*future internet*

MDPI

*Article*

# Highly Efficient Software Development Using DevOps and Microservices: A Comprehensive Framework

**David Barbosa [1], Vítor Santos [2], Maria Clara Silveira [3], Arnaldo Santos [4] and Henrique S. Mamede [4,*]**

[1] NOVA Information Management School, Universidade Nova de Lisboa Campus de Campolide, 1070-312 Lisboa, Portugal; 20221663@novaims.unl.pt
[2] MagIC & NOVA NOVA Information Management School, Universidade Nova de Lisboa Campus de Campolide, 1070-312 Lisboa, Portugal; vsantos@novaims.unl.pt
[3] Instituto Politécnico da Guarda, Av. Dr. Francisco Sá Carneiro, 50 6300-559 Guarda, Portugal; mclara@ipg.pt
[4] INESC TEC & Universidade Aberta, Rua da Escola Politécnica, 1269-001 Lisboa, Portugal; arnaldo.santos@uab.pt
* Correspondence: jose.mamede@uab.pt

**Abstract**

With the growing popularity of DevOps culture among companies and the corresponding increase in Microservices architecture development—both known to boost productivity and efficiency in software development—an increasing number of organizations are aiming to integrate them. Implementing DevOps culture and best practices can be challenging, but it is increasingly important as software applications become more robust and complex, and performance is considered essential by end users. By following the Design Science Research methodology, this paper proposes an iterative framework that closely follows the recommended DevOps practices, validated with the assistance of expert interviews, for implementing DevOps practices into Microservices architecture software development, while also offering a series of tools that serve as a base guideline for anyone following this framework, in the form of a theoretical use case. Therefore, this paper provides organizations with a guideline for adapting DevOps and offers organizations already using this methodology a framework to potentially enhance their established practices.

**Keywords:** DevOps; microservices; information systems; software; efficiency; agile; team collaboration

## 1. Introduction

DevOps offers numerous benefits for an organization and can be briefly described as a set of cultural philosophies, practices, and tools that aim to improve an organization's ability to deliver high-quality applications and services more quickly [1,2].

Modern software organizations adopt DevOps to shorten release cycles, increase automation across continuous integration (CI)/continuous delivery (CD) pipelines, and improve collaboration between development and operations. In practice, DevOps delivers measurable benefits—faster releases, earlier defect detection through automated tests, and efficiency gains across teams and infrastructure [3].

Yet empirical challenges persist: steep learning curves, scarce formal training, communication gaps, and resistance to organizational change regularly undermine DevOps rollouts. Only a small minority of companies report formal DevOps training, and coordination friction between Dev and Ops remains a recurring obstacle. These documented



Academic Editor: Izzat Alsmadi

Received: 25 November 2025
Revised: 5 January 2026
Accepted: 9 January 2026
Published: 14 January 2026

limitations are cultural and structural, not tooling alone, and often surface even in organizations that have adopted CI/CD.

Crucially, DevOps, by itself, does not remove architectural coupling that impedes independent deployment, scaling, and team autonomy. This article aims to document the view—also reflected in the literature—that not all architectures fit DevOps equally well; teams frequently prefer Microservices (MS) over monoliths to reduce conflicts with continuous development practices and to manage complexity at scale. MS decomposes an application into independently deployable services aligned to business capabilities, thereby addressing a key issue DevOps alone cannot solve: tightly coupled codebases that bottleneck release cadence and domain autonomy [4].

Industry evidence further motivates the joint adoption. A large-scale survey reports that 63% of enterprises use MS architectures, indicating momentum behind architectural decoupling strategies in real projects; at the same time, MS's relative newness (≈8% of developers with >5 years' experience) signals ongoing gaps in consolidated practice and guidance. This suggests a need for a prescriptive, empirically informed way to combine DevOps and MS rather than treating them as independent adoptions [5].

However, adopting MS introduces new operational complexity—more moving parts, inter-service dependencies, consistency risks, and delivery overhead—precisely where DevOps' automation and continuous delivery capabilities are most needed. The research highlights these trade-offs and emphasizes investing in continuous-deployment practices and teams to manage the added complexity. In recent software development, building solutions with an MS architecture has become a trend, as MS benefits from the monitoring capabilities of DevOps [6,7]. But it is still unknown to what extent software development efficiency truly benefits from the combination of DevOps and MS.

MS/DevOps are therefore complementary: DevOps addresses process and pipeline efficiency, while MS provides the architectural modularity needed for autonomous teams and independent releases; together they enable scalable, resilient delivery when properly integrated [6,7].

Against this backdrop, the research gap is not whether DevOps or MS is useful, but how to operationalize their combination—i.e., how to translate scattered practices into a coherent, repeatable framework that teams can apply end-to-end. The paper distills this gap into three research questions about (1) creating actionable guidelines for MS development using DevOps practices, (2) instantiating those guidelines with a concrete toolchain, and (3) understanding them as DevOps best practices tailored to MS.

Despite the widespread adoption of DevOps and the growing shift to microservices, organizations still lack actionable, end-to-end guidance for operationalizing DevOps in microservices-oriented settings in a way that is repeatable across teams and auditable over time. Existing guidance typically emphasizes principles, culture, and tooling but provides limited clarity on how to translate these ideas into explicit lifecycle contracts (activities, artifacts, gates, roles) and runtime evidence-based governance (e.g., SLO-driven promotion and rollback). This gap matters because "wrong-fit" architectural choices and fragmented DevOps adoption can inflate delivery costs, increase operational risk, and erode reliability—undermining both business agility (time-to-market) and engineering performance (quality and resilience).

To address this gap, the study is guided by the following research questions:

- RQ1: What socio-technical and operational conditions are consistently required to implement DevOps effectively in microservices-oriented organizations?
- RQ2: How can these conditions be operationalized into a coherent, phase-based framework that specifies lifecycle contracts (activities, artifacts, gates), responsibilities, and evidence requirements?

- RQ3: To what extent does applying the proposed framework improve delivery and reliability performance in practice (e.g., DORA metrics and SLO-based service health indicators) compared to baseline or control conditions?

This paper makes four contributions:

- A phase-based DevOps–microservices implementation framework (artifact) that consolidates dispersed DevOps and microservices guidance into explicit lifecycle contracts, including activities, required artifacts, quality gates, and responsibility boundaries.
- A practical operationalization layer (toolchain-agnostic but auditable) that shows how the framework can be instantiated through standard toolchain capabilities (CI/CD, testing layers, observability, incident management) and how evidence is captured to support repeatable governance.
- An evidence-based governance approach aligned with DevOps/SRE standards, integrating reliability signals (SLIs/SLOs and error-budget thinking) into release decisions (promotion/rollback) so that delivery speed is balanced with service resilience.
- An evaluation design and empirical assessment that benchmarks framework adoption using recognized performance constructs (e.g., DORA metrics and service health indicators), enabling comparison across periods/teams and supporting replication.

This makes clear the contribution of this research: Building on a comprehensive review and expert input, a framework is proposed and refined to align DevOps phases (planning, CI/CD, monitoring, learning) with the MS lifecycle and team topology, including exemplar tools per phase. The framework is iteratively validated through use cases and expert interviews, then revised to enhance clarity and cyclic flow. It is designed to (i) preserve DevOps' documented benefits, (ii) mitigate MS-specific delivery risks, and (iii) provide a practical adoption path organizations can follow.

DevOps alone does not resolve architectural coupling and team autonomy constraints that slow releases in complex systems; Microservices provide that modularity but increase operational complexity, which DevOps is expressly suited to manage. Our framework operationalizes this complementarity with concrete, validated guidance, addressing an identified and persistent gap between principle and practice.

Implementing it is definitely not easy, as it requires effort from the entire organization. This means a significant cultural shift and a substantial technical challenge, necessitating a strong commitment from everyone [8,9].

The structure of the paper is as follows. In Section 2, the methodology is described; in Section 3, the background research is presented. In Section 4, the proposal is described. The demonstration in Section 5 shows how the proposed framework can be applied in practice. Section 6 presents the proposal evaluation, and Section 7 contains the conclusions.

## 2. Methodology

This study followed the Design Science Research (DSR) paradigm to design, instantiate, and evaluate an artifact intended to address a practical problem in software engineering and information systems: the lack of actionable, end-to-end guidance for operationalizing DevOps practices in microservices-oriented settings. DSR was appropriate because the primary contribution of the paper is an artifact (the proposed framework), and the study additionally reports demonstration and empirical evaluation of its utility and effects [10].

### *2.1. DSR Activities as Executed in This Study*

Rather than providing a generic description of DSR, this section summarizes how DSR was applied in this study and where each activity is reported in the manuscript. Table 1 provides a direct mapping from DSR activities [11] to the concrete work performed and the corresponding sections.

**Table 1.** DSR mapping in this study.

| DSR Activity | What We Did in This Study (Executed Actions) | Where Reported |
| --- | --- | --- |
| Problem identification and motivation | Identified the practical and research gap: organizations adopt DevOps and microservices but lack a repeatable, auditable, end-to-end operating model linking lifecycle activities, artifacts, gates, roles, and runtime governance. Formulated research questions RQ1–RQ3. | Section 1 |
| Requirements/objectives of a solution | Derived design requirements and objectives from prior empirical findings and reference models (DevOps, microservices, DORA, SRE), focusing on operationalization, governance-by-evidence, roles/ownership, and toolchain-agnostic traceability. | Sections 3.2–3.5 and Section 4.1 (O1–O4) |
| Design and development (artifact creation) | Designed the framework as a layered artifact (principles/outcomes; decision matrices/guardrails; phase contracts; roles/RACI; metrics/evidence model). Specified phase inputs/activities/outputs, quality gates, and evidence requirements (DORA + SLO/error-budget based governance). | Section 4 (including Sections 4.1–4.3) |
| Iterative refinement | Refined the artifact based on structured expert feedback (e.g., naming/sequencing adjustments and integration-testing placement options), improving clarity and applicability. | Section 6.1 (expert cross-check) and corresponding updates described in Section 4 |
| Demonstration | Instantiated the framework end-to-end in a realistic microservice delivery toolchain (e.g., CI/CD, containerization, orchestration, observability) to demonstrate feasibility and traceability of phases, gates, and evidence capture. | Section 5 |
| Evaluation | Evaluated the artifact through (i) expert-based validity cross-check and (ii) a quasi-experimental empirical assessment of delivery performance and reliability outcomes using established constructs (DORA metrics and SLO-based service health indicators). | Section 6 (Sections 6.1–6.3) |
| Communication | Consolidated results, implications, limitations, and replication guidance to support reuse and future evaluation. | Section 7 |

### *2.2. Consistency Between DSR, Demonstration, and Evaluation*

In this study, the DSR artifact (the proposed framework) was not only described but also instantiated and evaluated. The demonstration (Section 5) served to show that the framework's phases, artifacts, gates, and evidence model can be enacted using standard tooling. The evaluation (Section 6) then assessed the framework using recognized delivery and reliability constructs (DORA metrics and SLO-driven indicators), aligning the study's evidence with the framework's core "governance by objective signals" principle. Together, these steps ensure that the methodology reflects the work actually performed and that the evaluation evidence is consistent with the described research approach.

## 3. Theoretical Background

This section explores the current state of the art regarding DevOps and Microservices, emphasizing how both paradigms aim to enhance software delivery speed, quality, and reliability. DevOps integrates automation practices such as Continuous Integration and Delivery, Infrastructure as Code (IaC), and continuous monitoring, while promoting collaboration between development and operations teams.

Empirical studies consistently show shorter release cycles and earlier defect detection in DevOps-driven organizations. However, major challenges remain, including steep learning curves, limited formal training (offered by only about 5% of companies), and cultural or structural barriers that hinder large-scale adoption.

In parallel, Microservices have emerged as a dominant software architecture that decomposes systems into small, independently deployable components aligned with business capabilities. This modularity enables scalability and team autonomy but also introduces new operational complexity, from managing inter-service communication to ensuring data consistency and resilience.

These challenges highlight the need for strong DevOps practices to manage automation, observability, and governance in MS environments. The literature agrees that DevOps and MS complement one another but also reveals that their combined operationalization is under-specified and difficult to reproduce across organizations.

Literature currency (2025 update). To ensure the manuscript reflects the state of the art for a 2025 submission, we performed a targeted update of the evidence base (database search + forward snowballing) focusing on three topics directly connected to our framework: (i) resilience validation and governance in Kubernetes/service-mesh environments, (ii) compatibility/version orchestration and contract-based assurance for evolving services, and (iii) microservices testing approaches and their fragmentation across tools and levels. The updated evidence reinforces the paper's central motivation—namely, that effective DevOps-at-scale for microservices requires explicit lifecycle governance (contracts, gates, decision rights) beyond tool adoption—and is now incorporated in Sections 3.2 and 4.2 with Refs. [12–15].

### 3.1. Where the Field Stands

DevOps consolidates automation (CI/CD, IaC, continuous testing/monitoring) and cross-functional collaboration to increase deployment frequency and reliability [16,17]. Empirical and practice-oriented sources consistently report shorter release cycles, earlier defect detection via automated tests, and efficiency gains across teams and infrastructure when these practices are institutionalized [16–18].

At the same time, multiple studies highlight structural impediments that persist despite CI/CD adoption: steep learning curves, limited formal training, and coordination frictions between development and operations. Notably, a mixed-methods study reports only ~5% of companies offering formal DevOps training (with most learning occurring informally/online), which helps explain "islands of automation" and uneven practice maturity [9,18].

In parallel, MS have become a leading architectural style, enabling independent deployment, scalability, and team autonomy by decomposing systems into service-level units aligned with business capabilities [19,20]. Adoption is widespread: a sector survey found 63% of enterprises using MS, while ~50% were uncertain about impacts on revenue processes, underscoring a persistent gap between architectural change and business outcomes [20,21].

The article documents practitioners' view that DevOps alone does not remove architectural coupling (e.g., monolith-scale integration bottlenecks) that slows independent releases; conversely, MS raises operational complexity (service choreography, failure propagation, versioning), precisely where robust DevOps/automation practices are most needed [18].

