Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

WORKLOAD-AWARE DEPLOYMENT DECISION-MAKING IN MICROSERVICE ARCHITECTURES: AN EMPIRICAL FRAMEWORK WITH PERFORMANCE INDEX

NIDHI VANIYAWALA [1], KAMLENDU KUMAR PANDEY [2]

[1] Assistant Professor, Shree Ramkrishna Institute of Computer Education and Applied Sciences, Sarvajanik University, Surat, India

[2] Associate Professor, Department of Information Technology, Veer Narmad South Gujarat University, Surat, India

E-mail: [1]nidhi.vaniyawala@srki.ac.in, [2]kspandey@vnsgu.ac.in

ABSTRACT

Microservice architectures support flexible deployment across heterogeneous environments; however, selecting an appropriate deployment strategy for varying workload characteristics remains a significant challenge. This study presents an empirical workload-aware evaluation of monolithic, microserver, and orchestrated deployment environments using a normalized Performance Index (PI) to measure deployment efficiency. Two workload categories were experimentally evaluated: read-intensive single-service operations and multi-service transactional workflows. The proposed PI provides a unified quantitative metric for comparing heterogeneous deployment environments based on performance behaviour under varying workloads. Experimental results demonstrate a strong workload-dependent variation in deployment efficiency. For read-intensive workloads, monolithic deployment achieved the highest efficiency (PI = 0.85), outperforming microserver (PI = 0.34) and orchestrated deployments (PI = 0.13). Conversely, for multi-service transactional workloads, orchestrated deployment achieved the best performance (PI = 0.70), while monolithic deployment exhibited minimal efficiency (PI = 0.06) and microserver deployment showed moderate performance (PI = 0.31). The findings confirm that no single deployment paradigm is universally optimal for all workload types. The proposed PI-based evaluation framework supports workload-aware deployment selection and enables data-driven optimization of microservice systems within Intelligent DevOps environments.

Keywords: *Microservice Architecture, Empirical Software Engineering, Containerization, Intelligent DevOps, Performance Benchmarking, Observability*

1. INTRODUCTION

1.1 Background and Motivation

Microservice Architecture (MSA) is a software architectural style in which applications are composed of independently deployable, loosely coupled services that communicate through lightweight protocols. MSA has emerged as a dominant paradigm for building scalable and modular software systems due to its support for independent service evolution, rapid deployment, and flexible scalability. In modern DevOps ecosystems, microservices are deployed across heterogeneous runtime environments, including monolithic execution platforms, lightweight standalone microservers, and fully orchestrated containerized infrastructures.

A monolithic deployment refers to an execution environment in which multiple application components operate within a single deployable unit and shared runtime environment. In contrast, a microserver-based deployment executes services independently using lightweight runtime containers such as Payara Micro, without centralized orchestration overhead. Orchestrated deployments utilize container orchestration platforms such as Kubernetes to automate service deployment, scaling, load balancing, and fault management across distributed environments. Although these deployment paradigms provide flexibility and scalability, deployment efficiency in microservice systems is highly sensitive to workload characteristics.

Workload characteristics describe the operational behaviour of an application, including

communication intensity, service interaction patterns, concurrency demands, and transaction complexity. Microservice systems do not exhibit uniform performance behaviour across operational contexts [1]. For example, read-intensive single-service interactions are typically latency-sensitive and require minimal execution overhead, whereas multi-service transactional workflows involve inter-service coordination, distributed communication, state propagation, and concurrency management, making them more sensitive to communication overhead and consistency constraints.

In practical enterprise systems, microservice applications commonly consist of heterogeneous and co-existing workload patterns rather than isolated operational behaviours. Consequently, different components of the same application may exhibit distinct performance and scalability requirements. However, deployment strategies are often selected at the system level without explicitly accounting for workload-specific behaviour. This may lead to inefficient resource utilization, increased operational overhead, and inconsistent system performance.

1.2 Problem Statement

Existing research on microservice performance primarily emphasizes aggregate system behaviour or platform-specific optimizations, with limited focus on how distinct workload patterns interact with deployment paradigms. Furthermore, current approaches lack a unified, multi-dimensional framework for translating performance and observability data into actionable deployment strategies. As a result, deployment decisions in practice often rely on heuristic or trial-and-error approaches, leading to inefficient resource utilization, increased operational costs, and inconsistent system performance.

1.3 Objectives and Research Questions

To address these limitations, this study proposes a workload-aware empirical framework for analyzing performance behaviour and guiding deployment decisions in microservice architectures. The investigation is guided by the following research questions:

- **RQ1:** How do different deployment paradigms impact the performance characteristics of read-intensive single-service interactions and multi-service transactional workflows?
- **RQ2:** Which operational metrics yield the most actionable insights for environment tuning and selection?
- **RQ3:** Can a normalized, multi-metric analysis be designed to derive a Performance Index (PI) for consistent and fair ranking of deployment environments?
- **RQ4:** How can these empirical insights be integrated into Intelligent DevOps workflows to enable continuous, data-driven performance optimization?

1.4 Contributions

This paper makes the following key contributions:

- **Workload-Aware Evaluation Framework**
Establishes that deployment efficiency in microservice architectures is workload-dependent and introduces a structured approach for aligning deployment strategies with workload characteristics.

- **Performance Index (PI) as a Decision-Support Metric**
Proposes a normalized Performance Index (PI) [0, 1] that aggregates heterogeneous performance metrics into a single, comparable measure for evaluating deployment efficiency across environments.

- **Empirical Evidence of Deployment Trade-offs**
Demonstrates, through experimental evaluation, the inversion in deployment suitability—monolithic deployment excels for read-intensive workloads, while orchestrated deployment performs best for multi-service transactional workloads.

- **Decision-Support for Deployment Selection**
Provides a quantitative, data-driven mechanism for selecting appropriate deployment environments, enabling workload-aware optimization in microservice systems.

These contributions aim to help both practitioners in optimizing real-world deployments and researchers in building reproducible benchmarks and frameworks for intelligent DevOps workflows.

2. RELATED WORK

The performance of microservices has been studied from multiple perspectives, ranging from service-level execution characteristics to deployment and orchestration strategies. Recent literature reflects a growing effort to systematically evaluate microservice workloads, yet findings remain fragmented across dimensions such as performance evaluation, environment-specific trade-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

offs, empirical benchmarking, and DevOps-driven decision support. To provide structure, we categorize prior work into four themes: (i) performance evaluation in microservices, (ii) deployment environment analysis, (iii) empirical studies and benchmarking techniques, and (iv) DevOps and decision support strategies.

2.1 Performance Evaluation in Microservices

Cortellessa et al. (2024) proposed an approach that leverages genetic algorithms (NSGA-II) to optimize trade-offs among performance, cost, and energy consumption in cloud-based microservice architectures. Their method evaluates deployment configurations with an eye toward sustainability and runtime efficiency [2]. Ayas et al. (2023), in an empirical study on microservices migration, highlighted the complexity arising from inter-service communication, transactional consistency, and fault handling. They underscore the importance of robust orchestration, which supports workload-specific deployment strategies [3]. Meijer et al. (2024) conducted an experimental evaluation of architectural design patterns—namely Gateway Aggregation, Gateway Offloading, and Pipes and Filters—showing that model-based performance predictions closely align with real-world benchmarks in terms of latency and resource utilization, reinforcing the need to validate theoretical models with empirical observations [4]. X. Zhou et al. (2023), investigated industrial practitioners' experiences via qualitative interviews across 20 companies, highlighting common microservice pains like decomposition, monitoring, and evaluation [5]. Xu et al. (2024) addressed the joint optimization of microservice instance deployment and request routing in cloud data centers, where traditional models often fail to capture the dynamic nature and interdependencies of microservices. They proposed a hybrid genetic and local search algorithm for instance deployment and a probabilistic forwarding mechanism for routing. Using an open Jackson queuing network for performance analysis, their approach demonstrated a 37%–67% reduction in average response latency and an 8%–115% increase in request success rates compared to baseline strategies [6]. Singh et al. (2022) presented an Event-Driven Architecture (EDA) model for data-centric microservices integrated with distributed version control platforms such as Git and SVN. Their work demonstrated that EDA-based systems support efficient real-time event processing, minimize latency, and enhance decision-making capabilities when compared with conventional message-driven communication approaches [7]. Kolla (2024) examined the evolution of enterprise applications from monolithic systems toward distributed architectures based on microservices and event-driven computing. The study emphasized improvements in scalability, resilience, and operational efficiency across domains including finance, e-commerce, and logistics, while also discussing emerging trends such as artificial intelligence integration, edge computing, and advanced observability mechanisms [8]. Muthusamy (2025) explored event-driven data engineering within microservice ecosystems, focusing on asynchronous communication among loosely coupled services. The study highlighted the role of event producers, routers, and consumers in enabling scalable and real-time data processing while addressing challenges such as event duplication and data consistency through techniques including Command Query Responsibility Segregation (CQRS) and event sourcing [9]. Srijith et al. (2022) analyzed asynchronous inter-service communication in microservice architectures using Kafka Connect. Their research proposed an optimization methodology for publisher–subscriber communication that reduced CPU and memory overhead while improving throughput and response time during microservice interactions [10].

