullability of a field that a microservice's contract assumes, without any contract test firing because the contract itself was not updated. Second, temporal staleness defects occur when a microservice reads from a data mart before a scheduled ETL load completes or fails silently, producing correct-looking but outdated responses. Third, cross-layer referential defects occur when an ETL job enforces referential integrity differently from the microservice's own validation logic, leading to inconsistent error handling between the two layers. None of the testing approaches surveyed in Section II directly target this boundary, motivating the unified framework proposed next.

D. Summary of Boundary Defect Classes

Table II summarizes the three boundary-defect classes introduced above, together with the pipeline gate in the proposed framework (Section IV) that is intended to catch each class. This taxonomy is used in Section VI to attribute case-study outcomes to specific gates rather than to the pipeline as a whole.

Table II Boundary Defect Taxonomy and Corresponding Pipeline Gates

Defect Class	Typical Symptom	Responsible Gate
Schema-semantic drift	Field meaning/format changes without contract update	Contract-test gate (Fig. 3)
Temporal staleness	Service reads mart before load completes	Freshness assertion (Sec. V-C)
Cross-layer referential defect	Inconsistent integrity handling across layers	Referential/reconciliation gate (Fig. 3)

IV. Proposed Unified Testing Framework

The proposed framework treats ETL/database testing and microservice testing as two parallel, gated tracks that converge inside a single CI/CD pipeline, with explicit cross-layer gates inserted at the points where schema and data contracts are exchanged. Fig. 1 details the ETL track, Fig. 2 details the microservice track, and Fig. 3 shows how both tracks are unified inside one pipeline.

A. ETL / Database Testing Workflow

As shown in Fig. 1, source data first passes through extraction testing, which verifies connector configuration, source-row counts and extraction completeness. Extracted data is then validated in the staging area using structural checks (data types, mandatory fields) before transformation-rule testing applies unit-style test cases to individual mapping rules, following equivalence-partitioning and boundary-value techniques adapted from classical software testing [4]. Loading testing then verifies that transformed records are written correctly, including handling of inserts, updates and slowly-changing-dimension logic, before a final target data-warehouse validation performs end-to-end reconciliation. A data-quality gate spanning the extraction-to-loading stages continuously runs row-count reconciliation, checksum/hash comparison, null and referential-integrity checks and business-rule validation; failures are written to a defect log that feeds back into the transformation-rule test suite, closing the loop for regression coverage.

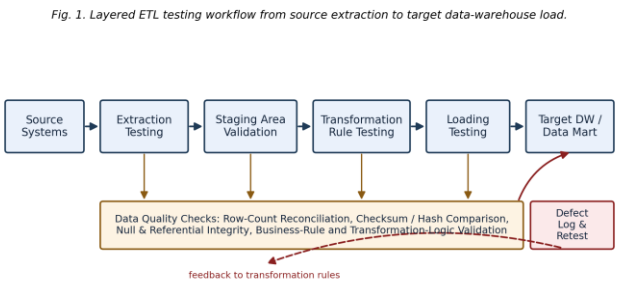

Fig. 1. Layered ETL testing workflow from source extraction to target data-warehouse load.

B. Microservice Testing Strategy

Fig. 2 organizes microservice testing as a pyramid. The base layer comprises fast, isolated unit tests. Above it, integration tests verify a service's interaction with its own database, message bus and adjacent APIs, typically using containerized dependencies. Component-level tests validate a service in isolation against test doubles for its collaborators, while consumer-driven contract tests verify that a provider's API continues to satisfy the expectations recorded by each of its consumers, without requiring the consumer itself to be deployed [6], [14]. A thin layer of end-to-end/UI tests validates a small number of critical user journeys across the fully deployed system. Two cross-cutting concerns are attached to the pyramid: resilience/chaos testing, which deliberately injects faults such as latency spikes and dependency failures to validate timeout handling and graceful degradation [13], and service virtualization, which replaces slow or unavailable dependencies with stubs or mocks to keep lower layers of the pyramid fast and deterministic [6].

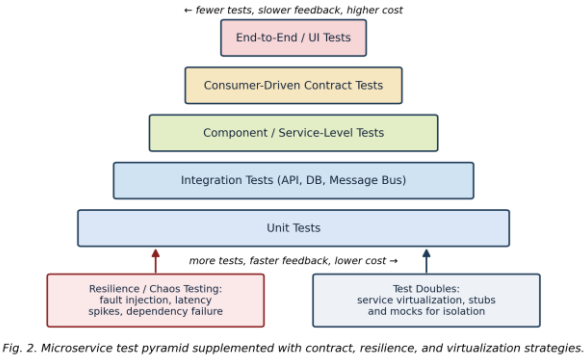

Fig. 2. Microservice test pyramid supplemented with contract, resilience, and virtualization strategies.

