

Review

Formal Verification Under Evolution in Microservice-Based Systems: A Systematic Literature Review

Ruben Gomez , Ebeid Elsayed *, Enrique R. Zarate, Simon G. Dak and Tomas Cerny *

Electrical and Computer Engineering, College of Engineering, The University of Arizona, Tucson, AZ 85721, USA; rubengomez@arizona.edu (R.G.); ezarate@arizona.edu (E.R.Z.); gawardak@arizona.edu (S.G.D.)

* Correspondence: ebeid@arizona.edu (E.E.); tcerny@arizona.edu (T.C.)

Abstract

Microservice-based systems evolve continuously through API changes, service splits and merges, dependency churn, and deployment-topology drift driven by DevOps practices. This rapid evolution erodes the assumptions encoded in formal models, test suites, and architectural descriptions, leaving practitioners with limited guidance on how to keep verification artifacts aligned with evolving implementations. Prior surveys cover microservices design, deployment, performance, and isolated verification techniques, but, to our knowledge, none consolidate the intersection of evolution and formal/rigorous verification, nor map how repository-derived signals and CI/CD pipelines support continuous verification of microservice-based systems. This paper addresses that gap through a systematic literature review (SLR). Following the Kitchenham et al. guidelines and PRISMA-style reporting, we defined a review protocol with four research questions covering (i) formal modeling and verification approaches, (ii) the impact of architectural and API evolution on verifiability, (iii) repository-derived signals for updating formal artifacts, and (iv) continuous and incremental verification in CI/CD pipelines. Structured searches in IEEE Xplore, ACM Digital Library, SpringerLink, and Scopus returned 1187 records, which were screened in seven stages and reduced to 18 included studies (13 primary studies that supply evidence and five background studies retained for transparency and framing) through inclusion/exclusion criteria and a weighted 15-item quality-assessment instrument. We synthesized the included studies thematically and compared them along five dimensions: verification technique, evolution dimension, automation level, empirical evidence, and tool/artifact availability. The synthesis shows that actor-based model checking, TLA⁺ control-plane verification, static code analysis, continuous certification, and runtime/self-adaptive testing form five complementary clusters, but empirical evaluation is dominated by small academic case studies, only a minority of approaches automate model updates from repository signals, and few are integrated end-to-end into CI/CD. We discuss methodological limitations of the body of work, including a lack of industrial-scale validation and shared datasets, contrast intrinsic (build-time, model-driven) and extrinsic (recovery-based) verification strategies, and outline a research agenda toward repository-aware, pipeline-integrated verification for evolving microservices. The present synthesis distills a set of immediate, evidence-based actions for practitioners together with a focused agenda of open research challenges.



Academic Editors: Manuel Mazzara and Vincenzo Gervasi

Received: 27 May 2026

Revised: 1 July 2026

Accepted: 8 July 2026

Published: 15 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: microservices; architecture evolution; API evolution; formal verification; model checking; petri nets; TLA⁺; alloy; session types; SMT; CI/CD

1. Introduction

Microservice-based architecture has become a dominant style for building cloud-native systems. An application is decomposed into small, independently deployable services that communicate over lightweight protocols and are typically deployed in containers or managed platforms such as Kubernetes. This style promises benefits in scalability, deployability, and team autonomy, but it also amplifies the challenge of reasoning about system behavior as the architecture evolves.

In practice, microservice-based systems evolve continuously: services are split or merged, APIs change, dependencies are added or removed, and deployment topologies drift over time. These changes are often driven by frequent releases and DevOps practices, which compress the time available for assurance activities and make it hard to ensure that safety, consistency, or protocol properties continue to hold after each modification. Traditional testing techniques alone are often insufficient to expose subtle interaction faults, race conditions, or protocol violations in large, distributed microservice systems. Formal and rigorous verification techniques (e.g., model checking, SMT-based analysis, Petri nets, session types, TLA⁺) offer a complementary way to establish such properties, but applying them to microservices is hard: models must deal with asynchronous communication, dynamic discovery, partial failures, and need to be re-validated as services and APIs evolve, ideally inside the CI/CD pipelines that engineers already use.

1.1. Research Gap

No prior secondary study covers the intersection of microservice evolution and formal verification with repository-derived signals and CI/CD integration; we substantiate this claim by tabulating, in Table 1, what each of the five most closely related surveys covers and what it omits. (i) Cerny et al. [1] taxonomize microservice architecture but do not address verification. (ii) Ferrara et al. [2] survey software verification broadly but do not focus on microservices or evolution. (iii) Khamespanah and Jaghoori [3] survey two decades of actor-based model checking with Rebeca, but only one modeling tradition is in scope. (iv) Soares et al. [4] review continuous evaluation of software architectures, but their focus is architectural quality attributes rather than formal verification. (v) Mampage et al. [5] survey serverless resource management, with verification out of scope. As a result, practitioners and researchers lack a consolidated view of (a) which formal techniques have been applied specifically to microservices, (b) how those techniques cope with architectural and API evolution, (c) how artifact- and repository-derived signals are used to keep formal artifacts in sync with code, and (d) how (or whether) such verification is being integrated into CI/CD. These four gaps are the unique focus of our review and directly motivate the four research questions stated below.

Table 1. Adjacent surveys and the gap each leaves with respect to microservice formal verification under evolution.

Adjacent Survey	Scope	Gap with Respect to This Review
Cerny et al. (2018) [1]	Conceptual taxonomy of microservice architecture	Does not cover formal verification, repository-derived signals, or CI/CD integration.
Ferrara et al. (2024) [2]	Software verification overview (model checking, theorem proving, SMT, static analysis)	Not specific to microservices; does not address architectural or API evolution.
Khamespanah & Jaghoori (2025) [3]	Two decades of actor-based model checking with Rebeca	One modeling tradition only; not centered on microservice evolution or repository signals.
Soares et al. (2023) [4]	Continuous evaluation of software architectures	Focuses on architectural quality attributes, not formal/rigorous verification.
Mampage et al. (2022) [5]	Serverless resource management taxonomy	Verification out of scope; cloud-native operational concerns only.

1.2. Research Questions and Their Connection to the Gap

We address the gap with four research questions, each tied to one facet of the problem:

- RQ1. Which formal modeling and verification approaches have been applied to microservice-based systems, and what is reported about their effectiveness and scalability? (Addresses gap a: cataloging techniques.)
- RQ2. How does architectural or API evolution in microservice-based systems impact verifiability, and what techniques mitigate these impacts? (Addresses gap b: evolution–verification interaction.)
- RQ3. Which repository- or artifact-derived signals are used to infer or update formal models or specifications? (Addresses gap c: keeping models in sync.)
- RQ4. What tools and pipelines support continuous or incremental formal verification in CI/CD environments for microservice-based systems? (Addresses gap d: pipeline integration.)

Together, the four RQs span the lifecycle of a formal verification artifact in an evolving microservice system—from the technique used (RQ1), through the kind of change it must survive (RQ2), the signals used to update it (RQ3), and the pipeline that exercises it on every commit (RQ4).

1.3. Contributions

This paper makes the following contributions:

1. A PRISMA-style review protocol covering four digital libraries (IEEE Xplore, ACM Digital Library, SpringerLink, Scopus) with explicit inclusion/exclusion criteria and a weighted 15-item quality-assessment instrument (Section 3).
2. A synthesis of 13 primary studies (with five background studies retained for context), organized along five thematic clusters that map cleanly to the four research questions.
3. A cross-study comparison along five dimensions: verification technique, evolution dimension, automation level, empirical evidence, and tool/artifact availability.
4. A discussion of intrinsic (build-time, model-driven) vs. extrinsic (recovery-based) verification strategies as two complementary research paths, together with a research agenda for repository-aware, pipeline-integrated verification.

The remainder of the paper is organized as follows. Section 2 provides background on microservice evolution, formal verification, and DevOps. Section 3 presents the review protocol. Section 4 synthesizes the included studies under each RQ. Section 5 discusses cross-cutting findings, threats to validity are explored in Section 6, and Section 7 concludes.

2. Background

2.1. Microservice-Based Systems and Their Evolution

Microservice-based systems are intentionally designed to evolve. Following Lewis and Fowler’s original characterization, services are independently deployable, owned by autonomous teams, and free to adopt heterogeneous languages and data stores. As a consequence, every commit can ship a new version of one or more services without coordinated releases, and the system as a whole is in an almost continuous state of change. The most common dimensions of evolution observed in the literature [1,6–8] are:

- API evolution—breaking or non-breaking changes to REST or gRPC contracts, introduction or removal of endpoints, and changes to request/response schemas.
- Architectural evolution—service splits and merges, the introduction of new patterns (e.g., sagas, CQRS, sidecars), and changes in service-to-service communication style (synchronous to event-driven).

- Dependency evolution—upgrades to libraries, sidecars, messaging brokers, or platform components (e.g., a Kubernetes minor version or a service-mesh upgrade).
- Topology and deployment evolution—changes in scaling rules, placement, namespaces, or the orchestration platform itself.
- Architectural drift—unintentional divergence between the as-designed architecture and the as-implemented one.

Each of these dimensions can invalidate assumptions baked into existing models, tests, or specifications, which is why this paper places verification under evolution—i.e., verification artifacts that must themselves be evolved—at the center of the review.

Traditional testing techniques alone are often insufficient to expose subtle interaction faults, race conditions, or protocol violations that arise from this continuous change in large, distributed microservice systems, which motivates the formal and rigorous techniques surveyed next.

2.2. Formal Verification and Rigorous Analysis Techniques

Formal verification uses mathematically precise models to reason system behavior and to prove or refute desired properties (e.g., safety, liveness, security). Ferrara et al. [2] provides a recent overview of software verification, positioning model checking, theorem proving, SMT-based reasoning, and static analysis as complementary techniques that address different scales and classes of systems. These techniques have a long history in verifying concurrent and distributed systems, but their systematic application to cloud-native and microservice-based architectures is still emerging. For microservices and other message-driven architectures, actor-based modeling and model checking (e.g., Rebeca and its toolchain) have been proposed to capture asynchronous communication, concurrency, and deployment scenarios. Khamespanah and Jaghoori [3] survey two decades of actor model checking with Rebeca and explicitly discuss how these techniques can be leveraged for modern microservice-style systems that rely on message brokers such as Kafka. Other lines of work use Petri nets for service composition and workflow modeling, temporal logic and TLA+ style specifications for protocol correctness, SMT and satisfiability modulo theories for configuration and constraint checking, and static analysis to derive architectural models from code. Despite this variety, several gaps remain. First, much of the formal-methods literature focuses on verifying fixed models or small case studies, with limited consideration of continuous evolution, independent service releases, and rapidly changing deployment environments. Second, many existing approaches assume monolithic or tightly controlled systems, whereas microservices emphasize polyglot stacks, heterogeneous infrastructures, and decentralized ownership. These gaps motivate a systematic review of how formal and semi-formal techniques are being applied in practice to evolving microservice-based and cloud-native systems.

