The Eastasouth Journal of Information System and Computer Science
Vol. 1, No. 01, August, pp. 187 - 195

Contract-Drift Detection in Event-Driven Microservices Using Runtime Message Semantics: A Literature Synthesis

Srinivas Nune
Independent Researcher, Franklin, USA

Article Info

Article history:

Received, Aug, 2023
Revised, Aug, 2023
Accepted, Aug, 2023

Keywords:

API Testing;
Consumer-Driven Contract Testing;
Contract Drift;
Distributed Observability;
Event-Driven Architecture;
Message Semantics;
Microservices;
Runtime Verification;
Schema Evolution;
Semantic Versioning

ABSTRACT

Event-driven microservice architectures largely dispense with synchronous interfaces, in which producer and consumer services are bound to a shared interface at compile time, and rely instead on asynchronous message contracts that may evolve between services without explicit notice — a condition termed contract drift. This paper synthesises fourteen peer-reviewed and archival papers published between 2017 and 2022 across five domains: microservice architecture, consumer-driven contract testing, semantic versioning, runtime-verification theory, and distributed observability. The synthesis finds that prevailing assurance mechanisms operate predominantly offline and pre-deployment, and therefore cannot detect drift that emerges once a service is in production. Building on an established runtime-verification taxonomy, the paper proposes a structural-similarity measure and a type-agreement measure for quantifying the extent of contract drift; both are computed in flight against a registered baseline contract, and an alert is raised when their weighted combination exceeds a configurable threshold. The available evidence indicates pronounced recent interest in the problem — six of the fourteen papers synthesised were published in 2022 — and suggests that a decentralised, broker-side monitoring architecture is a viable design approach. Open challenges remain in baseline governance, behavioural drift detection, and the absence of standardised benchmarks for comparing runtime contract-monitoring techniques.

Corresponding Author:

Name: Srinivas Nune
Institution: Independent Researcher, Franklin, USA
Email: Srinivas.nune9792@gmail.com

1. INTRODUCTION

In event-driven microservice systems, an application is decomposed into loosely coupled services that communicate asynchronously through message brokers rather than by direct synchronous calls [1], [2]. This decoupling yields independent deployability and scalability, both recognised as principal drivers of microservice adoption [2]. The same loose coupling, however, eliminates the compile-time correctness guarantees that a shared code base provides, relying instead on mutual agreements between services — partially enforced or entirely informal — known as contracts [3], [4].

In this setting, a contract defines the structure, the data types, and the expected semantics of messages exchanged between a producer and one or more consumers. Contract drift occurs when a producer alters a message schema, the meaning of a field, or the ordering of emitted events without breaking

the contract outright: the running system continues to appear healthy while consumer expectations are gradually eroded [3]. Event-driven systems lack a synchronous request-response boundary at which such mismatches would surface immediately, so the effects of drift often manifest only later, as processing errors, data-quality degradation, or cascading downstream failures [5], [6].

This exposure is only partially covered by existing quality-assurance mechanisms. Consumer-driven contract testing compares the producer's declared behaviour against the consumer's stated expectations at build time, but that verification is inherently a snapshot taken when the tests run, not a continuous check throughout production execution [4]. While static semantic-versioning analysis can identify structurally breaking changes in published artefacts, empirical replication studies show that a substantial proportion of releases do not observe the versioning conventions they declare, making version numbers a poor proxy for drift [7], [8]. Automated application-programming-interface (API) test-generation tools improve test coverage for request-response contracts, but still achieve limited coverage of complex or evolving schemas, particularly where semantically meaningful tests are required [9], [10]. None of these practices treats the message stream itself, at runtime, as a source of ground truth for detecting drift as it occurs.

This paper synthesises findings from fourteen peer-reviewed and archival papers published between 2017 and 2022 across five areas: microservice architecture, consumer-driven contract testing, semantic versioning, runtime-verification theory, and observability and anomaly detection in distributed systems. On that basis, it presents a conceptual account of contract-drift detection grounded in runtime message semantics — the continuous, in-flight comparison of observed message structure and behaviour against a registered contract baseline. The aim is not to report new empirical results, but to synthesise, compare and formally organise existing conceptual and empirical contributions into a coherent account that clarifies their overlaps, their differences, and the gaps that remain open to future research.