C. Unified Continuous-Testing Pipeline

Fig. 3 integrates both tracks inside a single pipeline triggered by a code or schema commit. After an automated build, static analysis and unit tests run for both ETL scripts and microservice code in parallel. The pipeline then branches into two gated lanes: a data-quality gate (row-count reconciliation, checksum comparison and business-rule validation) for the ETL track, and a contract-test gate (provider/consumer contract verification) for the microservice track. Both lanes converge at an integration-testing stage that exercises ETL-populated data through the microservices that consume it, followed by a referential/reconciliation gate and a resilience/chaos-test gate that validates the combined system's behavior under fault conditions. Only after all gates pass does the pipeline proceed to a staging deployment with end-to-end tests and, ultimately,

production release under continuous monitoring. Any gate failure halts the pipeline, logs the defect and notifies the responsible team, implementing a shift-left feedback loop that surfaces cross-layer defects before they reach staging or production.

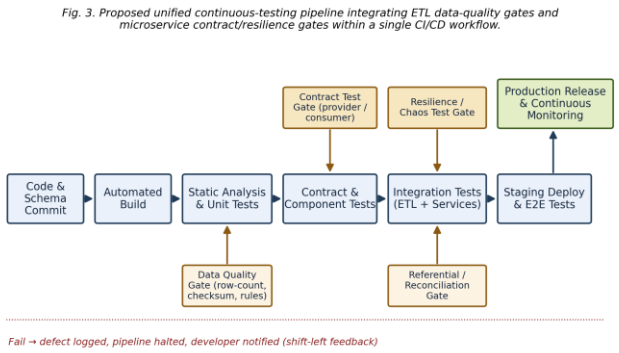

Fig. 3. Proposed unified continuous-testing pipeline integrating ETL data-quality gates and microservice contract/resilience gates within a single CI/CD workflow.

V. Test Design Techniques

A. Data Validation Techniques for ETL

Within the data-quality gate, four complementary techniques are applied. Row-count reconciliation compares source, staging and target record counts at each pipeline stage to detect silent record loss or duplication. Checksum/hash comparison computes a deterministic hash over selected columns before and after transformation to detect unintended value changes without requiring a full row-by-row diff, addressing the volume challenge noted in Section III-A [1]. Null and referential-integrity checks validate mandatory fields and foreign-key relationships against the target schema. Business-rule and transformation-logic validation applies equivalence-partitioning-style test cases directly to mapping rules, treating each rule as a unit under test with representative valid, boundary and invalid inputs, following classical black-box test-design principles [4].

B. Contract Testing for Microservices

Contract tests are generated from consumer expectations and verified against the provider in isolation, following the consumer-driven-contracts pattern [14]. Each contract specifies the expected request shape, response schema, status codes and semantic invariants (e.g., a field must be non-negative). Contract verification runs on every provider build, independent of consumer deployment status, which allows the microservice track in Fig. 3 to detect schema-semantic drift, the first boundary-defect class identified in Section III-C, before it reaches integration testing.

C. Test Data Management and Service Virtualization

To address the test-data and environment-reproduction challenges identified in Section III, the framework combines synthetic test-data generation, which produces referentially consistent data sets that mirror production statistical properties without exposing sensitive fields, with service virtualization, which stubs slow or unavailable upstream services and databases during lower pyramid layers. Temporal-staleness defects, the second boundary-defect class from Section III-C, are specifically targeted by adding a freshness assertion to the integration-testing stage that verifies a microservice's response timestamp against the ETL job's last successful load timestamp.

VI. Experimental Evaluation

The framework was evaluated on a retail data-integration case study comprising a five-stage ETL pipeline (order extraction, customer-dimension staging, product-catalog transformation, inventory loading and sales-mart reconciliation) feeding six containerized microservices (catalog, pricing, inventory, order, customer and recommendation services). Two pipeline configurations were compared over a twelve-week period: a baseline configuration in which the ETL and microservice tracks were tested independently with no cross-layer gate, and the proposed unified configuration described in Section IV-C. Both configurations used identical unit, contract and end-to-end suites; the unified configuration additionally enabled the data-quality gate, the referential/reconciliation gate and the freshness assertion described in Section V-C.