2.2 Deployment Environment Analysis

Fritzsch et al. (2023) performed a rapid review and case study on application of microservices and DevOps to distributed, industrial systems, revealing that dynamic placement, monitoring, and deployment strategies are key to performance and reliability [11]. Kang et al. (2021) presented a detailed evaluation of multiple Kubernetes CNI plugins, showing how network stack selection significantly impacts latency, throughput, and CPU overhead. The findings provide actionable guidance for tuning environments to achieve desired performance targets [12]. Chippagiri (2024) investigates Kubernetes networking performance by evaluating CNIs such as Cilium, Flannel, Calico, and Antrea, along with system tuning profiles, providing insights into trade-offs between simplicity, security, throughput, and scalability for optimizing containerized applications [13]. Koukis et al. (2024) analysed Kubernetes networking in constrained edge environments, revealing how limited resources amplify network bottlenecks. Their study highlights the need for careful CNI selection to ensure reliability in edge deployments [14]. Zhu et al. (2023) broke down the overheads introduced by service mesh sidecars, quantifying CPU, memory, and latency penalties across different workloads. Their insights help

architects optimize configurations to maintain SLO compliance [15]. Noor et al. (2025) benchmarked Kubernetes application performance across heterogeneous CPU architectures, such as x86 and ARM, exposing cost–performance trade-offs. The work informs hardware selection for optimized deployments [16]. Balis et al. (2022) address the challenge of running scientific workflows on Kubernetes by proposing a predictive auto-scaling policy that anticipates resource demands based on workflow structure. Their evaluation with HyperFlow and Montage workflows on Google Cloud shows that predictive scaling outperforms reactive approaches in elasticity, execution time, and cost efficiency [17]. Turin et al. (2023) built predictive models from telemetry data to forecast container resource consumption, enabling better request and limit configurations. This proactive approach minimizes throttling and performance regressions [18].

2.3 Empirical Studies & Benchmarking Techniques

Wyciślik et al. (2023) conducted a systematic performance comparison of JVM frameworks—including Spring Boot, Quarkus, and Micronaut—for building microservices. The study analyses response time quantiles, CPU and memory utilization, and container size, highlighting trade-offs in resource efficiency and performance footprints [19]. Henning & Hasselbring (2024) carried out extensive scalability benchmarks of modern stream processing frameworks—such as Flink, Kafka Streams, Hazelcast Jet, and Apache Beam—as deployed within microservice architectures on Kubernetes and private clouds. Their findings reveal nearly linear scalability across environments but note differing resource efficiency profiles, making trade-offs evident for practitioners [20]. Smith et al. (2023) introduced an end-to-end testing benchmark for microservices, leveraging tracing-based service-dependency discovery to enable full functional test coverage across complex distributed systems. The work helps simplify experimentation and highlight bottlenecks in user-flow testing of microservices [21]. Henning et al. (2024) presented ShuffleBench, a configurable benchmark suite for distributed stream processing frameworks (Flink, Hazelcast, Kafka Streams, Spark), emphasizing latency, throughput, and scalability in Kubernetes-based cloud environments. The tool includes open-source implementations and empirical evaluations, improving reproducibility in performance research [22]. Vogel et al. (2024) offer a comprehensive benchmarking study on fault recovery performance in stream processing frameworks (Flink, Kafka Streams, Spark) using chaos engineering. Their results highlight differences in recovery stability and latency post-failure, providing empirical guidance for resilient microservice deployments [23]. Mochniej and Badurowicz (2023) compared reactive vs. imperative microservices using Spring WebFlux vs. Spring MVC, finding that reactive services offer lower RAM usage and better latency under communication delays, particularly in data-intensive operations. This empirical comparison illuminates how programming paradigms impact operational performance [24]. Plecinski et al. (2022) evaluate the performance of different microservices technologies for sensor network applications, highlighting trade-offs in latency, throughput, and resource usage to guide technology selection in IoT-oriented projects [25]. Laigner et al. (2025) introduced Online Marketplace, a benchmark that addresses core data management challenges in microservices, including transaction and query processing, event handling, constraint enforcement, and replication, thereby exposing performance overheads often overlooked by existing benchmark suites [26]. In a recent study, Sun et al. (2024) proposed MicroServo, a live, scenario-oriented benchmarking framework for evaluating AIOps algorithms in microservice systems by generating real-time datasets and simulation environments tailored to varying operational scenarios [27].

2.4 DevOps and Decision Support Strategies

The study by Port et al. (2024) investigates the effectiveness of DevOps-based policies in NASA's Mission Design and Navigation Software group, using 15 years of data from the MONTE system. Their analysis shows that while variability in productivity and quality remains a continual risk, the instituted DevOps practices are largely complied with and effective in mitigating maintenance risks [28]. Azad and Hyrynsalmi (2023) conducted a systematic literature review to identify critical success factors (CSFs) for DevOps adoption, synthesizing organizational, cultural, process, tooling, and measurement aspects that shape successful DevOps transformations; their taxonomy is practical for building decision checklists and maturity assessments [29]. Kumar et al. (2024) proposed a multicriteria decision-making framework (fuzzy best–worst method) for prioritizing and selecting DevOps practices in organizations, enabling practitioners to weigh trade-offs (e.g., automation vs. cultural change) and make structured deployment/process decisions [30]. Eramo et al. (2024) developed an architecture for model-based

© Little Lion Scientific

and intelligent automation that integrates runtime telemetry with model-driven decision logic, demonstrating how predictive analytics and rule-based planning can drive automated reconfiguration and deployment choices in cloud/DevOps settings [31]. Díaz-de-Arcaya et al. (2023) introduced Orfeon, an AIOps framework that formulates goal-driven optimization for operational pipelines—combining multiple objectives (latency, cost, reliability) to recommend deployment and routing decisions—showing the practicality of AIOps for automated decision support [32]. Zhang et al. (2025) surveyed the emerging role of large language models (LLMs) in AIOps, analysing 183 studies to understand how LLMs optimize IT operations tasks, handle diverse failure data, and support novel AIOps methods. The study highlights trends, evaluation approaches, and gaps in LLM-based AIOps, providing directions for future research [33]. Rzig et al. (2024) conducted the empirical study on CI/CD configuration evolution in machine learning projects, analyzing 343 commits from 508 open-source ML repositories. They identify common change categories, co-evolution patterns between CI/CD and ML components, and developer expertise impacts, revealing frequent build policy changes, some suboptimal practices, and insights into how ML-specific workflows influence CI/CD evolution [34].

While prior studies provide valuable insights into microservice performance and deployment environments, they largely lack a unified, workload-aware framework for guiding deployment decisions. In particular, the interaction between distinct workload patterns and deployment paradigms remains insufficiently explored, and observability data is rarely leveraged for structured decision-making. This work addresses these limitations through a workload-aware empirical framework and a normalized Performance Index.

3. METHODOLOGY

An empirical methodology is adopted to examine the relationship between deployment environments and microservice performance across distinct workload conditions. The approach integrates controlled experimentation with multi-metric analysis to facilitate consistent and comparable evaluation of deployment behaviour.