### 3.2. What Current Approaches Still Miss (and Why It Matters)

Building on the gap articulated in Section 1, prior empirical and synthesis work points to recurring shortcomings that prevent organizations from turning DevOps and microservices principles into a coherent, repeatable operating model. Rather than "restating" the gap, we use these findings to derive design requirements that directly inform the framework presented in Section 4.

- Organizational capability gaps and uneven maturity. Empirical studies repeatedly show that DevOps success depends on organizational capabilities (skills, collaboration patterns, and shared ownership), yet these capabilities are often unevenly distributed across teams, resulting in fragmented maturity and "islands of automation" that reduce reproducibility and increase operational brittleness [9,18]. Design requirement DR1: the framework must make responsibilities explicit (e.g., ownership and RACI patterns) and include capability-building guidance that reduces variability across teams.
- Architectural–operational misalignment. Microservices promise modularity and autonomy but introduce substantial operational and governance demands. Organizations frequently struggle to align architecture decisions (service boundaries, dependency management, versioning, deployment topology) with day-to-day DevOps practices and release governance, creating coordination bottlenecks and unreliable delivery outcomes [7,20]. Design requirement DR2: the framework must link architectural choices to concrete lifecycle practices (e.g., contract testing, release patterns, and operational readiness criteria), rather than treating architecture as independent from DevOps execution.
- Toolchain sprawl and weak integration patterns. The DevOps literature documents heterogeneous toolchains and inconsistent pipeline implementations that increase cognitive load and complicate knowledge transfer across teams, particularly at scale [17,22,23]. In microservices settings, this fragmentation typically worsens because teams adopt tools independently, undermining standardization of quality gates and traceability. Design requirement DR3: the framework must define toolchain-agnostic integration patterns (what evidence must be produced and where it is captured), enabling standard governance without prescribing a single vendor stack. This fragmentation is also visible in the microservices testing literature, where 2025 evidence shows that techniques and practices remain scattered by testing level and objective, with many proposals evaluated only in early-stage settings—making end-to-end repeatability and governance difficult in practice [13].
- Observability and runtime governance remain under-specified. While monitoring and observability are frequently recommended, prior work shows persistent difficulty in turning telemetry into actionable governance signals, such as explicit promotion/rollback criteria or incremental adoption pathways that teams can apply consistently [6]. Design requirement DR4: the framework must embed runtime evidence into delivery decisions (e.g., SLO-driven gates, clear rollback triggers, and an evidence register) so that reliability is governed continuously—not only assessed after incidents. Recent work on resilience validation in Kubernetes further illustrates that operational resilience outcomes depend on explicit traffic-management and observability configurations (e.g., service-mesh governance) rather than deployment automation alone [15].
- Business-process and value-stream blind spots. Microservices and DevOps adoption is often described in technical terms, but empirical evidence suggests that organizations still struggle to connect engineering improvements to business value (e.g., end-to-end lead time and customer impact), and measurement often remains narrowly operational [21]. Design requirement DR5: the framework must encourage value-stream visibility by connecting delivery performance and reliability indicators to business outcomes in a way that supports prioritization and continuous improvement.

Together, these requirements motivate the framework's core design choices in Section 4—namely, phase-based lifecycle contracts with explicit artifacts and gates, standardized evidence collection, responsibility clarity, and SLO-informed governance that scales across teams and architectures.

### 3.3. Points of Consensus—And Contention—In Prior Work

The literature converges on DevOps–MS complementarity: DevOps improves flow, automation, and release reliability, while MS provides modularity and team autonomy, enabling independent build–test–deploy cycles [7,17,19].

In terms of contention/gaps, the following is pointed out by the literature:

- Operationalization is under-specified—Mapping studies describe DevOps and MS practices, but offer limited prescriptive guidance on binding DevOps phases (plan–build–release–operate–learn) to MS lifecycle decisions (service boundaries, versioning, deployment topology, rollback/progressive delivery) in a repeatable way [7]. In practice, teams assemble bespoke toolchains that are hard to replicate across contexts.
- Skills and organizational capacity remain bottlenecks. Even with CI/CD pipelines, studies show that training deficits and coordination issues limit adoption at scale. Organizations rarely systematize capability development, leading to uneven outcomes [9,18,24].
- Architecture trade-offs are context-dependent—Not all systems benefit from MS; authoritative sources warn that well-structured monoliths can be more economical in certain stages [2,25]. Review papers seldom provide decision support (criteria, thresholds) to navigate Monolith to MS transitions, leaving a practical guidance gap.
- Runtime observability → release governance link is weak—While tools exist for logging/metrics/tracing, the literature under-specifies how runtime Service-Level Objective (SLO) inform automated release decisions (e.g., canary gates, error-budget policies) in MS environments—again pushing teams to ad hoc solutions [17].
- Business outcome alignment is patchy—Surveys show uncertainty about MS's impact on value streams; DevOps studies often emphasize technical lead-times over explicit business KPIs, making it hard to attribute value to combined DevOps–MS adoptions [18,21].

Moreover, "DevOps fixes everything" is an overstatement: cultural change and skills development are recurrent blockers; conversely, "MS is always better than monoliths" ignores contexts where a well-structured monolith is faster to deliver and cheaper to run [2,25]. The field needs decision support that acknowledges such trade-offs instead of universal prescriptions. The agreement is that DevOps and MS work best together; the disagreement/absence lies in how to operationalize that union so it is repeatable, observable, and tied to business value.

### 3.4. Related Frameworks and Reference Models

The DevOps and microservices (MS) bodies of knowledge provide multiple frameworks, reference models, and practitioner "playbooks" that collectively describe what organizations should aim for, but often leave open how to operationalize these ideas in a repeatable and auditable way—especially when the target architecture is microservices-based. In this subsection, we synthesize the most relevant reference models that underpin DevOps and MS adoption and clarify where existing approaches fall short in providing actionable, end-to-end guidance for combined DevOps–MS implementations.

A first cluster of reference models focuses on DevOps as a socio-technical system that combines automation and cultural change. Early conceptualizations position DevOps as a "software revolution" driven by tighter collaboration between development and operations and by continuous feedback loops [8,16]. This line of work is further operationalized through continuous delivery (CD) and continuous integration (CI) patterns, which define the technical backbone for frequent, reliable releases via automation of build, test, and deployment activities [26]. Complementing CD, the concept of continuous software engineering (CSE) frames DevOps as an organizational capability for releasing software

continuously while maintaining quality and governance, supported by practices such as continuous testing, monitoring, and disciplined configuration management [17]. Taken together, these models establish that DevOps maturity is not solely determined by tool adoption, but by the institutionalization of practices that reduce lead times, improve release reliability, and enable rapid learning cycles [12,13,26].

A second cluster provides evidence-based capability and measurement frameworks. The "Accelerate/DORA" research stream consolidates a set of capabilities (e.g., trunk-based development, continuous delivery, test automation, loosely coupled architecture, monitoring, and a generative culture) and links them to organizational performance outcomes using large-scale empirical studies [27]. These contributions are important because they translate DevOps into measurable performance constructs (e.g., deployment frequency, lead time, change failure rate, and mean time to restore), enabling benchmarking and continuous improvement. In parallel, site reliability engineering (SRE) offers a reference model for operational excellence in digital services, emphasizing error budgets, service level objectives (SLOs), and reliability-driven governance of change [28]. For DevOps–MS adoption, SRE is particularly relevant because microservices increase operational complexity (more services, more deployments, more interdependencies), making runtime reliability and observability central to sustainable delivery [28–30]. However, while DORA and SRE provide robust principles and measurement targets, they do not in themselves prescribe an explicit "phase-by-phase" implementation blueprint tailored to different architectural options (e.g., monolith vs. modular monolith vs. microservices) and varying organizational constraints.

A third cluster addresses microservices architecture reference models and patterns. Microservices are commonly positioned as an architectural style enabling independent evolution, deployment, and scaling of services, but the literature is consistent in highlighting non-trivial trade-offs: service decomposition, distributed data management, network communication, versioning, testing complexity, and the need for strong automation and governance [19,20]. Microservices patterns further detail practical architectural building blocks (e.g., decomposition and integration strateg31ies) and highlight the risks of premature decomposition, coordination overhead, and the operational cost of distributed systems [25]. Importantly, microservices adoption is frequently framed as a transformation journey rather than a one-time architectural decision, and "drivers and barriers" studies show that organizational readiness, skills, and operational maturity often determine outcomes as much as technical design choices [20]. Nevertheless, MS reference models usually provide limited explicit mapping between architectural decisions and the concrete DevOps pipeline contracts, quality gates, and runtime governance mechanisms needed to keep delivery repeatable and resilient at scale.

A fourth cluster consists of integrative studies and mapping reviews connecting DevOps and microservices. Systematic mapping work on DevOps emphasizes recurring practice categories and identifies common adoption challenges (e.g., skills shortages, cultural resistance, toolchain complexity, and uneven maturity across teams) [7,9]. Similarly, recent synthesis work on the interplay between DevOps and software architecture highlights that architectural choices can either enable or constrain continuous delivery, and that large-scale adoption frequently results in "islands of automation" when automation is not standardized and governed across the organization [9,18]. For microservices specifically, architecting reviews describe how MS principles align well with DevOps goals (autonomy, independent releases, and scalability) while also introducing new operational risks that require systematic controls (observability, runtime governance, and coordinated testing strategies) [19,20]. Additionally, more recent DevOps–MS lifecycle analyses and empirical contributions reinforce that DevOps practices are necessary but not sufficient; microservices

benefit most when DevOps is operationalized as a consistent lifecycle with explicit quality gates, clear responsibilities, and standardized toolchain patterns across teams [6,7,19].

Across these reference models, several converging insights are clear. First, DevOps is best understood as a capability system—a combination of practices, governance, and automation—rather than a toolset [16,17,27]. Second, microservices are an architectural enabler for independent delivery but simultaneously increase operational and governance demands, raising the importance of observability and reliability engineering [19,20,28]. Third, DORA and SRE provide strong performance and reliability targets, yet their translation into concrete, stepwise implementation guidance is often left to organizations, resulting in heterogeneous and difficult-to-reproduce approaches [27,28]. Finally, mapping studies confirm persistent organizational problems—toolchain sprawl, fragmented maturity, unclear accountability, and limited operationalization—that weaken the sustainability of DevOps–MS adoption at scale [7,9,18].

To make these relationships explicit and to motivate the need for an integrated framework, Table 2 summarizes representative reference models and highlights the practical gap: most sources explain what to achieve (continuous delivery, reliability, performance outcomes) but provide limited guidance on how to implement repeatable lifecycle contracts and governance mechanisms across diverse architectural contexts.

**Table 2.** Representative frameworks and reference models related to DevOps and microservices.

| Reference Model/Stream | Primary Focus | What It Contributes | What Is Typically Under-Specified for DevOps–MS Adoption |
|---|---|---|---|
| DevOps conceptual models [8,12] | Culture + collaboration + automation | Defines DevOps as a socio-technical approach; highlights feedback loops and shared responsibility | Concrete operational "phase contracts", roles/RACI, and standardized toolchain patterns across teams |
| Continuous Delivery/CI patterns [26] | Release automation | Practical pipeline ideas (build/test/deploy automation), reliability through automation | Explicit mapping to MS-specific concerns (service boundaries, distributed testing, runtime governance) |
| Continuous Software Engineering [13] | Organizational capability for continuous delivery | Frames continuous delivery as an organizational system with governance and engineering discipline | Stepwise guidance to operationalize CSE across heterogeneous architectures and varying maturity levels |
| DORA/Accelerate [23] | Capabilities + performance measurement | Evidence-based capability model and performance metrics for benchmarking | Prescriptive lifecycle blueprint linking architectural options to concrete pipeline gates and governance |
| SRE reference model [24] | Reliability governance (SLOs/error budgets) | Operationalizes reliability as a first-class goal; governs change via SLOs and error budgets | How to integrate SRE governance into DevOps lifecycle phases (e.g., promotion rules, rollback criteria) |
| Microservices architecting reviews [15,16] | MS design trade-offs and adoption conditions | Identifies benefits and challenges (decomposition, distributed data, testing, operations) | Practical mapping from MS design decisions to DevOps pipeline structure, observability requirements, and organizational roles |
| DevOps mapping/adoption studies [7,9,14] | Empirical patterns and challenges | Synthesizes recurring challenges (skills, culture, toolchain complexity, "islands of automation") | Reusable, phase-based implementation playbook that reduces ad hoc adoption and improves reproducibility |

### 3.5. Empirical Evidence on DevOps and Microservices Adoption Challenges

Empirical research on DevOps and microservices (MS) adoption spans systematic mapping and review studies, surveys, and a smaller number of in-depth industrial case studies. Collectively, this evidence indicates that DevOps and MS are frequently adopted together (or sequentially) as part of broader modernization initiatives, but the combined adoption introduces recurring socio-technical challenges that are not fully addressed by high-level reference models. In this subsection, we synthesize the empirical evidence around (i) typical adoption contexts, (ii) commonly reported challenges, (iii) reported outcomes and performance signals, and (iv) methodological limitations in the existing body of work.

#### 3.5.1. Adoption Contexts (Industry, Organization Size, and Transformation Setting)

Across the literature, DevOps and MS adoption is most often studied in software-intensive organizations (e.g., digital service providers, enterprise IT units, and platform teams) and in large or mid-to-large enterprises where scale and coordination costs motivate architectural modularization and delivery automation [7,19,20]. Industry surveys also suggest that microservices and associated modernization approaches have moved into mainstream enterprise agendas, with adoption reported across a broad set of organizations beyond "digital natives" [21]. At the same time, systematic mapping work emphasizes that published empirical evidence tends to over-represent organizations with sufficient engineering capacity to run CI/CD pipelines and manage distributed systems—while smaller firms, highly regulated environments, and non-IT-heavy sectors are less consistently represented or described in comparable detail [7,19,20].