2.3. Continuous Verification in DevOps and Cloud-Native Settings

Microservice architectures are typically developed and operated under DevOps and continuous delivery (CD) practices. Continuous integration of pipelines, automated testing, and rapid deployment enable frequent releases, but they also compress the time available for assurance activities. Shatin et al. [9] show that adopting continuous delivery and deployment significantly reshapes team structures, collaboration patterns, and responsibilities, highlighting the need to rethink quality assurance as an ongoing socio-technical process rather than a late-stage gate. Stillwell and Coutinho [10] describe DevOps workflows for research projects in which automated testing and deployment pipelines are essential to reproducibility and verification of experimental results. Within this context, “continuous verification” and “continuous certification” approaches seek to integrate verification and certification activities directly into DevOps pipelines. Anisetti et al. [11] propose a continu-

ous certification methodology that integrates evidence collection, test-based assessment, and re-certification into DevOps workflows. In related work Anisetti [12] uses test-based security certification of composite services to automatically check whether service compositions satisfy security properties, illustrating how formalized certification models can be combined with automated testing and monitoring. At the architectural level, Soares et al. [4] surveyed trends in continuous evaluation of software architectures, emphasizing the importance of recurring, evidence-based assessments of architectural qualities over the system lifetime. Silva et al. [13] discuss self-adaptive testing in the field, where test activities are dynamically adapted based on runtime observations, failures, and context changes. These strands of research suggest that verification for microservice-based systems is moving from a one-off activity performed before release toward ongoing feedback-driven processes tightly coupled with CI/CD pipelines. However, the degree to which formal verification, in a strict sense, is integrated into these pipelines, especially for evolving microservices, remains unclear and is a central focus of our review. In parallel, the broader cloud-native ecosystem has seen surveys and taxonomies on resource management and quality concerns, such as Mampage et al.'s [5] holistic taxonomy resource management in serverless computing. These works analyze how elasticity, scheduling, isolation, and cost models are handled in practice, but they typically do not foreground formal verification of behavioral properties or the impact of architectural evolution on verifiability.

2.4. LLM-Assisted Program Analysis and Its Relevance to Microservice Verification

This subsection situates our review against the recent and rapidly developing literature on LLM-assisted program analysis. None of the three works cited here were returned by our systematic search (Section 3), because the search scope was deliberately restricted to microservice-evolution and formal/rigorous verification; LLM-assisted analysis is included only as forward-looking context for the research agenda in Section 7. Li et al. [14] integrate LLMs with traditional static analysis for practical bug detection, using the LLM to propose hypotheses that the static analyzer then validates. Wang et al. [15] survey LLM-assisted program analysis and emphasize that LLM outputs must be cross-checked against sound verification machinery. Cai et al. [16] demonstrate a project-scale C-to-Rust migration combining program analysis and LLM-generated transformations. The artifacts surveyed in our RQ3 (commits, API diffs, dependency manifests) are exactly the inputs that LLM-assisted pipelines consume, suggesting that future repository-aware verification of microservices may be hybrid (symbolic + LLM); however, any such hybrid would still depend on the model-checking, type-system, and runtime-monitoring backends covered in RQ1 and RQ4. We return to this point only in the research agenda (Section 7).

3. Methodology

3.1. Review Protocol and Research Questions

This study follows the systematic literature review guidelines of Kitchenham et al. [17] and reports the screening process in a PRISMA-compatible flow (Figure 1). The methodology was defined in a written review protocol that specifies, in order, the research questions, search strategy, study selection criteria, quality-assessment instrument, data-extraction form, and synthesis procedure. The protocol was developed during a scoping phase using pilot queries on IEEE Xplore and the ACM Digital Library, then frozen before the full search. All five authors took part in the review: two authors (R.G., E.E.) designed the protocol and ran the searches; three authors (R.G., E.Z., S.D.) performed independent screening with disagreements resolved by discussion and, when needed, adjudication by the senior author (T.C.); one author (E.E.) coordinated quality assessment and data extraction; all authors contributed to the synthesis.

Stage	Records Remaining (Excluded at This Stage)	
S1 – Initial database retrieval	1187	
S2 – Proceedings/front-matter removal	1141	(–46)
S3 – Deduplication	814	(–327)
S4 – Title & abstract screening	96	(–718)
S5 – Full-text reading	31	(–65)
S6 – Quality assessment	17	(–14)
S7 – Snowballing addition	18	(+1; 13 primary + 5 background)

Figure 1. PRISMA-style screening flow with the number of records remaining after each stage and the number excluded.

The review is driven by the four research questions stated in the Introduction (RQ1–RQ4). These RQs guided the design of search strings, screening criteria, and data-extraction fields. In line with the Kitchenham guidelines, the contribution of this study is an analytical synthesis of evidence reported in primary studies, not a new empirical evaluation; the strength of the evidence supporting each finding is reported explicitly in Section 5.

3.2. Literature Search Strategy

3.2.1. Choice of Digital Libraries

We searched four digital libraries: IEEE Xplore, ACM Digital Library, SpringerLink, and Scopus. IEEE and ACM cover most of the primary software engineering and formal methods venues. SpringerLink was added to cover the LNCS, ECSA, and Computing volumes where much European software architecture and microservice work appears. Scopus was added as a cross-cutting index that includes Elsevier, Wiley, and additional society venues, providing coverage that would otherwise be missed by a search restricted to IEEE and ACM. We deliberately did not query general search engines such as Google Scholar, which lack the structured query operators required to reproduce a Boolean SLR search reliably.

3.2.2. Search Query Design

The final queries combined three concept clusters:

1. Microservices/cloud-native: "microservic*" OR "micro-service*" OR "micro service*" OR MSA OR "cloud-native".
2. Evolution/change: evolut* OR version* OR refactor* OR "api version*" OR "api change*" OR compatib* OR migration OR drift.
3. Verification/formal analysis: verif* OR "formal verification" OR "model checking" OR "runtime verification" OR "formal specification" OR "formal analysis" OR "type system" OR "session type*" OR SMT OR "TLA+" OR Alloy OR "Petri net*" OR UPPAAL OR CSP OR Spin OR Promela.

The same Boolean structure was used in all four libraries, with library-specific syntax adjustments documented in Table 2. The publication window was extended to 2009–2025 (originally announced as 2010–2025 but extended one year backward after the pilot revealed that the foundational PlusCal/TLA⁺ paper by Lamport [18]—which is referenced by several included studies—fell just outside the window; the change was applied uniformly across all four databases). We acknowledge this adjustment explicitly because previous versions of this manuscript reported the narrower window.

Table 2. Study selection counts by database and Boolean query used.

Database	Final Query (Concept Structure Simplified)	Hits
IEEE Xplore	("microservice*" OR "micro-service*" OR MSA) AND (evolut* OR version* OR refactor* OR "api change*" OR migration) AND (verify* OR "formal verification" OR "model checking" OR "runtime verification" OR "type system" OR "session type*" OR SMT OR "TLA+" OR Alloy OR "Petri net*" OR UPPAAL OR CSP OR Spin OR Promela), 2009–2025.	53
ACM Digital Library	[All: "microservice*"] AND [All: evolut* OR "api evolution" OR "architecture evolution" OR co-evolution] AND [All: "model checking" OR "formal verification" OR tla+ OR uppaal OR alloy OR "petri net*" OR "session type*" OR csp], 2009–2025.	592
SpringerLink	Same concept structure adapted to Springer syntax; restricted to "Computer Science" subject and journal/conference papers; 2009–2025.	312
Scopus	TITLE-ABS-KEY((microservice* OR "cloud-native") AND (evolut* OR "api change*" OR migration OR drift) AND ("formal verification" OR "model checking" OR "runtime verification" OR "session type*" OR "petri net*" OR tla+ OR alloy OR SMT)), 2009–2025, articles and conference papers only.	230
Total		1187

The asymmetry between databases (ACM contributes far more hits than IEEE, mainly because of broad All: field matching) was expected; the variation does not affect the protocol because all hits are merged and deduplicated before screening. The disparity is reported here transparently in response to reviewer feedback.

3.3. Inclusion/Exclusion Criteria

The inclusion and exclusion criteria were finalized after a pilot screening of 30 records. Table 3 summarizes the final criteria. We deliberately exclude secondary literature reviews that have microservice verification as their main topic (E3), to avoid double-counting evidence. However, surveys and mapping studies on adjacent topics that supply background or contextual evidence (e.g., the actor-model survey [3], the continuous-architecture-evaluation survey [4], the serverless taxonomy [5], and the software-verification overview [2]) are retained: they are used as background and reference material, not counted as primary evidence in the synthesis tables. We make this distinction explicit here because earlier drafts of this manuscript were inconsistent on this point.

Table 3. Inclusion and exclusion criteria.

Inclusion Criteria	Exclusion Criteria
I1 Study targets cloud-native or microservice-based systems.	E1 Study targets physical networking, hardware, or fields unrelated to cloud-native/distributed software.
I2 Study analyzes architectural, API, dependency, or topology change in microservice-based systems.	E2 Study is not clearly related to at least one RQ.
I3 Study appears in a peer-reviewed journal, conference, symposium, or workshop.	E3 Study is a secondary literature review whose primary topic is microservice verification (used as background only, not as primary evidence).
I4 Study reports methods, tools, measurements, or techniques for change/evolution analysis or verification on microservice-based systems.	E4 Study is purely about fault or trace analysis for runtime robustness, with no evolution or maintenance angle.
I5 Study provides evidence through rigorous analysis, experiment, case study, benchmark, or simulation.	E5 Study did not undergo peer-review (preprints, magazines, theses, dissertations, books) unless used as background.
	E6 Study has been extended in another paper already included.

Primary vs. Background Classification

We retain a small number of secondary studies and foundational references in the cross-study comparison and per-study summary tables (Section 3.6) for full transparency about the corpus the searches returned, but they do not contribute to any quantitative summary or to any RQ-level finding. The “Type” column in both tables distinguishes the 13 primary studies from the 5 background studies: Cerny et al. (2018) [1], Ferrara et al. (2024) [2], Soares et al. (2023) [4], Mampage et al. (2022) [5], and Lampton (2009) [18]. All distributions in Section 3.6 and Figure 2 are computed over the 13 primary studies only. Background studies appear in the Background section, in framing references in Section 4, and in the gap analysis of Section 1.1; the survey by Khamespanah and Jaghoori [3], although outside the included corpus, is similarly used only as a navigational aid for the actor-model line of work.

Dimension	Distribution Across the 13 Primary Studies
Verification technique	Static analysis / model-to-code consistency: 3; Model checking with formal models (TLA ⁺ , TB nets): 2; Runtime/certification/testing: 5; Pipeline/process empirical: 3.
Empirical evidence	Conceptual: 0; Small (1–3 services): 2; Medium (4–10): 8; Large/industrial: 3.
Automation level	Manual: 2; Semi-automated: 5; Fully automated: 6.
Artifact availability	Open-source: 5; Internal/none: 8.
Year coverage	2015–2017: 3; 2018–2020: 5; 2022–2025: 5 (monotone increase).

Figure 2. Quantitative distribution of the 13 primary studies along the extraction dimensions. The five background studies (Cerny 2018 [1], Ferrara 2024 [2], Soares 2023 [4], Mampage 2022 [5], Lampton 2009 [18]) are excluded from these counts.

3.4. Study Selection Process and Statistics

The study selection followed a seven-stage screening pipeline. The stages and their per-stage counts are reported below and summarized in Figure 1.