3.1 Experimental Environment

Experiments were conducted on a controlled testbed to ensure consistency and reproducibility. The system was deployed using Java-based microservices across three representative runtime environments: a monolithic application server (Payara Server), a lightweight microservice runtime (Payara Micro Server), and a containerized orchestration platform (Kubernetes). The underlying infrastructure comprised a multi-core processing environment with constrained memory allocation to simulate realistic deployment conditions. Supporting components included a relational database (Mysql) for persistence and an observability framework (Prometheus and Grafana) for metrics collection. The modular decomposition of the microservice system further enables realistic performance evaluation by supporting heterogeneous workload patterns and facilitating scalable, isolated, and flexible execution across diverse deployment environments [35,36].

3.2 Workload Modelling and Execution

Two representative workload patterns were evaluated:

- **Read-intensive single-service interactions**, representing lightweight, latency-sensitive requests
- **Multi-service transactional workflows**, involving coordinated inter-service communication and state updates

Workloads were executed under controlled conditions with incrementally increasing request volumes to evaluate system behaviour across varying load levels. A consistent ramp-up strategy was applied to simulate realistic request arrival patterns and avoid abrupt system saturation.

3.3 Performance Measurement

System behaviour was assessed using a comprehensive set of metrics:

- **Resource utilization:** CPU, memory, and thread concurrency
- **Performance:** throughput, average latency, and tail latency
- **Reliability:** request success rate and failure count

Metrics were collected at fine temporal granularity using an observability framework, enabling accurate performance profiling under different deployment conditions. Real-time visualization further supported continuous monitoring of performance trends and rapid identification of system anomalies during load execution [37].

Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3.4 Data Processing and Normalization

Collected metrics were pre-processed to ensure consistency and comparability. Missing values were addressed using linear interpolation to preserve continuity in time-series data [38]. All metrics were normalized to a common [0,1] scale based on their relationship with system performance. Negatively correlated metrics (e.g., latency and resource usage) were inversely scaled, while positively correlated metrics (e.g., throughput) were directly normalized. This ensured unbiased aggregation across heterogeneous metrics.

3.5. Derivation of Performance Index

To enable holistic comparison, a Performance Index (PI) is computed by aggregating standardized metrics across categories: efficiency (throughput, latency), scalability (success rate under load), and resource utilization (CPU/memory/thread) [39,40]. This composite measure acts as a unifying benchmark for evaluating system performance across workloads and deployment environments. The derived value ranges between 0 and 1, with higher values indicating superior performance. The derivation process proceeds in five systematic steps:

Step 1: Metric Normalization

For each performance parameter p, its normalized value $N_{p,l}$ at load level l is computed as:

$$N_{p,l} = \frac{X_{p,l} - X_p^{min}}{X_p^{max} - X_p^{min}} \qquad (1)$$

Where, $X_{p,l}$ denotes the observed value, and $X_p^{\min}, X_p^{\max}$ represent the minimum and maximum values across all experimental conditions.

Step 2: Parameter Weighting

Each performance parameter is assigned a relative weight subject to the constraint:

$$\sum_{i=1}^{n} W_i = 1 \qquad (2)$$

Where, W_i denotes the weight of the ith parameter.

The distribution reflects the differentiated contribution of each parameter to both Quality of Service (QoS) as perceived by end-users and the sustainability of system resources.

Overall, as shown in table 1, 70% of the weight is allocated to user-facing performance parameters (latency and throughput), while the remaining 30% captures resource efficiency (CPU, memory, and concurrency). This ensures that the performance index emphasizes real-world user experience without neglecting backend sustainability.

Table 1: Assigned Weights and Rationale for Performance Parameters

Performance Parameter	Weight (W_i)	Rationale
$W_{p99latency}$	0.25	Critical for tail performance.
$W_{avglatency}$	0.20	Reflects overall request-handling smoothness; critical for consistent quality of service.
$W_{throughput}$	0.25	Reflects scalability and service capacity; equally important as latency.
W_{cpu}	0.10	Ensures computational efficiency; secondary to user-facing QoS.
W_{memory}	0.10	Indicates stability and capacity management; less visible to users.
$W_{threadcount}$	0.10	Captures concurrency handling; modest weight to prevent overshadowing latency and throughput.

Step 3: Load Weighting

Since performance must be evaluated across a spectrum of request intensities, incremental loads (l) ranging from the minimum (1000 requests/min) up to the maximum sustainable threshold load l_{max} are considered. To ensure that higher loads — which better expose system bottlenecks — are weighted more heavily, a two-step load weighting scheme is adopted.

i. Raw Load Weight Calculation

For each load level L_i, a raw weight is first computed as a normalized distance from the lowest load:

$$RW_l = \frac{l - l_{min}}{l_{max} - l_{min}} \qquad (3)$$

Where:

- l_i = current load,

Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

- l_{min} = minimum load (1000 RPS),
- l_{max}=threshold load for the given operation/environment.

This ensures that loads closer to the threshold receive proportionally higher significance.

ii. Normalization of Raw Weights

Since raw weights do not sum to 1, they are normalized across all n loads to obtain the final load weights:

$$\boldsymbol{LW_i} = \frac{\boldsymbol{RW_i}}{\sum_{j=1}^{n} \boldsymbol{RW_j}} \qquad (4)$$

Where, LW_i is the normalized load weight assigned to load L_i. The weighting scheme prioritizes higher load levels, as they represent system scalability and resilience under stress. Lower loads are included for completeness but contribute less to the aggregate index. Normalization ensures that weights sum to unity across all tested loads, thereby balancing baseline stability with peak-load behaviour. This approach avoids bias toward trivial low-load performance while capturing the critical degradation points that distinguish deployment environments.

Step 4: Aggregated Parameter Score per Load

For each applied load level l, the contributions of all performance parameters are aggregated into a single score. This ensures that the multi-dimensional performance profile (latency, throughput, CPU, memory, and thread utilization) is represented as a unified measure. The aggregated parameter scores for load l, denoted as S(l), is computed as:

$$\boldsymbol{S(l)} = \sum_{k=1}^{n} (\boldsymbol{N_k(l)} \cdot \boldsymbol{w_k}) \qquad (5)$$

Where:

- $N_k(l)$ = normalized value of performance parameter k at load l,
- w_k = assigned weight of performance parameter k,
- n = total number of performance parameters considered.

This formulation captures the weighted influence of each parameter, ensuring that higher-priority metrics such as latency and throughput exert stronger influence on the aggregated score compared to auxiliary system-level parameters (CPU, memory, threads).

Step 5: Performance Index (PI) Computation

To incorporate workload intensity into the evaluation, the aggregated performance score $S(l)$at each load level $l$is weighted using the corresponding load weight W_l, ensuring that higher-intensity workloads contribute proportionally more to the overall evaluation. The Performance Index (PI) is then computed as:

$$\boldsymbol{PI} = \sum_{l=1}^{m} \boldsymbol{S(l)} \cdot \boldsymbol{W_l} \qquad (6)$$

Where:

- $S(l)$= aggregated performance score at load level l
- W_l= normalized weight assigned to load level l, with $\sum_{l=1}^{m} W_l = 1$
- m= total number of load levels

The resulting PI lies within the range [0, 1], providing a unified measure of system performance that captures trade-offs between efficiency, scalability, and resource utilization under varying workload intensities. Higher PI values indicate superior deployment performance. This derived PI is subsequently employed to conduct a comparative evaluation across operations and deployment environments.

3.6 Evaluation Procedure

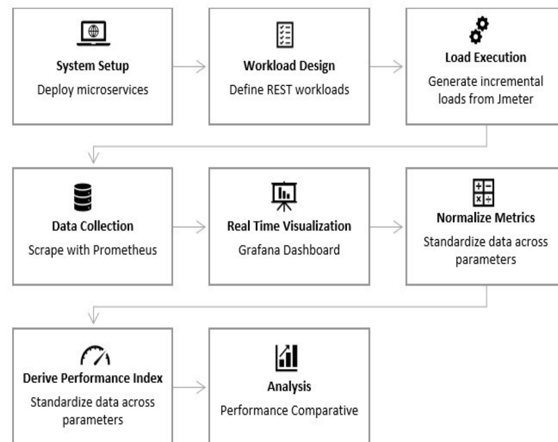

Figure 1: Research Methodology

As shown in figure 1, the evaluation followed a consistent and repeatable process:

© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

1. Deployment of the system in the selected environment
2. Execution of workload scenarios under controlled load conditions
3. Collection and preprocessing of performance metrics
4. Computation of performance scores
5. Comparative analysis across deployment paradigms

Each experiment was repeated multiple times, and results were averaged to minimize variability and improve statistical reliability.