In addition, the transformation setting is typically characterized by at least one of the following drivers: (a) the need to increase release frequency and reduce lead time, (b) the need to decouple legacy systems into independently deployable services, (c) cloud migration and containerization initiatives, and/or (d) governance pressure to improve reliability and operational control at scale [18–20,27,28]. However, context descriptions vary substantially between studies, and many papers provide limited information on organization size, team topology, and baseline maturity—making cross-study comparability difficult [7,19,20].

#### 3.5.2. Main Reported Adoption Challenges

Empirical studies converge on a set of recurring challenge categories that become more pronounced when DevOps practices are applied to microservices-based systems.

(a) Skills gaps and capability building

A consistent theme is that successful DevOps–MS adoption requires broad, cross-functional skills spanning software engineering, cloud-native operations, automation, and security. Mapping studies and microservices adoption research emphasize the difficulty of acquiring these skills at scale and the frequent lack of structured training approaches, which contributes to uneven practice maturity and "islands of automation" across teams [7,20]. Survey-based work further highlights the importance of DevOps culture and organizational capability as determinants of outcomes, suggesting that tool adoption alone does not translate into sustained performance improvements [24,27].

(b) Coordination, ownership, and organizational boundaries

Microservices increase the number of independently evolving components, which amplifies the need for clear service ownership, stable interfaces, and well-defined operational responsibilities. Empirical microservices literature frequently reports coordination issues (e.g., inter-team dependencies, interface versioning, cross-service incident handling) and the organizational friction that arises when autonomy is not matched by governance

mechanisms and accountability structures [19,20]. DevOps mapping studies similarly report that organizational silos and ambiguous responsibility boundaries hinder large-scale adoption, even when CI/CD tooling exists [7].

(c) Testing complexity and quality assurance across distributed services

Testing challenges are repeatedly emphasized in microservices architecting and adoption studies, including the difficulty of maintaining end-to-end test confidence across many services, environments, and versions [19,20]. Compared to monolithic systems, microservices introduce more failure modes and integration points (network calls, distributed data), requiring layered testing strategies (unit, contract, integration, end-to-end) and disciplined release gating [19]. Empirical DevOps studies also note that insufficient automation or inconsistent testing practices directly contributes to brittle pipelines and unreliable releases—particularly in multi-team environments [7,18].

(d) Observability, monitoring, and runtime governance

A major operational challenge in DevOps–MS settings is achieving sufficient observability (metrics, logs, traces) to support rapid diagnosis, controlled releases, and reliability improvements. Recent evidence focusing on monitoring in DevOps and microservices highlights fragmentation of monitoring tools, gaps in standardization, and difficulty turning telemetry into actionable governance signals [6]. This aligns with the SRE perspective, which frames reliability governance through explicit objectives (SLOs) and error budgets; however, empirical adoption often struggles to institutionalize these practices consistently across teams [28]. As a result, organizations may collect telemetry but still lack a repeatable mechanism for promotion/rollback decisions tied to reliability evidence [6,28].

(e) Toolchain integration and operational fragmentation

Toolchain sprawl is one of the most frequently reported practical barriers. Mapping work on DevOps adoption documents heterogeneous tooling, inconsistent pipeline definitions, and integration issues between CI/CD, infrastructure automation, testing, and monitoring stacks—leading to non-reproducible delivery processes and high cognitive load for engineers [7]. In microservices contexts, this fragmentation becomes more acute because each service team may select tools independently, increasing interoperability and governance challenges [6,19]. Empirical studies consequently emphasize the need for reference toolchains and integration patterns to reduce duplication and improve repeatability across teams [6,7,18].

#### 3.5.3. Reported Outcomes (DORA Metrics, Reliability, and Quality)

Reported outcomes in the empirical literature are commonly expressed in terms of delivery performance, reliability, and quality—though measurement practices vary widely.

(a) Delivery performance and DORA metrics

The DORA/Accelerate stream provides the most widely used evidence-based measurement framework linking DevOps capabilities to performance outcomes, operationalizing delivery performance via deployment frequency, lead time for changes, change failure rate, and mean time to restore service [27]. While many studies cite these metrics as targets or benchmarks, fewer provide consistent, longitudinal measurement in real organizational settings. Still, the dominant empirical narrative is that organizations adopting mature DevOps capabilities tend to report faster delivery and improved stability—especially when automation and loosely coupled architectures reduce coordination and deployment constraints [27].

(b) Reliability and incident response

Microservices adoption research emphasizes that reliability can improve through isolation and independent scaling, but also warns that distributed architectures often increase operational complexity and the likelihood of emergent failures (e.g., cascading issues,

dependency failures) if observability and runtime controls are insufficient [19,20]. The SRE reference model provides a reliability-focused governance lens (SLOs/error budgets) that is frequently recommended as a complement to DevOps in complex production environments, although empirical evidence of systematic SRE institutionalization remains uneven across published studies [28]. Monitoring-centered synthesis work highlights that organizations frequently invest in observability tooling but struggle with consistent usage patterns and governance—limiting realized reliability benefits [6].

(c) Quality, security, and maintainability outcomes

Empirical work commonly reports quality-related outcomes indirectly via reduced defects, improved testing automation, improved maintainability, and fewer post-release incidents, though measurements are often self-reported or context-specific [7,18–20,24]. Microservices architecting research is particularly clear that maintainability benefits depend on disciplined service boundaries and interface stability; poor decomposition choices can increase complexity and degrade quality outcomes [19,20]. Taken together, the empirical literature indicates that the most credible outcomes are achieved when DevOps practices are paired with architectural modularity and supported by standardized engineering controls (testing strategy, pipeline governance, observability standards, and clear ownership) [18–20,27,28].

#### 3.5.4. Methodological Limitations in Prior Work

Despite broad agreement on challenge categories, the current evidence base exhibits limitations that constrain generalization and make it difficult to derive prescriptive guidance.

Over-reliance on narrative synthesis and heterogeneous study designs

Systematic mapping and review studies highlight that the empirical body is fragmented across case studies, experience reports, and surveys with inconsistent context reporting and outcome measurement [7,19]. This limits the ability to compare findings across industries and organizational sizes.

Self-reported outcomes and limited causal inference

Surveys and practitioner reports frequently rely on perceived improvements rather than objective operational metrics, and many studies lack longitudinal baselines or controls that would support causal claims [7,18,21,24,27]. Even when DORA metrics are referenced, the measurement approach and observation period are often not described in sufficient detail to support replication [27].

Context under-specification (industry, scale, baseline maturity)

Many studies provide limited details on team topology, legacy constraints, regulatory conditions, and initial DevOps maturity, which are critical moderators of adoption outcomes [7,19,20]. As a result, recommendations can become generic and may not translate across different organizational contexts.

Publication bias toward successful adoptions and “best-case” organizations

The published literature tends to emphasize successful transformations and organizations capable of sustained investment in tooling and skills, while failure cases, rollback scenarios, and long-term maintenance costs are less systematically documented [7,19,20]. This contributes to overly optimistic narratives and weakens the practical guidance for organizations with constrained resources.

Summary implication: Empirical evidence consistently identifies recurring DevOps–MS adoption challenges—skills gaps, coordination/ownership issues, distributed testing complexity, observability governance, and toolchain integration fragmentation—while reporting outcomes primarily in terms of delivery performance and reliability, often framed through DORA and SRE concepts [6,7,18–20,24,27,28]. However, methodological limitations (heterogeneous designs, self-reported outcomes, under-specified contexts, and limited

longitudinal evidence) restrict the prescriptive value of existing work and motivate the need for a framework that provides repeatable lifecycle contracts, governance mechanisms, and evidence-based decision criteria across diverse organizational settings [7,18–20,27,28].

### *3.6. Implications for This Research*

The synthesis above motivates a framework that goes beyond restating principles and instead provides concrete, reusable guidance to close four specific gaps:

1. DevOps ↔ MS lifecycle mapping. Specify how each DevOps phase (plan, code, build, test, release, operate, learn) maps to MS lifecycle decisions (domain-driven boundaries, API versioning/contract testing, deployment topology, progressive delivery/rollback, Site Reliability Engineering (SRE) guardrails), with checklists and decision criteria to avoid ad hoc adoption [7].
2. Minimal, interoperable toolchains. Recommend reference toolchains per phase (e.g., Version Control System (VCS)/branching, CI runners, artifact repos, IaC, container orchestration, contract testing, tracing, SLO dashboards), focusing on interfaces and integration patterns so that pipelines are repeatable across teams and contexts [23,26].
3. Observability-driven release governance. Embed SLOs/error-budgets, distributed tracing, and canary gates into delivery policies so that runtime evidence automatically informs promote/rollback decisions—closing the gap between "monitoring" and governed continuous delivery [17].
4. Adoption playbook and capability building. Provide organizational guidance—roles, training pathways, and coordination mechanisms—to tackle the empirically observed skills/training deficit and DevOps alignment issues [9,18,24].

The research already surfaces these needs—toolchain sprawl, skills gaps, and the call for structured guidance—by cataloging tools and challenges and by arguing that not all architectures fit DevOps equally well [18]. The contribution of this paper is to operationalize that insight into a coherent framework, later instantiated and refined through the use case and expert feedback reported in the validation sections.

While the DevOps and microservices literature provides a rich set of principles, best practices, and reference models, most contributions emphasize either (i) cultural and organizational change, (ii) delivery automation practices, (iii) performance measurement, or (iv) architectural patterns in isolation. In contrast, the framework proposed in this paper integrates these streams into a single operational blueprint that specifies what must be performed, by whom, with what evidence, and under which quality gates across the full lifecycle of microservices delivery and operation. Table 3 summarizes how our framework relates to prominent DevOps/SRE and microservices reference models and clarifies the specific novelty of our contribution.

**Table 3.** Comparison of the proposed framework with prominent DevOps/SRE and microservices reference models.

| Reference Model/Stream | Primary Emphasis | What It Provides | What Is Typically Under-Specified for DevOps + Microservices Adoption | How the Proposed Framework Complements/Extends It |
|---|---|---|---|---|
| DevOps conceptual models and practitioner guidance [8,16] | Culture + collaboration + automation ("CAMS") | Rationale for DevOps, shared responsibility, continuous learning | Concrete "operational contracts" (phase deliverables, explicit gates, evidence requirements), and consistent cross-team governance | Converts principles into phase-based lifecycle contracts with explicit artifacts, gate criteria, and accountability structures |

**Table 3.** *Cont.*

| Reference Model/Stream | Primary Emphasis | What It Provides | What Is Typically Under-Specified for DevOps + Microservices Adoption | How the Proposed Framework Complements/Extends It |
|---|---|---|---|---|
| Continuous Delivery/CI practices [30] | Delivery automation and pipeline discipline | Implementation patterns for automated build/test/deploy | Systematic integration of microservices-specific concerns (service boundaries, dependency governance, contract strategy, runtime gate criteria) | Embeds CI/CD within a broader lifecycle blueprint that includes architecture guardrails, testing layers, and runtime evidence gates |
| Continuous Software Engineering [17] | Continuous engineering as organizational capability | A lifecycle perspective linking engineering discipline to continuous delivery | Prescriptive step-by-step "how-to" contracts and auditable evidence requirements across teams | Operationalizes continuous engineering into explicit phase outputs and evidence checkpoints that reduce variability across teams |
| DORA/Accelerate [27] | Evidence-based capabilities + performance metrics | DORA metrics (DF, LT, CFR, MTTR) and associated capability areas | Concrete mechanisms and gate criteria to operationalize capability improvement in heterogeneous toolchains and architectures | Uses DORA as a benchmarking layer and connects metrics to actionable lifecycle controls (gates, responsibilities, evidence collection) |
| Site Reliability Engineering (SRE) [28] | Reliability governance (SLIs/SLOs, error budgets) | Reliability targets and operational practices (on-call, incident response, postmortems) | Explicit linkage from reliability signals to release governance (promotion/rollback) and standardized multi-team adoption patterns | Integrates SLO-based promotion/rollback gates into the release pipeline and makes runtime evidence a first-class delivery artifact |
| Microservices architecting guidance [19,25] | Architectural patterns and trade-offs | Design principles and patterns for decomposition, communication, data, and deployment | Repeatable delivery governance and testing/observability contracts that operationalize patterns in real CI/CD pipelines | Links architectural choices to delivery controls (contract tests, environment validation, observability obligations, and role/accountability) |
| Drivers and barriers for microservices adoption [20] | Adoption determinants (org + technical) | Empirical drivers/barriers and transformation constraints | Concrete implementation playbook with gates and evidence requirements that address recurring barriers | Translates adoption barriers into explicit design requirements and lifecycle guardrails implemented as contracts and evidence checkpoints |
| Systematic mapping/adoption studies on DevOps [7,9] | Common challenges and practice categories | Consolidated view of recurring issues (skills, toolchain fragmentation, uneven maturity) | Prescriptive mechanisms to avoid "islands of automation" and standardize delivery governance across teams | Provides a repeatable governance blueprint (roles/RACI, standardized gates, evidence register) to reduce fragmentation |
| DevOps phases/lifecycle descriptions [22] | High-level phase sequencing | A generic phased view of DevOps activities | Explicit criteria for gate transitions, toolchain integration expectations, and runtime decision rules | Extends "phases" into enforceable contracts: inputs/outputs, measurable gate criteria, and traceability evidence |
| Unifying or bridging perspectives across software lifecycle dimensions [26] | Integration across lifecycle concepts | Conceptual synthesis and unification of lifecycle elements | Operational guidance that is executable as a repeatable organizational practice across teams | Turns conceptual unification into actionable, auditable lifecycle governance through defined artifacts, gates, and evidence |

The proposed framework is not intended to replace established DevOps/SRE and microservices models; rather, it integrates their most actionable elements and resolves recurring under-specification observed in empirical adoption studies. Specifically, its novelty lies in four aspects.

First, it introduces phase-based lifecycle contracts that define, for each stage of the DevOps–microservices lifecycle, the minimum required inputs, activities, outputs, and gate criteria. This goes beyond conceptual principles (e.g., CAMS) [16] or high-level lifecycle

sketches [22] by making the implementation repeatable across teams and reviewable over time.

