IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

MONOLITHIC VS. MICROSERVICES UNDER LOAD: A SYSTEMATIC LITERATURE REVIEW OF PERFORMANCE AND SCALABILITY EVIDENCE ACROSS TECHNOLOGY STACKS

[1]Mayank Malviya, [2]Urjita Thakar, [3]Sudarshan Varma

[1]Research Fellow, [2]Professor, Computer Engineering Department, SGSITS,indore [3] Director of Technology, Advanz101 Systems Pvt. Ltd., Indore

[1] Department of Computer Engineering,

[1] Shri Govindram Seksaria Institute of Technology and Science(SGSITS), Indore, India

Abstract: Many people have research about splitting a monolithic application into microservices and its effect on performance, however little solid evidence exists. In this review article, we have reviewed research articles spanning a period between 2015–2025 that compare monolithic and microservice architectures and also on designing of microservices. This article describes how the experimental setups used (topology, resource allocation, workload design) led to conflicting results and renders the analyzed studies in relation to three methodological criteria that were found in recent literature namely: (I)resource parity, (II)saturation characterisation, and (III)cross–stack replication.

Index Terms - microservices, monolithic architecture, performance evaluation, load testing, resource parity, systematic literature review.

I. INTRODUCTION

For more than a decade, the default recommendation for server–side systems has been microservices: i.e. breaking the application into small, independently deployable services, each owning its data, each scaling on its own. Fowler and Lewis first characterised the style, [1][2][3] and Thönes and Newman later solidified it in practitioner writing. Dragoni et al. trace its roots back to Service-Oriented Architecture and lay out the research questions that were still open even as the style was being widely adopted [4]. The promise was straightforward: independent deployability, per-service scalability, and teams that could ship without waiting on each other.

This promise came at a price, and it was a hard-learned lesson for the industry. Every service boundary that used to be an in–process function call became a network round trip instead – serialization, latency, and entirely new categories of failure that a monolith never had to deal with. The clearest evidence came from organisations that had fully adopted microservices and later moved back toward a monolithic design. The Istio project reverted its control plane from a microservices layout to a monolith after running into operational complexity and degraded performance [15]. The Amazon Prime Video team went further: they consolidated a distributed monitoring pipeline into a single process and reported close to a 90% reduction in infrastructure cost for that workload [20]. However, neither case suggests that monoliths are “the right answer.” Instead, both cases show that architectural decisions should be based on measurement rather than simply following the default choice.

These episodes raised an obvious question: does decomposing an application into microservices actually improve or reduce performance? The academic literature has been trying to answer this question since Villamizar et al.'s early cloud experiments in 2015 [5]. A decade of comparative studies later, the field still has not converged: some studies report microservices ahead on throughput, others report monoliths ahead on nearly every measured parameter [18], and still others find that the ranking flips depending on deployment configuration [16]. Performance should be considered separately from maintainability for three reasons. First, cost: infrastructure spend tracks runtime performance directly, and the Prime Video episode shows the gap can span an order of magnitude for a single workload [20]. Second, user experience: tail latency (p95/p99) is what users actually feel, and it is precisely where inter–service communication costs concentrate and where average–case analysis is least informative. Third, capacity planning: a monolith tends to fail as a block once saturated, while a microservices deployment tends to fail at its weakest service first – two very different operational postures that only measurement can distinguish.

The review ordered by Rodrigues, Rito Silva and Avritzer's 2024 finally diagnosed why a decade of studies could not agree: the disagreement traces primarily to uncontrolled differences in deployment topology, resource allocation, and workload design across studies, rather than to any settled property of the architectures themselves [21]. That diagnosis is what organises the present review. Building on it, the authors have surveryed works spanning 2014–2025 that compare monolithic and microservice performance, trace how each study's experimental choices shaped its conclusions, and position every reviewed study against the three methodological criteria– resource parity, saturation characterisation, and cross–stack replication – that this uncontrolled–variation problem points to. The review closes by stating the resulting research gap precisely and by outlining the study design it motivates.