4. RESULTS

The experimental evaluation provides a comparative analysis of deployment behaviour across two representative workload patterns: read-intensive single-service interactions and multi-service transactional workflows. Performance is examined across four key dimensions—resource utilization, latency characteristics, throughput scalability, and threshold behaviour—to identify deployment-specific trade-offs. The analysis focuses on how deployment paradigms respond to varying workload intensities, highlighting efficiency, scalability limits, and performance degradation patterns. These observations are further interpreted to derive workload-aware deployment insights.

4.1 Read-Intensive Single-Service Interactions

A single-service read request serves as a controlled baseline for assessing environment responsiveness under minimal transactional overhead. Evaluating this operation first isolates the core efficiency of each deployment model in handling lightweight query workloads, thereby establishing a reference point against which the more complex, multi-service transaction can later be contrasted.

4.1.1. Parameter-level analysis under read-intensive workload

The parameter-wise comparison of this operation provides a multidimensional view of system efficiency under different deployment models. The results, captured across CPU usage, thread utilization, memory consumption, throughput, and latency, highlight the architectural trade-offs in handling single-service read operations.

a) Resource Utilization (CPU, Threads, and Memory)

Monolithic deployment demonstrates the most efficient resource profile, maintaining consistently low CPU and memory utilization across increasing loads, with stable thread behaviour until saturation (Figure 2–4). This behaviour indicates that, for lightweight workloads, performance is primarily governed by runtime overhead, where tightly coupled execution minimizes coordination costs and enables efficient resource utilization [41].

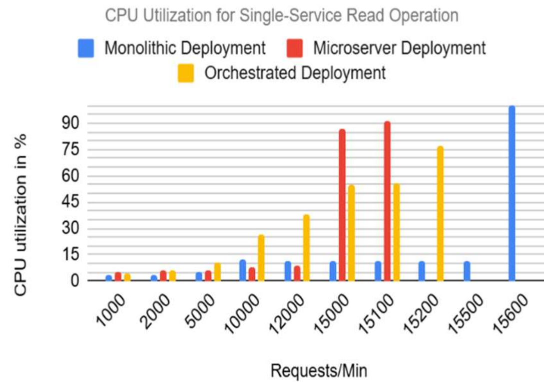

Figure 2: CPU Utilization for Single-Service Read-Intensive Workload

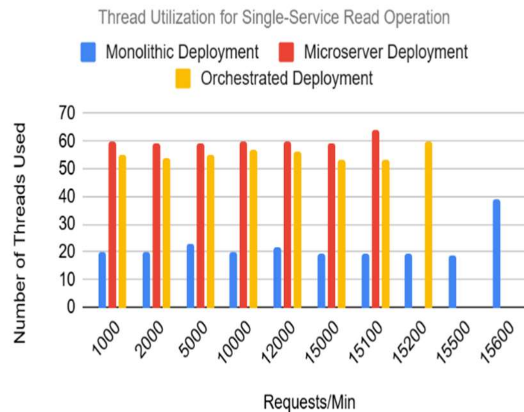

Figure 3: Thread Utilization for Single-Service Read-Intensive Workload

Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

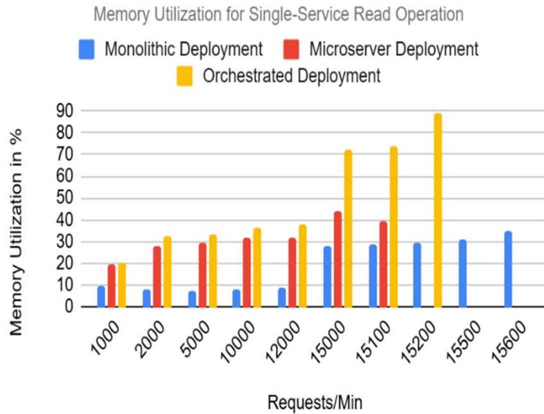

Figure 4: Memory Utilization for Single-Service Read-Intensive Workload

In contrast, the microserver runtime exhibits higher baseline resource consumption, with CPU utilization increasing more rapidly under load, suggesting that reduced structural overhead is offset by increased computational demand associated with service isolation (Figure 2–4). The orchestrated deployment shows the highest resource consumption, with steep increases in both CPU and memory usage, highlighting the impact of containerization and orchestration layers, which introduce additional scheduling, communication, and state management overhead.

These observations indicate that for read-intensive workloads, resource efficiency is more strongly influenced by execution overhead than by scalability mechanisms, making monolithic and micro server deployments more suitable than fully orchestrated environments.

b) Throughput and Latency Characteristics

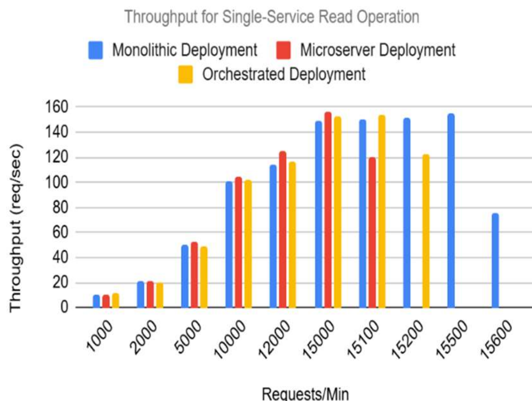

Figure 5: Throughput for Single-Service Read-Intensive Workload

Throughput trends indicate that all three deployment paradigms achieve comparable peak request-handling capacity, with Monolithic and Microserver deployments scaling nearly linearly up to ~150–156 requests/sec near their respective thresholds (Figure 5). This suggests that for read-intensive workloads, raw throughput is not a distinguishing factor, as the orchestrated deployment also attains similar peak throughput. However, this is achieved at significantly higher resource cost, indicating that orchestration overhead does not translate into efficiency gains for lightweight operations.

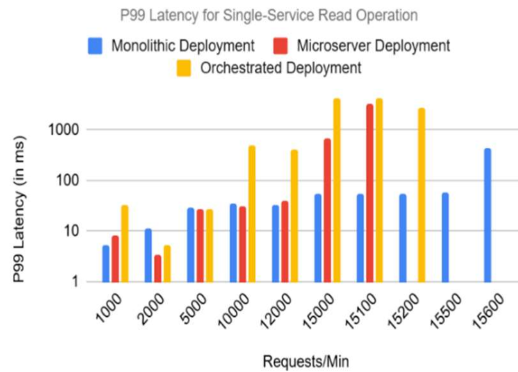

Figure 6: P99 Latency for Single-Service Read-Intensive Workload

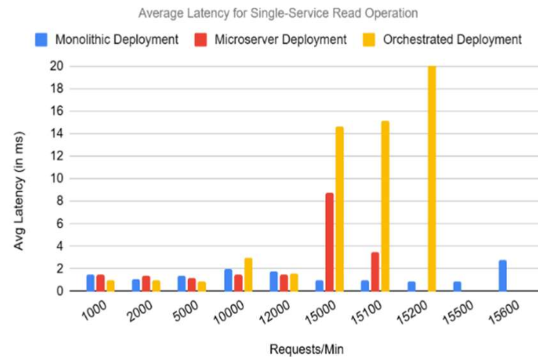

Figure 7: Average Latency for Single-Service Read-Intensive Workload

Latency behaviour, in contrast, reveals clear divergence across deployment environments (Figure 6, 7). Monolithic deployment maintains the most stable latency profile, with average latency consistently below 2 ms and P99 latency remaining under 60 ms until saturation. This reflects efficient

thread scheduling and minimal coordination overhead within a tightly coupled execution model.

The microserver runtime maintains competitive average latency under moderate load but exhibits sharp degradation in tail latency at higher loads, with P99 latency increasing significantly (e.g., 672 ms at 15,000 requests/minute). This indicates the onset of queuing delays under increasing CPU pressure, despite sustained throughput performance. The orchestrated deployment demonstrates the most pronounced latency degradation. While throughput remains comparable, P99 latency increases substantially at earlier load levels, exceeding 400 ms around 10,000 requests/minute, while average latency rises sharply near saturation. This behaviour is attributable to orchestration overheads, including inter-container communication and scheduling delays, which disproportionately impact tail latency even for lightweight workloads [42].