2. LITERATURE REVIEW

2.1 Microservice Contracts and Consumer-Driven Testing

In an industrial case study, Lehvä, Mäkitalo and Mikkonen examined the application of consumer-driven contract testing, in which contracts derived from consumer expectations are evaluated ahead of the provider's implementation [4]. They found that provider-side contract verification catches many incompatibilities at lower cost than end-to-end integration testing, but that the approach presupposes a discrete build pipeline in which explicit test suites are executed. That precondition does not hold for asynchronous integration, where a consumer may be one of many anonymous subscribers to a topic. In a complementary study of automated REST API test-generation tools, Kim, Xin, Sinha and Orso observed that although the tools themselves are mature, the test suites they generate remain insufficiently deep to cover the full functional and semantic contract of an API [9]. Liu et al. introduced Morest, a model-based approach that uses execution feedback to guide RESTful API test generation dynamically, achieving improved coverage of contract states [10]. Taken together, these contributions indicate that contract-testing methods are comparatively mature for synchronous, request-response interfaces, but remain at an early stage of development for asynchronous, event-driven message contracts.

2.2 Semantic Versioning and Breaking Changes

Ochoa, Degueule, Falleri and Vinju conducted a large-scale replication study of semantic-versioning practice in the Maven Central package ecosystem, examining how frequently and how severely releases introduce breaking changes [7]. Their analysis shows that a substantial proportion of version

increments do not reliably signal the magnitude of the accompanying contract-breaking change, and that version numbers are therefore an unreliable indicator of compatibility. To address this gap, Zhang et al. proposed a static semantic-differencing technique that automatically determines whether a release violates the semantic-versioning guarantees it declares [8]. Both studies examine library and package interfaces rather than the event-driven message contracts at issue here. Their shared conclusion — that declared versioning metadata is a poor indicator of true compatibility — nevertheless carries directly over, and arguably with greater force: event-driven contracts in message-based microservice integration typically lack declarative versioning metadata altogether, and so cannot be enforced by a package manager or dependency resolver.

2.3 Runtime Verification Foundations

Falcone, Krstić, Reger and Traytel introduced a taxonomy of runtime-verification tools organised along three dimensions: the type of specification formalism, central versus decentralised monitor placement, and the timing of verification (online versus offline) [11]. This classification supplies the vocabulary needed to position contract-drift detection methods within the broader discipline of runtime verification. Addressing decentralised and distributed systems specifically, Francalanza, Pérez and Sánchez examine the problem of reaching a globally consistent verdict across nodes that fail independently and communicate asynchronously — conditions characteristic of real-world event-driven microservice deployments [12]. Mettler, Mueller-Gritschneder and Schlichtmann demonstrated the feasibility of a distributed hardware-level monitoring architecture for runtime verification on multi-tile systems, showing that mechanisms for coordinating monitors and aggregating properties across a distributed substrate extend naturally to the extraction and correlation of properties across distributed software components [13]. Collectively, these foundations supply the formal vocabulary and the monitor-placement strategies that a contract-drift detector can adapt to an event-driven message stream.

2.4 Observability and Anomaly Detection in Distributed Microservices

Li et al. surveyed industrial practice in microservice tracing and observability, finding that although distributed tracing is widely adopted, engineers still struggle to relate trace data to the behavioural and contractual properties of the services being traced [6]. Kohyarnejadfard, Aloise, Azhari and Dagenais applied natural-language-processing techniques to distributed tracing and log data in order to detect anomalies in microservice environments, demonstrating that combining structured trace metadata with unstructured log content surfaces behavioural deviations that neither source reveals on its own [5]. In an interview-based and grey-literature study of microservice evolvability, Bogner, Fritzsch, Wagner and Zimmermann report that practitioners consistently identify a central, persistent challenge: services cannot be evolved independently with confidence, because doing so depends on contract stability between components that change at different rates [3]. Earlier work by Fritzsch, Bogner, Wagner and Zimmermann on microservice migration strategy echoes this finding, observing that organisations undertaking migration are frequently unprepared for the operational discipline required to maintain decoupled service interfaces once decomposition is complete [14]. Read together with the systematic review of deployment and communication patterns by Aksakalli, Çelik, Can and Tekinerdoğan [1], this body of evidence indicates both the breadth of architectural variation that a drift-detection mechanism must accommodate and the

practical urgency of the underlying problem.