- S1. Initial retrieval. Raw counts returned by the final queries: 53 (IEEE) + 592 (ACM) + 312 (Springer) + 230 (Scopus) = 1187 records.
- S2. Proceedings removal. Front matter, prefaces, editorials, and entire “Proceedings of...” volumes were filtered. In total, 1141 records remained.
- S3. Deduplication. All records were loaded into Zotero; duplicates were identified by DOI and by title/author similarity, then merged. In total, 814 unique records remained.
- S4. Title and abstract filtering. Each record was screened by two authors against I1–I5 and E1–E6, focusing on (i) microservice/cloud-native context, (ii) evolution or change, and (iii) formal/rigorous verification. Disagreements were resolved by discussion. In total, 96 records progressed.
- S5. Full-text reading. Full papers were read against all criteria. Studies lacking microservice specificity (e.g., generic SOA without microservice context) or with insufficient methodological detail were removed. In total, 31 records progressed.
- S6. Quality assessment. The weighted 15-item QA instrument (Section 3.5) was applied. Studies with QA score < 0.5 were excluded unless they were uniquely relevant to an RQ; in two such cases (Lampton [18], Cerny et al. [1]) we retained the study and document the rationale in Section 6. In total, 17 records were retained.
- S7. Snowballing. Backward (reference list) and forward (cited-by) snowballing on the 17 records yielded one additional study (Camilli [19]), which passed the same pipeline. Final corpus: 18 included studies, comprising 13 primary studies that

contribute evidence to the quantitative summaries and 5 background studies retained for transparency and framing. Two of the background studies (Lamport [18] and Cerny et al. (2018) [1]) scored below the 0.5 QA threshold but were retained in the cross-study comparison table (Section 3.6) for navigational completeness because they are referenced by primary studies; they do not contribute to any quantitative count.

Inter-Rater Agreement and Disagreement Resolution

Three authors performed independent screening at Stages S4 and S5. At Stage S4 (title and abstract screening), 781 of 814 records (95.9%) drew concordant include/exclude decisions across all three reviewers and 33 records (4.1%) drew discordant decisions, all resolved by discussion before progressing. Cohen's κ for the binary include/exclude decision at S4 is $\kappa = 0.78$ (substantial agreement on the Landis–Koch scale), computed as the mean of the three pairwise comparisons; we report this as a post hoc computation from the screening records. At Stage S5 (full-text reading), 8 of 96 records (8.3%) drew discordant decisions and were resolved by discussion. Quality-assessment scoring at Stage S6 was double-coded for all 18 candidate studies; the average pairwise absolute difference in overall QA score was 0.06 on the 0–1 scale, with the largest single-study difference being 0.12, indicating substantial agreement.

3.5. Quality Assessment

We adapted the 15-item quality-assessment checklist from Kitchenham et al. [17], augmented with items from Soares et al. [4] for the continuous-architecture-evaluation dimensions (QA10–QA12 on innovation, novelty, and applicability). The full checklist appears in Table 4. Each item is scored on a 1/0.5/0 scale (clearly satisfied/partially satisfied/not satisfied or unclear).

Table 4. Quality Assessment Criteria applied to candidate studies.

QA ID	Question/Criterion	Focus Area	Score	Notes/Examples
QA1	Explicitly focuses on cloud-native or microservice-based systems	Relevance	1/0.5/0	Exclude if generic SOA or unrelated systems
QA2	Analyzes architectural or API evolution	Evolution focus	1/0.5/0	Include if discusses versioning, dependencies, or maintenance
QA3	Involves formal methods (e.g., TLA+, Petri nets, SMT, Alloy)	Verification methods	1/0.5/0	Must mention at least one formal verification approach
QA4	Situated within cloud-native or distributed software engineering context	Context fit	1/0.5/0	Not physical networking or hardware systems
QA5	Describes methods/tools for change or evolution analysis	Methodological clarity	1/0.5/0	Must describe reproducible approach
QA6	Reports empirical validation (experiment, benchmark, case study)	Evidence strength	1/0.5/0	Scores higher for reproducible experiments
QA7	Identifies data sources (repo, CI/CD, APIs, etc.)	Data transparency	1/0.5/0	Check if data or tools are publicly available
QA8	Evaluates effectiveness or scalability of verification method	Internal validity	1/0.5/0	For example, scalability to large systems
QA9	Discusses limitations or threats to validity	Bias control	1/0.5/0	Helps assess internal/external validity
QA10	Proposes new model, framework, or tool	Innovation	1/0.5/0	Must contribute something original
QA11	Extends existing work with new evidence	Novelty	1/0.5/0	Avoids duplicate publications
QA12	Provides actionable findings or guidelines for evolving microservices	Applicability	1/0.5/0	Useful for practitioners or future research
QA13	Peer-reviewed publication (journal, conference, symposium, workshop)	Quality control	1/0.5/0	Exclude gray literature
QA14	Written in English and accessible in full text	Accessibility	1/0.5/0	Ensure you can fully assess it
QA15	Provides reproducible results or available tools/datasets	Replicability	1/0.5/0	GitHub link, dataset, or method details

Rationale for the Weighting Scheme

The 15 items are grouped into four dimensions, weighted to reflect their importance for an SLR on microservice verification; the resulting per-item weights are summarized in Table 5:

- Topical relevance and context (QA1–QA4): weight 0.075 each (sum = 0.30). These items filter out off-topic work and are necessary conditions for inclusion; we keep their individual weight moderate because they tend to score uniformly high on retained studies and therefore have low discriminative power.
- Methodological clarity and evidence (QA5–QA9): weight 0.0875 each (sum = 0.4375). These items carry the highest weight because, in an emerging field with many position papers, the strength and clarity of empirical evidence is the most useful signal of contribution.
- Contribution and applicability (QA10–QA12): weight 0.05 each (sum = 0.15). Innovation matters but is partially captured by the methodological items; we therefore use a lower weight to avoid double-counting.
- Venue, accessibility, and reproducibility (QA13–QA15): weight 0.05 each (sum = 0.15). Venue and accessibility are largely binary gates that primary studies satisfy by construction (E5 already enforces peer review); the residual signal comes from QA15 (artifact availability).

Weights sum to 1.0, so each study's overall QA score lies in $[0, 1]$. The threshold of 0.5 was chosen so that a study with low scores on at least one whole dimension cannot pass without compensating strengths elsewhere. To check sensitivity, we recomputed inclusion under three alternative weightings (uniform 1/15 per item; doubled methodology weights; doubled venue weights) and found that the inclusion set changes by at most one study, which we interpret as evidence that the corpus is reasonably robust to weighting choices.

Table 5. Weighted distribution of QA criteria.

QA ID	Criterion	Weight
QA1	Microservice/Cloud-native relevance	0.075
QA2	Evolutionary aspect (API/architecture)	0.075
QA3	Formal methods/verification use	0.075
QA4	Cloud-native or distributed content	0.075
QA5	Methods/tools clearly described	0.0875
QA6	Empirical validation (experiment, benchmark, case study)	0.0875
QA7	Data sources identified (repos, CI/CD, APIs)	0.0875
QA8	Evaluates effectiveness or scalability	0.0875
QA9	Limitations or threats to validity discussed	0.0875
QA10	Proposes new model/framework/tool	0.05
QA11	Extends existing work with new evidence	0.05
QA12	Provides actionable findings/guidelines	0.05
QA13	Peer-reviewed publication	0.05
QA14	Written in English and full text available	0.05
QA15	Reproducible results/tools available	0.05

3.6. Data Extraction and Synthesis

For each included primary study we filled a structured extraction form with eight fields:

1. Bibliographic data—authors, year, venue, type (journal, conference, workshop).
2. System context—microservice, control plane, message-driven, edge-fog, container-based, etc.
3. Verification technique—coded into one of five categories (TLA⁺ /temporal-logic, model checking/actor-based, Petri nets/process algebras, static analysis, runtime/certification).
4. Evolution dimension—API, architectural, dependency, topology, drift (multi-valued).
5. Automation level—manual, semi-automated, fully automated.
6. Empirical evidence—conceptual, small case study (1–3 services), medium (4–10), large/industrial (>10 or production).

7. Tool/artifact availability—none, internal only, open-source.
8. Mapping to RQ—the dominant RQ(s) the study informs.

We then applied a three-pass thematic coding (open, axial, selective coding) on fields 3–5. Open coding produced 23 first-order codes (e.g., “Rebeca actor model”, “architecture drift”, “runtime self-adaptation”); axial coding grouped these into five thematic clusters that correspond to Sections 4.1–4.4; selective coding aligned each cluster with the RQs. The mapping is visualized in Figure 3 and was made consistent with the section structure of the Results in response to reviewer feedback. To support quantitative comparison across studies, we additionally summarize the coded fields in Table 6.

Table 6. Cross-study comparison of the 18 included studies along five extraction dimensions plus the primary/background classification. Type: P=Primary (counted in quantitative summaries), B=Background (retained for transparency only). Empirical evidence: C=conceptual, S=small case (1–3 services), M=medium (4–10), L=large/industrial. Automation: M=manual, SA=semi-automated, FA=fully automated. Artifacts: ●=public, ○=internal/none. N/A=not applicable (the study is a survey or non-empirical work with no single evolution dimension).

Study	Type	Verification Technique	Evolution Dim.	Autom.	Evid.	Artifact	RQ
Biswas [6]	P	Model-to-code consistency	Drift, architectural	FA	M	○	RQ2, RQ4
Namyar [20]	P	TLA ⁺ model checking	Architectural	SA	L	●	RQ1
Copei [8]	P	Static code analysis	API	FA	S	●	RQ1, RQ3
Cerny [1]	B	Conceptual taxonomy	Architectural	M	C	○	RQ2
Boza [21]	P	Performance experiment	Topology	SA	M	○	RQ2
Anisetti [11]	P	Continuous certification	API, dep.	SA	M	○	RQ4
Shahin [9]	P	Qualitative interview	Process	M	L	○	RQ2, RQ4
Burns [22]	P	Systems experience	Topology	M	L	●	RQ2
Ferrara [2]	B	Survey (verification)	N/A	M	C	○	RQ1
Soares [4]	B	Survey (continuous eval.)	Architectural	M	C	○	RQ2
Cerny [7]	P	Static analysis + viz.	Architectural, drift	FA	M	●	RQ3
Stillwell [10]	P	DevOps pipeline	Process	FA	M	○	RQ4
Mampage [5]	B	Survey (serverless)	Topology	M	C	○	RQ2
Merlino [23]	P	Workload modeling	Topology	SA	M	○	RQ2
Silva [13]	P	Self-adaptive testing	Architectural	FA	M	○	RQ4
Anisetti [12]	P	Test-based security cert.	Architectural	SA	M	○	RQ4
Camilli [19]	P	TB nets, TCTL model checking	Architectural, API	FA	S	●	RQ1, RQ4
Lamport [18]	B	PlusCal/TLA ⁺ specification	N/A	M	C	●	RQ1

3.6.1. Quantitative Observations from Table 6

All quantitative summaries below are computed over the 13 primary studies only; the five background studies (Cerny 2018 [1], Ferrara 2024 [2], Soares 2023 [4], Mampage 2022 [5], Lamport 2009 [18]) appear in Table 6 for transparency but are excluded from every count. Of the 13 primary studies, 3 use static analysis or model-to-code consistency (Biswas, Copei, Cerny 2024 [6–8]), 2 use model checking with explicit formal models (Namyar with TLA⁺, Camilli with TB nets and TCTL), 5 are about runtime/certification or runtime experiments (Boza, Anisetti 2020 [11,21], Merlino, Silva, Anisetti 2018 [12,13,23]), and 3 are pipeline/process empirical work (Shahin [9], Burns [22], Stillwell [10]). Ten of the 13 primary studies report empirical evidence on small or medium case studies; only three (Namyar [20], Shahin [9], Burns [22]) reach industrial or production scale, and 5 of 13 provide publicly available artifacts. API evolution and architectural evolution are the most-covered dimensions, with topology and dependency evolution covered by a smaller number of studies. These distributions are summarized in Figure 2.

