Review

Not peer-reviewed version

Analyzing Evolution of Microservice-Based Systems: Vision—An Extensible Methodology for Formal Software Verification in Microservice Systems

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

Posted Date: 25 March 2026

doi: 10.20944/preprints202603.1951.v1

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

 doi:10.20944/preprints202603.1951.v1

Review

Analyzing Evolution of Microservice-Based Systems: Vision—An Extensible Methodology for Formal Software Verification in Microservice Systems

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

Department of Electrical and Computer Engineering, College of Engineering, The University of Arizona, Tucson, AZ, USA
* Correspondence: ubengomez@arizona.edu (R.G.); ebeid@arizona.edu (E.E.); ezarate@arizona.edu (E.Z.); gawardak@arizona.edu (S.G.D.); tcerny@arizona.edu (T.C.)

Abstract

Microservice-based architecture has become a dominant style for building cloud-native systems, but their rapid and continuous evolution makes it difficult to understand, verify, and control the impact of change. Although many works address microservices from the perspectives of design, deployment, and operation, there is still limited consolidated knowledge on how the evolution and verification of microservice-based systems are supported in the research literature. This paper reports on a systematic literature review (SLR) on methods and tools for analyzing evolution and providing verification support in microservice-based systems. Following established SLR guidelines, we defined a review protocol with focused research questions, executed structured searches in major digital libraries, and applied explicit inclusion and exclusion criteria tailored to cloud-native microservices. Candidate studies were screened through multiple stages, supported by an academic queries table, a statistics table summarizing the selection process, and a quality assessment checklist. For each included primary study, we extracted data on the type of evolution or change considered, the analysis or verification technique used, the form of tool support, and the nature of empirical evaluation. The review highlights a small but emerging body of work that combines microservice evolution with formal verification, runtime monitoring, and model-based analysis. However, empirical evidence is often limited to small case studies, with few industrial-scale evaluations and little shared tooling or datasets. We conclude by identifying gaps and outlining a research agenda for more robust, empirically grounded approaches to microservice evolution and verification.

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. In microservice architecture, an application is decomposed into a set of small, independently deployable services that communicate over lightweight protocols and are often deployed in containers or managed platforms such as Kubernetes. This style promises benefits in terms of scalability, deployability, and team autonomy, but it also amplifies the challenges of reasoning 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 can make it difficult to ensure that critical 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 software verification techniques, such as model verification, SMT-based analysis, Petri nets, and session types, offer a complementary way to reason for correctness and safety properties.

 doi:10.20944/preprints202603.1951.v1

However, applying such techniques in the context of microservices introduces several challenges. Formal models must deal with asynchronous communication, dynamic discovery, and partial failures, requiring updates as the system evolves, and integrate with existing tool-chains and continuous integration / continuous delivery (CI/CD) pipelines in a way that is practical for engineers. While there is growing interest in verification for cloud-native systems, there is still limited consolidated knowledge about how evolution and verification of microservice-based systems are supported in the research literature. This paper reports on a systematic literature review (SLR) that investigates the intersection of microservice evolution and formal or rigorous analysis methods. Following established SLR guidelines in software engineering, we design and execute a review protocol to identify primary studies that (i) explicitly target microservice-based systems in a cloud-native context, and (ii) provide methods, tools, or empirical evidence related to evolution, change, or verification. The review protocol includes a set of research questions, search queries over major digital libraries, inclusion and exclusion criteria, a multi-stage screening process, and a quality assessment procedure. The goals of this review are to (i) characterize the types of evolution and change that are studied in microservice-based systems, (ii) catalog the formal or rigorous techniques used to analyze or verify these systems, and (iii) assess the available empirical evidence and tool support. Ultimately, the review aims to identify gaps in current research and outline a research agenda for more robust and empirically grounded approaches to evolution and verification in microservice-based architectures.

2. Background

2.1. Microservice-based Systems and Their Evolution

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 can make it difficult to ensure that critical 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.

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. 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 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.

 doi:10.20944/preprints202603.1951.v1

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. Shahin et al. 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. Still well and Coutinho 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. propose a continuous certification methodology that integrates evidence collection, test-based assessment, and re-certification into DevOps workflows. In related work, they use 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. 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. 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 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.

3. Methodology

3.1. Review Protocol and Research Questions

This study follows established guidelines for systematic literature reviews in software engineering, with the goal of making the review process transparent and reproducible. The methodology was defined in a review protocol that specifies the research questions, search strategy, study selection criteria, quality assessment, data extraction, and synthesis procedures. The protocol was developed during the feasibility/scoping phase of the project and refined iteratively as our understanding of the topic matured. In that phase, we initially explored literature from credible sources, such as IEEE Xplore and the ACM Digital Library using pilot queries and trial screening, then refining our queries as our inclusion/exclusion criteria were stable. The review is driven by the following research questions (RQs):

- RQ1: What formal modelling and verification approaches (e.g., TLA+, Petri nets, session types, SMT-based techniques, process algebras) have been applied to microservice-based systems, and what evidence is reported about their effectiveness and scalability?
- RQ2: How does architectural or API evolution in microservice-based systems impact verifiability (e.g., specification drift, protocol violations), and what practices or techniques are proposed to mitigate these impacts?
- RQ3: Which repository- or artefact-derived signals (e.g., commits, API diffs, service dependency graphs) are used to infer or update formal models or specifications automatically (i.e., an MSR angle)?

 doi:10.20944/preprints202603.1951.v1

- RQ4: What tools and pipelines support continuous or incremental formal verification in CI/CD environments for microservice-based systems?

These RQs guided the design of search strings, screening criteria, and data extraction fields.

3.2. Literature Search Strategy

We used a database-driven search strategy centered on IEEE Xplore and the ACM Digital Library as the scoping phase showed they contain the bulk of relevant work on microservices and formal verification. The final queries combined three main concept clusters:

1. Microservices / cloud-native (e.g., "microservic*", "micro-service*", "micro service*", "MSA"),
2. Evolution / change (e.g., "evolut*", "version*", "refactor*", "API change", "migration"), and
3. Verification / formal analysis (e.g., "formal verification", "model checking", "runtime verification", "type system", "session type*", "SMT", "TLA+", "Alloy", "Petri net*", and specific tools such as UPPAAL, CSP, Spin).

Table 1. Study selection statistics from database searches

Indexer	Final Query	Sources	Count
IEEE Xplore	("microservic*" OR "micro-service*" OR "micro service*" OR MSA) AND (evolut* OR version* OR refactor* OR "api version*" OR "api change*" OR compatib* OR migration) AND (verify* 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)	IEEE Xplore	53
ACM DL	[[All: "microservice"] OR [All: "microservices"]] AND [[All: evolut*] OR [All: "api evolution"] OR [All: "architecture evolution"] OR [All: co-evolution]] AND [[All: "model checking"] OR [All: "formal verification"] OR [All: tla+] OR [All: uppaal] OR [All: alloy] OR [All: "petri net*"] OR [All: "session type*"] OR [All: csp]] AND [E-Publication Date: (01/01/2010–12/31/2025)]	ACM Digital Library	592

3.3. Inclusion/Exclusion Criteria

The inclusion and exclusion criteria were defined in the protocol and refined slightly after the pilot screening, adjusting criteria to be stricter on their requirements. The final criteria are as follows:

Table 2. Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
I1 Study is internal to the cloud-native domain. We are only interested in analyzing change impacts on cloud-native systems.	E1 Study is about physical networking or other fields not directly related to cloud-native systems, or study is not about distributed systems.
I2 Study is about analyzing changes related to microservice systems.	E2 Study is not clearly related to at least one aspect of the specified research questions.
I3 Study comes from an acceptable source such as a peer-reviewed scientific journal, conference, symposium, or workshop.	E3 Secondary literature reviews.
I4 Study reports on methods, tools, measurements, or any techniques concerning change analysis on microservice systems.	E4 Fault analysis or trace analysis papers intended for system robustness rather than evolution or maintenance.
I5 Study describes solid evidence on microservice change analysis, for instance, by using rigorous analysis, experiments, case studies, benchmarks, or simulations.	E5 Study did not undergo a peer-review process, such as a non-reviewed journal, magazine, or conference paper, master theses, books, and doctoral dissertations (in order to ensure a minimum level of quality).
	E6 Study has been extended in another paper included by the SLR.

 doi:10.20944/preprints202603.1951.v1

These criteria were initially trialed on a subset of hits during the scoping phase and then applied consistently in the full screening.