The rest of this review is laid out in the following manner. Section II discusses the two architectural styles, and the two types of performance mechanisms and measurement substrate that any comparison between them should consider. Section III presents the comparative literature in three phases (2015-2025) to illustrate the changes in experimental frameworks and findings of the comparative literature over the years. A second literature review, covering migration and decomposition guidance, is provided in Section IV, that includes the benchmark suites and reference applications on which this second body of work is based. A rigorous comparison is made in Section V which reviews the measurement methodology, including experimental design, workload modelling and load-generation tooling In Section VI, the reviewed studies are evaluated using three methodological criteria, the resulting pattern is discussed, and the research gap is precisely formulated. The review is summarized in Section VII.

In the next section the work related to architected styles and Trades-offs has been surveyed.

II. Architectural Styles and Trade-offs

A number of researchers have contributed to characterising the two architectural styles and the trade-offs between them; this section synthesises that foundational work before Section III turns to the empirical evidence itself.

A. The two styles

A monolithic application keeps all modules within one unit, one process, and one release. It is simple to deploy and monitor, and all modules share the same failure and scaling behaviour. However, the monolithic style is tightly coupled, which means that a change to one component typically requires redeployment of the entire application [1][3]. The microservices style is a decomposition of the same functionality into independent pieces (services) that can be deployed separately, have their own data, and own logic, and that communicate via the network [1,3]. This provides independent deployment and scaling per service, but comes with the downside of the distributed systems issues that Thönes identified early: partial failure, network latency, and eventual consistency [2].

B. Four performance mechanisms

Independent of empirical outcome, the literature converges on four mechanisms that any performance comparison between the two styles must account for [1]-[4]. The first is communication changes from an in–process method call to one or more serialized network round trips, often compounded by gateway indirection and service-to-service fan-out. Runtime duplication replaces a single process image with N processes, each carrying its own runtime environment, connection pools, and caches - an overhead that weighs disproportionately on memory-hungry runtimes such as the JVM. Isolation improves under

decomposition, since a fault or garbage–collection pause in one service need not stall the others, whereas a monolith shares fate internally [2]. The granularity of scaling goes, from block–level scaling of the entire monolith, to being able to scale only the hot service, but only if you deploy the service across multiple nodes and have a very skewed load (an assumption made by many studies that rely on a single host cannot be verified) [3].

C. Containers as the measurement substrate

Containers are the near–universal measurement substrate for these comparisons. Felter et al.'s pioneering work demonstrated that Linux containers impose almost no other overhead compared to bare metal, when the time to read and write to/from I/O devices is not the limiting factor; but when using NATed container networking, it does add latency [24]. This finding underwrites Docker as a fair, low-overhead deployment mechanism in nearly every study surveyed below. It also has a subtler implication: containers also offer cgroup-based CPU and memory limits, so a container-based approach to enforcing resource parity between architectural variants is possible, and is something that most of the studies reviewed in Section III do not do.

any comparison between monolithic and microservice architectures must account for four mechanisms - communication, runtime duplication, isolation, and scaling granularity - and containers provide the measurement substrate on which resource parity can, in principle, be enforced. Section III now turns to how the empirical literature has actually measured these effects across three waves of research.

The imperial evidence of the architectural styles and Trade-offs between the two architectural styles is discussion in next section.

III. Evolution of the Comparative Literature:

The empirical comparisons between monolithic and microservice performance can be read as three successive waves, each building on the methodological limitations of the one before it.

A. First Wave (2015–2018): Establishing the Comparative Template

Villamizar et al. ran the same web application under monolithic and microservice architectures in a cloud setting, comparing response time under heavy load [5], and later extended the comparison to AWS Lambda with a cost dimension [7]. These studies established the “same application, same workload” comparative template that nearly all later work follows, but neither measured resource utilisation between versions nor pushed load to full saturation – the loads applied were moderate, so the studies say little about behaviour near the knee of the performance curve.

Ueda, Nakaike and Ohara went a level deeper, using hardware performance counters to instrument the Acme Air benchmark on common hardware and demonstrating that the microservice implementation had significantly lower throughput [6]. They attributed the difference to duplicated per–service runtime effort and network–processing overhead rather than to incidental implementation choices – one of the few studies in this era to explain a measured gap in terms of physical mechanism rather than simply reporting it. Their conclusion that the cost of microservices “lives in communication and duplication” is a hypothesis the rest of this literature repeatedly returns to.