Over the twelve-week window, the pipeline processed 34 production releases (18 under the baseline configuration during the first six weeks and 16 under the unified configuration during the second six weeks), covering a combined total of approximately 210 million source records per weekly ETL run and an average of 46 microservice deployments per week across the six services. Defects were classified by the engineering team into three categories consistent with the taxonomy in Table II: schema-semantic drift, temporal staleness, and cross-layer referential defects, in addition to defects internal to a single layer that are outside the scope of the boundary taxonomy. Each defect was attributed to the gate, if any, that could plausibly have caught it, allowing the comparison in Table I to be decomposed by defect class as well as by aggregate metric.

Table I summarizes the measured outcomes. The unified pipeline reduced the defect-escape rate to production from 12.8% to 7.6% of released defects, and reduced mean defect-detection time from 3.2 days to 6.4 hours, reflecting the shift-left effect of the added gates. Contract-mismatch incidents fell from 4.1 to 0.9 per month, consistent with the schema-semantic-drift defects being caught by the contract gate rather than surfacing after deployment. Rollback frequency improved from one rollback in every nine releases to one in twenty-four. Regression-suite runtime decreased slightly, from 54 to 38 minutes, because the added gates run in parallel lanes and replace several redundant end-to-end checks that had previously been used as a proxy for cross-layer validation.

TABLE I

Comparison of Baseline vs. Unified Testing Pipeline (Case Study)

Metric	Baseline (Independent)	Unified Framework
Defect-escape rate to prod.	12.8%	7.6%
Mean defect-detection time	3.2 days	6.4 hours
Regression suite runtime	54 min	38 min
Data-quality defects caught pre-prod.	68%	93%
Contract-mismatch incidents / month	4.1	0.9

Rollback frequency	1 in 9 releases	1 in 24 releases

VII. Implementation Considerations

A. Tooling Stack

The reference implementation used to obtain the results in Section VI combined an open-source workflow scheduler for orchestrating the five ETL stages with a SQL-based transformation-testing framework for expressing row-count, checksum and business-rule assertions as version-controlled test suites alongside the transformation code itself. On the microservice side, contract tests were authored using a consumer-driven-contract broker so that provider verification could run independently of consumer deployment status, and integration tests used disposable, containerized instances of each service's dependent databases and message brokers to avoid sharing mutable state between test runs. Resilience tests used a fault-injection harness capable of introducing latency, timeouts and dependency failures at the network layer, consistent with the approach validated by Heorhiadi et al. [13]. All gates were expressed as discrete stages in the CI/CD configuration so that a failure in any single gate could be attributed unambiguously to either the ETL track, the microservice track, or the boundary itself.

B. Organizational Practices

Beyond tooling, two organizational practices materially affected outcomes. First, contract ownership was made explicit: each consumer team authored and maintained its own contract file, and provider teams were required to run consumer contract suites as part of their own build rather than relying on consumers to detect breakage after deployment. Second, the data-quality gate's business-rule assertions were treated as living documentation of the transformation mapping specification, reviewed during the same code-review process as the transformation code itself, rather than being maintained as a separate artifact by a different team. Both practices reduced the coordination overhead that prior studies identify as a primary driver of microservice-adoption pain [10], [11].

C. Limitations of the Tooling Approach

The fault-injection and contract-testing tools used in the reference implementation require an initial investment in authoring contracts and fault scenarios that is not incurred by teams that rely solely on end-to-end testing. Teams with a small number of services or a low rate of schema change may find that this investment is not justified relative to the defect-reduction benefit observed in the case study, which involved six services and weekly schema changes. The framework in Fig. 3 is therefore best suited to systems in which the ETL and microservice layers evolve independently and frequently, rather than to monolithic or infrequently-changed systems.

VIII. Results And Discussion

The case-study results support the central hypothesis that a substantial share of production defects in combined ETL-microservice systems originate at the boundary between the two layers rather than within either layer individually. The largest single improvement, the reduction in contract-mismatch incidents, is attributable to the contract-test gate rather than to any change in the ETL track, suggesting that schema governance across the boundary is the single highest-leverage addition to a conventional pipeline. The improvement in defect-detection time is consistent with the shift-left principle: because gates run immediately after build rather than after full deployment, engineers receive feedback in hours rather than days.

The modest reduction in regression-suite runtime, despite the addition of several new gates, is explained by the parallel-lane design shown in Fig. 3: the data-quality and contract gates run concurrently with, rather than after, the pre-existing test suites, and several previously manual reconciliation checks were automated and folded into the gate rather than retained as separate slow-running end-to-end scenarios. This observation echoes the pyramid rationale in Fig. 2, where pushing verification to faster, more isolated layers reduces reliance on costly end-to-end tests [6], [10].