3.4. Study Selection Process and Statistics

The study selection followed a multi-stage screening pipeline designed to progressively filter the raw search results down to a small set of high-relevant, high-quality primary studies. The process and its rationale are summarized below:

- Initial: This phase records the raw number of results returned by the final search string in each database before any screening. We simply recorded the count down by each site after applying the 2010-2025 publication window. This yielded 53 records from IEEE Xplore and 592 from the ACM Digital Library, for a total of 645 unique results at the outset.
- Proceedings: The proceedings stage filters through items that are not individual research articles, such as entire "Proceedings of …" volumes, front matter, prefaces, or editorials. These were identified by checking the record type and book title fields (for example, entries whose title or type clearly indicated they were a proceedings volume rather than a standalone paper). Once screened, we then moved forward to the next phase.
- Removal (duplicate elimination): The removal stage eliminates exact and near-duplicate records within and across databases. To do this, we exported all remaining items into Zotero and used its duplicate-detection functionality to find records with identical or highly similar titles, DOIs, and author lists. Duplicates were merged or removed so that each logical paper appeared only once in the dataset.
- Filtering (abstract screen): The filtering stage applies to a fast abstract screening against the inclusion and exclusion criteria (Section 3.3). At this point we focused on three main questions:

  (i) does the paper clearly concern microservices or cloud-native systems,
  (ii) is there any notion of evolution or change (e.g., architecture, API, configuration), and
  (iii) is there a link to formal, rigorous, or systematic analysis? Each record was then filtered to provide a higher-level initial filtering of our query results.

- Full Read: The full read stage corresponds to full text screening. For each study kept from the initial filtering, we carefully read the paper to verify clear alignment with our research questions and criteria: explicit microservice/cloud-native context, analysis of evolution or change, and the presence of a formal or otherwise rigorous verification/analysis method with sufficient methodological detail. Papers that lacked scope (e.g., generic SOA without microservice specificity) or that did not describe their methods in enough detail to judge or reuse them were excluded at this stage.
- Quality Assessment: For our final stage of screening, we then performed a quality assessment of screening through each paper we had left at this stage to determine the significance the paper had to our study. We applied the 15-item, we weighed QA checklist and computed each study's overall quality score. Studies below the predefined threshold or clearly failing key criteria were excluded.
- Extract: With the screening stages completed, we began the extraction stage. The extraction stage involves perfuming structured data extraction for all data items within Section 3.6. Concretely, this means the paper provided enough information to identify: the formalism/engine used, the types of evolution of events considered, the evidence and metrics used in evaluation, and any DevOps/CI/CD aspects. If a paper was conceptually relevant but too vague or incomplete to fill these fields, it did not progress.
- Snowballing: The snowballing stage captures additional candidate papers found through backward (reference list) and forward (cited-by) snowballing from the final studies we have. Each snowballed paper was re-subjected to the same multi-stage screening pipeline as our initial query results were.

 doi:10.20944/preprints202603.1951.v1

3.5. Quality Assessment

For the studies that reached the Quality Assessment stage, we applied a quality assessment criterion inspired by common SLR practice. The checklist is as follows:

Table 3. Quality Assessment Criteria applied to candidate studies

QA ID	Question / Criterion	Purpose / Focus Area	Score Options	Notes / Examples
QA1	Study 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

We had a total of 15 criteria that we would check throughout our full read of each individual paper, grading each criterion on a scale of:

- 1 – clearly satisfied
- 0.5 – partially satisfied
- 0 – not satisfied or unclear

 doi:10.20944/preprints202603.1951.v1

These QA items are grouped into four broad dimensions:

- Topical relevance and context (QA1-QA4)
  - QA1: Explicit focus on cloud-native or microservice-based systems
  - QA2: Analysis of architectural or API evolution
  - QA3: Use of formal methods or explicit verification techniques (e.g., TLA+, Petri nets, SMT, Alloy)
  - QA4: Placement within a cloud-native or distributed software engineering context (as opposed to pure hardware/physical networking)
- Methodological clarity and evidence (QA5-QA9)
  - QA5: Methods/tools for change or evolution analysis are clearly described and reproducible
  - QA6: Presence of empirical validation (experiment, benchmark, case study)
  - QA7: Data sources are identified (e.g., repositories, CI/CD pipelines, APIs)
  - QA8: Evaluation of effectiveness or scalability of the method
- Contribution and applicability (QA10-QA12)
  - QA10: Proposal of a new model, framework, or tool
  - QA11: Extension of existing work with new evidence (not duplicate publication)
  - QA12: Actionable findings or guidelines for evolving microservices (useful to practitioners or future research)
- Venue quality, accessibility, and reproducibility (QA13-QA15)
  - QA13: Peer-reviewed publication (journal, conference, symposium, or workshop)
  - QA14: Written in English and accessible in full text
  - QA15: Provision of reproducible results or artifacts (e.g., GitHub repository, dataset, or detailed method description)

To reflect the relative importance of each dimension, we assigned weights to the items. Relevance and context items (QA1-QA4) each have a weight of 0.075, methodological and evidence-related items (QA5-QA9) each have a slightly higher weight of 0.0875, and contribution / venue / reproducibility items (QA10-QA15) each have a weight of 0.05. The weights sum to 1.0, so that each study's overall quality score lies in the range [0, 1]. We used the QA scores as a final Inclusion threshold, where studies with an overall QA score below .5 were excluded from the final synthesis, unless they were uniquely relevant to a research question and the low score was due to minor reporting issues rather than fundamental methodological flaws. The full weighted table can be seen below to see the distribution between each QA item:

Table 4. 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

 doi:10.20944/preprints202603.1951.v1

3.6. Data Extraction and Synthesis

To organize the selected studies under the four research questions (RQ1–RQ4), the team applied a qualitative content-analysis procedure such as open, axial, and selective coding commonly used in thematic synthesis. First, each paper was reviewed to extract its key technical contributions, verification techniques, and the specific problems it addressed. These initial codes were then grouped into conceptually related categories (i.e. actor-based verification, control-plane validation, architectural drift, repository-derived signals, CI/CD-based runtime verification) by examining similarities in purpose, methodology, and verification intent. In the final stage, each thematic group was aligned with the research question it most directly supported, based on whether the contribution operated at design-time modeling, system evolution, repository-level inference, or runtime/incremental verification in deployment pipelines. The resulting structure is visualized in Figure X, where first-order concepts are grouped underneath higher-order themes for each RQ. Reference numbers (e.g., [6], [8], [17]) attached to each thematic box indicate which primary studies contributed evidence to that category, ensuring traceability from themes back to the individual papers. This structured and transparent approach ensures that the mapping of studies to research questions is concept-driven, reproducible, and founded in established qualitative synthesis methods for systematic literature reviews.