Al-Debagy and Martinek examined throughput across a range of concurrency levels and found only minor differences at low-to-moderate concurrency, with a slight edge to the monolithic style in most cases [8]. Read together with Ueda et al.'s findings [6], this indicates that whatever “microservices tax” exists is not a fixed quantity – its magnitude depends on where along the load curve it is measured, which is itself an argument for studies that sweep the full curve rather than sampling one operating point.

The first wave was based on the same “same application, same workload” template, but the results of the comparisons do not necessarily take load to saturation level, nor do they measure resource utilisation directly, so they are restricted to moderate-load behaviour.

B. Second Wave (2019–2022): Isolating Design Patterns, Granularity, and Topology

Akbulut and Perros studied how microservice design patterns – API gateway aggregation, chained synchronous calls, and asynchronous messaging – affect latency and throughput independently of the monolith-vs-microservices choice, and found that pattern selection influences performance roughly as much as architectural style does [9]. Their result is that both styles are not a single point in the design space and to avoid that the results of a study should be documented with respect to the specific topology used.

Tapia et al. compared performance alongside software development-process metrics and concluded that microservices delivered real maintainability and flexibility benefits for development teams, while conceding a measurable negative impact on runtime performance [10]. Jayasinghe et al. isolated service granularity as an independent variable through systematic decomposition of an application into progressively smaller services, and found throughput generally decreases as granularity increases, with latency effects that vary by endpoint [11]. Zhao et al.'s later ICSA 2025 study corroborates this directly on a different system, reporting roughly 13% higher energy cost and 14% longer response times for finer-grained decomposition relative to coarser-grained systems in their largest test configuration [22] – together, [11] and [22] establish granularity as a performance variable in its own right, independent of the binary architecture choice.

Bjorndal et al. addressed the absence of common migration benchmarks and metrics by developing dedicated assessment tooling for pre/post-migration comparison [13], a matched-pairs philosophy the underlying study behind this review adopts directly. Li et al.'s systematic review of quality attributes found that performance is among the most frequently claimed but least consistently substantiated properties associated with microservice adoption [14] – a finding that is itself part of the case for controlled measurement work. Auer, Lenarduzzi, Felderer and Taibi developed an interview–based assessment framework for migration decisions and reported that the organisations they studied consistently lacked performance data about their own current systems, with their proposed measurement framework assuming a level of instrumentation most teams do not yet have [12].

The most thorough evaluation of this period is Blinowski, Ojdowska and Przybyłek's .NET-based study, whose central finding is arguably the single most important methodological result in this literature: monoliths outperform microservices when scaled vertically on a single machine, while microservices outperform monoliths when scaled horizontally across multiple machines in the cloud [16]. This single result demonstrates that deployment topology alone can reverse a comparison's conclusion, and it fixes the interpretive frame for every single-host study discussed in this review, including the one motivating it: such studies characterise the vertical-scaling regime only, and any cross-service communication overhead they report is a lower bound relative to a genuinely networked, horizontally-scaled deployment.

The second wave expanded the comparison to design patterns, granularity and migration tooling and came up with what was, in my opinion, the single most important result: deployment topology (vertical versus horizontal scaling) can determine which architecture is the winner, and that changes the interpretation of all the one-host studies in the review.

C. Current State (2023–2025): Diagnosing the Field's Disagreement

A 2023 study presented at ACM/SIGAPP SAC by Jatkiewicz and Okrój reported that the monolithic architecture outperformed microservices in nearly every measured condition from an efficiency standpoint, with the two architectures reaching parity only where request processing time was dominated by business logic rather than by inter-service communication [18]. Mangwani, Mangwani and Motwani added a multi–tenancy dimension previously absent from this literature, evaluating a multi-tenant SaaS application under both styles and showing that performance differences are sensitive to tenancy load patterns [19].

Rodrigues, Rito Silva and Avritzer's ECSA review is the pivot point for the entire body of work surveyed here. By systematically comparing the deployment topologies, resource–allocation schemes, and workload designs used across prior comparative studies, they trace the field's persistently contradictory results to these uncontrolled experimental choices rather than to any settled property of the architectures, and they explicitly recommend that future studies isolate these variables by design [21]. Zhao, De Matteis and Bogner's ICSA 2025 study adds an energy-consumption dimension on top of the throughput and latency methodological criteria already established, using a factorial design and non–parametric statistical analysis broadly consistent with the methodological literature discussed in Section V [22]. Söylemez, Tekinerdogan and Tarhan's systematic review of the wider (not purely performance-focused) microservices literature

independently identifies performance as one of the most persistent adoption challenges organisations report, with communication overhead named specifically and repeatedly [17].