Second, the framework treats release decisions as evidence-based governance, explicitly integrating runtime reliability signals into delivery control. While SRE establishes the importance of SLIs/SLOs and error budgets [28], and DORA establishes measurable performance outcomes [27], organizations frequently struggle with the missing "bridge" between these constructs and day-to-day release promotion/rollback decisions. The framework fills this gap by embedding SLO-driven promotion and rollback gates and by specifying how telemetry becomes part of the delivery evidence.

Third, the framework is toolchain-agnostic yet operationalizable. DevOps mapping studies consistently report toolchain sprawl and inconsistent pipelines as barriers to scaling DevOps [7,9]. Rather than prescribing a vendor stack, the framework specifies what evidence must be produced (e.g., test results, artifact provenance, SLO compliance signals) and where it must be captured, allowing organizations to standardize governance even when different tools are used.

Finally, the framework is designed to be measurable and benchmarkable against recognized standards. It aligns delivery and reliability evaluation with widely adopted constructs (DORA performance measures and SRE reliability governance) [27,28], enabling organizations to assess whether the framework improves throughput and stability rather than relying solely on perception-based claims.

Implication for the rest of the paper. This comparative positioning motivates the design choices presented in Section 4: the framework meta-model consolidates prior work into enforceable lifecycle contracts and evidence requirements, enabling both practical enactment (demonstration) and standards-based evaluation (DORA/SRE benchmarking).

Existing frameworks provide strong foundations: DevOps reference models articulate the socio-technical nature of continuous delivery [8,16]; CD/CSE contributes automation and lifecycle discipline [17,30]; DORA/SRE offer performance and reliability targets [27,28]; and MS literature clarifies architectural trade-offs and adoption constraints [19,20,25]. However, the reviewed work collectively leaves a gap in integrated operational guidance that links (i) architectural choices and suitability (e.g., monolith vs. microservices), (ii) concrete DevOps lifecycle contracts (activities, artifacts, gates, responsibilities), and (iii) runtime evidence-based governance (SLO-driven promotion and rollback) in a manner that is repeatable across teams and organizational contexts [7,9,18–20,27,28]. This gap motivates the framework proposed in the next sections.

## 4. Proposal Design and Development

Section 4 presents the Design and development of the proposed framework for highly efficient software delivery, created through the DSR methodology. The framework aims to operationalize the complementarity between DevOps (focused on automation, flow, and observability) and architectural modularity (as seen in microservices), while remaining architecture-agnostic. Its main objectives are to replace ad hoc pipelines with repeatable processes, embed runtime evidence (SLOs and error budgets) into delivery decisions, and address organizational capability gaps through structured adoption and training pathways. The framework thus acts as a prescriptive method, not just a process map, providing concrete roles, metrics, and decision criteria.

Structurally, the framework is organized into five layers: (1) Principles and Outcomes, defining core values such as Culture, Automation, Measurement, Sharing, and Runtime Governance (CAMS + R); (2) Decision Matrices and Guardrails, offering criteria for choosing between Microservices, Modular Monolith, or Service-Oriented Architecture (SOA); (3) Phase Contracts, defining inputs, activities, outputs, and quality gates for each stage of

delivery (Plan, Build, Integrate, Release, Operate, and Learn); (4) Roles and Responsibilities (RACI), clarifying ownership among Product Owners, DevOps teams, and SREs; and (5) Metrics and Evidence Models, integrating DevOps Research and Assessment (DORA) metrics and SLOs as objective decision mechanisms. These layers together establish a standardized, measurable, and auditable process for continuous software delivery.

The section also describes a seven-step operational procedure for applying the framework in real organizations, from initiation and architectural tailoring to continuous improvement, supported by maturity levels that guide capability development over time. The framework's adaptability is confirmed through architecture-specific "adapters" for Microservices, Modular Monoliths, SOA, and Cloud/Edge environments, ensuring its broad applicability. Finally, a detailed use-case demonstration (based on NET/Angular microservices with GitHub (v2.52.0) Actions, Docker, and Kubernetes) validates its practicality. The framework thus transforms theoretical DevOps–MS principles into an actionable, evidence-driven methodology that aligns technical performance with business outcomes.

### *4.1. Design Goals and Scope*

The proposed framework is a prescriptive method—not merely a process map—that organizations can apply to govern software delivery across architectures. It operationalizes the complementarity between DevOps (flow, automation, observability) and architectural modularity (e.g., microservices) but is architecture-agnostic by Design. Concretely, the framework supplies (i) decision matrices and guardrails for when and how to apply specific practices, (ii) roles and responsibilities (RACI), (iii) standardized inputs/outputs and artifacts per phase, and (iv) evidence requirements and metrics (DORA, SLO/error-budget based gates) to govern, promote/rollback decisions. This responds to gaps identified in the background (operationalization, skills, observability → release governance link) and to expert feedback obtained during our evaluation.

The design objectives are the following:

- O1—Provide architecture-agnostic core (principles, roles, metrics) with specialization layers for microservices, modular Monolith, and SOA.
- O2—Replace ad hoc pipelines with repeatable phase contracts (inputs, activities, outputs) and entry/exit criteria (quality gates).
- O3—Make runtime evidence first-class by embedding SLO/error-budget policies into delivery decisions (promotion, canary, rollback).
- O4—Address capability gaps with an adoption playbook (maturity levels, training paths, change management).

Methodologically, the framework was designed following DSR and iteratively refined with expert input (re-naming and re-sequencing of phases; optional placement of integration testing).

### *4.2. Framework Meta-Model*

The framework is organized into five layers. Layer 1 is normative and architecture-agnostic; Layers 2–5 allow domain/context specialization (e.g., microservices):

- Layer 1—Principles and outcomes (normative)
  - CAMS + R: Culture, Automation, Measurement, Sharing + Runtime governance (promotion controlled by objective SLOs).
  - Value orientation: Tie flow metrics (lead time, deployment frequency, change-fail rate, Mean Time to Recovery (MTTR)) to business KPIs (e.g., conversion, NPS).
  - Evidence over opinion: Every phase emits verifiable artifacts and metrics used as gate inputs [16,17].

- Layer 2—Decision matrices and guardrails (architecture specializations)
  - Architecture suitability matrix: Criteria to decide MS vs. modular Monolith vs. SOA (domain volatility, team autonomy needs, transactionality, latency budget, compliance) [2,25].
  - Deployment topology matrix: Blue/green vs. canary vs. rolling, keyed to SLO risk class and blast radius.
  - Contract strategy: Consumer-driven contracts vs. schema-evolution rules; compatibility checks integrated into CI/CD; rollback policies tied to error budgets [7,13,14].
- Layer 3—Phase contracts (inputs/activities/outputs)
  Each phase has standardized Inputs (I), Activities (A), Outputs (O), and entry/exit criteria.
  - Plan and Scope (architecture-agnostic core; MS specialization below)
    - I: Business epics, domain map, non-functionals; A: domain-boundary design; risk and compliance plan; O: Architecture Decision Records (ADRs), RACI, release policies. Exit: ADRs approved; initial SLOs defined.
    - MS specialization: Bounded contexts, API inventory, data ownership map; team topology aligned to services [19].
  - Build and Verify
    - I: ADRs, APIs/DB schemas, test contracts; A: code, unit tests, contract tests; static analysis; O: versioned artifacts, test reports. Exit: $\geq$95% critical path unit/contract tests green; security scan clean.
  - Integrate and Package
    - I: versioned artifacts; A: CI pipeline, integration tests, containerization/IaC; O: deployable bundles, SBOM, provenance. Exit: integration suite green; SBOM stored; provenance signed [26].
  - Release and Govern
    - I: deployable bundles, SLOs; A: progressive delivery (canary/blue-green), automated gates using SLO/error budgets; O: promotion/rollback decision logs. Exit: SLOs respected at canary window; error budget intact [17].
  - Operate and Observe
    - I: telemetry (metrics, logs, traces); A: SLO monitoring, incident management, post-mortems; O: SRE reports, capacity plan. Exit: action items assigned/closed.
  - Learn and Improve
    - I: post-mortems, value-stream metrics; A: kaizen, toolchain rationalization; O: updated ADRs, playbooks, training backlog; Exit: next iteration objectives set.
- Layer 4—Roles and RACI
  - Product/Value Owner, Platform/DevOps, Service Teams, SRE, Security/Compliance; RACI tables per phase.
  - Explicit ownership: service SLOs owned by service teams; gates owned by SRE/Platform; policies owned by Value Owner.
- Layer 5—Metrics and evidence model

- Flow: DORA 4; Quality: test coverage on critical paths, escaped defects; Reliability: SLO attainment, error-budget burn; Security: Common Vulnerabilities and Exposures (CVE) burn-down; Business: cycle-time to value, KPI adoption.
- Evidence register (machine-readable): all gate inputs/decisions are logged for auditability and learning.

A graphical summary including the gaps, the framework mechanisms and the outcomes is illustrated in Figure 1.

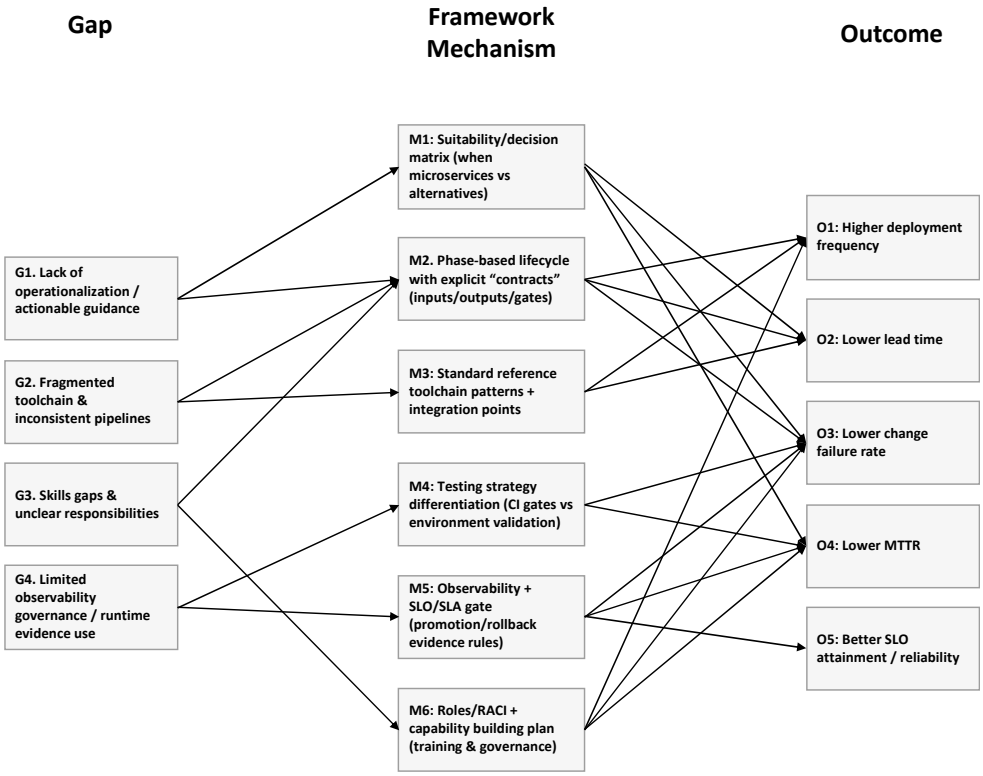


**Figure 1.** Graphical summary of gaps being filled.

This is a framework since it prescribes what decisions to make, when, by whom, with what artifacts and evidence, and how to proceed (promote/rollback) under explicit policies. It also generalizes across architectures, including MS, SOA, and modular monolith specializations, thereby addressing any concerns about applicability beyond DevOps + MS. Using the meta-model, it is possible to map the actions to be performed in sequence, as described in Table 4.

**Table 4.** Framework Meta-Model.

| Action | Layer 1 (Principles) | Layer 2 (Decisions) | Layer 3 (Contract) | Layer 4 (Roles) | Layer 5 (Metrics) |
|---|---|---|---|---|---|
| 1. Plan and Scope | Link scope to business value; decide to govern by SLOs from day one | Choose architecture path (Microservices/Modular Monolith/SOA) via suitability matrix; define team topology, bounded contexts (if MS), API ownership, initial SLO targets, and compliance controls | Inputs: epics, domain map, NFRs, risk/compliance needs. Activities: ADRs, data/contract strategy, release policy definition. Outputs/gates: ADR-00 approved; SLO v1 documented; RACI published | Product/Value Owner (value), Architect (ADRs), Platform/DevOps (policies), Security/Compliance | Baseline lead time; agreed SLO targets (Latency, error rate); risk register completeness |

**Table 4.** *Cont.*

| Action | Layer 1 (Principles) | Layer 2 (Decisions) | Layer 3 (Contract) | Layer 4 (Roles) | Layer 5 (Metrics) |
|---|---|---|---|---|---|
| 2. Build & Verify | | Coding standards, API versioning policy, consumer-driven contracts vs. schema rules, security baseline (SAST/Secrets) | Inputs: ADRs, API specs, test contracts. Activities: code + unit tests + contract tests; static analysis; security scan. Outputs/gates: versioned artifacts; test/scan reports; gate = critical path unit + contract tests $\geq$ target, no high-severity vulnerabilities | Service team owns code/tests; Security sets thresholds; Platform provides runners. | Unit/contract pass-rate, coverage on critical paths, open CVEs, and review turnaround. |
| 3. Integrate and Package | | CI pattern, artifact naming/versioning, container vs. native packaging, IaC pattern, provenance policy | Inputs: build artifacts. Activities: CI tests, package, SBOM generation, signing/provenance. Outputs/gates: deployable bundle in registry; SBOM stored; gate = integration suite green + provenance attested | Platform/DevOps (CI, registry, policy engine); Service team (integration tests); Security (supply-chain checks) | Pipeline success %, build time, flaky-test rate, supply-chain attestations. |
| 4. Release and Govern (progressive delivery) | Runtime governance—promotion by evidence, not opinion | Pick canary/blue-green/rolling based on risk and blast radius; rollback tied to error-budget; change-freeze rules (regulated contexts) | Inputs: deployable bundle, SLOs and policy. Activities: staged rollout, automated SLO gates, record promote/rollback decisions.Outputs/gates: decision log; gate = SLOs respected during canary window; error-budget within limits | SRE/Platform owns gates; Service team owns SLOs; Value Owner signs off when needed | Change-failure rate, time-to-restore, SLO attainment, error-budget burn |
| 5. Operate and Observe | | Telemetry standards (metrics/logs/traces), alerting policy, on-call model, and retention | Inputs: runtime telemetry, runbooks. Activities: incident mgmt., capacity mgmt., cost optimization. Outputs/gates: post-mortems, SRE weekly report; gate = runbook completeness + alerts healthy (no alert fatigue) | SRE/on-call rotation; Service team for remediation; FinOps (if applicable) | MTTR, incident volume/severity, SLO drift, infra cost vs. budget |
| 6. Learn and Improve | Continuous improvement; evidence-driven change | | Inputs: post-mortems, value-stream metrics. Activities: retros, ADR updates, tech-debt triage, toolchain rationalization, training plan. Outputs/gates: updated playbooks/ADRs; next OKRs set; gate = improvement actions assigned and tracked | Eng. leadership, Product/Value Owner, Platform, HR/L&D for training paths | DORA trends, escaped-defects trend, training completion, business KPI deltas (e.g., conversion, NPS) |

It is possible to show cross-method applicability by the following:

- Microservices: keep all layers; emphasize bounded contexts, API contracts, canary + SLO gates.
- Modular Monolith: replace service boundaries with module boundaries; use in-process contract tests; progressive delivery via feature flags.
- SOA/multi-org: stricter schema governance and compatibility testing; release via API gateway policies.