Table 5. Summary of included studies on microservice evolution and verification

Authors & Year	Study Type	System Context	Method Used	Verification/ Evaluation	Contribution Summary
Biswas et al. (2023) [3]	Empirical + Tool	Microservice architecture models	Model-to-code consistency checking	Detects architecture drift	Continuous consistency checking reduces divergence
Namyar et al. (2025) [13]	Formal verification + system	Cloud control-plane	TLA+ model checking	Safety verification; scalability evaluation	Presents ZENITH: formally verified control plane
Copei et al. (2023) [8]	Static analysis	Microservice implementations	Static code analysis	Detects API mismatches, structural violations	Static analysis improves microservice implementation quality
Cerny et al. (2018) [7]	Conceptual taxonomy	Microservice architecture	Contextual analysis	Conceptual reasoning	Defines contextual understanding of microservice architectures
Boza et al. (2019) [4]	Performance experiment	Container-based deployments	Empirical performance evaluation	Measures resource variability and impact	Argues for performance-aware container deployment
Anisetti et al. (2020) [1]	DevOps certification	Cloud service pipelines	Continuous certification pipeline	Certification via automated tests	Introduces continuous certification for DevOps
Shahin et al. (2017) [14]	Empirical study	Microservice-based DevOps teams	Interviews and qualitative analysis	Process/coordination evaluation	Shows how CD impacts team structures and responsibilities
Burns et al. (2016) [5]	Systems design	Cloud-scale cluster systems	Systems architecture analysis	Observational; production-scale reliability	Summarizes Borg, Omega, and Kubernetes evolution
Ferrara et al. (2024) [9]	Survey	Software verification	Verification methods survey	Discusses scalability limits	Identifies major challenges in software verification

 doi:10.20944/preprints202603.1951.v1

Table 5. *Cont.*

Authors & Year	Study Type	System Context	Method Used	Verification/ Evaluation	Contribution Summary
Soares et al. (2023) [16]	Survey	Software architecture evaluation	Continuous evaluation literature review	Identifies evaluation gaps	Shows trends in continuous architecture evaluation
Cerny et al. (2024) [6]	Static analysis + visualization	Microservice codebases	Architecture recovery from code	Structural violation detection	Generates updated architecture models from code
Stillwell & Coutinho (2015) [17]	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) [11]	Survey	Serverless / FaaS systems	Resource management taxonomy	QoS / performance discussion	Provides holistic overview of resource management challenges
Merlino et al. (2019) [12]	Systems/platform	Edge–Fog–Cloud hierarchical systems	Workload modeling and placement	Empirical performance evaluation	Proposes workload engineering patterns across layers
Silva et al. (2024) [15]	Runtime testing method	Service-based applications	Self-adaptive testing framework	In-the-field fault detection	Introduces dynamic adaptive testing in deployed environments
Anisetti et al. (2018) [2]	Security testing	Composite SOA/service compositions	Test-based certification	Security violation detection	Connects testing evidence to certification for evolving services
Camilli (2020) [19]	Formal verification	Microservice process flows	Continuous formal verification	Model checking	Continuous formal verification of microservice-based process flows
Lamport (2009) [18]	Formal specification	Distributed systems	PlusCal/TLA+ specification	Protocol correctness	Introduces PlusCal algorithm language for TLA+

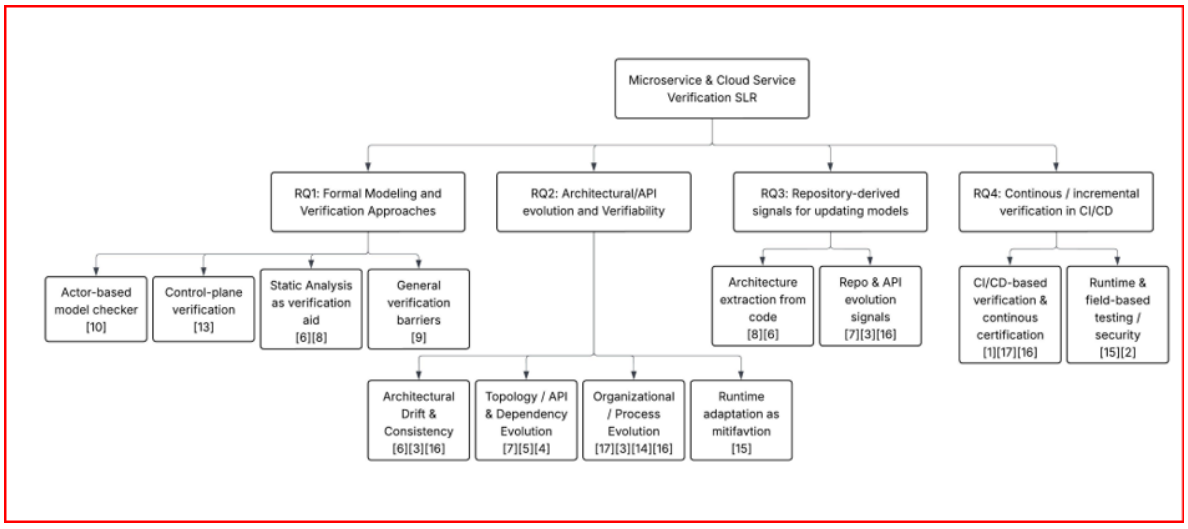

Figure 1. Hierarchical mapping of research questions (RQ1–RQ4) to thematic categories and representative studies in the systematic literature review (SLR).

 doi:10.20944/preprints202603.1951.v1

4. Results

This section provides the findings of the systematic literature review and is organized by research questions (RQ1 – RQ4). Each RQ has sub-themes that emerged from the selection of papers during data extraction. The data extraction for this paper was focused more on concepts rather than quantitative metrics. These papers include verification techniques, architectural challenges, evolution patterns, tool support, and evidence of scalability. Each subsection highlights the most significant contributions, challenges, and identified gaps.

4.1. Formal Modeling and Verification Approaches in 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. This section examines the main classes of approaches identified in the selected studies, with a focus on how well they scale microservice-like workloads. Subsections first look at model checking techniques, including actor-based modeling languages such as Rebeca and their evolution toward scalable verification engines. The discussion then shifts to control-plane verification and fault tolerance in microservice-based controllers, where formal specifications and proofs are used to guarantee consistency in distributed settings. Building on this, the section reviews how static code analysis can serve as a practical aid to verification by extracting communication structures and architecture views directly from source code. Finally, it situates these results within broader, long-term trends in software verification to highlight persistent barriers like state-space explosion and technological diversity that remain especially relevant in microservice environments.