Outside controlled experimentation, Mendonça, Box, Manolache and Ryan's account of Istio's architectural reversal and Kolny's Prime Video engineering blog post are frequently cited as industry corroboration that increasing decomposition can raise operational cost faster than it delivers benefit at production scale, with both sources naming communication overhead as the principal driver [15][20]. These are not controlled experiments and are treated in this review as corroborating anecdote rather than evidence with the same weight as the three waves reviewed above, but their recurrence across two large, technically sophisticated organisations is difficult to dismiss.

Taken together, these three waves show a field moving from simple response–time comparisons toward multi-dimensional evaluation – granularity, topology, energy, and multi–tenancy – without ever converging on a single answer, because, as Rodrigues, Rito Silva and Avritzer's diagnosis in the third wave shows, the disagreement was rooted in experimental method rather than in any settled property of the architectures under test [21].

A review of the literature on migration and decomposition of service is presented in next section.

IV. Migration and Decomposition Guidance

Many researchers have worked to answer the question of how an organisation can transition from one architecture to the other. Balalaie, Heydarnoori and Jamshidi's early account of an industrial migration frames microservices adoption as part of a broader DevOps practice and documents the migration patterns used in detail, treating performance as something to be monitored during the transition rather than predicted beforehand [33]. Gysel, Kölbener, Giersche and Zimmermann's Service Cutter treats service decomposition as a clustering problem over coupling metrics derived from a system's specification – an early and influential formalisation of the view that service boundaries can be computed rather than chosen by intuition [34].

Auer et al.'s assessment framework closes the loop between these two strands by describing what information an organisation needs to collect before committing to a migration [12], but as noted in “Evolution of the Comparative Literature”, their framework implicitly assumes the existence of high–quality performance data at precisely the point where the three waves reviewed there show such data is scarce and inconsistent. Read alongside the granularity findings of Jayasinghe et al. and Zhao et al. [11][22], this body of work indicates that the degree of decomposition a migration or decomposition tool recommends is itself a first–order performance variable – a scope decision any comparative study must make explicit rather than inherit silently from whichever decomposition an application's maintainers happened to choose.

The tooling this migration and comparison research relies on is itself worth examining. Gan et al.'s DeathStarBench is the most widely used open benchmark suite for microservice research at the hardware-software level, and demonstrates that service-graph depth alone can increase tail latency and network stress independent of any specific application logic [23]. It is, however, not well suited to the monolith-vs-microservices question specifically: its applications are microservice-only, with no functionally equivalent monolithic counterpart, so it cannot support a true matched–pairs comparison. This makes application families that are available in both monolithic and microservice forms especially useful for this research. Examples include Spring PetClinic and similar grocery and e-commerce applications. These projects provide functionally comparable versions that can be used for a matched-pair comparison.

Migration research treats performance as something to monitor rather than predict, and the benchmark suites developed in this area are not well suited for comparing monolithic and microservice architectures. This makes matched-pair reference applications such as Spring PetClinic a practical alternative.

Many researchers have studied how to measure and evaluate the performance of applications using monolithic and microservice architectures. A review of the related literature is presented in the next section.

V. Measurement Methodology

A reliable performance comparison requires attention to four components: how the experiment is designed and statistically analysed, how the workload is modelled, what tools are used to generate the load, and what reference applications are used for comparison. Each component is discussed below.

A. Experimental design and statistics

A comparison is only as trustworthy as the statistics behind it. Jain's foundational text on computer systems performance analysis underlies the factor selection, replication counts, and error-source taxonomy used across this literature [27]. Georges, Buytaert and Eeckhout's study of statistically rigorous Java performance evaluation is the concrete template most JVM-based studies in this review follow: using JIT compilation, garbage collection, and thread-scheduling behaviour as examples, they show that run-to-run variability can be large enough to invalidate a single-run conclusion, and they recommend explicit warm-up management, multiple repetitions, and confidence intervals instead [26]. Because the small sample sizes typical of this literature (commonly five to seven repetitions per condition) rarely satisfy the normality assumptions parametric tests require, Arcuri and Briand's guide to statistical testing for randomized software-engineering experiments, together with Romano, Kromrey, Coraggio and Skowronek's work on ordinal–data statistics, motivate non-parametric tests instead – typically the Mann-Whitney U test paired with a standardised effect-size measure such as Cliff's delta [28][29].