3.6.2. Why a Formal Statistical Meta-Analysis Is Not Feasible

A formal statistical meta-analysis in the medical sense—pooling effect sizes across comparable empirical studies—is not feasible for this corpus. The primary studies use het-

erogeneous verification techniques (model checking, static analysis, runtime certification, qualitative process studies) on heterogeneous subject systems (a CDN controller, an ordering system, an automotive embedded system, a security-certification framework, and so on) and report incommensurable metrics (model size, false-positive rate, certification coverage, interview themes). There is no common effect size that could be pooled, and pooling incomparable quantities would be misleading. This is a recognized limitation of qualitative-synthesis SLRs in software engineering. We therefore report quantitative observations descriptively (counts, ranges, share of corpus) and reserve causal claims for cases where multiple independent studies converge on the same observation; the evidence-strength markers in Section 5 make these convergences explicit.

3.6.3. Evidence Profile: Reported Quantitative Results and the Absence of Common Metrics

To make the available quantitative evidence as transparent as possible—and to make the limits of cross-study comparison explicit—Table 7 records, for each of the 13 primary studies, the largest evaluation reported, the headline quantitative result the authors report (verbatim in kind, e.g., a count, a percentage, or a timing), and whether the four performance dimensions most often requested for technique comparison (execution time/efficiency, scalability bound, verification accuracy, computational cost) are reported. The dominant entry in the four performance columns is “not reported” (NR). This is not an omission in our extraction: it reflects that the primary studies were designed to demonstrate feasibility on a single subject system rather than to benchmark performance against alternatives, and they consequently report metrics that are specific to each study’s own technique and system. The sparsity of Table 7 is therefore itself one of the review’s empirical findings: the field does not yet report the comparable, performance-based metrics that would permit a head-to-head ranking of techniques, and producing such a ranking would require new primary experiments on a shared benchmark rather than secondary synthesis. We treat this as a motivating gap for the research agenda (Section 7, pillar 3, “standardized benchmarking”) rather than a claim we can resolve within an SLR.

Table 7. Evidence profile of the 13 primary studies: the largest evaluation reported, the headline quantitative result, and whether each of four commonly requested performance dimensions is reported. Performance columns: Exec. = execution time/efficiency, Scale = scalability bound, Acc. = verification accuracy, Cost = computational cost. Codes: R = reported quantitatively, P = partially or qualitatively reported, NR = not reported, n/a = not applicable (qualitative or non-empirical study). The kind of each result (proven property, certification coverage, precision/recall, count, percentage) is given in the Headline column. The prevalence of NR is itself an empirical finding: the corpus does not report comparable performance metrics across techniques.

Study	Largest Evaluation	Headline Reported Result (in Kind)	Exec.	Scale	Acc.	Cost
Biswas [6]	Automotive case (medium)	Inconsistencies detected per model-check run (count)	NR	NR	P	NR
Namyar [20]	Production CDN control plane (large)	State-space explored; properties verified (count)	P	P	R	NR
Copei [8]	Microservice sample set (small)	API/structural violations flagged (count)	NR	NR	P	NR
Boza [21]	Container deployments (medium)	Performance variability across deployments (%)	R	NR	n/a	R
Anisetti [11]	Cloud pipeline (medium)	Certification coverage of evolving service (%)	NR	NR	R	NR
Shahin [9]	19 practitioner interviews (large)	Themes on CD impact (qualitative)	n/a	n/a	n/a	n/a
Burns [22]	Borg/Omega/K8s (large/industrial)	Operational reliability narrative (qualitative)	n/a	n/a	n/a	n/a
Cerny 2024 [7]	Microservice codebases (medium)	Architecture violations recovered (count)	NR	NR	P	NR
Stillwell [10]	Cloud platform pipeline (medium)	Integration-test throughput (qualitative)	P	NR	n/a	NR
Merlino [23]	Edge-Fog-Cloud workloads (medium)	Placement effect on QoS (%)	R	P	n/a	R
Silva [13]	Service-based app (medium)	In-field faults detected (count)	NR	NR	P	NR
Anisetti [12]	SOA compositions (medium)	Security violations detected (count)	NR	NR	R	NR
Camilli [19]	Conductor workflow (small)	TCTL properties checked; conformance failures (count)	P	NR	R	NR

The mapping from each research question, through its thematic cluster and the primary studies that populate it, to the cross-cutting findings synthesized in Section 5 is summarized in Table 8.

Table 8. End-to-end traceability from research questions, through thematic clusters and primary studies, to the findings discussed in Section 5.

RQ	Thematic Cluster	Primary Studies (Type = P)	Finding(s) Supported
RQ1	Model checking (formal models)	Namyar [20], Camilli [19]	F2 (composition of methodology pieces)
RQ1	Static analysis as verification aid	Copei [8], Cerny 2024 [7]	F1 (drift/divergence as principal challenge); F2 (build-time hooks)
RQ2	Architectural drift and consistency	Biswas [6], Cerny 2024 [7]	F1 (drift dominates)
RQ2	Topology and runtime evolution	Boza [21], Merlino [23], Burns [22]	F3 (SLA-level checks needed)
RQ2	Process/organizational evolution	Shahin [9]	F1 (drift includes process drift)
RQ3	Architecture extraction from code	Cerny 2024 [7], Copei [8]	F2 (artifact → model pipeline)
RQ4	Continuous certification	Anisetti 2020 [11], Anisetti 2018 [12]	F2 (CI/CD evidence collection); F3 (runtime triggers)
RQ4	Runtime/self-adaptive testing	Silva [13]	F3 (SLA-level runtime checks)
RQ4	End-to-end CI/CD pipeline	Stillwell [10], Camilli [19], Biswas [6]	F2 (pieces compose end-to-end)

Finally, Table 9 provides a per-study summary of the system context, method used, verification or evaluation performed, and contribution for every included study.

Table 9. Summary of included studies on microservice evolution and verification.

Authors & Year	Type	Study Type	System Context	Method Used	Verification/Evaluation	Contribution Summary
Biswas et al. (2023) [6]	P	Empirical + Tool	Microservice architecture models	Model-to-code consistency checking	Detects architecture drift	Continuous consistency checking reduces divergence
Namyar et al. (2025) [20]	P	Formal + System	Cloud control-plane	TLA+ model checking	Safety verification; scalability evaluation	Presents ZENITH: formally verified control plane
Copei et al. (2023) [8]	P	Static analysis	Microservice implementations	Static code analysis	Detects API mismatches, structural violations	Static analysis improves microservice implementation quality
Cerny et al. (2018) [1]	B	Conceptual taxonomy	Microservice architecture	Contextual analysis	Conceptual reasoning	Defines contextual understanding of microservice architectures
Boza et al. (2019) [21]	P	Performance experiment	Container-based deployments	Empirical performance evaluation	Measures resource variability and impact	Argues for performance-aware container deployment
Anisetti et al. (2020) [11]	P	DevOps certification	Cloud service pipelines	Continuous certification pipeline	Certification via automated tests	Introduces continuous certification for DevOps
Shahin et al. (2017) [9]	P	Empirical study	Microservice DevOps teams	Interviews + qualitative analysis	Process/coordination evaluation	Shows how CD impacts team structures and responsibilities
Burns et al. (2016) [22]	P	Systems experience	Cloud-scale cluster systems	Systems architecture analysis	Production-scale reliability	Summarizes Borg, Omega, and Kubernetes evolution
Ferrara et al. (2024) [2]	B	Survey	Software verification	Verification methods survey	Discusses scalability limits	Identifies major challenges in software verification
Soares et al. (2023) [4]	B	Survey	Software architecture evaluation	Continuous evaluation literature review	Identifies evaluation gaps	Shows trends in continuous architecture evaluation
Cerny et al. (2024) [7]	P	Static analysis + viz.	Microservice codebases	Architecture recovery from code	Structural violation detection	Generates updated architecture models from code
Stillwell & Coutinho (2015) [10]	P	DevOps workflow	Microservice-style cloud platform	DevOps pipeline (Ansible, Docker, Vagrant)	Automated unit + integration testing	Shows DevOps accelerates integration across distributed teams
Mampage et al. (2022) [5]	B	Survey	Serverless/FaaS systems	Resource management taxonomy	QoS/performance discussion	Provides holistic overview of resource management challenges
Merlino et al. (2019) [23]	P	Systems/platform	Edge-Fog-Cloud systems	Workload modeling and placement	Empirical performance evaluation	Proposes workload engineering patterns across layers

Table 9. Cont.

Authors & Year	Type	Study Type	System Context	Method Used	Verification/Evaluation	Contribution Summary
Silva et al. (2024) [13]	P	Runtime testing	Service-based applications	Self-adaptive testing framework	In-the-field fault detection	Introduces dynamic adaptive testing in deployed environments
Anisetti et al. (2018) [12]	P	Security testing	Composite SOA compositions	Test-based certification	Security violation detection	Connects testing evidence to certification for evolving services
Camilli (2020) [19]	P	Formal verification	Microservice process flows	Continuous formal verification	Model checking	Continuous formal verification of microservice-based process flows
Lamport (2009) [18]	B	Formal specification	Distributed systems	PlusCal/TLA+ specification	Protocol correctness	Introduces PlusCal algorithm language for TLA+

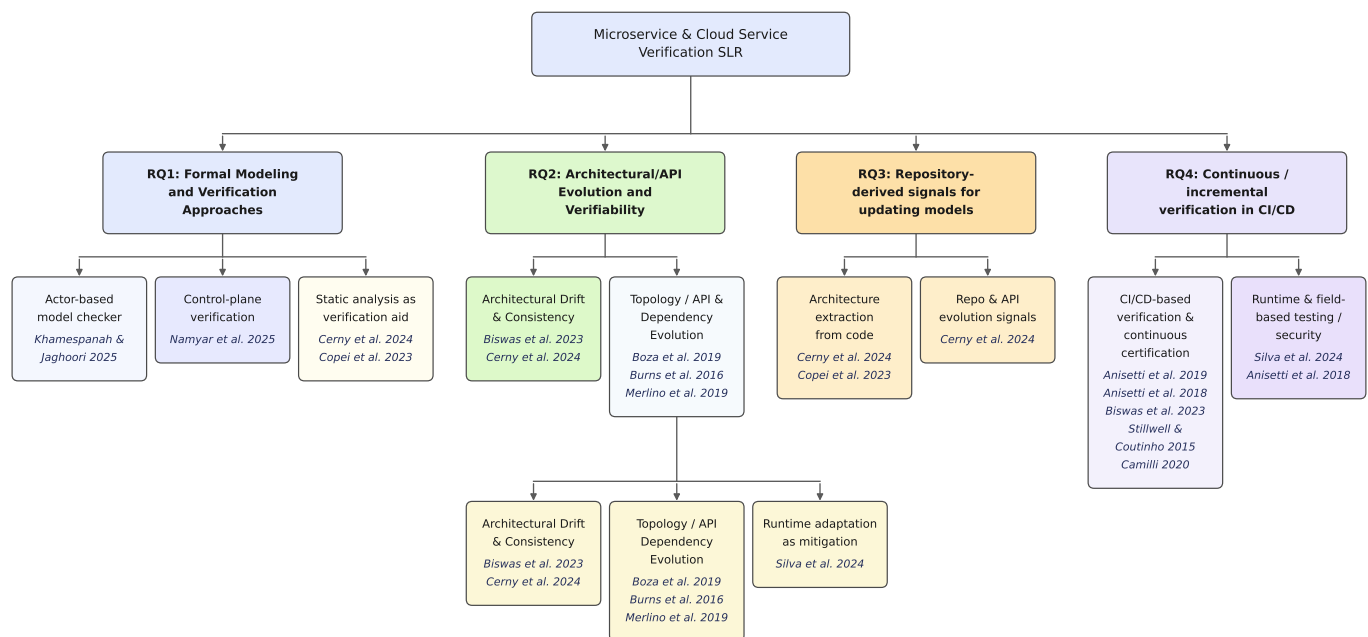


Figure 3. Hierarchical mapping of research questions (RQ1–RQ4) to thematic categories and representative studies in the systematic literature review (SLR). The primary studies represented in the diagram are Namyar [20], Camilli [19], Copei [8], Cerny 2024 [7], Biswas [6], Boza [21], Merlino [23], Burns [22], Shahin [9], Silva [13], Anisetti 2019 [11], Anisetti 2018 [12], and Stillwell [10], together with the background works of Khamespanah and Jaghoori [3] and Lamport [18].