4.1.1. Model Checking Approaches

Rebeca is an actor-based modeling language that structures systems as independent actors (classes) that communicate through asynchronous messages. This is very similar to how microservices interact through decoupled services and continuous message passing. The paper illustrates this using an ordering-system example with a customer actor, a shipping actor, an order-processing actor, and an inventory actor, all coordinating through message exchanges [10]. This interaction pattern is intentionally aligned with the communication patterns seen in distributed microservice architectures. Across its evolution, Rebeca has demonstrated potential as a scalable model-checking framework capable of verifying systems with large numbers of concurrent actors, like microservice deployments where concurrency and distributed communication lead to state-space growth. The paper looked at real world examples that involve systems with complicated interactions. These examples show using an actor-based approach to checking errors can scale with microservice-style systems [10]. Rebeca's dedicated model-checking engine, Modere, was a major milestone in this evolution. Modere did not rely on repeatedly calling itselsf during a Nested Depth-First Search (NDFS), which helped prevent crashes and reduced memory usage during verification [10]. Modere added two techniques that assisted with system checks. Microservices can run many tasks at the same time, and the number of possible scenarios to verify can increase in number very quickly. These additional techniques help ensure that verification remains feasible as systems scale. The next major step in Rebeca's tooling was the development of Afra, the current model checker. Afra enhances modular verification and supports more complex communication and timing behaviors. In the paper, one of the industrial examples demonstrates how Rebeca models can interact with messaging systems such as Kafka to approximate high-throughput, distributed event-driven architectures [10]. Although the paper does not directly model Kubernetes, it notes design similarities between Kafka-based distributed systems and orchestrated microservice deployments, especially regarding how both deal with concurrency and potential state-space growth. Because of this, Rebeca shows evidence of being a strong model-checking approach that can scale to the levels of concurrency observed in modern microservice systems.

 doi:10.20944/preprints202603.1951.v1

4.1.2. Control-Plane Verification and Fault Tolerance

ZENITH is a microservice-based controller that, by design, avoids state inconsistencies between control and data planes. These inconsistencies result from bugs, failures, or race conditions [18]. Current solutions rely on periodic reconciliation, which causes reduced network availability, scales poorly, and high tail latency. ZENITH introduced an architecture that consists of two layers and a tool. The first component is ZENITH-core, a microservice-based CDN controller that is model-checked for eventual consistency and proven using TLA+. The second component is ZENITH-apps, a framework for formally verifying CDN applications. Applications just specify behavior as PlusCal [19] specifications. These specifications are verified using Abstract Core and guarantee end-to-end correctness when composed with ZENITH-core. To help in app development, a tool is built to generate Python code from PlusCal specifications. Generated code preserves the correctness of the previously verified specification. ZENITH used formal verification methodology and considered the largest TLA+ specification to date with 8.6K lines. Along the way, it introduced novel scaling techniques for model checking complex systems. ZENITH is a production-ready verified controller implementation that shows how microservices decompose. It also demonstrates the feasibility of formally verifying complex microservices. ZENITH is still limited to Python while Java is the major microservices language so far. It is also limited to OpenFlow-like protocols. Another verification tool observed is Anvil. Anvil observes liveness of controller clusters which monitors that tasks are being completed or delayed. Since controllers may not have any responses such as crashes, Anvil focuses on searching for controllers that are delayed or stuck [35]. Problems such as this are more nuanced when it comes to traditional testing and may better represent real world issues. With Zenith searching for consistency and correctness, Anvil may be used with Zenith to be more efficient while ensuring operations and required actions are completed. Moving beyond individual controllers, formal models of container orchestration platforms such as Kubernetes. Kubernetes is a prime example of a platform like others that includes concepts such as scheduling decisions and reconciliation loops that can be systematically analyzed [36]. The model itself does not verify application code, but it can help explain how microservices are deployed, scaled, and managed. By observing Kubernetes, deductions can be made about how orchestration decisions affect services. This provides information about process flows and services interactions discussed in the next section.

4.1.3. Formal Verification of Service Behavior, APIs, and Process Flows

Previous examples such as model checking and control-plane verification address change at the component level, but several studies extend verification to APIs and end-to-end process flows. These are highly relevant to microservices where failures can occur between the services. One group of studies observes verification through RESTA APIs. Rauf et al. Treat REST APIs as stateful systems which help control states of the services. Using event-based triggers, they track all verifications by ensuring that all states remain true, and when they are not, it notifies developers that the endpoints may need to be reviewed. This allows developers to use event-based triggers for monitoring the status of APIs and implement continuous monitoring failures. Other studies move beyond the single service and will extend the scope to entire process flows across microservices. These represent more practice in business cases with how microservices are used in the field. Examples may include ordering, receiving payment, or fulfillment. The goal is to ensure the process is occurring as it needs to and if it happens in the correct order. It is important that this is continuous in case there are any changes to any processes; they do not interfere with the process of completion from end-to-end across the microservice. This is what may occur when microservices evolve. With both approaches, verification can occur through APIs and workflows where microservices interact. In many cases, failures between microservices are not a defect on one microservices part but could be a misunderstanding between the two services, and continuous monitoring and testing may help mitigate these issues. The challenges identified also grow with the ideas to help maintain and verify microservices. The more microservices are involved,

 doi:10.20944/preprints202603.1951.v1

the more complex verification becomes. These challenges present better options or tools to assist or automate verification.

4.1.4. Static Code Analysis as a Verification Aid

Static code analysis plays an important role in supporting verification efforts for microservice-based systems. Copei et al. show how static analysis can reveal the underlying communication patterns between microservices, especially the API calls and service-to-service interactions that are often difficult to track in distributed architectures [8]. Their work centers around SIARest, an IDE-integrated tool designed to help developers identify potential communication issues early in the development cycle. SIARest flags incorrect or outdated API endpoints and detect possible violations of expected service contracts, which allows developers to resolve these problems before they evolve into architectural inconsistencies [8]. It also provides autocompletion support that is based on extracted communication structures, helping ensure the implemented code aligns with the intended microservice interactions. This early detection is especially important for verification, because static analysis outputs can effectively serve as the boundary conditions for later formal modeling. By surfacing structural and communication issues early, static analysis reduces the complexity of the models that need to be verified and helps prevent scalability problems when applying formal verification tools. Similar approaches in the literature, such as Cerny et al.'s work on extracting architecture models directly from source code, also show how static analysis can generate candidate architecture diagrams that support more formal verification steps [6]. Overall, these static analysis techniques provide a practical way to strengthen the verification workflow by ensuring more accurate representations of the system before the formal methods phase even begins. Cerny et al. works using static code analysis to produce visual models of microservice architecture. Although these visual models are not used directly for verification, they help the development teams have a better holistic perspective on the system. Which always helps uncover dependencies and implicit connections [6]. Cerny et al.'s work consisted of four phases of process for software architecture reconstruction of SAR. Phase one is the extraction phase which does Abstract Syntax Tress AST analysis of individual microservice codebases to extract all graphs, controller endpoints, data entities, and component types. Phase two uses the data collected in phase one to generate component call graphs that show components' connections implied by calls. Phase three merges microservices' representations using overlapping entities and entities matching Wu-Palmer algorithm. Phase four is for representation analysis and insights generation. It detects design spells, anti-patterns, security policy violations, and system bottlenecks. All four phases have been implemented in a tool that was evaluated by practitioners. Tool also released as open-source software foundations that other researchers in the field. The tool is currently limited to Java microservices written in the Spring framework. It cannot detect event-driven communication like queues (Kafka, JMS.).

4.1.5. General Verification Barriers and Challenges

The evolution of software over the years has exposed new verification challenges. Ferrara et al.'s work presents a comprehensive historical analysis of these challenges from the 1940s to present day [9]. Ferrara et al.'s work organized software verification evolution into distinct phases by decades. For each phase, the paper considered technologies and programming languages that emerged in that phase, types of software developed, verification challenges raised for these types of software developed, and scientific advancements in that phase. Identified challenges have been cataloged into certain patterns and their root causes. The identified patterns are inconsistency issues, scalability problems, and technological diversity problems. The paper also identified future trends and challenges for practitioners with a focus on AI-generated code and emerging technologies like quantum computing and novel deployment approaches. The historical verification lens of Ferrara et al.'s work helps practitioners and researchers anticipate the microservices challenges like language heterogeneity, independent deployments, and distributed runtime nature.

 doi:10.20944/preprints202603.1951.v1

4.2. Architectural / API Evolution and Impact on Verifiability

Microservice architectures are designed to evolve services that are added, split, merged, and deprecated, while APIs changes to accommodate new features and client needs. However, this continuous evolution has direct consequences for verifiability. As architecture and APIs change, previously valid assumptions encoded in models, test suites, and architecture descriptions can become outdated, leading to drift between intended and implemented designs. This section explores how different forms of evolution shape the verification landscape. Subsections first consider how architecture drift can be detected and controlled through continuous consistency checking, then discuss how large-scale topology changes and platform shifts complicate integration and system-level verification. The remaining subsections examine how team structures, DevOps practices, and self-adaptive testing techniques can help mitigate these challenges by aligning verification activities with the pace of architectural and API evolution.