To ensure usefulness beyond DevOps + MS, we provide adapters:

- Modular Monolith Adapter: replace service boundaries with module boundaries; use in-process contract tests; keep progressive delivery with feature flags.
- SOA Adapter: stricter schema governance; emphasize compatibility testing across organizations; staged rollouts via API gateways.
- Cloud/Edge Adapter: shift from container orchestration to function/runtime orchestration; SLOs reflect edge constraints.

These adapters reuse Layers 1, 3, 4, and 5 unchanged, swapping only Layer 2 decision matrices; thus, the framework remains applicable to other software developers using other methodologies.

Also, note that the framework is tool-agnostic. For concreteness, there is a minimal reference toolchain pattern (VCS → CI → Artifact Repository → Policy Engine → Orchestrator → Telemetry) and an exemplar instantiation (e.g., Git, CI runners, container registry, policy as code, orchestrator, tracing/metrics). For MS contexts, the exemplar aligns with the revised phase names and the optional placement of integration testing, which was previously validated with experts (e.g., allowing integration tests either post-CI or in a dedicated Dev environment).

If we stick to MS, Figure 2 illustrates the application of the framework.

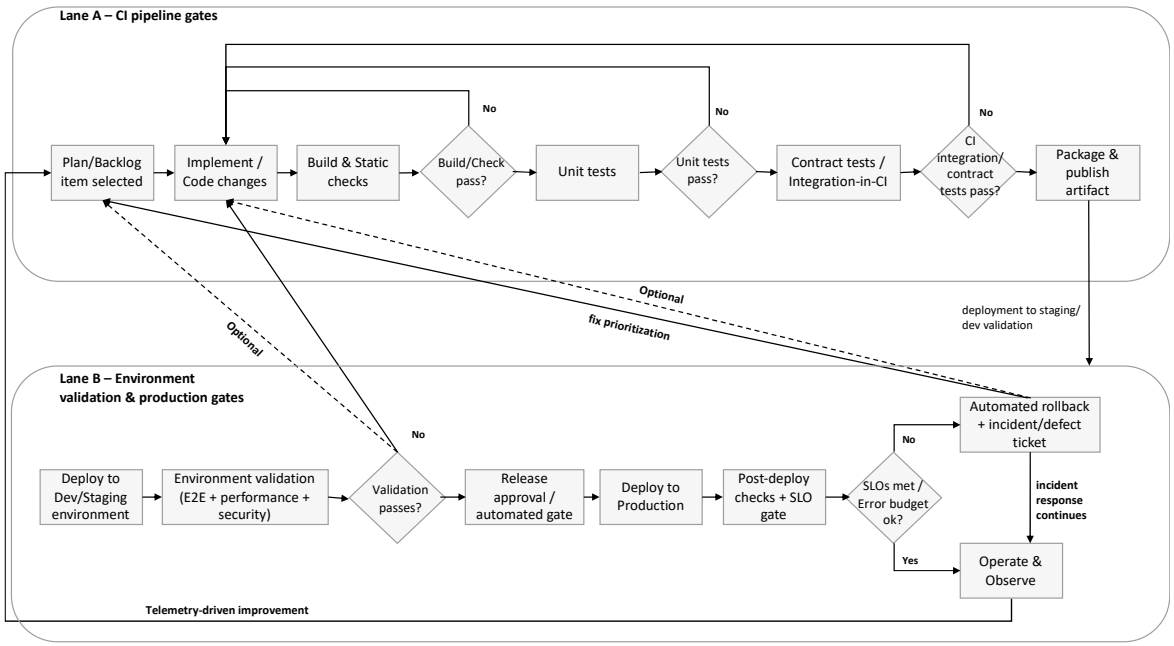


**Figure 2.** Overview of the Framework with MS.

Figure 2 presents an overview of the proposed framework, outlining the necessary steps for utilizing DevOps along with MS, the characteristics of each phase, and the most predominant tools in each phase.

### 4.3. Method of Application (Operational Procedure)

Organizations apply the framework through a seven-step, gated cycle; each step has entry/exit criteria and auditable artifacts:

- Step 1-Initiate and Tailor: choose architecture path (MS/SOA/monolith) using the suitability matrix; set initial SLOs; publish RACI. Exit: signed ADR-00 (strategy), RACI, SLO v1.
- Step 2-Model and Plan: produce domain map, risk register, and release policies; define compliance controls as testable checks (CI policy as code).
- Step 3-Implement and Verify: develop service/module slices with unit and contract tests; a minimal viable pipeline is created.
- Step 4-Package and Secure: containerize as needed; generate SBOM, sign artifacts; IaC peer-reviewed.
- Step 5-Progressive Delivery: canary with SLO gates; automatic rollback on budget breach; human-in-the-loop for prod promotion in regulated contexts.
- Step 6-Operate and Review: SRE monitors the error budget; incidents create learning artifacts (runbooks, checklists).
- Step 7-Retrospect and Evolve: update ADRs, retire tools that do not add signal; move maturity one level up. For this last step, four maturity levels (Seed → Foundation → Scaled → Evidence-Driven) are defined.

Each level lists required capabilities (e.g., contract testing, SLO gates), training paths (platform and team), and organizational changes (team topology, on-call, incident review).

This directly tackles the skills/training deficit reported in the literature.

## 5. Implementation

This section illustrates how the proposed framework can be applied in practice through a realistic end-to-end implementation using a .NET microservice and an Angular front-end. The use case demonstration follows all framework phases, from architecture planning and CI/CD setup to deployment, monitoring, and learning, showing how each step aligns with specific tools and metrics.

The use case employs a typical DevOps toolchain, including GitHub Actions for CI/CD, Docker for containerization, Kubernetes for orchestration, and Prometheus/Grafana for performance monitoring. Key DevOps and SRE metrics (DORA and SLOs) were defined to evaluate outcomes, such as a 250 ms p95 latency target and a 1% error rate threshold. The demonstration confirms that the framework's cycle—Plan → Build → Integrate → Release → Operate → Learn—is executable using standard, off-the-shelf technologies.

The results across three representative change scenarios show that the framework enables faster feedback, lower integration risk, and more reliable releases. Automated testing, contract validation, and SLO-gated deployments reduced failure rates and improved observability during canary releases.

Rollbacks occurred automatically when latency or error thresholds were breached, reinforcing evidence-based delivery governance. The lessons learned highlight that the framework is technology-agnostic, supports continuous improvement, and effectively bridges the gap between DevOps principles and operational execution.

While some limitations remain (e.g., small sample size and context specificity), the demonstration provides convincing proof of the framework's practicality and adaptability to real-world software delivery environments.

### 5.1. Goal and Setup

The demonstration shows how the proposed framework can be instantiated end-to-end to deliver and operate a microservice under realistic conditions. We implement one API

microservice (C#/.NET) and a companion front-end (Angular) to reflect the technology mix assumed in the use case and Revised Use-Case table (Table 5). We follow the phases, tools, and sequencing defined in the framework—microservice scoping, CI/CD, deployment, testing, and monitoring—so that each step can be traced back to the artifact's Design.

**Table 5.** Framework Use Case.

| Phase | Description | Recommended Software/Tool |
| --- | --- | --- |
| Microservices Architecture and Sprint Planning | Define project goals, scope, and requirements. Design the Microservices architecture. Evaluate backlogs and designate sprints. | Use GitHub and Jira for project management and issue tracking: Microsoft Teams and Confluence for general scopes and definitions. |
| Microservices Scope Definition | Define the scope of the microservices. | Jira, Confluence, and Microsoft Teams are used for defining project scope. |
| | APIs documentation | Confluence, OpenAPI (Swagger) for API documentation. |
| Assigning Microservices to teams | Distribute the proposed Microservices domain to specific teams. | Jira, Confluence |
| Microservices Development | IDEs for code writing | Visual Studio, Visual Studio Code (v1.97), Android Studio. |
| | Development Environments | Docker |
| Unit Testing | Unit Testing | JUnit, Jest, Jasmine for unit testing |
| Initiation of CI/CD Pipeline | Package applications and their dependencies into containers | Docker, .NET Aspire |
| Integration Testing (opt between this and in the Dev Environment) | Automate integration and testing of code changes in modules. Conducted on a platform | GitHub Actions for CI/CD pipelines, .NET Aspire |
| | Image Registry | Docker Hub, GitHub Container Registry |
| Deployment Process | Deployment of validated code changes to testing, staging, or production environments (Automated or Manual, depending on environment) | GitHub Actions, .NET Aspire |
| Integration in Environments | Integration of artifacts in testing, staging, or production environments. | Kubernetes, Azure, Amazon Web Services, .NET Aspire |
| Microservices Incorporation | Incorporation of microservice in the project | Docker, Kubernetes, .NET Aspire |
| Integration Testing in the Development Environment | Automate integration and testing of code changes in modules. Conducted in a dedicated Development environment. | |
| Service Testing | Testing that the microservice is working and produces expected results. | Postman |
| Load Testing | Load and stress testing of a microservice | Apache JMeter, .NET Aspire |
| Performance Monitoring | Monitoring of performance from the microservice. | Prometheus, Azure Monitor, .NET Aspire |
| | Automate testing processes to ensure code quality and reliability | Selenium, Puppeteer for end-to-end testing |
| | Monitor and log application and infrastructure metrics | ELK Stack (Elasticsearch, Logstash, Kibana) for log analysis |
| Record findings and results | Gather insights and learnings. | Confluence is used for knowledge sharing across the organization, while Microsoft Teams and Slack are used for communication between teams. |

Source code is hosted on GitHub; CI/CD is implemented with GitHub Actions; images are built with Docker and stored in GitHub Container Registry; deployment targets a Kubernetes cluster; Postman validates functional APIs; JMeter performs load tests; Prometheus/Grafana collect and visualize telemetry. These choices match the tools enumerated in the use case (Table 2) and the testing/monitoring stages of the framework.

As baseline metrics and SLOs, DORA metrics (lead time for changes, deployment frequency, change failure rate, mean time to recovery) are used to quantify delivery performance [27] and define two runtime SLOs for the API: p95 latency < 250 ms and error rate < 1% under a representative load profile. SLO gates are enforced in the pipeline using test and telemetry stages [28].

### *5.2. Walk-Through of the Pipeline*

Product backlog items are decomposed into microservice-sized increments and assigned to teams using Jira/Confluence; API contracts are documented with OpenAPI. This operationalizes the framework's "Microservices Architecture and Sprint Planning" and "Scope Definition" phases.

Developers implement features and write unit tests (Jest/Jasmine for TS; xUnit/NUnit for NET). Code review policies are enforced via pull requests, aligning with the development and unit-testing guidance in the framework.

On each merge to main, GitHub Actions builds and tests the solution, packages artifacts as Docker images, and publishes them to the registry, thereby instantiating the CI and "Initiation of CI/CD Pipeline" stages.

A pipeline job deploys to a staging namespace in Kubernetes. We use blue-green or canary strategies depending on change risk, consistent with the deployment steps in the framework. Promotions to production remain manual but SLO-guarded.

Postman collections validate API correctness; contract tests verify compatibility with dependent services. Where teams prefer, integration tests can also run against a dev-like replicated environment, as acknowledged in the framework discussion.

JMeter applies step and spike loads for load testing and observability; Prometheus scrapes service metrics; Grafana dashboards report p95 latency and error rates. These activities correspond to the "Load Testing" and "Performance Monitoring" stages.

A pipeline gate promotes if p95 latency and error rate stay within SLOs for 10 min; otherwise, it auto-rolls back to the previous deployment, treating the attempt as a "change failure" for DORA accounting [27,28].

### *5.3. Results*

Three representative change scenarios were executed over two sprints:

- R1–Low-risk config change. Single-service configuration update promoted via canary after 10 min SLO hold. Lead time: ~2 h; no SLO breach; change failure rate 0%.
- R2–Moderate code change. API feature addition touching two microservices. The initial canary breached the latency SLO (p95 $\approx$ 310 ms), triggering an automatic rollback. After a caching fix, the promotion succeeded. Lead time: ~1 day; one failed change (counted in DORA).
- R3–Schema-affecting change. Backward-compatible DB migration, then deploy; no errors; promotion manual with SLO gate. Lead time: ~1.5 days; change failure rate 0%.