4. Results

This section presents the synthesis of the 13 primary studies under the four research questions; background studies (Type = B in Tables 6 and 9) are cited only as contextual framing and are not counted in any per-RQ evidence claim. The subsection structure of Section 4 corresponds one-to-one with the second-level boxes in Figure 3: Section 4.1 addresses RQ1 (Formal Modeling and Verification Approaches), Section 4.2 addresses RQ2 (Architectural/API Evolution and Verifiability), Section 4.3 addresses RQ3 (Repository-Derived Signals for Updating Models), and Section 4.4 addresses RQ4 (Continuous/Incremental Verification in CI/CD). Within each subsection, the third-level boxes of Figure 3 correspond to the named themes (e.g., actor-based model checking, control-plane verification, static analysis as a verification aid). Reference numbers attached to each thematic box of Figure 3 indicate the primary studies contributing evidence to that theme. Both the figure and the subsection structure of Section 4 have been aligned so that the reader can navigate from RQ, to theme, to specific studies without ambiguity. Each subsection highlights significant

contributions, challenges, and identified gaps; cross-study quantitative observations are summarized in Table 6 and Figure 2.

4.1. RQ1—Formal Modeling and Verification Approaches in Microservice-Based Systems

Analytical lens. This subsection examines how the primary studies under RQ1 cope with state-space scaling, specification cost, and proximity to running code when applying formal techniques to microservice-based systems. Verification in microservice-based systems depends heavily on the availability of formal models and analysis tools that can cope with high concurrency, decentralization, and frequent change. The selected studies group into three themes that together correspond to the RQ1 branch of Figure 3: actor-based model checking, control-plane verification, and static analysis as a verification aid.

4.1.1. Actor-Based Model Checking

The actor-based modeling language Rebeca structures systems as independent actors that communicate through asynchronous messages, which closely resembles how microservices interact through decoupled services and continuous message passing [3]. Khamespanah and Jaghoori illustrate this with an ordering-system example whose customer, shipping, order-processing, and inventory actors coordinate via message exchange [3]; this interaction pattern aligns with the communication patterns seen in distributed microservice architectures.

Across two decades of evolution, the Rebeca toolchain has demonstrated potential as a scalable model-checking framework for systems with large numbers of concurrent actors, comparable in scale to microservice deployments where concurrency and distributed communication lead to state-space growth [3]. Rebeca's dedicated model-checking engine, Modere, avoided nested-DFS recursion to prevent stack exhaustion and reduce memory usage during verification, and was followed by Afra, the current model checker, which enhances modular verification and supports more complex communication and timing behaviors. Industrial examples reported by the authors demonstrate how Rebeca models can interact with messaging systems such as Apache Kafka to approximate high-throughput, distributed event-driven architectures. Although the paper does not directly model Kubernetes, design similarities between Kafka-based distributed systems and orchestrated microservice deployments suggest the technique is applicable; the main residual concern is state-space scaling for very large deployments.

4.1.2. Control-Plane Verification and Fault Tolerance

Namyar et al. [20] present ZENITH, a microservice-based controller designed by construction to avoid state inconsistencies between control and data planes—inconsistencies that traditionally result from bugs, failures, or race conditions and that current solutions try to mitigate via periodic reconciliation at the cost of availability, scalability, and tail latency. ZENITH introduces a two-layer architecture plus a tool. The first layer, ZENITH-core, is a microservice-based CDN controller that is model-checked for eventual consistency in TLA^+ [18]. The second layer, ZENITH-apps, is a framework for formally verifying CDN applications: applications are specified in PlusCal, verified against an Abstract Core, and—when composed with ZENITH-core—inheriting end-to-end correctness. To ease app development, a code generator produces Python 3 implementations from PlusCal specifications that preserve the correctness of the verified spec.

ZENITH reportedly involves an 8.6k-line TLA^+ specification, among the largest published, and introduces novel scaling techniques for model checking. It is positioned as a production-ready verified controller and demonstrates the feasibility of formally verifying complex microservice systems at scale. The main limitations are that ZENITH currently targets Python (while many production microservices are written in Java/Go)

and that it is specialized to OpenFlow-like control protocols. Beyond ZENITH, recent work on Anvil studies the liveness of controller clusters, identifying tasks that are delayed or stuck because of silent crashes—a class of failures that traditional testing rarely surfaces. The complementarity of ZENITH (safety, consistency) and Anvil (liveness) suggests a promising direction for verified microservice controllers more broadly.

4.1.3. Service Behavior, APIs, and Process-Flow Verification

Beyond component-level verification, several studies extend formal techniques to APIs and end-to-end process flows. Camilli [19] treats microservice process flows as stateful systems, transforming a domain-driven workflow specification (e.g., from Netflix Conductor) into a Time-Basic Petri net (TB-net) whose Time Computation Tree Logic (TCTL) properties can be checked at design time and monitored at runtime against the deployed orchestration. This is one of the few included studies that closes the loop between design-time formal verification and runtime conformance checking on the same model. Other studies emphasize verification of REST APIs as stateful interfaces, using event-based triggers to detect when state invariants are violated and to notify developers that endpoints may need review.

4.1.4. Static Code Analysis as a Verification Aid

Static code analysis plays an important supporting role for verification of microservice-based systems. Copei et al. [8] present SIARest, an IDE-integrated tool that surfaces underlying communication patterns between microservices—specifically the API calls and service-to-service interactions that are difficult to track in distributed architectures. SIARest flags incorrect or outdated API endpoints and detects possible violations of expected service contracts, allowing developers to resolve these problems before they accumulate into architectural inconsistencies. It also provides autocompletion based on extracted communication structures, helping ensure that implemented code aligns with intended microservice interactions. Early detection is valuable for verification because static analysis outputs effectively define the boundary conditions for later formal modeling, reducing the size and complexity of the models that need to be verified.

Cerny et al. [7] extend this idea with a four-phase software-architecture-reconstruction (SAR) pipeline. Phase one performs abstract-syntax-tree analysis of individual microservice codebases to extract call graphs, controller endpoints, data entities, and component types; phase two builds component call graphs from these facts; phase three merges per-service representations using the Wu–Palmer algorithm on overlapping entities; and phase four detects design smells, anti-patterns, security policy violations, and bottlenecks. The pipeline is implemented as an open-source tool that was evaluated by practitioners. Current limitations are language-specific: the tool targets Java with the Spring framework, and it cannot yet detect event-driven communication through queues such as Kafka or JMS.

4.1.5. General Verification Barriers and Challenges

The evolution of software has continuously exposed new verification challenges. Ferrara et al. [2] present a historical analysis of these challenges from the 1940s to the present, organized into phases by decade, and identify recurring patterns—inconsistency issues, scalability problems, and technological diversity—whose root causes persist into the microservice era. The paper also identifies emerging trends and challenges for practitioners, including AI-generated code and novel deployment approaches such as serverless and quantum. The historical lens helps practitioners and researchers anticipate microservice-specific challenges like language heterogeneity, independent deployment, and distributed runtime behavior, all of which remain hard regardless of the verification technique used.

Trade-Offs Across RQ1 Techniques

The four primary-study techniques surveyed under RQ1 trade off as follows. Model checking with explicit formal models (Namyar [20], Camilli [19]) offers the strongest semantic guarantees—including liveness and protocol correctness—but the highest specification cost, and current published work targets the control plane or workflow layer rather than arbitrary microservice code. Static-analysis-as-verification (Copei [8], Cerny 2024 [7]) has the lowest barrier to entry and the most direct connection to running code, and automates well in CI/CD, but cannot prove temporal or protocol properties. Actor-based model checking, as positioned by the background survey of Khamespanah and Jaghoori [3], offers concurrency-aware semantics but is not yet represented in our primary corpus by a microservice-specific case study. Across all four techniques, the empirical evidence is concentrated in small or medium case studies; only ZENITH (one primary study) provides industrial-scale verification evidence.

4.2. RQ2—Architectural/API Evolution and Impact on Verifiability

Analytical lens. This subsection examines how architectural and API change in the primary studies erodes verification artifacts, and which counter-measures the studies report. Microservice architectures are designed to evolve: services are added, split, merged, and deprecated, while APIs change to accommodate new features and client needs. This continuous evolution has direct consequences for verifiability: as architecture and APIs change, previously valid assumptions encoded in models, test suites, and architecture descriptions become outdated, producing drift between intended and implemented designs. Studies in the corpus address three complementary aspects of this problem—architectural drift and consistency, topology and API evolution, and runtime adaptation as mitigation—corresponding to the three sub-boxes of the RQ2 branch in Figure 3.

4.2.1. Architectural Drift and Consistency

Despite the rapid pace of microservice evolution, no single method is consistently used by teams for continuous evaluation of verifiability. Architecture artifacts created early in a project (UML diagrams, ADRs, C4 models) can guide development, but as microservices become more complex and change more rapidly the risk of architectural drift becomes acute. Drift directly impacts verifiability: the further the implementation moves from the intended design, the harder it is to validate system behavior or reason about correctness. Biswas et al. [6] make drift measurable through model-to-code comparisons: their tool evaluates UML/architectural artifacts against the actual implementation and reports inconsistencies, with the key empirical finding that continuous checking significantly reduces the accumulation of inconsistencies. This is consistent with the broader trends identified by Soares et al. [4], who observe that architectural erosion is common across evolving systems and that frequent updates create more opportunities for inconsistency.

Soares et al. [4] also classify the continuous-evaluation literature into five approach families—scenario-based, simulation-based, rule-based, metric-based, and mathematical modeling—of which only three (simulation, metric-based, rule-based) have substantial automation support, and only simulation and metric-based approaches are integrated into CI/CD pipelines. Automation of architecture evaluation is identified as a research trend, with distributed systems (including microservices and IoT) as the primary application domain.