These results highlight that for read-intensive operations, throughput alone is insufficient to characterize system performance, and latency—particularly tail latency—emerges as the critical differentiator across deployment paradigms. These latency-driven differences are reflected in the Performance Index (PI), where monolithic deployment achieves the highest efficiency score, followed by the microserver runtime, while the orchestrated deployment is penalized due to early latency degradation.

c) Threshold Behaviour

Threshold behaviour further differentiates the practical limits of each deployment paradigm. Monolithic deployment sustains the highest load (~15,600 requests/minute) with gradual and predictable degradation, indicating robust handling of lightweight workloads until CPU saturation. The microserver runtime reaches its threshold slightly earlier (~15,100 requests/minute), not due to throughput limitations but as a result of CPU exhaustion and increasing tail latency, reflecting reduced stability under sustained load. Although the orchestrated deployment sustains a comparable load (~15,200 requests/minute), its latency degradation occurs significantly earlier, with response times escalating to impractical levels at higher loads. This indicates that while throughput capacity is maintained, usability is compromised due to severe latency inflation.

Overall, these observations highlight that threshold capacity alone does not determine deployment effectiveness; rather, the point at which latency becomes unacceptable defines the practical operational limit of each environment.

4.1.2. Performance Index (PI) Analysis for Read-Intensive Workload

The Performance Index (PI) results (Figure 8) provide a consolidated view of deployment efficiency across varying load levels for the read-intensive single-service workload. As this operation involves minimal coordination overhead, the PI primarily reflects how effectively each deployment environment utilizes resources to sustain lightweight requests.

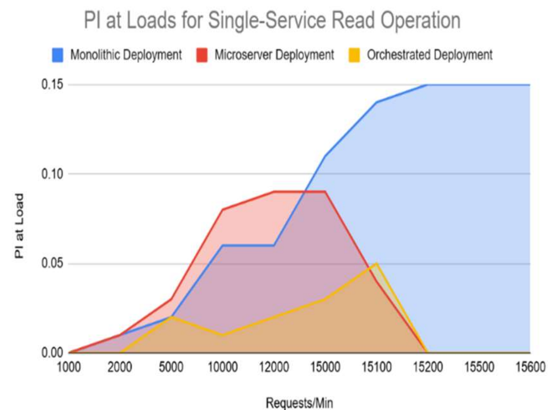

Figure 8: Performance Index across Loads for Single-Service Read Operation

At lower load levels, both Monolithic and Microserver deployments exhibit modest gains in efficiency, while the Orchestrated deployment remains near zero. This indicates that orchestration overhead introduces unnecessary cost in simple read scenarios, limiting efficiency under low-load conditions.

As the workload increases to mid-range levels (≈ 5,000–12,000 req/min), both Monolithic and Microserver deployments show progressive improvement in PI. The Microserver runtime slightly outperforms the monolithic environment at this stage, suggesting that its lighter execution model is better suited for sustaining moderate levels of concurrent read requests. In contrast, the Orchestrated deployment shows only marginal improvement, reinforcing that container orchestration provides limited benefit for workloads with minimal coordination requirements.

Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

At higher load levels (≥ 15,000 req/min), the deployment behaviours diverge significantly. Monolithic deployment continues to improve, reaching the highest PI at its maximum sustainable load (~15,600 req/min), indicating strong resilience and sustained efficiency. The Microserver runtime, however, exhibits a sharp decline beyond its threshold (~15,100 req/min), with PI dropping abruptly due to resource saturation and latency degradation. The Orchestrated deployment remains the weakest performer, with limited PI growth and eventual collapse near its threshold (~15,200 req/min).

Overall, these results demonstrate that Monolithic deployment provides the most stable and efficient performance for read-intensive workloads, particularly at higher load levels. The Microserver runtime remains competitive under moderate load but shows reduced resilience under sustained stress. In contrast, the Orchestrated deployment consistently underperforms in this scenario, as the overhead of containerization and orchestration does not yield proportional benefits for lightweight, read-dominated operations.

4.2 Multi-Service Transactional Workload

This workload introduces a fundamentally different execution profile compared to the single-service operation. Instead of a lightweight query, this transaction spans multiple microservices, combines read and write operations, and enforces inter-service coordination with database consistency requirements. As a result, system behaviour is no longer dictated solely by local execution efficiency but also by cross-service communication, synchronization delays, and orchestration overheads [43, 44]. These additional factors manifest in more volatile resource consumption patterns, sharper latency spikes, and earlier saturation points across the deployment environments.

4.2.1. Parameter-level analysis under multi-service transactional workload

a) Resource Utilization (CPU, Threads, and Memory)

Resource utilization patterns under transaction-intensive workloads reveal clear divergence in how deployment paradigms handle coordination overhead (Figure 9–11).

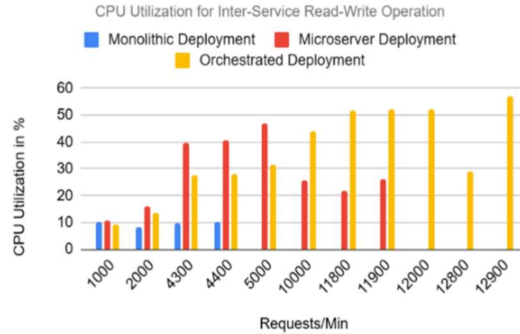

Figure9: CPU Utilization for Multi-Service Transactional Workload

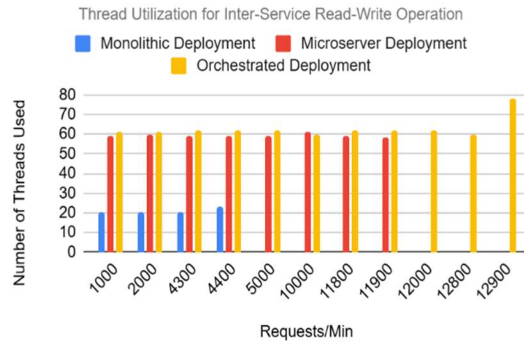

Figure 10: Thread Utilization for Multi-Service Transactional Workload

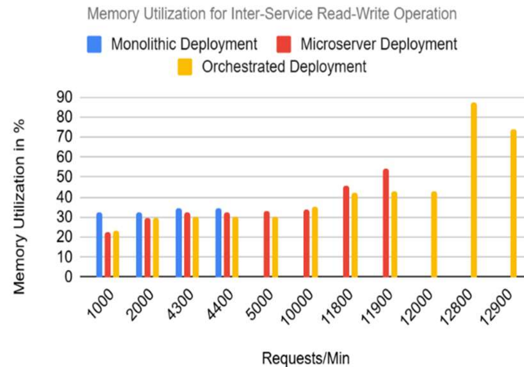

Figure 11: Memory Utilization for Multi-Service Transactional Workload

CPU utilization highlights the contrasting scalability characteristics of the environments (Figure 9). Monolithic deployment maintains low CPU usage (~8–10%) up to moderate load levels, reflecting efficient execution in a tightly coupled runtime. However, this stability is followed by abrupt collapse near its threshold (~4,400 requests/min), indicating limited elasticity when subjected to coordination-heavy workloads. In contrast, the microserver runtime exhibits higher

Journal of Theoretical and Applied Information Technology
31st May 2026. Vol.104. No.10
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

initial CPU consumption, increasing rapidly with load before stabilizing, suggesting that distributed execution improves load distribution but incurs additional computational overhead due to inter-service coordination. The orchestrated deployment shows the highest and most consistent CPU growth, reaching elevated utilization levels as load increases, reflecting the compounded overhead of container management and orchestration under multi-service interaction.

Thread utilization further reinforces these observations (Figure 10). Monolithic deployment maintains a low and stable thread count until near saturation, after which a sharp increase precedes system failure, indicating limited concurrency scaling. The microserver runtime and orchestrated deployment maintain consistently higher thread counts across all load levels, reflecting controlled concurrency models designed for distributed transaction handling. However, this stability comes at the cost of increased CPU and memory consumption.