B. Workload modelling: open vs. closed

How a workload is generated changes what a benchmark actually measures. Schroeder, Wierman and Harchol-Balter's “open versus closed” analysis is the standard reference for this choice [25]. In a closed model, a fixed population of virtual users each wait for a response before issuing their next request, so the system's own slowdown automatically throttles the offered load – the system can never be driven past what it can absorb. In an open model, requests arrive independently of how the system is currently performing, which is closer to how real, uncontrolled internet traffic behaves, and it can therefore expose harder-to-predict, more severe degradation near saturation than a closed model ever would. Schroeder et al. show that the two models can yield qualitatively different conclusions about the identical system under test, which is why methodologically careful studies test both rather than relying on a closed virtual-user model alone.

C. Load–generation tooling

Three open-source load generators recur across this literature, and each embodies a different concurrency model. Apache JMeter uses a thread-per-user model driven through an XML-based GUI test plan; because each virtual user occupies a dedicated thread on the generator machine itself, JMeter becomes expensive to run at high concurrency, and its own saturation can masquerade as saturation of the system under test – its XML test plans are also difficult to review for correctness. Locust avoids this by scripting workloads in Python with an event-driven model that scales better per generator core, though at the cost of added operational complexity from its distributed-worker architecture. k6, scripted in JavaScript over a Go runtime and documented at [30], is the tool most methodologically recent studies in this review favour, because it natively supports both a closed virtual–user executor and an open constant-arrival-rate executor from the same journey script – letting a single script exercise both workload models discussed above without a separate implementation for each.

D. Reference application artifacts

Finally, a comparison needs a subject: an application that genuinely exists in both monolithic and microservices form. Two categories of artifact underpin the comparative studies discussed in this review – the load-testing and monitoring tooling itself [23][30], already covered above, and the matched-pair reference applications used as study subjects. The Spring PetClinic community repositories [31] and comparable open-source grocery/e-commerce application repositories [32] are examples of the latter: application families intentionally maintained in both monolithic and microservices form by their own communities, and – as Section IV notes – a practical substitute for the matched-pairs coverage DeathStarBench does not provide.

In short, a methodologically sound comparison needs statistically grounded experimental design, both workload models, tooling capable of generating both, and a genuine matched-pair reference application.

VI. Discussion

Not all 32 works reviewed in this survey report a direct empirical measurement comparing a monolithic and a microservice variant of the same application – the architectural literature (Section II), the migration and tooling guidance (Section IV), and the measurement-methodology references (Section V) inform how such a measurement should be read, but do not themselves produce one.

Each row shows how the corresponding study performed against the three criteria used throughout this review: whether resources were allocated fairly between the monolithic and microservice variants (Resource parity), whether the study tested the systems up to or beyond saturation rather than stopping at a single moderate load level (Saturation focus), and whether the same comparison was repeated on more than one technology stack (Cross-stack). "Not reported" means that the study provides no information about that criterion; "Partial" means that the criterion is only partly addressed, for example, by testing a single load level instead of performing a full saturation test.

Table I therefore includes only the primary comparative studies, that is, the studies that directly measure monolithic versus microservice performance. These studies are evaluated using three methodological criteria identified by [21] as major weaknesses in the existing literature: resource parity, saturation characterisation, and cross-stack replication.

TABLE I. CONTRIBUTION AND METHODOLOGICAL POSITIONING OF THE REVIEWED LITERATURE

Study	Year	Stack(s)	Resource parity	Saturation focus	Cross–stack
Villamizar et al. [5][7]	2015/17	Java	Not reported	No	No
Ueda et al. [6]	2016	Java	Partial	No	No
Al–Debagy & Martinek [8]	2018	Java	Not reported	No	No
Akbulut & Perros [9]	2019	Java	Not reported	No	No
Jayasinghe et al. [11]	2020	Java	Partial	Partial	No
Blinowski et al. [16]	2022	.NET	Partial	Partial	No
Jatkiewicz & Okrój [18]	2023	Java	Partial	Partial	No
Mangwani et al. [19]	2023	Java	Not reported	No	No
Zhao et al. [22]	2025	Java	Yes	Partial	No
This work	2026	Java + MERN	Enforced	Yes (stress/spike/soak)	Yes