Table 1. Summary of Reviewed Studies Synthesised in This Paper

Ref.	Author(s) / Year	Thematic Focus	Contribution Type
[1]	Aksakalli et al., 2021	Microservice deployment & communication patterns	Systematic literature review
[3]	Bogner et al., 2021	Microservice evolvability assurance	Interview study & grey-literature review
[2]	Dragoni et al., 2017	Microservice architecture foundations	Conceptual overview
[11]	Falcone et al., 2021	Runtime verification taxonomy	Classification framework
[12]	Francalanza et al., 2018	Decentralised runtime verification	Theoretical / tutorial chapter
[14]	Fritzsch et al., 2019	Microservice migration strategy	Empirical industry study
[9]	Kim et al., 2022	Automated REST API test generation	Tool survey
[5]	Kohyarnejadfard et al., 2022	Microservice anomaly detection	NLP-based tracing analysis
[4]	Lehvä et al., 2019	Consumer-driven contract testing	Industrial case study
[6]	Li et al., 2022	Microservice tracing & observability	Industrial survey
[10]	Liu et al., 2022	Model-based RESTful API testing	Tool proposal (Morest)
[13]	Mettler et al., 2020	Distributed hardware runtime monitoring	System design & evaluation
[7]	Ochoa et al., 2022	Semantic versioning compliance	Large-scale replication study
[8]	Zhang et al., 2022	Semantic versioning violation detection	Static analysis technique

Source: Compiled by the authors from the fourteen sources synthesised in this review.

3. METHODS

This study was conducted as a narrative literature review, a design appropriate to drawing together heterogeneous sources from several domains into a conceptual synthesis rather than a statistical meta-analysis. Source selection focused on peer-reviewed journal articles, high-quality conference papers, and archival preprints published between 2017 and 2022 that address one or more of five thematic areas relevant to contract-drift detection: microservice architecture and evolution, contract or API testing, semantic versioning, runtime-verification theory, and distributed observability. Fourteen sources satisfied these criteria; the distribution of their contributions across the review period is summarised in Figure 2, which shows a marked concentration in the final years of the window.

For each source, the review extracted the stated problem and the proposed technique or finding, identified the architectural and methodological assumptions on which that technique rests, and cross-referenced sources addressing overlapping concerns in order to establish where they converge, contradict one another, or provide complementary coverage. The outcome of this process for the runtime-verification domain is represented schematically in Figure 3, which locates contract-drift detection within the taxonomy of Falcone et al. [11] along the dimensions of monitor placement, verification timing and specification formalism.

Building on this synthesis, the paper proposes a conceptual framework for contract-drift detection based on runtime message semantics. The framework formalises a message contract observed at a given point in time as a structured object C comprising the set of expected fields F, the declared data type T(f) of each field $f \in F$, and, where the event sequence is ordered, an expected ordering constraint over those events. A structural-similarity measure is defined from the overlap of the field sets of

the baseline contract C_0 and the observed contract C_t:

$$S(C_0, C_t) = |F_0 \cap F_t| / |F_0 \cup F_t| \quad (1)$$

where $|\cdot|$ denotes set cardinality. This ratio, equivalent to a Jaccard similarity coefficient, equals unity when the observed message structure matches the registered baseline exactly and approaches zero as the two field sets diverge. A complementary type-agreement measure, $\tau(C_0, C_t)$, is defined as the proportion of shared fields whose declared type remains unchanged between C_0 and C_t. A composite drift score D, integrating both structural and type-level divergence, is then expressed as a weighted combination:

$$D(C_0, C_t) = w_1 \cdot [1 - S(C_0, C_t)] + w_2 \cdot [1 - \tau(C_0, C_t)] \quad (2)$$

subject to $w_1 + w_2 = 1$, where the weights w_1 and w_2 allow the relative importance of structural versus type-level change to be tuned to the sensitivity requirements of a given consumer. A drift alert is raised whenever the composite score exceeds a configurable threshold θ:

$$\text{drift}_{\text{detected(t)}} = \text{TRUE if } D(C_0, C_t) > \theta; \text{ FALSE otherwise} \quad (3)$$

The choice of θ governs a familiar sensitivity/specificity trade-off, consistent with the general observation from the runtime-verification literature that monitors placed closer to the point of message emission achieve lower detection latency at the cost of higher susceptibility to transient false positives, whereas monitors aggregating evidence over longer observation windows achieve higher precision at the cost of slower detection [11], [12].