4.2.1. Architectural Drift & Inconsistency

With the evolution of microservices, including architectural and API changes, there still is no consistent method that teams use for continuous evaluation or verifiability. Architecture artifacts created early on, like UML diagrams or higher-level architectural descriptions, can help guide development, but as microservices become more complex and change more rapidly, the risk of architectural drift becomes a real challenge. This drift directly impacts verifiability because the further the implementation moves away from the intended design, the harder it is to validate system behavior or reason about correctness over time. One way to help reduce this drift is through automation and continuous consistency checking, which helps keep architecture and implementation aligned. As seen in Continuous Evaluation of Consistency in Software Architecture Models [3], drift can be made visible and measurable through model-to-code comparisons. Their approach [3] uses simulation and analysis tools to evaluate artifacts such as UML diagrams, check them against the actual implementation, and provide reports highlighting inconsistencies. The evaluation results in their paper also show that performing these checks continuously reduces the accumulation of inconsistencies over time. These findings are further supported by broader trends identified by Soares et al. [16], who note that architectural erosion is common across evolving systems and that frequent updates introduce more opportunities for inconsistency. Together, these studies suggest that automated consistency checking, along with shifts in team processes and culture to support ongoing verification, will be increasingly important for managing architectural drifts in microservice environments. Soares et al. have conducted a systematic mapping study examining evaluation of software architecture. The study identified five evaluation approaches: scenario-based, simulation-based, rule-based, metric-based, and mathematical modeling. Out of these five approaches, only three emerged through automation. Simulation-based approaches used automation for handling uncertainty. Metric-based approaches used automation for automated quality assessment. These two approaches used the CI/CD pipeline to integrate evaluation into the SDLC. Rule-based approaches used IDE plugins. The study identified automation of the architecture evaluation as one of the research trends in this area with a focus on distributed systems like microservices and IoT.

4.2.2. Evolution of Microservice Topology – Complexity to Verification Failure

While architecture evolution and architecture drift have different characteristics such as phenomena. For verification purposes, architecture evolution introduces similar challenges to Architecture drift. Architecture drift usually happens unintentionally over the lifetime of a code base, but architecture evolution is usually intentional. The software community comes to conclusions based on practical challenges and battle-tested techniques. Cerny et al. [7] conducted a systematic mapping study that analyzed and compared microservices architecture to service-oriented architecture. Although the focus of the study is architecture, it helps in setting up the frame of reference for verification strategy and scope. One of the key differences between SOA and microservices is the central and holistic

 doi:10.20944/preprints202603.1951.v1

nature of SOA implementation. It puts governance and scalability bottlenecks for developers, but it can make verification efforts easier. Microservices introduce loose development where teams move independently of each other. This speed up development but makes verification very hard, especially integration and system testing. The same difference impacts other aspects of the software development life cycle that are beyond the scope of this paper. Although microservices don't have to be deployed using containers, most developers will use containers to deploy their microservices if they can. (5) Burns et al. studied the containers' technology and its evolution [5]. They studied the container management systems built by Google that preceded Kubernetes, which arguably became the most successful system in that domain. They studied Borg and Omega, which were internal systems used for production and research, respectively. One major lesson learned is the shift from machine-oriented management to application-oriented management. This aligns with microservices' general approach of breaking the system into small independently deployable services handling specific responsibility instead of a monolithic system that gets deployed as a whole. Also, the container ecosystem challenges identified resemble similar patterns of microservices' development and verification challenges. Challenges like service discovery, namespaces, application-aware load balancing, rollout tools for deployments, monitoring and alerting. Paper also compared the different philosophy behind each system and the technical innovations each introduced, which is beyond the focus of the current paper.

4.2.3. Organizational & Process Evolution Impacts

Teams today are expected to operate with increasingly fast release cycles, and this constant pace influences how architectures and APIs evolve through ongoing iterations in agile and DevOps environments. As releases accelerate, organizations face more frequent API changes and a greater chance of architectural drift, which connects back to the drift and inconsistency issues discussed earlier. Shahin et al. look at how various organizations adopt Continuous Delivery and Continuous Deployment, focusing on team structures and processes that help keep verification manageable under rapid change [14]. Although organizations differ in how they structure Dev and Ops responsibilities, about 40% of respondents reported having some form of centralized team or shared group that supports CI/CD practices across development teams [14]. The study also found that roughly 68% of participants observed stronger collaboration after adopting CI/CD, which suggests that organizational evolution can help offset some verification challenges by improving communication and shared awareness [14]. This is tied into RQ2 because better collaboration can help reduce verification drift when APIs or service boundaries change quickly. Even with increased collaboration, the authors note that bottlenecks still appear when project status or pipeline information is not communicated clearly, showing how communication becomes even more important as release frequency increases. Responsibilities themselves did not drastically shift, but many practitioners said they needed to expand their skill sets to operate effectively in continuous development environments [14]. Overall, this paper highlights that organizational and process evolution—while not directly architectural still affects verifiability because teams must adapt at the same pace as the system's rapid changes.

4.2.4. Self-Adaptive Testing for Evolving Systems

As systems and microservices continue to evolve at rapid rates, there is an increasing need to consider alternative testing approaches that can keep pace with frequent architectural and API changes. Silva et al. discuss this challenge through the lens of self-adaptive testing in the field, arguing that traditional in-house testing alone becomes insufficient when upgrades and new features are released continuously [15]. Because system evolution often outpaces the ability to manually update tests, the paper highlights how verification drift can occur when the underlying architecture changes faster than the test suite meant to validate it. To address this, they outlined mechanisms for automated, feedback-driven testing that operate during deployment and runtime, helping teams detect defects earlier and streamline overall defect management [15]. Although presented broadly, Silva et al. provide examples that align closely with microservice-based systems. They describe scenarios where runtime logs and session data are collected and analyzed in the field, allowing teams to evaluate how

 doi:10.20944/preprints202603.1951.v1

well services perform as individual components evolve independently [15]. This approach becomes especially important in microservices, where localized regressions may surface in one service even when the rest of the system remains stable. As a result, self-adaptive testing supports verifiability by allowing tests to adapt alongside the architecture itself rather than relying solely on pre-release, static verification practices. This directly connects back to the broader theme of RQ2, emphasizing the need for verification strategies that evolve in response to architectural and API change.

4.3. Repository-Derived Signals for Updating Models

Think about trying to truly understand a microservices-based system, not by reading through dusty documents but by seeing how the system behaves and evolves every day. Repository-Derived signals make this possible happen. Through extracting insights from source code, commits, and dependencies recorded in repositories, we obtain a window into the real system architecture—as it lives, changes, and sometimes even misbehaves.