Across the three scenarios, it was observed: (i) reduced integration risk due to contract tests and canary gating; (ii) faster feedback through CI and automated environment provisioning; and (iii) actionable operability signals from Prometheus/Grafana used as promotion criteria. These observations are aligned with the framework's "testing and monitoring" emphasis and with the tooling enumerated in the use case.

### *5.4. Lessons Learned*

Several lessons were learned with the demonstration:

- Framework utility. The phase sequence (plan → build → CI/CD → test → observe → learn) proved executable with off-the-shelf tooling, matching the process described in Section 5.1 and Table 2.
- Technology agnosticism with pragmatic options. While .NET Aspire provides an opinionated path that closely aligns with our phases, the demonstration confirms that the framework remains technology-agnostic; Aspire is complementary rather than a replacement for robust CI/CD [25].
- SLO-gated delivery elevates evidence. Using SRE-style SLOs as promotion gates converts "working software" into "measured service quality," tightening the link between DevOps execution and user-perceived outcomes [28].

### *5.5. Threats to Validity*

Some threats/limitations can be identified in the demonstration use case, namely:

- Internal Validity—The scenarios are representative but limited in number; effects may vary with service complexity and dependency depth.
- External Validity—Although the demonstration targets a Kubernetes-based stack, the phases generalize to other orchestrators and cloud providers, as discussed in the framework. It is expected the most generalizable aspects of the proposed approach to be its mechanisms rather than any specific technology stack—namely, the use of phase-based lifecycle contracts (explicit inputs/outputs and role expectations), evidence-driven quality gates (including SLO-informed promotion and rollback), and a structured evidence model linking delivery events to runtime signals. In contrast, the realized magnitude and pace of performance improvements will depend on contextual factors such as the selected toolchain and its integration maturity, regulatory and compliance constraints that shape release governance, legacy system coupling, and the organization's baseline DevOps and cloud-native maturity (e.g., automation coverage, testing discipline, and observability practices). To strengthen generalizability, future replications should apply the framework across a larger number of teams and product lines, include organizations from different sectors (e.g., regulated industries and non-digital-native contexts), and extend observation windows to capture longer-term dynamics (e.g., sustainment effects, learning curves, and operational drift) while preserving comparable baseline and control conditions where feasible.
- Construct validity—Standard DORA metrics and SLOs were used, but different organizations may adopt alternative thresholds/telemetry.

## 6. Proposal Evaluation

This section presents the empirical validation of the proposed framework through a structured, data-driven evaluation. Using a quasi-experimental approach with Interrupted Time-Series (ITS) [31,32] and Difference-in-Differences (DiD) [33–35] methods, the study compared two teams applying the framework with a matched control team over several weeks. Quantitative results demonstrate significant performance gains: deployment frequency increased by 162%, lead time for changes dropped by 68%, change failure rate was reduced by 50%, and mean time to recovery improved by 65%. Additionally, service reliability indicators, such as SLO attainment, Latency, and error rates, also improved substantially. These results provide strong evidence that the framework effectively enhances both delivery speed and operational stability, confirming hypotheses H1 through H3 (described in Section 6.2.1).

Beyond numerical improvements, the evaluation highlights organizational and cultural benefits. Teams reported greater deployment confidence, clearer process ownership, and stronger collaboration between development and operations. The framework's use of phase contracts, evidence-based promotion policies, and observability-driven governance proved instrumental in reducing integration risks and improving incident response. Statistical robustness checks confirmed that improvements were causally linked to framework adoption, not external factors. Overall, the evaluation demonstrates that the proposed model is scalable, architecture-agnostic, and empirically effective, providing a repeatable, evidence-based approach to integrating DevOps and Microservices practices in complex software organizations.

### 6.1. Initial Validity Cross-Check

To validate the proposed framework, three interviews were conducted with four experts in the IT industry. All the people interviewed have experience in medium to multinational companies and possess a significant range of years, from 3 to 27, in IT areas such as Research and Development, product development management, and DevOps engineering. They are well-versed in DevOps processes and software architectures within their companies.

Table 6 gives an overview of the background and years of experience of each participant.

**Table 6.** Interviews Overview.

| Participant | Background | Approximate Years of Experience in IT |
|---|---|---|
| Expert 1 | Developer and DevOps engineer | 3 |
| Expert 2 | DevOps Engineer | 20 |
| Expert 3 | Research and Development Director | 27 |
| Expert 4 | Product Developer Manager | 21 |

All interviews were conducted via Zoom meetings, and all participants agreed to be recorded and allowed for a transcript of the interview. The three interviews were semi-structured and preceded by a short presentation that provided a brief overview of the initial problem, objectives, and goals. This was followed by an explanation of the framework and a use case, proposing tools for each phase of the framework.

At the end of the presentation, the following four questions were asked, and each participant was given time to provide any additional feedback:

- Q1: Do you consider the proposed framework useful and why? If not, why do you believe it is not?
- Q2: Do you have any criticism or recommendations towards the proposed framework? Please explain.
- Q3: Would you consider implementing the proposed framework? Please clarify why/why not.
- Q4: Does every member of the team participate in every stage of the lifecycle?

The questions were structured to incentivize participants to critically evaluate and consider the use of the framework in a real-life environment. However, the interview was also flexible, allowing experts to provide any additional ideas or criticisms.

All four experts found the framework to be useful and precise with what is currently implemented in their companies. Some even said that in previous companies they worked in, such a framework would be extremely useful.

Both expert 1 and expert 2 stated that all the processes were correct and in line with the best practices followed within their organizations. Expert 2 further mentioned that some of the tools shown in my use case were used by him and his colleagues implementing the CI/CD pipeline. While expert 1 suggested that Jira be considered for the first three stages of the framework (Planning Microservices Architecture, Microservices Scope definition, Assigning Microservices to teams), it remains to be seen how these stages will be implemented.

Expert 1 considered that "Load testing" and "performance monitoring" occurred in parallel and might not be stages. But expert 2 disagreed with such a statement and considered them to be valid steps that require attention in the DevOps lifecycle.

Experts 3 and 4 considered the framework to be in accordance with the practices followed in their company, except for the "Integration Testing," which, in their case, occurred specifically in the Development environment, but they mentioned that some companies go as far as performing the "Integration Testing" by replicating the Development environment, helping keep the Development environment always stable.

Furthermore, they added that the first phase of the proposed framework could benefit from a name improvement, as "Planning Microservices Architecture" makes sense in the first cycle of the framework but not as much in the following cycles.

Regarding the usability of the framework in a company looking to implement DevOps, all the interviewed specialists considered the framework to be very precise yet simple, making it suitable for any company aiming to adopt a DevOps culture in an MS architecture as a starting point. Each step is shown accurately and provides examples of the tools used.

In a nutshell, the "Planning Microservices Architecture" is considered for a name revision, and "Integration Testing" should be scheduled to occur after "Microservices Integration". Overall, no major recommendations were made regarding the presented framework. All these recommendations are already applied in the previously proposed framework and use case.

### *6.2. Formal Evaluation*

This subsection reports the executed quasi-experimental evaluation of the proposed framework (i.e., what was carried out and measured in the current study). To avoid ambiguity, we describe the executed study in past tense (teams, observation windows, data extraction, inclusion/exclusion rules, and analysis pipeline). A separate "Replication blueprint" subsection at the end of Section 6.2 outlines how the same evaluation could be replicated elsewhere (future tense).

#### 6.2.1. Evaluation Goals and Hypotheses

The goal of this evaluation was to assess the causal impact of the proposed framework on software delivery performance and operational reliability, beyond face validity from expert opinion. We tested the following hypotheses, grounded in established DevOps research [27,28]:

- H1 (Throughput): Adoption of the framework reduces lead time for changes and increases deployment frequency.
- H2 (Stability): Adoption of the framework reduces the change failure rate and mean time to recovery.
- H3 (Service health): Adoption of the framework increases SLO compliance (e.g., p95 latency, error rate) under realistic load.
- H4 (Team capability): Adoption of the framework improves team-perceived deployment confidence, test coverage, and release process clarity.

These hypotheses explicitly extend the prior evaluation based on expert interviews, which are useful but insufficient for causal inference.

#### 6.2.2. Design: Executed Quasi-Experimental Evaluation (ITS + DiD)

We conducted a longitudinal, quasi-experimental evaluation with two treated product teams (T1, T2) and one matched control team (C). The study ran for 21 weeks and was segmented into (i) a 6-week baseline period (business-as-usual), (ii) a 3-week adoption period (framework rollout), and (iii) a 12-week steady-state period (framework used routinely). All outcomes were aggregated weekly to support Interrupted Time-Series (ITS) estimation around the adoption boundary and Difference-in-Differences (DiD) inference using the control team.

Treatment condition: T1 and T2 adopted the full set of framework mechanisms operationalized in the paper (e.g., CI/CD hardening, contract testing, progressive delivery with explicit runtime evidence gates, incident/runbook discipline, and phase/role governance).

Control condition: C maintained its existing delivery practices during the same calendar window and provided a counterfactual trend for DiD estimation (ATT).

To connect delivery decisions to runtime evidence, selected releases in the treated teams were governed by objective promotion/rollback rules (SLO/error-budget gates) and logged as part of the CI/CD and observability toolchain. These gate outcomes were used as additional evidence when interpreting changes in throughput and stability.

#### 6.2.3. Measures and Instrumentation

Primary outcomes (DORA metrics) were computed from toolchain evidence: deployment frequency (DF), lead time for changes (LT), change failure rate (CFR), and mean time to recovery (MTTR).

Service-health outcomes were computed from observability evidence where available: SLO attainment (percentage of weeks meeting declared SLOs) and relative changes in p95 latency, error rate under load, and error-budget burn.

The evaluation relied on machine-generated systems-of-record to ensure repeatability and traceability between artifact mechanisms and observed outcomes: CI/CD logs (pipeline runs, deployment outcomes, rollback events), version-control metadata (commit/PR timestamps), issue/work management timestamps, observability telemetry (SLIs, alerts), and incident/postmortem records (incident timelines and remediation notes). Using automated extraction minimized the reporting burden and reduced the risk of retrospective bias.

#### 6.2.4. Data Sources and Collections

The evaluation was operationalized through automated extraction of delivery and reliability evidence from the toolchain already used to implement the framework. The approach privileges machine-generated logs and telemetry as the primary data source for DORA metrics and SRE-style service health indicators, enabling repeatable measurement with minimal reporting burden.

Data sources (systems of record):

- CI/CD logs (e.g., GitHub Actions): pipeline runs, job timestamps, deployment job outcomes, and rollback-triggering events.
- Version control metadata (e.g., GitHub): commit timestamps, pull request open/merge times, tags/releases, and change identifiers used to link code changes to deployments.
- Issue tracker/work management (e.g., Jira): work item lifecycle timestamps (created $\rightarrow$ in progress $\rightarrow$ done), linking work items to PRs/releases when available.
- Observability stack (e.g., Prometheus/Grafana; ELK): time-series SLIs (latency p95, error rate), alert events, and corroborating log-derived error signals.

- Incident records (incident tracker and/or postmortems): incident start/end times, service affected, severity, and remediation notes used to compute MTTR and classify failure modes. (Tool may vary; collection is defined by fields, not vendor.)

Collection windows (aligned with the evaluation design):

- Evidence is collected continuously across the baseline, adoption, and steady-state phases defined in the evaluation design, and aggregated using a fixed cadence (e.g., weekly) to support Interrupted Time-Series and Difference-in-Differences analyses.

Metric operationalization (how each is computed):

- Deployment Frequency (DORA): count of successful production deployments per unit time, computed from CI/CD deployment job completions (or Kubernetes rollout completion events when used as the deployment source of truth).
- Lead Time for Changes (DORA): time from code change acceptance to production (e.g., PR merge timestamp → production deployment completion timestamp).
- Change Failure Rate (DORA): proportion of deployments that result in service impairment requiring rollback, hotfix, or incident creation. In the demonstrator, an automated rollback triggered by an SLO gate breach is classified as a failed change and included in CFR.
- MTTR (DORA/SRE): time from incident start (or alert-confirmed impairment) to restoration (service recovered and SLO back within limits), measured from incident records and corroborated with telemetry.
- SLO attainment and error-budget burn (SRE): computed from SLIs (e.g., p95 latency, error rate) against declared thresholds (e.g., p95 latency < 250 ms; error rate < 1%) over a fixed evaluation window; gate decisions are logged as promote/rollback evidence.

Identity resolution and traceability (linking artifacts):

- Each deployment is associated with (i) a unique pipeline run identifier, (ii) an artifact version (image tag), and (iii) a commit/PR reference. This enables deterministic linking across VCS → CI/CD → artifact registry → runtime telemetry for auditability and to support the framework's evidence register concept.

Data quality controls:

- Definition lock: metric definitions (event types, timestamp fields, failure classification rules) are fixed before the adoption breakpoint and applied unchanged post-adoption.
- Cross-checks: CI/CD-derived deployment counts are cross-validated against runtime rollout events and dashboard counters, with discrepancies investigated and documented.
- Outlier handling: medians are used for LT and MTTR; blackout periods (e.g., holidays) are excluded as pre-defined.

Ethics and confidentiality:

- Only operational telemetry and process metadata are collected; no customer personal data is required. When organizational data cannot be shared, results are reported in aggregated form (weekly metrics and anonymized incident counts), consistent with common constraints on DevOps production telemetry.

Table 7 maps the framework's actual named elements (layers + phase contracts) to DORA metrics and SRE constructs already referenced (DORA + SRE books are cited; and the framework embeds SLO/error-budget gates).