4. RESULTS AND DISCUSSION

4.1 Conceptual Architecture for Runtime Contract-Drift Detection

The architectural and monitoring patterns identified in the reviewed literature can be combined into the conceptual pipeline presented in Figure 1. As in conventional publish-subscribe microservice integration [1], [2], a producer service publishes messages to an event broker. Rather than performing contract verification at the consumer, however, a separate message interceptor — deployed broker-side or as a sidecar — inspects messages in flight and validates them against a schema registry holding the current baseline contract. This arrangement decouples drift detection from both producer and consumer, and so follows the principle of decentralised monitor placement advocated by Francalanza et al. for systems in which no single node holds a global view [12]. Detected drift is surfaced through an alerting and versioning-feedback channel, closing the loop back to the producer team and supplying the continuous evidence of validity that semantic-versioning analysis alone has been shown not to provide [7], [8].

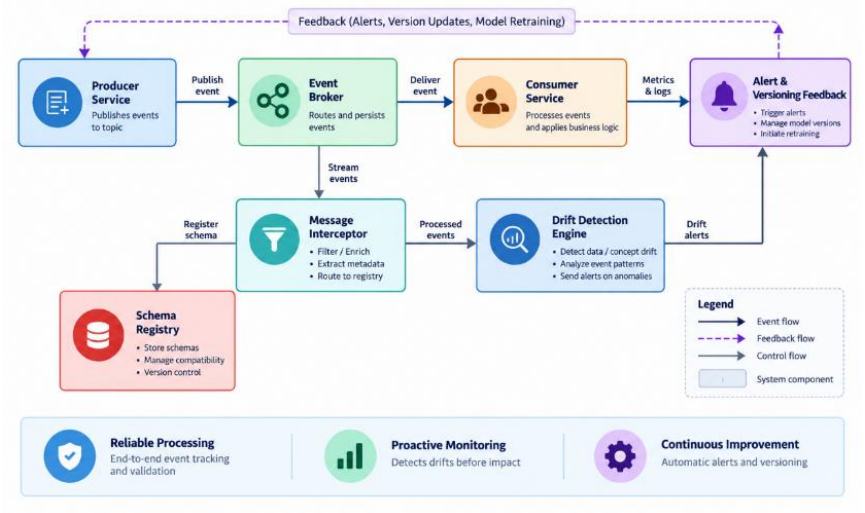

Figure 1. Conceptual Pipeline for Runtime Contract-Drift Detection

4.2 *Comparative Analysis of Detection Approaches*

The reviewed literature supports at least four broad strategies for making contract incompatibility explicit, each drawing on a distinct subset of the sources and compared in Table 2: pre-release consumer-driven contract testing [4]; static semantic-versioning or interface-differencing analysis [7], [8]; automated dynamic API test generation [9], [10]; and continuous runtime monitoring of message streams, the approach synthesised here. These strategies separate cleanly along the timing dimension of the Falcone et al. classification [11]: contract testing and test generation operate offline, prior to deployment, whereas runtime monitoring operates online, concurrently with production traffic. The distinction has a direct consequence for detection latency. Offline techniques can identify a breaking change before it reaches production, but cannot detect drift introduced after deployment through configuration change, data evolution or emergent behaviour. Online monitoring detects such post-deployment drift, but only after a non-zero exposure window between the onset of drift and its detection.