4.3.1. Architecture Extraction from Static Code

Static code analysis is like shining a spotlight on your microservices system's inner workings. Instead of relying on design documents that often get outdated quickly, researchers have shown that analyzing source code directly can reveal where service boundaries lie, the API endpoints they expose, and even how they communicate through HTTP calls or messaging [7]. This approach exposes the as-built architecture, giving developers a clear picture of what's really running. More than just a snapshot, it helps catch "smells" and anti-patterns creeping into the codebase—the subtle warning signs that architectural erosion or drift could be underway. Other studies emphasize how these static insights turn into early alerts for dependency hotspots or coding antipatterns that threaten the autonomy and evolvability of microservices [8]. The primary benefit of adopting this approach is support for automation and repeatability: static extraction pipelines can run on every commit, ensuring the architectural views are always fresh and synchronized with the code [3]. This is central to what scholars call continuous architecture, where architectural models aren't static documents but living, code-driven artifacts. However, the accuracy of these extractions relies on effective heuristics. As highlighted by [8], detecting service boundaries often means relying on framework annotations, container descriptors, and building modules, while interface contracts come from OpenAPI or protobuf/IDL definitions. If these low-level facts are lifted into visual models such as C4 diagrams or service-dependency graphs—as [6] describe—it becomes much easier for engineers to grasp complex architectures and spot design violations early. In contrast to the study perspective, this approach also offers valuable insight into architectural analysis.

4.3.2. Tracking Evolution via Commits & Dependency Changes

As any developer knows, software architecture is never static. Every commit, every pull request adds a thread to the ongoing story of a system's evolution. Mining these commit histories and associated metadata can unveil valuable signals about architectural changes—service splits or merges, interface updates, and shifting dependencies [16], [3]. Repositories are like detailed journals and argue by [16] that harnessing these records enables architectural models to grow in step with the codebase, significantly shrinking the usual gap between intended and actual architecture. By combining commit mining with static analysis, teams can selectively update only the changed services or modules, keeping analysis efficient and relevant. Dependency manifests—those Maven POM files, Node Package Manager (npm) package lists, Dockerfiles, and Helm charts—tell an equally important tale. Upgrading the library, adding a new sidecar container, or introducing a service mesh subtly reshapes the system's topology and non-functional qualities such as latency and resilience. Tracking this churn, as highlighted by [3] and others [16], helps teams identify "risk windows" where architecture consistency might be threatened. What makes this approach truly powerful is embedding architectural rules as automated checks in CI pipelines. In this way, every commit is scrutinized against standards: Is older API clients deprecated correctly? Are dependencies kept within recommended boundaries?

 doi:10.20944/preprints202603.1951.v1

Are new anti-patterns creeping into critical paths? The study by [3] emphasizes how automating these conformance checks turns architectural governance from a reactive chore into a proactive, continuous activity. Additionally, research studies by [18] and [10] see enormous potential in treating longitudinal commit data as "architectural event logs." These logs let us observe cause-effect relationships between changes and defects or performance issues, feeding into formal verification efforts where realistic, evolving models are gold.

4.3.3. Opportunities for Formal Model Generation

Going further than just visualization and changes tracking, repository-derived signals are paving the way toward smarter, more automated formal models, that is precise mathematical representations of the system's architecture that can be rigorously checked. The study by [3], [16] explores how techniques like graph-based learning and incremental verification could automate the generation and refreshing of these formal models on every commit or dependency update. This opens the door to "continuous verification," where key properties such as safety, availability, or fault tolerance are checked constantly as the system evolves. Besides bridging static analysis with formal verification, models cease to be static artifacts and instead become living, actionable entities that inform engineering decisions. Research study [8] acknowledge the difficulties involved in scaling formal methods to fast-changing microservice landscapes but point to lessons learned from two decades of actor model checking—like modularity, compositional reasoning, and domain-specific abstractions—as valuable tools to meet these challenges [10]. This approach creates a virtuous feedback loop: repositories feed continuously updated models; models drive automated architectural checks; these checks guide better commits, improving the system's evolution over time.

4.4. Continuous / Incremental Verification in CI / CD

Traditional verification activities were often performed at discrete milestones, but microservice-based systems and DevOps practices demand verification that is continuous, incremental, and tightly integrated into delivery pipelines. This section reviews tools and approaches that embed verification into CI/CD workflows and extend it into the runtime environment. It begins with work on continuous certification in DevOps, where compliance and formal checks are treated as first-class pipeline stages rather than afterthoughts. The next subsection focuses on runtime and field-based testing, highlighting self-adaptive techniques that update tests and evaluation logic in response to observed behavior in production. Building these ideas, we then examine how architecture validation can be integrated into CI/CD through automated model transformation, simulation, and conformance checking. Finally, the section considers emerging formal verification pipelines and orchestration-focused tooling that aim to make incremental verification feasible at the scale and speed characteristics of modern microservice deployments.

4.4.1. Continuous Certification in DevOps

Anisetti et al. [1] work reflects their experience in the EU H2020 Toreador project. It introduces continuous certification methodology for DevOps environments. This is very applicable to microservices systems since most teams building microservices also adopt DevOps tools and processes. The methodology implementation starts with modeling the development process. The methodology then embeds hooks throughout the development process to continuously collect data for inspection. Data collected is used to conduct evaluations on each stage of the development process. The main contribution of Anisetti et al. [1] work is identifying the requirements for continuous certification in DevOps environments. This includes modeling the development process, specification for each stage of evaluation, establishing trustworthy inspection mechanisms and owners, and embedded verification. Anisetti et al. [1] work reflects their experience in the EU H2020 Toreador project. It introduces continuous certification methodology for DevOps environments. This is very applicable to microservices systems since most teams building microservices also adopt DevOps tools and processes. The methodology implementation starts with modeling the development process. The methodology

 doi:10.20944/preprints202603.1951.v1

then embeds hooks throughout the development process to continuously collect data for inspection. Data collected are used to conduct evaluations on each stage of the development process. The main contribution of Anisetti et al. [1] work is identifying the requirements for continuous certification in DevOps environments. This includes modeling the development process, specification for each stage of evaluation, establishing trustworthy inspection mechanisms and owners, and embedded verification.

4.4.2. Continuous Certification Frameworks for Cloud and Microservices

Continuous certification in DevOps establishes certification as an ongoing activity for development; other bodies of work expand the idea and redefine certification itself. These works recognize that certification itself should continuously evolve alongside clouds and microservices. This shift is significant in a space where frequent deployments and changes to configurations can invalidate assumptions during verification. Multiple studies discuss certification as a feedback loop that assesses trust based on runtime evidence. Lins et al. Discusses dynamic certification where it is continuously re-evaluated using monitoring rather than assuming validity after approval. Runtime monitoring is the primary data to be surveilled for certification decisions and trust adjusted with system changes. Certification becomes an adaptive process that responds to its environment. Building on this, Anisetti et al. explores how the idea of continuous certification can be implemented with trust and scalability in mind. This is done by automated frameworks with human oversights since many systems will require some form of human interventions at some point. Highlighting the balance of automation and human interaction in this study reinforces that certification pipelines can be scaled but must be done so with trust. Certification evidence can also be observed in cloud environments. Greulich et al. reinforces the importance of monitoring data such as runtime metrics, logs, and performance indicators as input to certification decisions. The same claim reinforces that certification in microservices systems shall always be considered together when observing infrastructure. Other works continue the momentum by adding to current models to understand system behavior. Anisetti et al. started to incorporate timing constraints and probabilistic guarantees into certification frameworks. These are relevant to microservices in the continuing evolution, and many deployments and a similar model may be used to help with continuous certification pipelines. Another study by Anisetti et al. used security-focused studies to reinforce previous studies on continuous certification. In A Test-Based Incremental Security Certification Scheme, certification would be updated incrementally based on testing which would allow the certificate to evolve. Another study by Krotsiani et al. supports that runtime monitoring in the cloud environment can be used for security properties over time. The key information is that these services moved from one step to a continuous piece.