Memory utilization trends provide additional insight into system behaviour (Figure 11). Monolithic deployment exhibits efficient memory usage with minimal growth prior to collapse, indicating disciplined resource management but limited adaptability under sustained transactional load. The microserver runtime shows gradual memory growth, suggesting increased per-request state handling that remains manageable across its operational range. In contrast, the orchestrated deployment demonstrates the steepest memory escalation, with significant increases under high load, highlighting the overhead introduced by container isolation, state propagation, and orchestration mechanisms.

Overall, these results indicate that for multi-service transactional workloads, resource utilization is dominated by coordination overhead rather than raw execution efficiency. While monolithic deployment offers high efficiency at low load, it lacks scalability under coordinated workloads. The microserver runtime provides a balanced trade-off between efficiency and scalability, whereas the orchestrated deployment incurs substantial resource overhead in exchange for distributed execution capabilities.

b) Throughput and Latency Characteristics

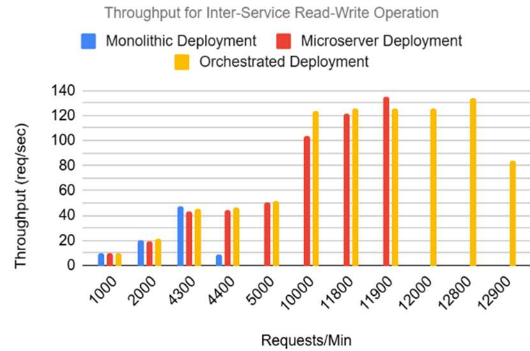

Figure 12: Throughput for Multi-Service Transactional Workload

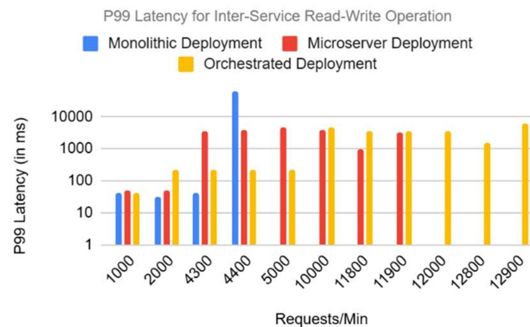

Figure 13: P99 Latency for Multi-Service Transactional Workload

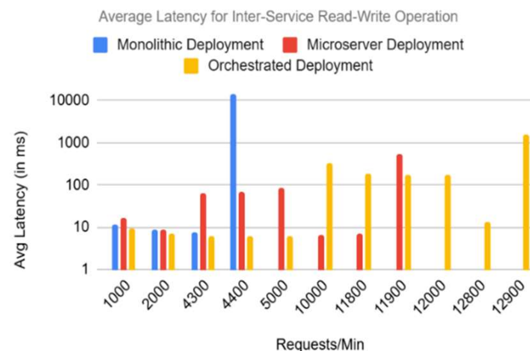

Figure 14: Average Latency for Multi-Service Transactional Workload

Throughput trends reveal distinct scalability characteristics across deployment paradigms (Figure 12). Monolithic deployment scales linearly up to ~47 requests/sec but collapses abruptly beyond its threshold (~4,400 requests/min), indicating limited capacity to sustain inter-service coordination. In contrast, the microserver runtime achieves steady growth, peaking at ~135 requests/sec, while the orchestrated deployment

sustains comparable throughput over a broader load range. This demonstrates that distributed and orchestrated environments extend scalability for transaction-heavy workloads, albeit with higher overhead.

Latency behaviour provides a clearer distinction (Figure 13–14). Monolithic deployment maintains low and stable latency at lower loads (Avg ~8–12 ms, P99 ~30–40 ms) but exhibits catastrophic degradation beyond its threshold, reflecting an all-or-nothing performance profile under coordination pressure. The microserver runtime shows improved endurance but reduced stability, with sharp increases and high variability in tail latency as load increases, indicating growing contention and queuing delays despite sustained throughput. The orchestrated deployment exhibits the most gradual latency degradation. Although baseline latency is higher, both average and tail latency increase more smoothly with load, indicating greater resilience under sustained multi-service interaction. This reflects the ability of orchestration mechanisms to distribute load and delay saturation, though at the cost of increased overhead.

Overall, these results demonstrate a shift from execution efficiency to coordination-driven scalability in multi-service workloads. Monolithic deployment is efficient but fragile, the microserver runtime provides moderate scalability with instability, while orchestrated deployment offers the most resilient scaling behaviour despite higher resource cost.

a) Threshold Behaviour

Threshold analysis clearly distinguishes the operational limits of each deployment paradigm. Monolithic deployment reaches its limit abruptly at ~4,400 requests/min, where throughput collapses and latency becomes untenable, making it unsuitable for sustained inter-service workloads. The microserver runtime extends scalability up to ~11,900 requests/min, maintaining throughput growth but experiencing increasing CPU pressure and tail-latency degradation, which ultimately constrains its practical usability. The orchestrated deployment sustains the highest load (~12,900 requests/min), preserving throughput under increasing concurrency. However, this comes at the cost of significantly elevated latency (average >1.5 seconds, P99 ~6 seconds), indicating a trade-off between resilience and user experience.

Overall, these results demonstrate a fundamental shift in deployment suitability for multi-service workloads. While monolithic deployment remains highly efficient at low load, it lacks scalability under coordination-intensive operations. The microserver runtime provides extended scalability with moderate stability limitations, whereas the orchestrated deployment offers the highest resilience, albeit with substantial resource and latency overhead.

4.2.2 Performance Index (PI) analysis for multi-service transactional workload

The Performance Index (PI) trends (Figure 15) provide a consolidated view of how each deployment environment sustains efficiency under multi-service read–write workloads. Unlike single-service operations, this workload introduces coordination and synchronization overhead, making PI a stricter indicator of sustainable performance.

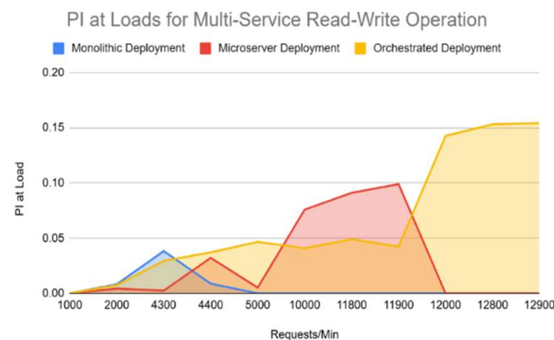

Figure 15: Performance Index across Loads for Multi-Service Transactional Workload

At lower load levels, Monolithic and Orchestrated deployments exhibit modest efficiency gains (PI ≈ 0.01), while the microserver runtime remains relatively flat, indicating early overhead from distributed coordination. As load increases to the lower–mid range (≈ 4,000–5,000 requests/min), Monolithic deployment shows a brief improvement but collapses soon after, with PI dropping to zero, confirming its inability to sustain inter-service workloads beyond moderate concurrency. The microserver runtime demonstrates limited recovery in this range but exhibits instability due to rising thread pressure and latency variability, eventually collapsing near its threshold (~12,000 requests/min). In contrast, the orchestrated deployment shows steady and consistent PI growth, indicating its ability to balance throughput and resource utilization under increasing coordination demand.

At higher load levels (≈ 10,000–12,000 requests/min), the divergence becomes pronounced. Monolithic deployment is no longer viable, while the microserver runtime briefly peaks (PI ≈ 0.08–0.10) before failing. The orchestrated deployment continues to improve, maintaining efficiency despite increased resource overhead. Beyond 12,000 requests/min, orchestrated deployment emerges as the only model sustaining performance, with PI increasing steadily up to ~0.15 at 12,900 requests/min. Both Monolithic and Microserver deployments record PI = 0, indicating exhaustion of their scalability limits.

Overall, these results demonstrate a clear shift in deployment suitability for coordination-intensive workloads. Monolithic deployment is efficient but non-scalable, the microserver runtime provides moderate scalability with limited stability, while orchestrated deployment delivers the highest resilience and sustained efficiency, making it the most suitable for multi-service transactional operations despite higher overhead.