Table 2. Comparative Analysis of Contract-Assurance Strategies for Event-Driven Microservices

Dimension	**Consumer-Driven Contract Testing**	**Static Semantic-Versioning Analysis**	**Automated API Test Generation**	**Runtime Message-Semantics Monitoring**
Verification timing	Offline (pre-release)	Offline (post-release, artefact-based)	Offline (pre-release)	Online (continuous, in-production)
Monitor placement	Build pipeline	Package repository / CI	Test harness	Broker-side / sidecar (decentralised)
Primary evidence source	Predefined test expectations	Release artefact differencing	Generated request/response pairs	Live message stream
Detects post-deployment drift	No	No	No	Yes
Reliance on developer discipline	High	High	Moderate	Low
Representative sources	[4]	[7], [8]	[9], [10]	Synthesised in this paper, informed by [11], [12], [13]

Source: Compiled by the authors from the synthesised literature; entries denote qualitative characterisation rather than measured benchmark values.

4.3 *Publication Trends and Research Maturity*

As illustrated in Figure 2, the distribution of the reviewed literature by year of publication is uneven: 6 of the 14 sources appeared in 2022 alone, while the remaining 8 are distributed across the preceding five years (2017: 1; 2018: 1; 2019: 2; 2020: 1; 2021: 3). This concentration is consistent with the trend reported by Bogner et al. and Fritzsch et al., who find that concerns such as microservice evolvability and contract stability become more salient as organisations accumulate experience in operating decomposed architectures [3], [14]. That only a single source falls in each of 2017, 2018 and 2020 further suggests that sustained attention to the runtime dimension of contract assurance, as distinct from static interface

testing, is a comparatively recent development in microservice-reliability research.

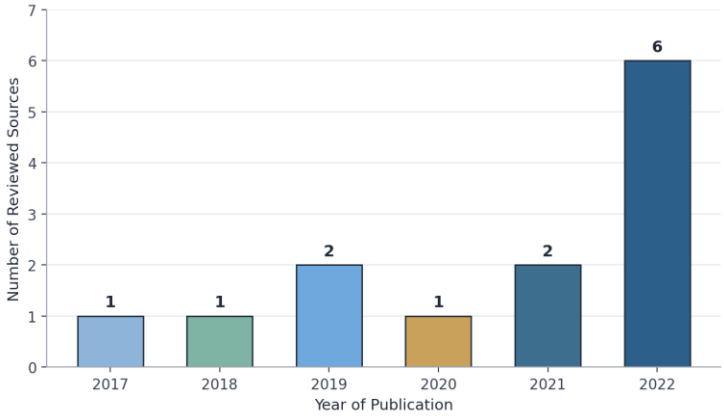

Figure 2. Distribution of Reviewed Sources by Year of Publication (2017–2022)

4.4 Runtime Verification Taxonomy and Open Challenges

Positioning contract-drift detection within the taxonomy of Falcone et al. [11], as shown in Figure 3, clarifies both the design space available to future detectors and the difficulties reported in the observability literature. On the monitor-placement dimension, the synthesised sidecar or broker-side interceptor design constitutes a decentralised configuration, comparable to the distributed monitor-coordination strategies of Mettler et al. [13] and subject to the decentralised-verification challenges identified by Francalanza et al. [12]. On the specification-formalism dimension, the drift measures defined in Section 3 are schema-based: they capture field-level and type-level divergence, but not behavioural drift in message ordering or timing. The reviewed literature suggests that this gap could be closed by combining schema-based measures with the trace-correlation and log-based anomaly-detection methods of Li et al. and Kohyarnejadfard et al. [5], [6]. Three open challenges follow from the synthesis. First, the reviewed sources provide no shared benchmark against which the detection latency and accuracy of competing runtime contract-monitoring techniques could be compared, which limits the field to qualitative comparison and represents a clear opportunity for future empirical work. Second, establishing a reliable and current baseline contract is difficult in the absence of centralised schema governance — a concern raised by Ochoa et al. in relation to versioning trust [7] and recurring across several of the synthesised sources. Third, overly sensitive drift thresholds carry a risk of alert fatigue, a failure mode already documented in industrial observability practice [6].