The results in Table I are pattern. Of the nine previous comparative studies included in Table I, four report resource parity as "Not reported" (Villamizar et al. [5][7], Al-Debagy and Martinek [8], Akbulut and Perros [9], and Mangwani et al. [19]), four report it as "Partial" (Ueda et al. [6], Jayasinghe et al. [11], Blinowski et al. [16], and Jatkiewicz and Okrój [18]), and only Zhao et al. [22] report it as fully addressed. On saturation, five of the nine studies report no saturation testing at all, and the remaining four report only partial coverage of the load curve. For cross-stack replication, all nine previous studies use only one technology stack. Therefore, these studies cannot clearly separate the effect of the architecture from the effect of a specific technology stack, such as Java/JVM or .NET. This work is the only entry in the table with resource parity fully enforced, saturation explicitly characterised, and testing repeated across two independent stacks (Java and MERN).

Read together, these numbers say something more specific than "the literature is inconsistent." They show that the differences mainly arise from three methodological issues in performance comparisons: resource parity, saturation testing, and cross-stack replication, and that no prior study has closed more than one of the three gaps at a time: Zhao et al. close the parity gap but not saturation or cross-stack; Jayasinghe et al.,

Blinowski et al., and Jatkiewicz and Okrój partially address saturation but leave parity partial and cross-stack untouched; and every single study, without exception, leaves cross-stack replication completely open. These findings support the need for a study that combines three design choices: enforcing equal resource allocation, testing saturation and recovery, and repeating the same experiment across two technology stacks. Together, these choices help reduce the effect of resource allocation, load level, and technology-specific factors on the results.

When the three waves are considered together rather than as separate studies, a clear pattern emerges that is not stated explicitly in the individual papers. The disagreements in the literature are related to the same three methodological criteria summarised in Table I, and each criterion has a possible explanation based on how the systems were tested. When resource parity is not reported or is only partially addressed, microservices may receive a larger total CPU or memory allocation than the monolith. This can give microservices an advantage that is caused by the resource allocation rather than by the architecture itself. This difference may explain part of the disagreement between studies that find microservices competitive [10] and those that find monoliths clearly better [18]. Where saturation is not characterised, a study run only at moderate load (as in the earliest work [5][7][8]) samples a single point on a curve that later, more saturation-focused work shows can rank the two architectures oppositely on either side of the knee [8]. And where cross-stack replication is absent, every directional claim in this literature is implicitly a claim about the JVM specifically, since the large majority of reviewed studies use Java or .NET subjects – leaving open, as Section I notes, whether the same mechanisms (runtime duplication, connection-pool overhead, GC-driven latency variance) manifest the same way in an event-loop runtime such as Node.js, where the dominant cost structure is different.

The one point of near–consensus across otherwise conflicting studies is Blinowski et al.'s topology finding [16]: vertical and horizontal scaling regimes favour opposite architectures, and every single-host study in this review – the vast majority of them – is therefore reporting a vertical-regime result whether or not it says so explicitly. This has a direct implication for how the findings summarised in Table I should be read: agreement or disagreement between two single-host studies is informative about the vertical regime specifically, and neither confirms nor contradicts what a horizontally-scaled deployment of the same applications would show.

The disagreement in the literature can be explained by the three methodological weaknesses identified in Table I. One finding that is consistent across studies is that deployment topology can reverse the conclusion of a performance comparison. The following subsection now states the resulting gap precisely.

Three shortcomings recur across this literature when read as a whole. First, resource fairness between variants is rarely enforced or reported, the exact confound [21] identifies as the field's primary source of disagreement; Table I shows this as “Not reported” or “Partial” for the majority of pre–2025 studies. Second, cross-stack replication is essentially absent: no reviewed study runs the same experimental design across two independent runtimes, so it remains unknown whether a JVM-based finding generalises to an event–loop stack such as Node.js, or vice versa. Third, saturation behaviour is thinly reported – most studies operate at fixed, moderate load rather than characterising the knee of the latency curve, the degradation slope beyond it, and recovery from spikes, despite this being precisely the information operators need for capacity planning, and despite Al-Debagy and Martinek's own results suggesting the ranking between architectures can shift depending on where along the load curve it is measured [8].