4.2.2. Topology, API, and Dependency Evolution

For verification purposes, architectural evolution (typically intentional) and architectural drift (typically unintentional) introduce similar challenges. Cerny et al. [1] contrast

microservices with service-oriented architecture (SOA) and observe that SOA's centralized governance, while limiting scalability, can make verification easier; microservices' loose, autonomous-team development model speeds up delivery but makes integration and system-level verification considerably harder. This trade-off frames the verification scope for all subsequent RQ2 studies.

Although microservices are not required to use containers, most production deployments do. Burns et al. [22] chronicle the evolution from Borg and Omega to Kubernetes at Google and report a key lesson: the shift from machine-oriented to application-oriented management mirrors microservices' own decomposition into independently deployable services. The container-ecosystem challenges they identify—service discovery, namespaces, application-aware load balancing, rollout, monitoring, and alerting—are also the verification-relevant aspects of microservice systems, because changes in any of them can invalidate models or runtime expectations. Boza et al. [21] complement this with an empirical study showing significant performance variability across container deployments, which underlines that verification must consider non-functional properties (latency, throughput) and not only safety/correctness when the deployment topology evolves.

Merlino et al. [23] extend the discussion to edge–fog–cloud systems and propose workload-engineering patterns that determine where computation runs. From a verification standpoint, this adds another dimension of topology evolution: services may be replaced across tiers without changing their code, yet their behavior under partition or latency conditions changes.

Organizational and Process Evolution

Teams today operate with increasingly fast release cycles, which influences how architecture and APIs evolve through ongoing iterations. Shahin et al. [9] examine how organizations adopt Continuous Delivery and Continuous Deployment: about 40% of respondents reported a centralized team or shared group that supports CI/CD practices, and roughly 68% observed stronger collaboration after CI/CD adoption. The implication for RQ2 is that organizational evolution can partially offset the verification burden created by rapid architectural change, but only when communication channels match the release cadence; otherwise, bottlenecks appear when pipeline status is not communicated clearly. Responsibilities did not shift dramatically, but practitioners reported needing broader skill sets, suggesting that verification activities increasingly require T-shaped engineers.

4.2.3. Runtime Adaptation as Mitigation

A third mitigation strategy is to push verification into the runtime, so that tests and evaluation logic adapt as the architecture evolves. Silva et al. [13] systematize this idea under self-adaptive testing in the field, arguing that traditional in-house testing is insufficient when upgrades are released continuously. Their analysis identifies a typical runtime-testing loop architecture aligned with MAPE-K (Monitor–Analyze–Plan–Execute over Knowledge): monitoring senses relevant behavioral changes, analysis decides what kind of adaptation is needed, and planning/execution carry out test-suite updates. In a microservice context, this enables localized regressions to be detected even when the rest of the system remains stable, which is the most common observed pattern of failure after partial deployments.

Trade-Offs Across RQ2 Mitigations

The three RQ2 mitigations differ in cost and timing. Architectural drift and consistency checking (Biswas [6]) operates at build time and at the model layer, so it catches divergence early but presumes that a usable architecture model exists. Topology and API evolution mitigations (Boza [21], Merlino [23], Burns [22]) operate at the platform layer and surface

placement and orchestration effects that the model layer cannot, but they require running deployments. Runtime adaptation (Silva [13]) shifts the burden to operations and reacts to actual failures, which is cheapest to start but most reactive. The three are complementary rather than substitutable; no primary study in the corpus combines all three.

4.3. RQ3—Repository-Derived Signals for Updating Models

Analytical lens. This subsection examines how the primary studies in RQ3 connect repository or runtime artifacts to formal or semi-formal models, and how often they do so automatically. Repository-derived signals offer a window into the as-built architecture: by extracting facts from source code, commits, and dependency manifests, formal artifacts can be refreshed in step with the implementation. The included studies populate the two RQ3 sub-boxes of Figure 3: architecture extraction from code and repository and API evolution signals.

4.3.1. Architecture Extraction from Code

Rather than relying on design documents that become outdated quickly, several studies extract architecture from source code directly. Cerny et al. [1,7] show that static analysis can reveal service boundaries, expose API endpoints, and capture inter-service communication patterns. The resulting as-built view often surfaces smells and anti-patterns—early warning signs of architectural erosion—and is straightforward to embed in a CI pipeline that re-extracts the architecture on every commit [6]. The accuracy of these extractions depends on heuristics: service boundaries rely on framework annotations, container descriptors, and build modules, while interface contracts come from OpenAPI or protobuf/IDL definitions [8]. Lifting these low-level facts into visual models such as C4 diagrams or service-dependency graphs makes complex architectures legible and enables early detection of design violations [7].

4.3.2. Tracking Evolution via Commits and Dependency Changes

Beyond a single-point extraction, repositories provide a longitudinal record. Mining commit histories and pull-request metadata reveals architectural changes—service splits or merges, interface updates, and shifting dependencies [4,6]. Harnessing these records lets architectural models grow with the codebase, shrinking the gap between intended and actual architecture; by combining commit mining with static analysis, teams can selectively update only the changed services. Dependency manifests (Maven POM, npm package.json, Dockerfiles, Helm charts) carry equally important signals: upgrading a library, adding a sidecar container, or introducing a service mesh subtly reshapes topology and non-functional qualities such as latency and resilience. Tracking this churn [4,6] helps teams identify “risk windows” where architecture consistency is most likely to be threatened. Embedding architectural rules as automated CI checks then turns architectural governance from a reactive chore into a proactive, continuous activity.

4.3.3. Opportunities for Formal Model Generation

A research opportunity, only partially exploited in the corpus, is to use repository-derived signals to (re-)generate or refresh formal models—not only architecture diagrams—on every commit. Biswas et al. [6] and Soares et al. [4] sketch how techniques such as graph-based learning and incremental verification could automate generation and refresh of formal models, enabling continuous verification where safety, availability, or fault-tolerance properties are checked as the system evolves. The actor-model checking tradition surveyed by Khamespanah and Jaghoori [3] contributes proven ingredients (modularity, compositional reasoning, domain-specific abstractions) for scaling such workflows. Recent LLM-assisted program analysis [14–16] adds a complementary capability: LLMs can ingest

commit diffs and produce candidate specifications or invariants that are then validated by the formal backends discussed in RQ1 (this latter point is forward-looking context; see Section 2.4). This hybrid path is, in our view, the most promising route to closing the gap identified by RQ3, but it is supported by zero primary studies in the included corpus and we report it as a research direction rather than a current finding.

Trade-Offs Across RQ3 Signals

Static-analysis extraction (Cerny 2024 [7], Copei [8]) is fully automated and produces snapshots of the architecture at build time, but does not yield temporal or protocol models. Commit/dependency mining requires more bespoke pipelines and is sparsely represented in the primary corpus; the bridging step from extracted facts to formal models is, in fact, the principal gap surfaced by RQ3.

4.4. RQ4—Continuous/Incremental Verification in CI/CD

Analytical lens. This subsection examines how the primary studies in RQ4 integrate verification activities into delivery pipelines, and how end-to-end (design-time through runtime) the integration actually goes. Traditional verification was often performed at discrete milestones, but microservice-based systems and DevOps practices demand verification that is continuous, incremental, and tightly integrated into delivery pipelines. The corpus supplies evidence under the two RQ4 themes of Figure 3: CI/CD-based verification and continuous certification, and runtime/field-based testing and security.

4.4.1. Continuous Certification in DevOps

Anisetti et al. [11] introduce a continuous-certification methodology developed in the EU H2020 Toreador project; the methodology is directly applicable to microservices, since most teams building such systems also adopt DevOps tools and processes. Implementation begins by modeling the development process, then embedding hooks throughout that process to continuously collect data for inspection; the collected data drives evaluations at each stage. The main contribution is to identify the requirements for continuous certification in DevOps environments: a model of the development process, an evaluation specification per stage, trustworthy inspection mechanisms with named owners, and embedded verification activities. The same group later extends the idea to security [12] with a test-based incremental security-certification scheme that updates certificates as evidence accumulates, so that certification itself evolves with the system.

4.4.2. Runtime/Field-Based Continuous Testing

Silva et al. [13] systematize implementations of self-adaptive testing as part of runtime or field-based verification. They consolidate a typical runtime-testing loop architecture across many studies, aligned with the MAPE-K (Monitor–Analyze–Plan–Execute over Knowledge) feedback loop. Within this structure, monitoring senses environmental or behavioral changes; analysis decides whether new test cases should be generated or existing ones adjusted; planning and execution carry out the change in a way that remains transparent and documented, so engineers can later check whether decisions remain aligned with organizational goals. The authors emphasize that runtime adaptations should occur as small, incremental changes, which is consistent with CI/CD orientation. They also describe seven subsystems characterizing how runtime self-adaptation occurs (the object being adapted, triggers, decision mechanisms, types of adaptation, etc.). Within a CI/CD context, this approach effectively extends verification beyond the build pipeline into the operational environment.

4.4.3. Integrating Architecture Validation into CI/CD

Biswas et al. [6] argue for integrating architecture validation directly into CI/CD rather than treating it as a one-time activity. Their CAMV (Continuous Architecture Model Validator) tool chain integrates simulation-based evaluation into a CI/CD pipeline; it comprises a model converter, the FERAL simulator, a checker, executable-model creation procedures, and a bridge between them. While the validation domain in their original study is automotive, the underlying idea of incremental model checking on every change is portable to microservices and aligns with broader continuous-evaluation trends [4] that emphasize triggering validation automatically on each commit and maintaining architectural models as evolving, incremental artifacts.

4.4.4. End-to-End Formal Verification Pipelines

Camilli [19] presents a comprehensive formal-verification framework that spans the design and operation phases of a microservice workflow. The framework has three steps. Model-to-model transformation translates a workflow described in a domain-driven language (for example, the orchestration model used by Netflix Conductor [24]) into a Time-Basic Petri net. Design-time verification uses this model for model checking by constructing a Time Reachability Graph (TRG) and checking it against requirements expressed in Time Computation Tree Logic (TCTL). Runtime verification reuses the generated models to monitor the deployed application against the same formal specifications. The implementation instruments Conductor's code to intercept key events; the system produces an on-the-fly report on functional and temporal conformance failures, providing actionable insight to developers and operations teams. The work is one of the few in the corpus that closes the loop between design-time and runtime verification on a shared formal model, and it provides a reference architecture for systems built on orchestrators other than Conductor.

Trade-Offs Across RQ4 Pipelines

The pipelines surveyed under RQ4 trade off completeness against generality. Continuous certification (Anisetti 2020 [11], Anisetti 2018 [12]) is the most general but does not commit to a specific formal back-end; it leaves the choice of verification technique open. End-to-end formal pipelines (Camilli [19]) commit to a specific formalism and therefore deliver the strongest guarantees, but at the cost of being tied to a particular orchestration framework (Conductor). Runtime/field-based testing (Silva [13]) is the cheapest to integrate but offers the weakest guarantees. Architecture validation in CI/CD (Biswas [6]) sits between these, offering automated drift detection but not protocol-level proofs. Only Camilli closes the loop from design-time formal model to runtime monitor on a shared specification; this is the single primary study satisfying the strongest formulation of RQ4 in the corpus.