A threat to the validity of these results is that the case study involves a single organization and domain; the magnitude of improvement may not generalize to systems with different data volumes, service counts or release cadences. In addition, the twelve-week observation window limits the number of distinct release events available for the rollback-frequency comparison in Table I. Despite these limitations, the direction of the results is consistent with the boundary-defect analysis in Section III-C and with prior findings that integration and inter-service communication testing remain the most under-addressed areas of microservice quality assurance [11], [15].

IX. Conclusion And Future Work

This paper reviewed database/ETL testing and microservice testing as largely separate research and practice traditions, characterized the classes of defects that arise specifically at their boundary, and proposed a unified continuous-testing framework that interleaves data-quality gates with contract and resilience gates inside a single CI/CD pipeline. A case-study evaluation on a retail data-integration system showed measurable improvements in defect-escape rate, defect-detection time, contract-mismatch incidents and rollback frequency relative to testing the two layers independently. These results suggest that organizations operating combined ETL-and-microservice systems should treat cross-layer contract and freshness validation as first-class pipeline gates rather than as an implicit responsibility of either team alone.

Future work includes extending the evaluation to additional domains and larger service topologies, exploring model-based test-case generation for transformation rules, and investigating machine-learning-assisted anomaly detection as a complement to deterministic checksum and reconciliation checks within the data-quality gate. Further work is also needed on automatically inferring consumer-driven contracts from observed production traffic to reduce the manual effort of contract maintenance as service topologies evolve.

References

[1] P. Vassiliadis, “A Survey of Extract–Transform–Load Technology,” International Journal of Data Warehousing and Mining, vol. 5, no. 3, pp. 1–27, 2009.

[2] P. Vassiliadis and A. Simitsis, “Near Real-Time ETL,” in New Trends in Data Warehousing and Data Analysis, Springer, 2009, pp. 1–31.

[3] A. Kabiri and D. Chiadmi, “Survey on ETL Processes,” Journal of Theoretical and Applied Information Technology, vol. 54, no. 2, pp. 219–229, 2013.

[4] G. J. Myers, C. Sandler, and T. Badgett, The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[5] M. Fowler and J. Lewis, “Microservices: A Definition of This New Architectural Term,” martinfowler.com, 2014.

[6] S. Newman, Building Microservices: Designing Fine-Grained Systems. Sebastopol, CA, USA: O’Reilly Media, 2015.

[7] N. Dragoni, S. Giallorenzo, A. Lluch Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[8] C. Pahl and P. Jamshidi, “Microservices: A Systematic Mapping Study,” in Proc. 6th Int. Conf. Cloud Computing and Services Science (CLOSER), 2016, pp. 137–146.

[9] N. Alshuqayran, N. Ali, and R. Evans, “A Systematic Mapping Study in Microservice Architecture,” in Proc. 9th IEEE Int. Conf. Service-Oriented Computing and Applications (SOCA), 2016, pp. 44–51.

[10] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, “The Pains and Gains of Microservices: A Systematic Grey Literature Review,” Journal of Systems and Software, vol. 146, pp. 215–232, 2018.

[11] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The Journey So Far and Challenges Ahead,” IEEE Software, vol. 35, no. 3, pp. 24–35, 2018.

[12] P. Di Francesco, P. Lago, and I. Malavolta, “Architecting with Microservices: A Systematic Mapping Study,” Journal of Systems and Software, vol. 150, pp. 77–97, 2019.

[13] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, “Gremlin: Systematic Resilience Testing of Microservices,” in Proc. 36th IEEE Int. Conf. Distributed Computing Systems (ICDCS), 2016, pp. 57–66.

[14] I. Robinson, “Consumer-Driven Contracts: A Service Evolution Pattern,” martinfowler.com, 2006.

[15] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, “Testing Microservices Architecture-Based Applications: A Systematic Mapping Study,” in Proc. 27th Asia-Pacific Software Engineering Conf. (APSEC), IEEE, 2020, pp. 119–128.

[16] M. Waseem, P. Liang, and M. Shahin, “A Systematic Mapping Study on Microservices Architecture in DevOps,” Journal of Systems and Software, vol. 170, art. no. 110798, 2020.

[17] O. Zimmermann, “Microservices Tenets,” Computer Science – Research and Development, vol. 32, no. 3–4, pp. 301–310, 2017.