4.3 Aggregate Performance Index Analysis Across Workload Types

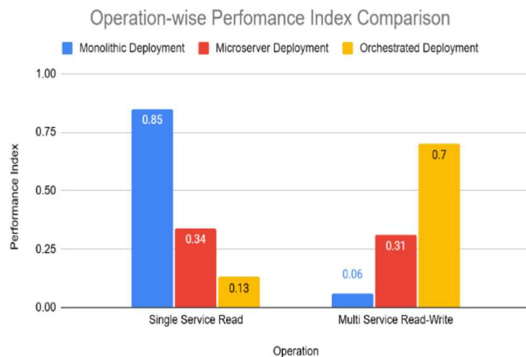

Figure 16: Performance Index Score across Deployment Environments

The aggregate Performance Index values (Figure 16) reveal a clear workload-dependent divergence in deployment suitability. For the single-service read operation, Monolithic deployment achieves the highest PI (0.85), significantly outperforming Microserver (0.34) and Orchestrated (0.13) deployments. This reflects the efficiency of centralized execution for read-dominated workloads, where minimal coordination overhead, efficient thread management, and in-memory data access enable near-optimal performance. In contrast, Microserver deployment incurs additional inter-process and coordination overhead, reducing efficiency, while Orchestrated deployment further amplifies this cost through containerization, networking, and scheduling layers, making it less suitable for lightweight operations.

The behaviour reverses under the multi-service read–write workload. Monolithic deployment drops sharply (PI ≈ 0.06), highlighting its inability to sustain coordination-intensive operations. The tightly coupled runtime becomes a bottleneck when inter-service communication and transactional consistency dominate execution. The microserver runtime achieves moderate performance (PI ≈ 0.31), offering improved flexibility for distributed interactions but lacking sufficient elasticity under sustained load. Orchestrated deployment, however, achieves the highest PI (≈ 0.70), demonstrating its strength in handling coordination-heavy workloads. Its ability to distribute load, scale services, and isolate execution enables sustained efficiency despite higher overhead.

Overall, these results confirm that deployment efficiency in microservice systems is fundamentally workload-dependent. Monolithic deployment is optimal for compute-localized, read-intensive operations, whereas orchestrated deployment is better suited for distributed, transaction-heavy workloads, where scalability and coordination outweigh raw execution efficiency

5. DISCUSSION OF FINDINGS

The experimental evaluation systematically examined REST-based microservice workloads across Monolithic, Microserver, and Orchestrated Deployments, revealing how environment–workload interactions dictate system efficiency. The findings are synthesized below with respect to the stated research questions.

RQ1: How do different deployment paradigms impact the performance characteristics of read-intensive single-service interactions and multi-service transactional workflows?

The results reveal a clear workload-dependent variation in deployment behaviour. For read-intensive single-service workloads, Monolithic Deployment consistently achieves the highest efficiency, with a PI of 0.85, stable latency, and sustained throughput under increasing load. Its tightly integrated runtime minimizes communication and orchestration overhead, allowing efficient processing up to 15,600 requests/min. In contrast,

Orchestrated Deployment performs poorly for lightweight workloads (PI = 0.13) because container management, service discovery, and inter-service communication introduce additional overhead and increased tail latency.

However, the behaviour reverses under multi-service transactional workloads. Monolithic Deployment rapidly degrades (PI = 0.06) due to thread exhaustion, centralized resource contention, and coordination bottlenecks. Conversely, Orchestrated Deployment demonstrates the highest resilience (PI = 0.70), benefiting from distributed scheduling, workload isolation, and horizontal scalability. Microserver Deployment exhibits intermediate behaviour, supporting moderate loads but saturating earlier than Orchestrated Deployment. These findings confirm that deployment efficiency is inherently workload-contingent rather than universally environment-specific.

RQ2: Which operational metrics yield the most actionable insights for environment tuning and selection?

Latency and throughput emerge as the most significant indicators of deployment viability. Average and P99 latency effectively capture both responsiveness and runtime stability under stress conditions. Monolithic Deployment maintains low latency during read-intensive workloads but exhibits sharp latency escalation under transactional workloads, indicating resource saturation.

Throughput trends further highlight scalability differences among deployment paradigms. Orchestrated Deployment maintains comparatively stable throughput growth during distributed operations, while Monolithic Deployment experiences rapid degradation under increasing coordination demands.

Thread utilization and CPU consumption provide additional diagnostic insight into concurrency behaviour and resource contention. Memory utilization remains relatively stable but helps identify saturation points preceding throughput collapse. Collectively, these metrics provide a multidimensional basis for workload-aware deployment tuning and environment selection.

RQ3: Can a normalized, multi-metric analysis be designed to derive a Performance Index (PI) for consistent and fair ranking of deployment environments?

The proposed Performance Index (PI) framework successfully integrates heterogeneous operational metrics, including CPU usage, memory utilization, latency, throughput, and thread behaviour, into a unified deployment evaluation model. The results demonstrate that PI effectively captures both operational efficiency and resilience under varying workload conditions.

The framework identifies not only high-performing environments but also critical threshold points where deployments become unstable or collapse. For example, Monolithic Deployment achieves high PI values for read-intensive workloads, whereas Orchestrated Deployment demonstrates superior PI values for distributed transactional workloads. This confirms that PI provides a normalized and workload-aware mechanism for fair comparison and deployment ranking across heterogeneous environments.

RQ4: How can these empirical insights be integrated into Intelligent DevOps workflows to enable continuous, data-driven performance optimization?

The findings support the development of workload-aware deployment strategies within Intelligent DevOps ecosystems. Read-intensive services can be efficiently deployed in monolithic environments to minimize overhead, while transaction-heavy services benefit from orchestrated deployments that provide scalability and fault isolation.

By continuously monitoring operational metrics and evaluating runtime PI values, DevOps systems can detect performance degradation and trigger adaptive actions such as autoscaling, workload redistribution, or redeployment. This enables a transition from reactive infrastructure management toward continuous and data-driven deployment optimization.

6. THREATS TO VALIDITY

Although the study was conducted with a structured methodology, several threats to validity must be recognized to maintain transparency and guide interpretation.

Internal Validity: Automation of workload generation, deployment, and metrics collection minimized bias, but uncontrollable factors such as host-level resource contention, network jitter, or container orchestration delays may have influenced

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

transient spikes in CPU, memory, or latency. Multiple runs and averaged values were used to reduce this risk, though hidden anomalies cannot be fully excluded. Additionally, missing metrics at certain load levels were addressed through linear interpolation to preserve dataset continuity. While this allowed consistent PI computation across environments, interpolation may have smoothed sharp anomalies, slightly underestimating volatility at system thresholds.

External Validity: The experiments evaluated two REST-based operations—one single-service read and one multi-service read–write transaction—under controlled workloads. While these operations were chosen to represent fundamental patterns, production microservices often involve mixed communication protocols, stateful transactions, or more dynamic scaling scenarios. Hence, results are most applicable to systems with similar characteristics, and generalization should be approached with caution.

Construct Validity: A composite Performance Index (PI) was employed to rank deployment environments by aggregating CPU, memory, threads, throughput, and latency with weighted importance. Throughput and latency were given higher weights, aligning with common service-level objectives for responsiveness and scalability. However, in contexts where resource efficiency or cost minimization is more critical, this weighting may underrepresent relevant trade-offs. Thus, while PI provides a consistent comparative measure, its interpretation remains sensitive to the assumed priorities embedded in the weighting scheme.

Overall, while these validity threats highlight the boundaries of applicability, they do not diminish the study's central contribution—providing empirical, data-driven insights into how deployment environments behave under different operation types and load intensities. Rather, they serve to clarify the conditions under which the findings are most relevant and how future extensions may broaden their scope.

7. CONCLUSION

This study presented a workload-aware empirical evaluation of microservice deployment across monolithic, microserver, and orchestrated environments, supported by a normalized Performance Index (PI) for unified, multi-metric assessment of efficiency, scalability, and resilience.