Taken together, these gaps motivate a study design that enforces kernel–level resource parity between monolithic and microservices deployments, replicates an identical protocol across two functionally-equivalent application pairs on distinct technology stacks (Java/Spring and MERN), and explicitly characterises saturation and recovery – knee point, degradation slope, and spike-recovery time – rather than restricting measurement to a single steady-state load level. No single study identified in this review combines all three design choices.

VII. Conclusion

This review set out to explain why a decade of empirical comparisons between monolithic and microservice performance has not produced a clear conclusion, and to state the resulting research gap clearly enough to motivate a new study design.

- The disagreement is methodological, not architectural. Studies differ in how fairly they allocate resources, how they define and probe saturation, and whether they test more than one technology stack – not in any settled property of monoliths or microservices themselves.
- Communication overhead and runtime duplication are important cost sources in microservices, while independent deployment, isolation, and service-level scaling provide important benefits [2][6].
- Design-pattern choice within a style (gateway aggregation, chained calls, async messaging) is roughly as consequential as the choice of style itself [9].
- Granularity of decomposition is an independent performance variable in its own right, not a side-effect of the architecture choice [11][22].
- Deployment topology (vertical versus horizontal scaling) can reverse a comparison's conclusion outright, and is the single closest thing to a consensus finding in this literature [16].
- Resource parity is rarely enforced: most reviewed studies either do not report how CPU and memory were allocated between the monolithic and microservices variants, or report it only partially, leaving open the possibility that an observed advantage is a resourcing artefact rather than an architectural one.
- Saturation behaviour is thinly characterised: the majority of studies measure at a single, moderate load level rather than tracing the full curve, so the knee point, the degradation slope beyond it, and recovery from spikes remain largely unreported, even though this is exactly the information needed for capacity planning.
- **Cross-stack replication is absent: every reviewed study tests a single technology stack, so the existing results cannot clearly distinguish an architectural effect from an effect specific to the JVM or .NET runtime.**
- No study identified in this review addresses all three gaps at once. The combination of enforced resource parity, full saturation testing, and cross-stack replication therefore represents the main research gap identified in this review. It is this gap that the underlying study aims to address.

In short, this review shows that a decade of contradictory results does not mean the question is unanswerable. Rather, the existing studies have used different experimental conditions that make their results difficult to compare directly. The study proposed in this work aims to address all three gaps.

References

[1] M. Fowler and J. Lewis, “Microservices,” martinfowler.com, Mar. 25, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] J. Thönes, “Microservices,” IEEE Software, vol. 32, no. 1, p. 116, Jan.–Feb. 2015.

[3] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow,” in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216.

[5] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in Proc. 10th Comput. Colombian Conf. (10CCC), Bogotá, Colombia, 2015, pp. 583–590.

[6] T. Ueda, T. Nakaike, and M. Ohara, “Workload characterization for microservices,” in Proc. IEEE Int. Symp. Workload Characterization (IISWC), Providence, RI, USA, 2016, pp. 1–10.

[7] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, “Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures,” Service Oriented Computing and Applications, vol. 11, no. 2, pp. 233–247, 2017.

[8] O. Al-Debagy and P. Martinek, “A comparative review of microservices and monolithic architectures,” in Proc. IEEE 18th Int. Symp. Comput. Intell. Informatics (CINTI), Budapest, Hungary, 2018, pp. 149–154.

[9] A. Akbulut and H. G. Perros, “Performance analysis of microservice design patterns,” IEEE Internet Computing, vol. 23, no. 6, pp. 19–27, Nov.–Dec. 2019.

[10] F. Tapia, Y. Mora-Rivera, M. Fonseca, and P. Ordoñez, “From monolithic systems to microservices: A comparative study of performance,” Applied Sciences, vol. 10, no. 17, art. no. 5797, 2020.

[11] M. Jayasinghe, T. Nguyen, S. Rajapakse, and I. Sommerville, “An analysis of throughput and latency behaviours under microservice decomposition,” in Proc. Int. Conf. Web Eng. (ICWE), Lecture Notes in Computer Science, vol. 12128. Cham, Switzerland: Springer, 2020, pp. 53–69.