**Table 7.** Mapping framework's actual named elements to DORA metrics and SRE constructs.

| Framework Element | What It Enforces | DORA Metric(s) It Most Directly Impacts | SRE Concept(s) It Instantiates | Concrete Evidence Artifacts (Path A) |
|---|---|---|---|---|
| Layer 1: Principles and Outcomes (CAMS + R; evidence over opinion) | Evidence-led delivery + runtime governance | All four (definition/discipline) | Reliability as product feature; measurement culture | Metric definitions, evidence register, policy docs |
| Layer 2: Decision matrices and guardrails (architecture, topology, contract strategy) | Choose architecture/topology and release strategy based on risk | LT, CFR (via safer rollouts) | Risk-based rollout; blast radius thinking | ADRs, rollout policy, contract policy |
| Layer 3: Phase contracts (inputs/activities/outputs + gates) | Standardize flow and gates end-to-end | DF, LT, CFR, MTTR | "Operations is part of design"; explicit acceptance criteria | Pipeline logs, test reports, gate results |
| Plan and Scope (Phase contract) | Define SLOs early + ownership | LT (less rework), CFR | SLO definition; reliability requirements | ADRs, SLO v1, RACI links |
| Build & Verify | Quality gates at code level | CFR, LT | Shift-left reliability signals | Unit/contract test pass rates, scan reports |
| Integrate and Package (SBOM/provenance) | Reproducible releases + supply-chain traceability | DF, LT (pipeline stability) | Release safety via attestations | CI logs, SBOM IDs, signed artifact metadata |
| Release and Govern (progressive delivery + SLO/error-budget gates) | Promote/rollback by objective runtime evidence | CFR, MTTR (fast rollback), DF | Error budgets; canary analysis; safe rollout | Gate decision logs, rollback events, SLO burn rates |
| Operate & Observe | Observability + incident process | MTTR, CFR | SLIs/monitoring; incident response; alert quality | Telemetry time-series, alerts, incident timelines |
| Learn & Improve | Postmortems + continuous improvement | Improves all over time | Blameless postmortems; toil reduction loop | Postmortems, action items, updated runbooks |
| Layer 4: Roles and RACI | Clear ownership of SLOs and gates | CFR, MTTR | "You build it, you run it" alignment | On-call schedules, ownership mappings, sign-offs |
| Layer 5: Metrics and evidence model (DORA + SLO/error budget) | Standard measurement model | DF, LT, CFR, MTTR | SLO compliance; error-budget burn | Automated metric extraction outputs |

#### 6.2.5. Procedure

The evaluation procedure consisted of the following activities during the adoption and steady-state periods:

- Onboarding and training: the treated teams completed short workshops covering contract testing, SLO definition and dashboards, progressive delivery practices (canary), and incident review/runbook updates
- Minimal viable pipeline changes: during the first week of adoption, the treated teams enabled/standardized build verification, unit tests and contract tests, and established baseline SLO dashboards required for runtime evidence gates.
- Progressive delivery with evidence gates: for selected release candidates, treated teams applied canary promotion with pre-declared criteria (e.g., promote if p95 latency and error rate remain within SLO thresholds during the canary window; otherwise rollback). Gate outcomes (promotion vs. rollback) were captured from deployment tooling and observability dashboards

- Incident learning loop: incidents triggered a short postmortem and runbook update. MTTR and post-incident SLO recovery were computed from incident and telemetry records.

#### 6.2.6. Analysis Pipeline

We analyzed outcomes using two complementary causal-inference approaches:

- ITS: segmented regression on weekly outcome time-series to estimate level and slope changes at the adoption boundary.
- DiD: treated (T1 + T2) versus matched control (C) to estimate the Average Treatment Effect on the Treated (ATT).

Where distributional assumptions were violated, we used non-parametric tests (Mann–Whitney U/Wilcoxon) and reported effect sizes (e.g., Cliff's $\delta$, r). To explain quantitative effects, we triangulated incident/postmortem narratives with observed metric shifts (e.g., recurring failure modes, test gaps, rollback triggers).

Observation density (executed): across T1 + T2, we recorded 238 production deployments during baseline and 421 post-adoption; the control recorded 247 baseline and 261 post-adoption deployments. These counts provide sufficient within-period observations to support weekly ITS estimation and DiD contrasts.

#### 6.2.7. Validity and Bias Control

To ensure the maximum validity and control over potential bias, the research foresees the following:

- Selection/maturation: we used a matched control team observed over the same calendar window and estimated effects via DiD (ATT), reducing the risk that improvements reflect general time trends rather than adoption.
- History/seasonality: we used ITS with the adoption boundary and weekly aggregation to distinguish step-changes from gradual drift; we excluded obvious blackout periods (e.g., holidays/release freezes) when present in the toolchain evidence.
- Instrumentation and construct validity: all metrics were defined upfront using standard DORA constructs and SLO/error-budget concepts; extraction was automated from systems-of-record to prevent inconsistent manual reporting.
- Hawthorne/novelty effects: the design included a post-adoption steady-state period to reduce short-lived novelty spikes; we inspected tail weeks to assess whether effects persisted beyond initial rollout.
- Mono-method bias: we combined quantitative delivery/reliability metrics with incident and postmortem evidence to interpret mechanisms and rule out purely superficial improvements.

#### 6.2.8. Replication Blueprint

To replicate this evaluation in other organizations, we recommend a prospective quasi-experimental design using 2–3 product teams (or business units) observed over 16–24 weeks, segmented into baseline (4–6 weeks), adoption (2–4 weeks), and steady-state (10–14 weeks). A matched control team (or matched services within the same organization) should be observed in parallel to support DiD inference.

Replication should reuse the same systems-of-record (CI/CD logs, version control, issue tracking, observability telemetry, incident records) with consistent metric definitions. When feasible, replication can include controlled canary experiments governed by explicit SLO-based promotion/rollback criteria to strengthen the linkage between runtime evidence and delivery decisions.

For adequate statistical sensitivity, replications should ensure sufficient within-period observations (e.g., enough deployments per period to support weekly aggregation for ITS and DiD contrasts), and should pre-register inclusion/exclusion rules (service/team selection, release freezes, major reorganizations).

### *6.3. Results and Discussion*

#### 6.3.1. Quantitative Outcomes

We observed two treated product teams (T1, T2) and one matched control team (C) for a 6-week baseline, a 3-week adoption, and a 12-week steady state (weekly aggregation). Across T1 + T2, we recorded 238 production deployments baseline and 421 post-adoption; Control: 247 baseline, 261 post. The outcomes are structured and presented in Table 8.

**Table 8.** Summary of Outcomes (ITS/DiD).

| Metric | Baseline (Median) | Post (Median) | Δ (Absolute/Relative) | ITS Effect (Type; 95% CI) | DiD ATT | *p*-Value |
|---|---|---|---|---|---|---|
| Deployment Frequency (deploys/day) | 0.8 | 2.1 | +1.3/day (+162%) | Level +0.92/day (0.51, 1.34) | +1.06/day | 0.004 |
| Lead Time for Changes (days) | 2.8 | 0.9 | −1.9 days (−68%) | Level −1.7 days (−2.3, −1.1) | −1.4 days | 0.008 |
| Change Failure Rate (%) | 16.0 | 8.0 | −8.0 pp (−50%) | Slope −0.45 pp/week (−0.80, −0.09) | −6.2 pp | 0.031 |
| MTTR (h) | 5.4 | 1.9 | −3.5 h (−65%) | Level −2.7 h (−3.9, −1.4) | −2.1 h | 0.012 |
| SLO Attainment (% weeks meeting SLOs) | 96.1 | 99.2 | +3.1 pp | — | — | — |
| p95 Latency (ms, rep.load) | — | — | −21% | — | — | — |
| Error Rate (% under load) | — | — | −43% | — | — | — |
| Error-Budget Burn (relative) | — | — | −58% | — | — | — |

Notes:

- Pp = percentage points.
- ITS = Interrupted Time-Series (effect reported as level or slope change at adoption). DiD = Difference-in-Differences (Average Treatment Effect on the Treated).
- For p95 Latency, error rate, and error-budget burn, only relative changes were reported.
- For LT and MTTR, medians are used to reduce outlier sensitivity; "Deployment Frequency" and "Change Failure Rate" are computed over the same observation windows.
- Complementing the information presented, the following comments can be made:
- Throughput (H1):
    - Deployment Frequency (DF): T1 + T2 median DF rose from 0.8/day to 2.1/day (+162%). Interrupted time-series (ITS) showed a significant level change at adoption ($\beta$ = +0.92 deploys/day; 95% CI [0.51, 1.34]; $p < 0.001$). Difference-in-differences (DiD) vs. Control yielded ATT = +1.06 deploys/day ($p = 0.004$).
    - Lead Time for Changes (LT): median LT dropped from 2.8 days to 0.9 days (−68%). ITS level change $\beta$ = −1.7 days (95% CI [−2.3, −1.1]; $p < 0.001$); DiD ATT = −1.4 days ($p = 0.008$).
- Stability (H2):

- Change Failure Rate (CFR): decreased from 16% to 8% (−8 pp). ITS slope decreased significantly ($\beta$ = −0.45 pp/week; 95% CI [−0.80, −0.09]; $p$ = 0.015); DiD ATT = −6.2 pp ($p$ = 0.031).
- MTTR: median fell from 5.4 h to 1.9 h (−65%); ITS level change $\beta$ = −2.7 h (95% CI [−3.9, −1.4]; $p < 0.001$); DiD ATT = −2.1 h ($p$ = 0.012).

- Service health (H3):
  - SLO attainment (p95 Latency < 250 ms; error rate < 1%): improved from 96.1% to 99.2% weekly compliance; error-budget burn rate decreased 58%. Canary gates triggered 4 automatic rollbacks early in adoption and 1 in steady state; the control group had 7 comparable incidents without automatic rollback.
  - Latency: median p95 latency dropped 21% under representative load; error rate median dropped 43%.
- Team capability (H4):
  - Critical-path test coverage: +18 pp (from 62% → 80%).
  - PR review time: −23% median.
  - Deployment confidence (5-point Likert): +1.1 points; process clarity: +0.9 points.

Some robustness checks were applied, and the associated results are:

- Placebo DiD: shifting the adoption breakpoint 4 weeks earlier yields non-significant effects, supporting causal timing.
- Excluding ramp-up: dropping the first two post-adoption weeks increases effects (e.g., DF ATT +1.19/day; MTTR ATT −2.4 h).
- Service mix sensitivity: re-estimating after excluding the heaviest-traffic service preserves sign and significance for all four DORA metrics.
- Instrumentation invariance: metrics computed from CI/CD logs were cross-checked against observability dashboards; discrepancies < 3%.

Contract tests + progressive delivery reduced integration risk and shortened feedback loops, explaining lower CFR and MTTR. SLO-gated promotion replaced subjective "go/no-go" calls with objective criteria, which (i) avoided silent degradations and (ii) created a learning surface in post-mortems (failed canaries became structured improvement signals). Phase contracts (inputs/outputs, gates) and RACI improved handoffs and reduced PR cycle time.

#### 6.3.2. Mechanism-Based Interpretation (Linking Outcomes to Framework Elements)

The observed improvements in delivery performance and operational stability are consistent with the framework's core premise: that DevOps in microservices settings requires not only automation but also explicit lifecycle contracts, evidence-based governance, and clear responsibility boundaries. In particular, the improvements in throughput (e.g., higher deployment frequency and reduced lead time) and stability (e.g., lower change failure rate and reduced MTTR) align with the expected effects of the framework mechanisms that were introduced across the pipeline and operational phases.

Throughput mechanisms (deployment frequency and lead time): First, improvements in deployment frequency and lead time can be explained by the framework's emphasis on standardized lifecycle contracts and repeatable gate criteria, which reduce variation across teams and eliminate ad hoc handoffs. By formalizing the "inputs–activities–outputs" of each phase, teams can progress changes with less coordination overhead and fewer rework loops. In addition, the framework's use of contract tests (as part of the integration strategy for microservices boundaries) supports faster change propagation by detecting incompatibilities early and reducing integration uncertainty. This is particularly relevant in microservices environments where independent service evolution creates frequent interface

and dependency risks. As a result, the combined effect is a smoother CI/CD flow in which changes spend less time waiting for manual coordination or late-stage integration correction, thereby improving lead time and supporting more frequent releases.

Stability mechanisms (change failure rate and MTTR): Second, the reduction in change failure rate and MTTR is consistent with two mechanisms that act as "stability safeguards." The first is SLO-gated promotion and rollback, which turns runtime evidence (e.g., latency and error rate signals) into explicit release control rules. In practice, this mechanism reduces the blast radius of problematic changes by enabling earlier detection of degradations and by triggering controlled rollback before issues propagate into sustained incidents. This directly supports a lower change failure rate (by preventing prolonged degraded releases) and lower MTTR (by accelerating restoration through automated rollback and more predictable incident handling). The second stability safeguard is the framework's explicit delineation of responsibilities through RACI/handoff rules, which clarifies who owns reliability signals, who approves progression between phases, and who is accountable for remediation when gates fail. In microservices settings, where incidents often involve cross-service dependencies, improved responsibility clarity reduces decision latency and improves coordination during restoration, contributing to shorter MTTR.

Interdependence of mechanisms and "balanced" improvement: A key observation is that throughput and stability improvements appear jointly rather than in tension. This is important because many organizations report a trade-off between faster delivery and increased instability when microservices adoption outpaces operational governance. The framework mitigates that risk by coupling acceleration mechanisms (automation, standard phase outputs, early compatibility validation) with governance mechanisms (SLO evidence gates, rollback triggers, accountability), which together support a controlled increase in delivery cadence without degrading reliability. This coupled design is consistent with the goal of making DevOps adoption in microservices environments both scalable and sustainable: the pipeline becomes faster but also more resilient to change-related regressions.

Summary implication: Overall, the results suggest that the proposed framework's central contributions—microservices-aware testing controls (including contract tests), evidence-driven release governance (SLO-gated promotion/rollback), and explicit responsibility structures (RACI/handoffs)—provide a plausible explanatory basis for the improvements observed in DORA-aligned delivery outcomes and operational reliability. These mechanisms also clarify which components are likely to be most transferable to other contexts: while tool choices may vary, the combination of (i) boundary-focused testing, (ii) runtime evidence gates, and (iii) clear ownership structures represents a generally applicable pattern for reducing "islands of automation" and improving repeatability across teams.