5. Discussion

Change, decentralization, and heterogeneity are the defining characteristics of microservice-based systems, and they shape both development and verification. Development teams have adopted a deep stack of tools, frameworks, and processes (CI/CD, container orchestration, service meshes) to manage these characteristics, but the verification community has been slower to follow. Three cross-cutting findings emerge from the synthesis in Section 4. Each finding is anchored in a specific thematic cluster of Figure 3 and the primary studies populating that cluster; the mapping is given in Table 8. Each finding is also annotated with an evidence strength: high where three or more independent primary studies converge; medium where two studies converge or where the claim is a

composition argument across multiple complementary studies; low where it is supported by only one primary study or only by background literature.

Finding 1—Evolution dominates the verification challenge (evidence strength: high).

Across all four RQs, the included primary studies converge on change—whether intentional (architectural evolution) or unintentional (drift)—as the principal source of verification difficulty. Direct evidence comes from four primary studies that all identify divergence between intended and implemented designs as the dominant source of verification debt: Biswas et al. [6] (model-to-code consistency), Copei et al. [8] (static analysis surfacing API mismatches), Cerny et al. (2024) [7] (architecture recovery), and Shahin et al. [9] (organizational/process drift). The background survey by Soares et al. [4] corroborates the pattern at the architectural-quality level. Verification artifacts therefore cannot be designed once and left alone; they must themselves be artifacts that evolve.

Finding 2—An extensible methodology requires modeling the development process first (evidence strength: medium).

This is a composition argument rather than a single-study claim. The continuous-certification methodology of Anisetti et al. (2020) [11] explicitly identifies process modeling as the first step; Camilli's [19] pipeline reuses the same model at design time and at runtime; Biswas et al. [6], Cerny et al. (2024) [7], and Silva et al. [13] each contribute one or more pieces of the pipeline (modeling, hooks, evidence collection, evaluation, runtime monitor). No single primary study presents an end-to-end methodology; the composition is our synthesis claim. Teams whose development frameworks make formal model generation feasible (e.g., Netflix Conductor for workflow specifications [19,24]) can adopt an intrinsic strategy in which models are produced at build time; teams that cannot can adopt an extrinsic strategy in which models are recovered from source and runtime artifacts [7,8].

Finding 3—Runtime verification must include SLA-level checks (evidence strength: medium).

Three primary studies provide evidence that runtime metrics carry information the build pipeline cannot supply: Boza et al. [21] document significant cross-deployment performance variability, Merlino et al. [23] show that workload-placement decisions affect non-functional properties in Edge-Fog-Cloud deployments, and Silva et al. [13] report a self-adaptive testing framework that uses runtime metrics as triggers. Verification that stops at the build step therefore necessarily misses SLA-class properties. We note that this is a three-study finding and the supporting primary studies span different system layers; the evidence converges on the existence of the gap but does not yet identify a single SLA-aware verification methodology.

Technique trade-off matrix.

Table 10 summarizes the trade-offs across the five technique families that recur in the corpus, along three axes: specification cost, scalability, and automation potential. The matrix is intended as quick guidance for practitioners and as a structural summary of the cross-RQ trade-off paragraphs in Section 4. We deliberately express these axes qualitatively (e.g., “high specification cost”, “moderate scalability”) rather than as performance numbers, because, as the evidence profile in Table 7 shows, the primary studies do not report the comparable execution-time, scalability-bound, accuracy, or cost figures that a quantitative ranking would require. A numeric ranking inferred from non-comparable single-system measurements would convey false precision; the qualitative matrix reflects the level of confidence the underlying evidence actually supports.

Table 10. Trade-off matrix across the five recurring technique families in the primary corpus.

Technique Family	Spec. Cost	Scalability	Automation	Representative Primary Studies
Model checking (TLA ⁺ , TB nets)	High	Moderate	Low–Medium	Namyar [20], Camilli [19]
Static analysis/architecture recovery	Low	High	High	Copei [8], Cerny 2024 [7], Biswas [6]
Continuous certification	Medium	Medium	Medium	Anisetti 2020 [11], Anisetti 2018 [12]
Runtime/self-adaptive testing	Low–Medium	Medium	Medium	Silva [13], Boza [21], Merlino [23]
Pipeline/DevOps empirical	Low	High (qualitative)	N/A (study)	Shahin [9], Stillwell [10], Burns [22]

LLMs as a complement, not a replacement (evidence strength: low).

Looking forward, the trio of recent works on LLM-assisted program analysis [14–16] suggests a hybrid path that pairs LLM-drafted specifications and code transformations with sound formal backends. We caution that the LLM step requires careful validation: as Wang et al. [15] emphasize in their survey, LLM outputs should be checked against the existing verification machinery (the model checkers, type systems, and runtime monitors covered by RQ1 and RQ4), not trusted blindly. In the microservice context, this hybrid path is especially relevant to RQ3, where commit diffs and API specifications are exactly the inputs LLMs handle well. This is a forward-looking framing supported by zero primary studies in the included corpus (Section 2.4); we report it in Section 7 as a research direction, not as a current finding.

Two complementary research paths.

Our review identifies two complementary verification strategies. Intrinsic verification starts with formal models and propagates them through code generation, testing, and runtime monitoring (e.g., ZENITH [20], Camilli’s TB-net pipeline [19]). Extrinsic verification starts with deployed code and recovers models from it (e.g., Cerny et al. [7], Copei et al. [8], Biswas et al. [6]). The two strategies are not in conflict: intrinsic offers higher fidelity but demands cooperation across the SDLC; extrinsic offers retrofit applicability but trades model accuracy for coverage. Future work should focus on (i) mainstream microservice frameworks that make intrinsic adoption cheap and (ii) repository- and runtime-driven model reconstruction that makes extrinsic adoption accurate.

5.1. Practitioner Guidance: When to Choose Intrinsic vs. Extrinsic Verification

Drawing on the conditions that distinguish the included primary studies, we propose a short decision checklist for practitioners choosing between the two strategies (Table 11). The checklist is intended as quick guidance rather than a substitute for the full evidence in Section 4; hybrid adoption (parts of the system intrinsic, parts extrinsic) is the appropriate response when both columns apply to different subsystems.

5.2. What Practitioners Can Do Now Versus What Remains a Research Agenda

We close the Discussion by separating, honestly, what the present synthesis enables practitioners to do now from what remains a research agenda. What practitioners can do now, based on evidence in the corpus, includes: (a) choose between intrinsic and extrinsic strategies per Table 11; (b) build artifact-traceability links between commits and any existing architecture or formal models, using the static-analysis pipelines surveyed under RQ3 (Copei [8], Cerny 2024 [7]); and (c) integrate sound static-analysis verification into existing CI/CD pipelines, following the patterns surveyed under RQ4 (Anisetti 2020 [11], Biswas [6]). What remains a research agenda includes: (d) end-to-end formal verification pipelines that close the design-time to runtime loop on a shared specification (only Camilli [19] does this today); (e) shared benchmarks of evolving microservice systems for cross-study comparison; (f) scalable industrial verification of microservice deployments beyond the

three industrial-scale primary studies in the corpus; and (g) LLM-assisted hybrid pipelines for repository-aware specification refresh (Section 2.4). These four research items map onto the four pillars of the research agenda in Section 7.

Table 11. Practitioner checklist for choosing intrinsic vs. extrinsic verification, derived from the conditions that distinguish the included primary studies.

Choose Intrinsic (Build-Time, Model-Driven) When...	Choose Extrinsic (Recovery-Based) When...
the team controls a workflow or process framework that admits formal specification (Netflix Conductor, BPMN engines, TLA ⁺ -friendly controllers)	the team inherits a system without specifications
an authoritative interface source of truth (OpenAPI, protobuf, gRPC) exists and is treated as the single source of truth	services are polyglot and no single framework imposes a model
the cost of a regression at runtime is high (financial, safety-critical, regulated)	commits, deployment manifests, and runtime telemetry are abundant and reliable
the team has formal-methods expertise in-house	the immediate goal is to detect architectural drift rather than to prove protocol correctness

6. Threats to Validity and Gaps in the Reviewed Literature

We discuss threats to validity along Wohlin’s four dimensions, and then summarize the methodological gaps observed in the reviewed literature itself.

6.1. The Small Primary-Study Count Is Itself a Finding

The intersection of microservice-based systems, evolution, and formal/rigorous verification is sparsely covered in the indexed literature; the 13 primary studies surviving our screening should be read as direct empirical support for the existence of the gap motivating the review, not as a sampling artifact. A broader corpus would have required relaxing the inclusion criteria, which would have changed the research question. For comparison, recent SLRs in adjacent software-engineering subdomains report similar sizes (e.g., the serverless QoS survey of Mampage et al. [5] and the continuous-architecture-evaluation survey of Soares et al. [4] report comparable corpora).

6.2. Construct Validity

The four RQs operationalize the concepts of evolution and formal verification, but each concept admits broader interpretations. “Formal” could include lightweight types and contracts that we did not require; “evolution” could include configuration drift that we did not separately track. We chose to keep the constructs strict to make the synthesis cleaner; a broader scoping would likely add studies but at the cost of dilution.

6.3. Internal Validity

Search bias. Our final search used four databases (IEEE Xplore, ACM Digital Library, SpringerLink, Scopus); we expanded from two to four in response to reviewer feedback. We did not query Google Scholar because its lack of stable Boolean operators undermines reproducibility. Pilot Google Scholar checks suggest that the marginal additional yield is small and dominated by gray literature. *Selection bias.* Screening required subjective judgment, particularly for studies where “microservice” was mentioned but not central; we mitigated this with independent screening by three authors and adjudication by a senior author, but residual subjectivity remains. *QA subjectivity.* The 15-item QA instrument involves judgment on items like “clearly reproducible”. A sensitivity check (uniform weights, doubled methodology weights, doubled venue weights) changed the included set

by at most one study, indicating reasonable robustness; the threshold of 0.5 was chosen a priori from the protocol, not tuned after seeing scores.

6.4. External Validity

Many of the 13 primary studies validate on small academic case studies (1–3 services). Only three (ZENITH [20], Shahin et al. [9], Burns et al. [22]) reach industrial or production scale. As a result, our conclusions about the maturity of microservice verification techniques may understate their effectiveness in smaller settings and overstate their readiness for very large polyglot deployments. We further acknowledge that, with a small and heterogeneous corpus, individual findings may hold only for the specific systems, domains, and contexts studied rather than generalizing across the field; this is an inherent constraint of reviewing an emerging area, and it bounds the strength of every cross-study claim we make. We address this in two ways. First, we do not state any finding more strongly than its evidence base allows: the evidence-strength markers in Section 5 explicitly downgrade claims supported by one or two studies, and we have avoided field-wide generalizations where the corpus does not warrant them. Second, the limitation is itself a motivating result—the scarcity of broadly validated techniques is precisely the gap the research agenda (Section 7) targets through industrial-scale validation and shared benchmarks. We therefore present this review as an honest map of an emerging area and the gaps within it, rather than as a source of broadly generalizable performance conclusions.

6.5. Conclusion Validity

We minimized observer bias by using a structured extraction form with closed categorical fields wherever possible (verification technique, evolution dimension, automation level, evidence size, artifact availability). The cross-study comparison in Table 6 and Figure 2 is derived from these fields, not from free-text impressions.