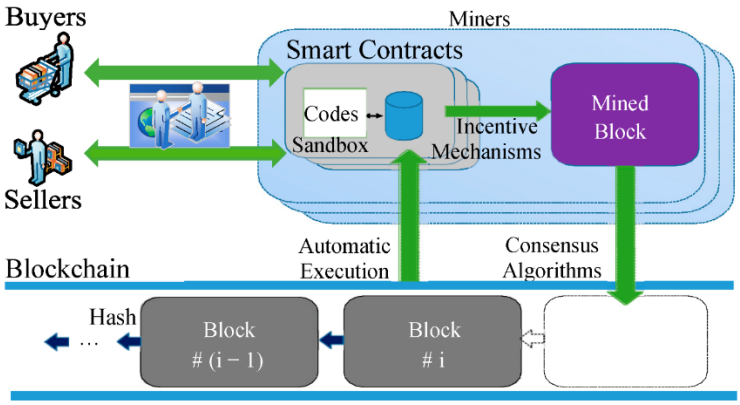

Figure 3. Contract-Drift Detection Positioned Within the Runtime-Verification Taxonomy, adapted from Falcone et al. [11]

5. CONCLUSION

This synthesis has brought together fourteen sources covering microservice architecture, contract testing, semantic versioning and runtime verification to form a coherent conceptual account of contract-drift detection in event-driven microservices. The review shows that established quality-assurance techniques — consumer-driven contract testing, static semantic-versioning analysis and automated API test generation — each contribute to interface reliability but share a common limitation: all operate offline, prior to deployment, and none provides a mechanism for detecting contract violations that arise from schema evolution, configuration drift or undocumented behavioural change in a running system. Drawing on runtime-verification theory, this paper has presented a conceptual framework in which contract drift is formalised through structural-similarity and type-agreement measures computed in flight against a registered baseline, and has situated that framework within an established taxonomy of runtime-verification design decisions concerning monitor placement, verification timing and specification formalism. The synthesis indicates that a decentralised, broker-side monitoring architecture assessed in schema-based terms is a sensible first step toward practical implementation, while behavioural and ordering-based drift detection, standardised benchmarks and baseline-governance mechanisms each warrant dedicated empirical investigation.

REFERENCES

[1] I. K. Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *J. Syst. Softw.*, vol. 180, p. 111014, 2021, doi: 10.1016/j.jss.2021.111014.

[2] N. Dragoni *et al.*, "Microservices: yesterday, today, and tomorrow," *Present ulterior Softw. Eng.*, pp. 195–216, 2017, doi: 10.1007/978-3-319-67425-4_12.

[3] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review," *Empir. Softw. Eng.*, vol. 26, no. 5, p. 104, 2021, doi: 10.1007/s10664-021-09999-9.

[4] J. Lehvä, N. Mäkitalo, and T. Mikkonen, "Consumer-driven contract tests for microservices: A case study," in *International Conference on Product-Focused Software Process Improvement*, Springer, 2019, pp. 497–512. doi: 10.1007/978-3-030-35333-9_35.

[5] I. Kohyarnejadfard, D. Aloise, S. V. Azhari, and M. R. Dagenais, "Anomaly detection in microservice environments using distributed tracing data analysis and NLP," *J. Cloud Comput.*, vol. 11, no. 1, p. 25, 2022, doi: 10.1186/s13677-022-00296-4.

[6] B. Li *et al.*, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empir. Softw. Eng.*, vol. 27, no. 1, p. 25, 2022, doi: 10.1007/s10664-021-10063-9.

[7] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study," *Empir. Softw. Eng.*, vol. 27, no. 3, p. 61, 2022, doi: 10.1007/s10664-021-10052-y.

[8] L. Zhang *et al.*, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12. doi:

10.1145/3551349.3556956.

[9] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for rest apis: No time to rest yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 289–301. doi: 10.1145/3533767.3534401.

[10] Y. Liu *et al.*, "Morest: Model-based restful api testing with execution feedback," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1406–1417. doi: 10.1145/3510003.3510133.

[11] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 2, pp. 255–284, 2021, doi: 10.1007/s10009-021-00609-z.

[12] A. Francalanza, J. A. Pérez, and C. Sánchez, "Runtime verification for decentralised and distributed systems," *Lect. Runtime Verif. Introd. Adv. Top.*, pp. 176–210, 2018, doi: 10.1007/978-3-319-75632-5_6.

[13] M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann, "A distributed hardware monitoring system for runtime verification on multi-tile mpsocs," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, pp. 1–25, 2020, doi: 10.1145/3430699.

[14] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019, pp. 481–490. doi: 10.1109/ICSME.2019.00081.