4.4.3. Runtime / Field-Based Continuous Testing

Silva et al. recommend several ways to implement self-adaptive testing as part of runtime or field-based verification, based on findings from their systematic literature review. One of the main outcomes of their work is a consolidated view of the typical runtime testing loop architecture, which they observe across multiple studies. Their analysis identifies at least six components that appear consistently, and these map closely to the Monitor–Analyze–Plan–Execute over Knowledge (MAPE-K) feedback loop [15]. Within this structure, monitoring is responsible for sensing relevant environmental or behavioral changes that may signal the need for adaptation. The analysis activity then examines those changes to determine what kind of adaptation might be required, including whether new test cases should be generated or if existing ones should be adjusted. Planning and execution coordinate these adaptations and testing activities, ideally in a way that remains transparent and documented so that engineers can later review whether the system's decisions remain aligned with organizational goals and expected behaviors [15]. Silva et al. also stresses that whenever possible, these runtime adaptations and test updates should occur as small, incremental changes, which helps maintain reliability and reduce verification risk, consistent with the incremental orientation of CI/CD practices already discussed in earlier sections. Beyond the loop, Silva et al. describe seven subsystems that characterize how

 doi:10.20944/preprints202603.1951.v1

runtime self-adaptation occurs. These subsystems include the object being adapted, triggers that initiate adaptations, how decisions are made, what types of adaptations are possible, and related factors [15]. These elements reinforce the idea that field-based testing is not a post-release add-on, but rather a continual process that runs alongside the deployed system. As microservices evolve or encounter unexpected runtime conditions, these self-adaptive mechanisms allow the testing process to evolve in parallel. Within a CI/CD context, Silva et al.'s approach effectively extends verification beyond the build pipeline into the operational environment. This supports the broader goal of RQ4: enabling continuous, incremental verification even after deployment, especially as microservices change frequently and may behave differently when running at scale or under real-world variability.

4.4.4. Integrating Architecture Validation into CI / CD

Another important part of microservice evolution relates to architectural changes and the need to continually track them as systems evolve. After discussing self-adaptive testing as a tool that operates post-release, it is also useful to look at approaches that keep architecture models consistent during development. Biswas et al. emphasize that detecting and resolving inconsistencies early is critical and argue that architecture validation should be integrated directly into the development workflow rather than treated as a static or one-time activity [3]. Their study in the automotive domain shows how simulation-based evaluation can be incorporated into a CI/CD pipeline, and while the context is different, the underlying idea of incremental model checking is portable to microservice engineering as well [3]. The proposed architecture Continuous Architecture Model Validator CAMV is an industry-oriented automated solution that integrates simulation-based evaluation tools into CI/CD pipelines. The approach includes: a model converter, the FERAL simulator, checker, executable model creation procedures, and a bridge. This tool chain demonstrates how architecture models can be automatically updated and validated on each change [3]. This aligns with broader trends in continuous evaluation research, which emphasize triggering validation tools automatically on each commit and maintaining architecture models as evolving, incremental artifacts rather than isolated snapshots [16]. In the context of RQ4, this work helps illustrate how architecture-level verification can become another automated stage in a CI/CD pipeline, supporting continuous and incremental verification as systems evolve.

4.4.5. Formal Verification Pipelines

The complex interactions between microservices pose a unique challenge for verification and special attempting to use formal methods. Matteo Camilli [20] addressed this challenge within the commonly used (CI/CD) pipelines. The paper's main contribution is a comprehensive formal verification framework that spans the design and operation phases of a microservice workflow. The framework consists of three steps. Model-to-model transformation transforms a model from a domain-driven language of a microservice orchestration platform like Netflix Conductor [21] to a formal model which then gets transformed Time Basic Petri Net (TB net). The next step, design-time verification, uses this model for model checking, constructing Time Reachability Graph (TRG), which gets used for system correctness check against requirements expressed as Time Computation Tree Logic (TCTL) properties. The last step, runtime verification, uses the generated models to monitor the behavior for the deployed application against its formal specifications. The framework implementation relied on instrumenting Conductor's code using an expected framework to intercept key events. This enables the system to produce an on-the-fly report about both functional and temporal conformance failures, providing critical insights to developers and operations teams. This provides a reference implementation that can guide future work for a system that doesn't use Netflix Conductor.

5. Discussion

Change, decentralization, and heterogeneity are main characteristics of Microservice based systems. These characteristics impact both development and verification of these systems. Development teams used a plethora of tools, frameworks, and processes to gain fruit and overcome the side effects of

 doi:10.20944/preprints202603.1951.v1

these characteristics. Verification teams still need to catch and implement similar practices. Our review highlighted how architecture and API change impacts verifiability of Microservices based systems. Whether this change is intentional, evolution, or unintentional, drift, the change impacts verifiability. The decentralized nature of organizations and teams building Microservices based systems amplify the processes and Organizational changes' impact on verification efforts. Our review suggested that an extensible methodology for verification teams in Microservices based environments can be achieved. The whole development process needs to be modeled and planned for the CI/CD pipelines. Different verification teams can use different techniques depending on the tools used by the development teams. For example, if development teams use a framework, language, or tool that make formal model generation feasible, the verification team can utilize formal methods throughout the development stages and implement a continuous certification framework around that. If verification can't implement formal methods due to lack of support, they can monitor code repositories for changes and use that to build tooling that guides the continuous verification process. Continuous verification at runtime should utilize what development time built and expand it to include service level agreements for microservices. Usually, this means expanding the automated verification to include runtime monitoring and alerting, along with any infrastructure scaling tuning. Future work to focus more on two areas: (i)frameworks used by development teams that could ease the adoption of formal methods like Netflix Conductor. (ii) building models that reverse engineer source code and logs to construct models. These are two approaches, proactive and reactive, for formal verification that rely on models. Depending on situation teams and apply one or another, or a combination of aspects from both.

6. Threats to Validity

In line with this systematic literature review, several threats to validity must be acknowledged to ensure transparency and to contextualize the findings:

(i). Search Bias: The search strategy we employed focused primarily on IEEE Xplore, SpringerLink and the ACM Digital Library, which are strong repositories for software engineering research. Moreover, relevant studies may exist in other venues such as Elsevier, or practitioner-oriented outlets. To limit the scope, we risk omitting contributions that address microservice evolution and verification from different perspectives, particularly industrial or interdisciplinary work. Future reviews should broaden the search base and consider complementary indexing services to reduce this bias.

(ii). Selection Bias: A multi-stage screening process (abstract filtering, full-text reading, and quality assessment) was designed to be rigorous, however inevitably involved subjective judgment. For instance, borderline cases where microservices were mentioned but not central to the study required interpretation. Even though all of us participated in screening, differences in interpretation could have influenced which studies were retained.

(iii). Publication Bias: Our investigation restricted the dataset to peer-reviewed venues to ensure methodological quality. Although this strengthens reliability, it excludes gray literature such as technical reports, industrial white papers, or blog posts. These sources often contain practical insights into microservice evolution and verification practices, especially in fast-moving DevOps environments. The absence of those insights may skew the review toward academic prototypes rather than industrial realities.

(v). Quality Assessment Subjectivity: Our QA checklist provided a structured way to evaluate studies; however, scoring inevitably involves interpretation. For instance, assessing whether a method was "clearly reproducible" or whether evidence was "sufficiently empirical" required judgment calls. Minor reporting issues may have led to lower scores for otherwise relevant studies, potentially affecting synthesis.