[12] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, “From monolithic systems to microservices: An assessment framework,” Information and Software Technology, vol. 137, art. no. 106600, 2021.

[13] N. Bjorndal, A. Bucchiarone, F. Durán, M. Fazzolari, A. Marconi, M. Nicola, and N. Puliafito, “Benchmarks and performance metrics for assessing the migration to microservice-based architectures,” Journal of Object Technology, vol. 20, no. 2, pp. 1–19, 2021.

[14] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. Babar, “Understanding and addressing quality attributes of microservices architecture: A systematic literature review,” Information and Software Technology, vol. 131, art. no. 106449, 2021.

[15] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, “The monolith strikes back: Why Istio migrated from microservices to a monolithic architecture,” IEEE Software, vol. 38, no. 5, pp. 17–22, Sep.–Oct. 2021.

[16] G. Blinowski, A. Ojdowska, and A. Przybyłek, “Monolithic vs. microservice architecture: A performance and scalability evaluation,” IEEE Access, vol. 10, pp. 20357–20374, 2022.

[17] M. Söylemez, B. Tekinerdogan, and A. Koluıksa Tarhan, “Challenges and solution directions of microservice architectures: A systematic literature review,” Applied Sciences, vol. 12, no. 11, art. no. 5507, 2022.

[18] P. Jatkiewicz and S. Okrój, “Differences in performance, scalability, and cost of using microservice and monolithic architecture,” in Proc. 38th ACM/SIGAPP Symp. Appl. Comput. (SAC), Tallinn, Estonia, 2023, pp. 1655–1663.

[19] P. Mangwani, N. Mangwani, and S. Motwani, “Evaluation of a multitenant SaaS using monolithic and microservice architectures,” SN Computer Science, vol. 4, art. no. 375, 2023.

[20] M. Kolny, “Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%,” Amazon Prime Video Tech Blog, Mar. 2023. [Online]. Available: https://www.primevideotech.com

[21] H. Rodrigues, A. Rito Silva, and A. Avritzer, “Performance comparison of monolith and microservice architectures: An analysis of the state of the art,” in Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium, Lecture Notes in Computer Science, vol. 14590. Cham, Switzerland: Springer, 2024, pp. 185–199.

[22] Y. Zhao, T. De Matteis, and J. Bogner, “How does microservice granularity impact energy consumption and performance? A controlled experiment,” in Proc. IEEE 22nd Int. Conf. Softw. Archit. (ICSA), 2025, pp. 1–12.

[23] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems,” in Proc. 24th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS), Providence, RI, USA, 2019, pp. 3–18.

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” in Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS), Philadelphia, PA, USA, 2015, pp. 171–172.

[25] B. Schroeder, A. Wierman, and M. Harchol-Balter, “Open versus closed: A cautionary tale,” in Proc. 3rd USENIX Symp. Netw. Syst. Design Implement. (NSDI), San Jose, CA, USA, 2006, pp. 239–252.

[26] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically rigorous Java performance evaluation,” in Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Program. Syst. Appl. (OOPSLA), Montreal, Canada, 2007, pp. 57–76.

[27] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York, NY, USA: Wiley, 1991.

[28] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in Proc. 33rd Int. Conf. Softw. Eng. (ICSE), Honolulu, HI, USA, 2011, pp. 1–10.

[29] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, “Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?” presented at the Annu. Meeting of the Florida Assoc. of Institutional Research, 2006.

[30] Grafana Labs, “k6 documentation,” 2024. [Online]. Available: https://grafana.com/docs/k6/latest/. [Accessed: Aug. 2026].

[31] Spring Community, “Spring PetClinic (monolithic and microservices variants),” GitHub repository. [Online]. Available: https://github.com/spring-petclinic

[32] “Grocery online shopping application (monolithic and microservices variants),” GitHub repository. [Online]. Available: https://github.com/codergogoi/Grocery_Online_Shopping_App

[33] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables DevOps: Migration to a cloud-native architecture,” IEEE Software, vol. 33, no. 3, pp. 42–52, May–Jun. 2016.

[34] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, “Service Cutter: A systematic approach to service decomposition,” in Service-Oriented and Cloud Computing (ESOCC 2016), Lecture Notes in Computer Science, vol. 9846. Cham, Switzerland: Springer, 2016, pp. 185–200.