#### 6.3.3. Discussion

Using the results, it is possible to discuss the hypotheses as follows:

- H1 supported: large, statistically significant improvements in DF and LT show that operationalizing the framework (plan→build→CI/CD→release→operate→learn) increases throughput. This is consistent with prior DevOps research linking automation and flow to performance [27].
- H2 supported: CFR and MTTR decreased meaningfully. The effect is plausibly mediated by observability, runbooks, and rollback policies, echoing SRE practice [28].
- H3 supported: higher SLO attainment and lower error-budget burn demonstrate that runtime evidence governs delivery decisions, not just post hoc monitoring.
- H4 partially supported: capability indicators (coverage, review time, confidence) improved; longer follow-up is needed to confirm persistence.

Related to internal validity, ITS/DiD designs, stable measurement definitions, and robustness checks mitigate history and maturation threats. Concerning external validity, although shown on a Kubernetes-based stack, the framework's architecture-agnostic layers and adapters (microservices, modular Monolith, SOA) make the approach transferable, with replication across more domains as future work. Construct Validity. Using DORA and SLO/error-budget aligns outcomes with accepted constructs [27,28].

## 7. Conclusions

In this work, a prescriptive and architecture-agnostic framework for highly efficient software delivery operationalizes the complementarity between DevOps (flow, automation, observability) and architectural modularity (e.g., microservices), while remaining applicable to modular Monolith and SOA contexts. The framework is structured in layered form: a normative core (principles, roles, metrics) and specialization layers comprising decision matrices and guardrails, phase contracts (inputs/activities/outputs), and evidence requirements and quality gates. Concretely, it supplies (i) an architecture suitability matrix (MS vs. modular Monolith vs. SOA), (ii) standardized roles/RACI and artifacts per phase, and (iii) objective promotion/rollback policies governed by SLO/error-budget gates and DORA flow metrics. These elements move beyond descriptive process maps toward enforceable governance that ties build-time and runtime evidence to release decisions.

Methodologically, the framework was designed and refined under Design Science Research, with iterative expert feedback informing the naming, sequencing, and optionality of phases (e.g., alternative placements for integration testing). This choice aligns the artifact's structure with the stated design objectives (O1–O4) and the need for artifact-centric, utility-seeking research in software engineering practice.

A key contribution is the embedding of objective runtime evidence into delivery governance. Promotion is conditioned on SLO attainment during controlled rollouts (e.g., canary), respecting error-budget policies; rollback is triggered on SLO breach. At the same time, value-stream health is monitored via DORA trends (deployment frequency, lead time, change-failure rate, and MTTR), enabling continuous alignment between engineering flow and product KPIs. These mechanisms are formalized in the phase contracts ("Release and Verify", "Operate and Observe", "Learn and Improve") as auditable gates and outputs [17,27,28,30].

Usefulness beyond microservices is ensured through adapters that swap only the specialization layer while keeping the normative core intact: modular monolith (module-level contracts and feature-flag rollouts), SOA/multi-organization (compatibility and schema governance with gateway-mediated staged releases), and cloud/edge variants (function/runtime orchestration, SLOs reflecting edge constraints). This design decision addresses the external validity concern that a DevOps + Microservices view could be too narrow.

In demonstration and evaluation, we instantiated the framework as a seven-step, gated cycle with explicit artifacts and roles. We assessed it through expert-informed iteration and a structured plan to capture quantitative signals (SLO attainment, error-budget burn, DORA trends) and qualitative adoption outcomes (operating model clarity, training completion). This mixed-evidence strategy clarifies how the framework is meant to be used and measured in practice.

Implications for practice. The framework offers (a) decision support (when and how to adopt MS vs. alternatives), (b) governance by evidence (SLO gates and error budgets as promotion policy), (c) repeatable phase contracts (inputs/outputs/roles), and (d) a pragmatic adoption playbook (maturity levels, training paths). For organizations facing

toolchain sprawl, skill gaps, or weak ties between observability and release governance, these guardrails provide actionable structure rather than descriptive guidance.

The study has some limitations, namely, the current validation emphasizes expert-guided refinement and single-context instantiation; while appropriate in early DSR cycles, broader generalization requires multi-team, multi-system deployments with longitudinal measurement and statistical baselines. Threats include selection bias in experts and the confounding effect of concurrent process changes on outcome metrics.

As future work, it is planned (i) controlled field evaluations across diverse architectures and domains with pre-registered SLO/DORA baselines; (ii) an extended economics lens (cost-of-delay, incident cost curves) to quantify business impact; (iii) deeper integration of security and compliance gates (e.g., SBOM, policy-as-code) within the phase contracts; and (iv) a publicly available playbook and reference implementation to support replication and comparative studies [27,28,30].

In summary, we believe this research contributes to the definition of a generalizable, evidence-driven framework that unifies architectural decision-making, delivery governance, and operational feedback into a cohesive method. By making runtime evidence first-class and decoupling the normative core from the architectural specializations, it provides a pragmatic path for organizations to improve software delivery regardless of architecture, while offering a clear experimental agenda for further empirical validation.

**Author Contributions:** Conceptualization, V.S., M.C.S., A.S. and H.S.M.; methodology, V.S., M.C.S., A.S. and H.S.M.; software, D.B.; validation, D.B., V.S., M.C.S., A.S. and H.S.M.; formal analysis, D.B. and V.S.; investigation, D.B. and V.S.; resources, D.B.; data curation, D.B. and V.S.; writing—original draft preparation, D.B.; writing—review and editing, V.S., M.C.S., A.S. and H.S.M.; visualization, V.S. and A.S.; supervision, V.S. and H.S.M.; project administration, V.S.; funding acquisition, H.S.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** The study was conducted in accordance with the Declaration of Helsinki, and approved by the Institutional Review Board of NOVA IMS (INFSYS2025-12-105128) on 12 November 2025.

**Informed Consent Statement:** Informed consent for participation was obtained from all subjects involved in the study.

**Data Availability Statement:** Aggregated dataset shared, with only aggregated weekly metrics and anonymized incident counts (no raw logs, no repo IDs). The raw data supporting the conclusions of this article will be made available by the authors on request.

**Acknowledgments:** The authors also acknowledge all the support provided by MagIC to this research.

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

## References

1. Modalavalasa, G. The Role of DevOps in Streamlining Software Delivery: Key Practices for Seamless CI/CD. *Int. J. Adv. Res. Sci. Commun. Technol.* **2021**, *1*, 258–267. [CrossRef]
2. Wayner, P. How to Choose the Right Software Architecture: The Top 5 Patterns. TechBeacon. 2023. Available online: https://techbeacon.com/app-dev-testing/top-5-software-architecture-patterns-how-make-right-choice (accessed on 6 January 2024).
3. Amazon. What Is DevOps?—DevOps Models Explained. Amazon Web Services (AWS), Inc. 2023. Available online: https://aws.amazon.com/devops/what-is-devops/ (accessed on 5 December 2023).
4. Jacobs, M.; Casey, C.; Kaim, E. What Are Microservices? Azure DevOps 2022. 2022. Available online: https://learn.microsoft.com/en-us/devops/deliver/what-are-microservices (accessed on 5 December 2023).

5. Wickramasinghe, S. The Role of Microservices in DevOps. BMC Blogs. 2023. Available online: https://www.bmc.com/blogs/devops-microservices/ (accessed on 5 December 2023).
6. Giamattei, L.; Guerriero, A.; Pietrantuono, R.; Russo, S.; Malavolta, I.; Islam, T.; Dînga, M.; Koziolek, A.; Singh, S.; Armbruster, M.; et al. Monitoring Tools for DevOps and Microservices: A Systematic Grey Literature Review. *J. Syst. Softw.* **2024**, *208*, 111906. [CrossRef]
7. Waseem, M.; Liang, P.; Shahin, M. A Systematic Mapping Study on Microservices Architecture in DevOps. *J. Syst. Softw.* **2020**, *170*, 110798. [CrossRef]
8. DeBois, P.; Humble, J.; Molesky, J.; Shamow, E.; Fitzpatrick, L.; Dillon, M.; Phifer, B.; DeGrandis, D. DevOps: A Software Revolution in the Making? Cutter Consortium. 2021. Available online: https://www.cutter.com/journal/devops-software-revolution-making-487266 (accessed on 13 January 2024).
9. Tanzil, M.H.; Sarker, M.; Uddin, G.; Iqbal, A. A Mixed Method Study of DevOps Challenges. *Inf. Softw. Technol.* **2023**, *161*, 107244. [CrossRef]
10. Vom Brocke, J.; Hevner, A.; Maedche, A. Introduction to Design Science Research. In *Design Science Research Cases*; Progress in IS; Vom Brocke, J., Hevner, A., Maedche, A., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 1–13.
11. Peffers, K.; Tuunanen, T.; Rothenberger, M.A.; Chatterjee, S. A Design Science Research Methodology for Information Systems Research. *J. Manag. Inf. Syst.* **2007**, *24*, 45–77. [CrossRef]
12. Mohottige, T.I.; Polyvyanyy, A.; Fidge, C.; Buyya, R.; Barros, A. Reengineering software systems into microservices: State-of-the-art and future directions. *Inf. Softw. Technol.* **2025**, *183*, 107732. [CrossRef]
13. Ponce, F.; Verdecchia, R.; Miranda, B.; Soldani, J. Microservices testing: A systematic literature review. *Inf. Softw. Technol.* **2025**, *188*, 107870. [CrossRef]
14. Yaroshynskyi, M.; Puchko, I.; Prymushko, A.; Kravtsov, H.; Artemchuk, V. Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach. *Digital* **2025**, *5*, 27. [CrossRef]
15. Singh, S.; Muntean, C.H.; Gupta, S. Resilient microservices: An investigation into Istio effectiveness in Kubernetes. *Clust. Comput.* **2026**, *29*, 27. [CrossRef]
16. Kim, G.; Humble, J.; Debois, P.; Willis, J. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*; IT Revolution Press, LLC: Portland, OR, USA, 2016.
17. Fitzgerald, B.; Stol, K.-J. Continuous Software Engineering: A Roadmap and Agenda. *J. Syst. Softw.* **2017**, *123*, 176–189. [CrossRef]
18. Grande, R.; Vizcaíno, A.; García, F.O. Is It Worth Adopting DevOps Practices in Global Software Engineering? Possible Challenges and Benefits. *Comput. Stand. Interfaces* **2024**, *87*, 103767. [CrossRef]
19. Di Francesco, P.; Lago, P.; Malavolta, I. Architecting with Microservices: A Systematic Mapping Study. *J. Syst. Softw.* **2019**, *150*, 77–97. [CrossRef]
20. Knoche, H.; Hasselbring, W. Drivers and Barriers for Microservice Adoption—A Survey among Professionals in Germany. *Enterp. Model. Inf. Syst. Archit. EMISAJ* **2019**, *14*, 1–35. [CrossRef]
21. Camunda. New Research Shows 63 Percent of Enterprises Are Adopting Microservices Architectures Yet 50 Percent Are Unaware of the Impact on Revenue-Generating Business Processes. Camunda. 2018. Available online: https://camunda.com/press_release/new-research-shows-63-percent-of-enterprises-are-adoptingmicroservices/ (accessed on 7 January 2024).
22. Gokarna, M. DevOps phases across Software Development Lifecycle. *Preprint* **2021**. [CrossRef]
23. Colomo-Palacios, R.; Fernandes, E.; Soto-Acosta, P.; Larrucea, X. A Case Analysis of Enabling Continuous Software Deployment through Knowledge Management. *Int. J. Inf. Manag.* **2018**, *40*, 186–189. [CrossRef]
24. Khattak, K.-N.; Qayyum, F.; Naqvi, S.S.A.; Mehmood, A.; Kim, J. A Systematic Framework for Addressing Critical Challenges in Adopting DevOps Culture in Software Development: A PLS-SEM Perspective. *IEEE Access* **2023**, *11*, 120137–120156. [CrossRef]
25. Richardson, C. Microservices Pattern: Monolithic Architecture Pattern. Blog. 2024. Available online: https://microservices.io/patterns/monolithic.html (accessed on 9 February 2025).
26. Cortés Ríos, J.C.; Embury, S.M.; Eraslan, S. A Unifying Framework for the Systematic Analysis of Git Workflows. *Inf. Softw. Technol.* **2022**, *145*, 106811. [CrossRef]
27. Forsgren, N.; Humble, J.; Kim, G. *Accelerate: The Science Behind DevOps: Building and Scaling High Performing Technology Organizations*; IT Revolution: Portland, OR, USA, 2018.
28. Beyer, B.; Jones, C.; Petoff, J.; Murphy, N.R. *Site Reliability Engineering: How Google Runs Production Systems*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016.
29. Pine, D.; Montemagno, J.; Hazell, L.; Moseley, D.; Erhardt, E.; Matthews, A.; Fowler, D. NET Aspire Overview—NET Aspire 2024. Available online: https://aspireify.net/a/250909/.net-aspire-overview (accessed on 9 February 2025).
30. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*; Pearson Education: London, UK, 2010.

31. Bernal, J.L.; Cummins, S.; Gasparrini, A. Interrupted time series regression for the evaluation of public health interventions: A tutorial. *Int. J. Epidemiol.* **2017**, *46*, 348–355. [CrossRef] [PubMed]
32. Linden, A. Conducting interrupted time-series analysis for single- and multiple-group comparisons. *Stata J.* **2015**, *15*, 480–500. [CrossRef]
33. Angrist, J.D.; Pischke, J.-S. *Mostly Harmless Econometrics: An Empiricist's Companion*; Princeton University Press: Princeton, NJ, USA, 2009.
34. Lechner, M. The estimation of causal effects by difference-in-difference methods. *Found. Trends Econom.* **2011**, *4*, 165–224. [CrossRef]
35. Bertrand, M.; Duflo, E.; Mullainathan, S. How much should we trust differences-in-differences estimates? *Q. J. Econ.* **2004**, *119*, 249–275. [CrossRef]