(vi). Generalizability: Many included studies rely on small-scale case studies, illustrated examples, or academic prototypes. Although these provide valuable insights, they may not generalize large-scale, polyglot, industrial microservice deployments. The lack of industrial-scale validation

 doi:10.20944/preprints202603.1951.v1

limits the external validity of our conclusions. To explicitly recognize these threats, we aim to provide transparency and encourage future work to mitigate them through broader search strategies, inclusion of industrial evidence, and open data practices.

7. Conclusions

This systematic literature review investigated the intersection of microservice evolution and formal verification techniques, guided by four research questions. Our synthesis highlights several important insights:

(i) Formal Methods in Microservices (RQ1): A various number of formal techniques have been applied, including Petri nets for workflow modeling, TLA+ for protocol correctness, SMT solvers for configuration checking, and session types for communication safety. However, most evaluations remain confined to small-scale case studies, with limited evidence of scalability to industrial systems.

(ii) Impact of Evolution (RQ2): The Architectural drift, API versioning, and dependency changes pose significant challenges to verifiability. Few approaches directly address specifications of drift or automated mitigation of protocol violations. This gap underscores the need for techniques that can adapt to formal models as systems evolve.

(iii) Signals for Model Updates (RQ3): The Repository-derived signals such as commits, API diffs, and dependency graphs are beginning to be leveraged to infer or update formal models. While promising, automation remains immature, and integration with tool chains is limited.

(iv) Continuous Verification in CI/CD (RQ4): There exists growing interest in embedding verification into DevOps pipelines, moreover, tool support is fragmented and empirical evidence of industrial adoption is scarce. Therefore, continuous verification remains more of a vision than a widely realized practice.

Alongside these findings, our review concludes that there are two main 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 life development 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 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 behaviors.

Based on the situation, teams may adopt one approach or combine aspects of both. This intrinsic path emphasizes rigor and proactive assurance, while the extrinsic path emphasizes practicality and adaptability. In general, the body of work is emerging, however, fragmented. However, Formal verification for microservice evolution is still in its infancy, with limited industrial-scale validation and little shared tooling or datasets. To promote further this field, suggest a research agenda emphasizing:

- The empirical grounding via large-scale industrial case studies that validate scalability and practicality.
- Through integration with CI/CD pipelines to enable continuous verification as part of everyday DevOps workflows.
- Then, open datasets and reproducible tooling to foster collaboration, benchmarking, and transparency.
- The automation of model updates using repository and runtime signals to keep specifications aligned with evolving systems.
- Inclusive adoption of intrinsic and extrinsic approaches, tailoring verification strategies to organizational maturity, resource availability, and system complexity.

 doi:10.20944/preprints202603.1951.v1

If these gaps 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.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study.

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

References

1. Anisetti, M.; Ardagna, C.A.; Gaudenzi, F.; Damiani, E. A Continuous Certification Methodology for DevOps. *MEDES '19: Proceedings of the 11th International Conference on Management of Digital EcoSystems*, 205–212. https://doi.org/10.1145/3297662.3365827
2. Anisetti, M.; Ardagna, C.; Damiani, E.; Polegri, G. Test-Based Security Certification of Composite Services. *ACM Transactions on the Web* **2018**, *13*(1), 1–43. https://doi.org/10.1145/3267468
3. Biswas, P.; Morgenstern, A.; Antonino, P.O.; Capilla, R.; Nakagawa, E.Y. Continuous Evaluation of Consistency in Software Architecture Models. *ECSA 2023 Proceedings*, 141–149. https://doi.org/10.1007/978-3-031-42592-9_10
4. Boza, E.F.; Abad, C.L.; Narayanan, S.P.; Balasubramanian, B.; Jang, M. A Case for Performance-Aware Deployment of Containers. *WOC '19 Proceedings*, 25–30. https://doi.org/10.1145/3366615.3368355
5. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, Omega, and Kubernetes. *Communications of the ACM* **2016**, *59*(5), 50–57. https://doi.org/10.1145/2890784
6. Cerny, T.; Abdelfattah, A.S.; Yero, J.; Taibi, D. From Static Code Analysis to Visual Models of Microservice Architecture. *Cluster Computing* **2024**, *27*(4), 4145–4170. https://doi.org/10.1007/s10586-024-04394-7
7. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual Understanding of Microservice Architecture. *ACM SIGAPP Applied Computing Review* **2018**, *17*(4), 29–45. https://doi.org/10.1145/3183628.3183631
8. Copei, S.; Schreiter, M.; Zündorf, A. Improving the Implementation of Microservice-Based Systems with Static Code Analysis. *Lecture Notes in Business Information Processing* **2023**, *489*, 31–38. https://doi.org/10.1007/978-3-031-48550-3_4
9. Ferrara, P.; Arceri, V.; Cortesi, A. Challenges of Software Verification. *International Journal on Software Tools for Technology Transfer* **2024**, *26*, 421–430. https://doi.org/10.1007/s10009-024-00765-y
10. Khamespanah, E.; Jaghoori, M.M. 20 Years of Actor Model Checking with Rebeca. *LNCS* **2025**, *15560*, 26–43. https://doi.org/10.1007/978-3-031-85134-6_2
11. Mampage, A.; Karunasekera, S.; Buyya, R. Resource Management in Serverless Computing. *ACM Computing Surveys* **2022**, *54*(11s), 1–36. https://doi.org/10.1145/3510412
12. Merlino, G.; Dautov, R.; Distefano, S.; Bruneo, D. Workload Engineering in Edge–Fog–Cloud Computing. *ACM Transactions on Internet Technology* **2019**, *19*(2), 1–22. https://doi.org/10.1145/3309705
13. Namyar, P.; Ghavidel, A.; Zhang, M.; Madhyastha, H.V.; Ravi, S.; Wang, C.; Govindan, R. ZENITH: Formally Verified Highly Available Control Plane. *SIGCOMM '25 Proceedings*, 409–433. https://doi.org/10.1145/3718958.3750533
14. Shahin, M.; Zahedi, M.; Babar, M.A.; Zhu, L. Adopting Continuous Delivery and Deployment. *EASE '17 Proceedings*, 384–393. https://doi.org/10.1145/3084226.3084263
15. Silva, S.; Pelliccione, P.; Bertolino, A. Self-Adaptive Testing in the Field. *ACM Transactions on Autonomous and Adaptive Systems* **2024**, *19*(1), 1–37. https://doi.org/10.1145/3627163
16. Soares, R.C.; Capilla, R.; dos Santos, V.; Nakagawa, E.Y. Trends in Continuous Evaluation of Software Architecture. *Computing* **2023**, *105*(9), 1957–1980. https://doi.org/10.1007/s00607-023-01161-1
17. Stillwell, M.; Coutinho, J.G. A DevOps Approach to Integration of Software Components. *QUDOS 2015 Proceedings*, 1–6. https://doi.org/10.1145/2804371.2804372
18. Lamport, L. The PlusCal Algorithm Language. *ICALP 2009 Proceedings*, pp. 36–60. https://doi.org/10.1007/978-3-642-03466-4_2

 doi:10.20944/preprints202603.1951.v1

19. Camilli, M. Continuous Formal Verification of Microservice-Based Process Flows. *ECSA 2020 Proceedings*, pp. 420–435. https://doi.org/10.1007/978-3-030-59155-7_31
20. Netflix Conductor Documentation. Available online: https://github.com/Netflix/conductor (accessed on 12 December 2025).