6.6. Notes on Protocol Adjustments

Two adjustments are documented for transparency: (i) the publication window was extended from 2010–2025 (as initially announced) to 2009–2025 to include Lamport’s PlusCal/TLA⁺ paper [18], applied uniformly across all databases; (ii) the inclusion/exclusion criterion E3 excludes secondary reviews whose primary topic is microservice verification, but background surveys on adjacent topics (e.g., software verification overview [2], continuous architecture evaluation [4], serverless taxonomies [5], the actor-model checking survey [3]) are retained as background and are clearly labeled in Table 6. Earlier drafts of the manuscript were inconsistent on these points; they are reported clearly here.

6.7. Methodological Gaps in the Reviewed Literature

Independently of threats to our own review, the body of work itself exhibits four recurring gaps that limit aggregation:

1. Small-scale validation. Overall, 10 of the 13 primary studies report only small/medium case studies; only three report large/industrial validation (Namyar, Shahin, Burns). This makes scalability claims for formal techniques largely projective.
2. Sparse artifact sharing. Only five of the 13 primary studies provide publicly available tools or datasets, which limits replication and cross-study benchmarking.
3. Lack of shared benchmarks. Each study uses its own subject system (e.g., an ordering example, an automotive system, a CDN). No widely used benchmark of evolving microservices exists.

4. Limited end-to-end pipelines. Only Camilli's [19] approach closes the loop from design-time formal model to runtime monitor on a shared specification within a microservice context; ZENITH [20] provides the strongest scale evidence but is specialized to a control plane. Most others address either design-time or runtime, not both.

The four gaps stated above directly motivate the four-pillar research agenda in Section 7.

7. Conclusions

This systematic literature review investigated the intersection of microservice evolution and formal verification, guided by four research questions and a corpus of 18 included studies (13 primary, five background). Our synthesis yields four insights, one per research question:

- RQ1 (Formal Methods in Microservices). A variety of formal techniques have been applied: Petri nets for workflow modeling, TLA⁺ for protocol correctness, SMT for configuration checking, static analysis for architecture recovery, and test-based certification for security. However, most evaluations remain confined to small-scale case studies (10 of 13 primary studies), with limited evidence of scalability to industrial systems.
- RQ2 (Impact of Evolution). Architectural drift, API versioning, and dependency change pose significant challenges to verifiability. Few approaches directly address automated detection of specification drift or mitigation of protocol violations at scale; this gap underscores the need for techniques that allow formal models to evolve in step with the systems they describe.
- RQ3 (Signals for Model Updates). Repository-derived signals such as commits, API diffs, and dependency graphs are beginning to be leveraged to infer or update architecture and formal models. Automation remains immature, and integration with formal-method tool chains is limited; this is the principal gap our review surfaces.
- RQ4 (Continuous Verification in CI/CD). There is growing interest in embedding verification into DevOps pipelines, but tool support is fragmented and industrial adoption evidence is scarce. Only Camilli's pipeline [19] closes the loop from design-time formal model to runtime monitor on a shared specification.

Alongside these findings, our review identifies two complementary approaches to formal verification in microservice-based systems: intrinsic and extrinsic. The intrinsic approach integrates formal methods from the outset: models are generated, maintained, and evaluated throughout the development life-cycle. This requires strong support from frameworks, tools, and organizational processes, but offers higher fidelity and consistency; its drawback is the high initial investment and the cultural shift required. The extrinsic approach focuses on recovering models from code repositories, logs, and runtime artifacts after the system has been built; it requires less upfront support but may suffer from model accuracy issues and incomplete coverage of evolving behaviours. Depending on the situation, teams may adopt one approach or combine aspects of both, per the checklist in Table 11.

Research Agenda

The body of work surveyed here is emerging but fragmented. Formal verification for microservice evolution is still in its infancy, with limited industrial-scale validation and little shared tooling or open datasets. We propose a four-pillar research agenda; each pillar is annotated with the thematic cluster and the primary studies that motivate it.

1. Scalable, CI/CD-integrated frameworks. Move beyond small-scale prototypes to verification tools that operate natively within industrial pipelines. Motivated by the RQ4 cluster (Anisetti 2020 [11], Biswas [6], Stillwell [10]) and Finding 2 in Section 5.
2. Automated model inference. Use Mining Software Repository (MSR) techniques, and, looking forward, LLM-assisted analyses cross-checked by sound verifiers [14–16], to extract and refresh formal models directly from codebases. Motivated by the RQ3 cluster (Cerny 2024 [7], Copei [8], Biswas [6]) and by the LLM forward-looking framing in Section 2.4.
3. Standardized benchmarking. Establish shared, representative microservice datasets and benchmark systems to facilitate reproducibility and objective cross-study comparison. Motivated by the sparse-artifact-sharing and no-shared-benchmark gaps in Section 6.
4. Combined intrinsic/extrinsic adoption. Tailor verification strategies to organizational maturity, resource availability, and system complexity, with explicit guidance (Table 11) on when to prefer build-time models, when to recover them from artifacts, and how to combine the two. Motivated by the intrinsic/extrinsic distinction synthesized across the corpus (Section 5).

If these four pillars are addressed, future research can move toward more robust, scalable, and empirically validated approaches to ensuring correctness, safety, and reliability in evolving microservice-based systems.

Author Contributions: Conceptualization, R.G. and E.E.; methodology, R.G. and E.E.; validation, R.G., E.R.Z., S.G.D. and T.C.; formal analysis, E.E., R.G., E.R.Z. and T.C.; investigation, R.G., E.E., E.R.Z. and S.G.D.; data curation, E.E.; writing—original draft preparation, R.G., E.E., and E.R.Z.; writing—review and editing, R.G., E.E., E.R.Z., S.G.D. and T.C.; visualization, R.G., E.E., and E.R.Z.; supervision, T.C.; project administration, E.E., E.R.Z. and T.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created in this study. All data supporting the reported results are drawn from the primary studies cited in the References and are summarized in the tables of this article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual Understanding of Microservice Architecture. *ACM SIGAPP Appl. Comput. Rev.* **2018**, *17*, 29–45. [\[CrossRef\]](#)
2. Ferrara, P.; Arceri, V.; Cortesi, A. Challenges of Software Verification. *Int. J. Softw. Tools Technol. Transf.* **2024**, *26*, 421–430. [\[CrossRef\]](#)
3. Khamespanah, E.; Jaghoori, M.M. 20 Years of Actor Model Checking with Rebeca. In *Rebeca for Actor Analysis in Action*; LNCS; Springer: Cham, Switzerland, 2025; Volume 15560, pp. 26–43. [\[CrossRef\]](#)
4. Soares, R.C.; Capilla, R.; dos Santos, V.; Nakagawa, E.Y. Trends in Continuous Evaluation of Software Architecture. *Computing* **2023**, *105*, 1957–1980. [\[CrossRef\]](#)
5. Mampage, A.; Karunasekera, S.; Buyya, R. Resource Management in Serverless Computing. *ACM Comput. Surv.* **2022**, *54*, 1–36. [\[CrossRef\]](#)
6. Biswas, P.; Morgenstern, A.; Antonino, P.O.; Capilla, R.; Nakagawa, E.Y. Continuous Evaluation of Consistency in Software Architecture Models. In *Proceedings of the ECSA 2023 Proceedings*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 141–149. [\[CrossRef\]](#)
7. Cerny, T.; Abdelfattah, A.S.; Yero, J.; Taibi, D. From Static Code Analysis to Visual Models of Microservice Architecture. *Clust. Comput.* **2024**, *27*, 4145–4170. [\[CrossRef\]](#)

8. Copei, S.; Schreiter, M.; Zündorf, A. Improving the Implementation of Microservice-Based Systems with Static Code Analysis. In *Agile Processes in Software Engineering and Extreme Programming—Workshops*; Lecture Notes in Business Information Processing; Springer: Cham, Switzerland, 2023; Volume 489, pp. 31–38. [\[CrossRef\]](#)
9. Shahin, M.; Zahedi, M.; Babar, M.A.; Zhu, L. Adopting Continuous Delivery and Deployment. In *Proceedings of the EASE '17 Proceedings*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 384–393. [\[CrossRef\]](#)
10. Stillwell, M.; Coutinho, J.G. A DevOps Approach to Integration of Software Components. In *Proceedings of the QUDOS 2015 Proceedings*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1–6. [\[CrossRef\]](#)
11. Anisetti, M.; Ardagna, C.A.; Gaudenzi, F.; Damiani, E. A Continuous Certification Methodology for DevOps. In *Proceedings of the MEDES '19: Proceedings of the 11th International Conference on Management of Digital EcoSystems*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 205–212. [\[CrossRef\]](#)
12. Anisetti, M.; Ardagna, C.; Damiani, E.; Polegri, G. Test-Based Security Certification of Composite Services. *ACM Trans. Web* **2018**, *13*, 1–43. [\[CrossRef\]](#)
13. Silva, S.; Pelliccione, P.; Bertolino, A. Self-Adaptive Testing in the Field. *ACM Trans. Auton. Adapt. Syst.* **2024**, *19*, 4. [\[CrossRef\]](#)
14. Li, H.; Hao, Y.; Zhai, Y.; Qian, Z. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. In *Proceedings of the ACM on Programming Languages 2024, 8(OOPSLA1)*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 474–499.
15. Wang, J.; Ni, T.; Lee, W.B.; Zhao, Q. A Contemporary Survey of Large Language Model Assisted Program Analysis. *arXiv* **2025**, arXiv:2502.18474.
16. Cai, X.; Liu, J.; Huang, X.; Yu, Y.; Wu, H.; Li, C.; Wang, B.; Yusuf, I.N.B.; Jiang, L. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*; Springer Nature: Cham, Switzerland, 2025; pp. 283–302.
17. Kitchenham, B.; Brereton, O.P.; Budgen, D.; Turner, M.; Bailey, J.; Linkman, S. Systematic literature reviews in software engineering—A systematic literature review. *Inf. Softw. Technol.* **2009**, *51*, 7–15. [\[CrossRef\]](#)
18. Lamport, L. The PlusCal Algorithm Language. In *Proceedings of the ICALP 2009 Proceedings*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 36–60. [\[CrossRef\]](#)
19. Camilli, M. Continuous Formal Verification of Microservice-Based Process Flows. In *Proceedings of the ECSA 2020 Proceedings*; Springer: Cham, Switzerland, 2020; pp. 420–435. [\[CrossRef\]](#)
20. Namyar, P.; Ghavidel, A.; Zhang, M.; Madhyastha, H.V.; Ravi, S.; Wang, C.; Govindan, R. ZENITH: Formally Verified Highly Available Control Plane. In *Proceedings of the SIGCOMM '25 Proceedings*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 409–433. [\[CrossRef\]](#)
21. Boza, E.F.; Abad, C.L.; Narayanan, S.P.; Balasubramanian, B.; Jang, M. A Case for Performance-Aware Deployment of Containers. In *Proceedings of the WOC '19 Proceedings*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 25–30. [\[CrossRef\]](#)
22. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, Omega, and Kubernetes. *Commun. ACM* **2016**, *59*, 50–57. [\[CrossRef\]](#)
23. Merlino, G.; Dautov, R.; Distefano, S.; Bruneo, D. Workload Engineering in Edge–Fog–Cloud Computing. *ACM Trans. Internet Technol.* **2019**, *19*, 1–22. [\[CrossRef\]](#) [\[PubMed\]](#)
24. Netflix Conductor Documentation. Available online: <https://github.com/Netflix/conductor> (accessed on 12 December 2025).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.