The results demonstrate a clear workload-dependent divergence in deployment suitability. For read-intensive workloads, monolithic deployment achieves the highest efficiency (PI = 0.85), significantly outperforming microserver (PI = 0.34) and orchestrated deployments (PI = 0.13). In contrast, for multi-service transactional workloads, orchestrated deployment achieves superior performance (PI = 0.70), while monolithic deployment exhibits minimal efficiency (PI = 0.06), highlighting its limitations under coordination-intensive conditions. The microserver runtime provides intermediate performance (PI = 0.31), balancing efficiency and scalability but with reduced stability at higher loads.

These findings establish that deployment efficiency is fundamentally workload-contingent rather than environment-inherent. The proposed PI framework contributes a reproducible and decision-oriented method for comparing heterogeneous deployment environments, enabling objective ranking and identification of operational thresholds. By translating multi-dimensional performance metrics into a unified measure, the framework supports data-driven deployment selection and provides a foundation for workload-aware optimization in microservice systems.

Future work will extend this framework to additional orchestration platforms, event-driven workloads, and cost–performance trade-offs in cloud-native environments, enabling broader applicability and multi-objective optimization of deployment strategies.

AUTHOR CONTRIBUTIONS

Nidhi Vaniyawala: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing -- original draft, Writing -- review & editing, Visualization.

Kamlendu Kumar Pandey: Conceptualization, Methodology, Resources, Writing -- review & editing, Supervision.

REFERENCES

[1] Vaniyawala, N., Pandey, K. K., "A bird's eye view of microservice architecture from the lens of cloud computing", *Advancements in Smart Computing and Information Security (ASCIS 2023)*, Communications in Computer and

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Information Science, Vol. 2040, Springer, 2024.

[2] [Cortellessa, V., Di Pompeo, D., Tucci, M., "Exploring sustainable alternatives for the deployment of microservices architectures in the cloud", *Proceedings of IEEE 21st International Conference on Software Architecture (ICSA)*, IEEE, 2024, pp. 34–45.

[3] Ayas, H. M., Leitner, P., Hebig, R., "An empirical study of the systemic and technical migration towards microservices", *Empirical Software Engineering*, Vol. 28, 2023, p. 85.

[4] Meijer, W., Trubiani, C., Aleti, A., "Experimental evaluation of architectural software performance design patterns in microservices", *Journal of Systems and Software*, Vol. 218, 2024, p. 112183.

[5] Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z., Babar, M. A., "Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry", *Journal of Systems and Software*, Vol. 195, 2023, p. 111521.

[6] Xu, B., Guo, J., Ma, F., et al., "On the joint design of microservice deployment and routing in cloud data centers", *Journal of Grid Computing*, Vol. 22, 2024, p. 42.

[7] Singh, A., Singh, V., Aggarwal, A., et al., "Event Driven Architecture for Message Streaming Data Driven Microservices Systems Residing in Distributed Version Control System", *2022 International Conference on Innovations in Science and Technology for Sustainable Development (ICISTSD)*, 2022, pp. 308–312.

[8] Kolla, D. P., "Resilient Enterprise Systems through Scalable Microservices and Event-Driven Architectures", *International Journal of Computer Engineering and Technology (IJCET)*, Vol. 15, No. 6, 2024, pp. 1080–1090.

[9] Muthusamy, K., "Event-Driven Data Engineering in Microservices Architectures", *International Journal of Emerging Technologies and Computer Science Information Technology (IJETCSIT)*, Vol. 6, No. 1, 2025, pp. 36–43.

[10] Srijith, K. B. R., G. N., and A. M. R., "Inter-Service Communication among Microservices using Kafka Connect", 2022 IEEE *13th International Conference on Software Engineering and Service Science (ICSESS)*, 2022, pp. 43–47.

[11] Fritzsch, J., Bogner, J., Haug, M., et al., "Adopting microservices and DevOps in the cyber-physical systems domain: A rapid review and case study", *Software: Practice and Experience*, Vol. 53, No. 3, 2023, pp. 790–810.

[12] Kang, Z., An, K., Gokhale, A., Pazandak, P., "A comprehensive performance evaluation of different Kubernetes CNI plugins for edge-based and containerized publish/subscribe applications", *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, October 17, 2021.

[13] Chippagiri, S., "Optimizing Kubernetes network performance: A study of container network interfaces and system tuning profiles", *European Journal of Theoretical and Applied Sciences*, Vol. 2, No. 6, 2024, pp. 651–668.

[14] Koukis, G., Skaperas, S., Kapetanidou, I., Mamatas, L., Tsaoussidis, V., "Performance evaluation of Kubernetes networking approaches across constraint edge environments", *Proceedings of IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2024.

[15] Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., Zhuo, D., Mahajan, R., "Dissecting overheads of service mesh sidecars", **Proceedings of the ACM Symposium on Cloud Computing (SoCC '23)*, ACM, 2023, pp. 1–16.

[16] Noor, J., Faysal, M. B., Amin, M. S., Tabassum, B., Khan, T. R., Rahman, T., "Kubernetes application performance benchmarking on heterogeneous CPU architecture: An experimental review", *High-Confidence Computing*, Vol. 5, No. 1, 2025, p. 100276.

[17] Balis, B., Broński, A., Szarek, M., "Auto-scaling of scientific workflows in Kubernetes", *Euro-Par 2021: Parallel Processing Workshops*, Lecture Notes in Computer Science, Vol. 13162, Springer, 2022, pp. 44–55.

[18] Turin, G., Borgarelli, A., Donetti, S., Damiani, F., Johnsen, E. B., Tapia Tarifa, S. L., "Predicting resource consumption of Kubernetes container systems using resource models", *Journal of Systems and Software*, Vol. 203, 2023, p. 111750.

[19] Wyciślik, Ł., Latusik, Ł., Kamińska, A. M., "A comparative assessment of JVM frameworks to develop microservices", *Applied Sciences*, Vol. 13, No. 3, 2023, p. 1343.

[20] Henning, S., Hasselbring, W., "Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud", *Journal of Systems and Software*, Vol. 208, 2024, p. 111879.

[21] Smith, S., Robinson, E., Frederiksen, T., Stevens, T., Černý, T., Bures, M., Taibi, D., "Benchmarks for end-to-end microservices testing", *arXiv*, 2023.
[22] Henning, S., Vogel, A., Leichtfried, M., Ertl, O., Rabiser, R., "ShuffleBench: A benchmark for large-scale data shuffling operations with distributed stream processing frameworks", *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24)*, ACM, 2024, pp. 2–13.
[23] Vogel, A., Henning, S., Perez-Wohlfeil, E., Ertl, O., Rabiser, R., "A comprehensive benchmarking analysis of fault recovery in stream processing frameworks", *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems (DEBS '24)*, ACM, 2024, pp. 171–182.
[24] Mochniej, K., Badurowicz, M., "Performance comparison of microservices written using reactive and imperative approaches", *Journal of Computer Sciences Institute*, Vol. 28, 2023, pp. 242–247.
[25] Plecinski, P., Bokla, N., Klymkovych, T., Melnyk, M., Zabierowski, W., "Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks", *Sensors*, Vol. 22, No. 20, 2022, p. 7759.
[26] Laigner, R., Zhang, Z., Liu, Y., Gomes, L. F., Zhou, Y., "Online Marketplace: A benchmark for data management in microservices", *Proceedings of the ACM on Management of Data*, Vol. 3, No. 1, 2025, pp. 1–26.
[27] Sun, Y., Wang, J., Li, Z., Nie, X., Ma, M., Zhang, S., Ji, Y., Zhang, L., Long, W., Chen, H., Luo, Y., Pei, D., "A scenario-oriented benchmark for assessing AIOps algorithms in microservice management", *arXiv*, 2024.
[28] Port, D., Taber, B., Emkani, P., "Investigating effectiveness and compliance to DevOps policies and practices for managing productivity and quality variability", *Journal of Systems and Software*, Vol. 213, 2024, p. 112030.
[29] Azad, N., Hyrynsalmi, S., "DevOps critical success factors — A systematic literature review", *Information and Software Technology*, Vol. 157, 2023, p. 107150.
[30] Kumar, A., Nadeem, M., Shameem, M., "A systematic literature review for investigating DevOps metrics to implement in software development organizations", *Journal of Software: Evolution and Process*, Vol. 37, No. 1, 2024, p. e2733.
[31] Eramo, R., Said, B., Oriol, M., Bruneliere, H., Morales, S., "An architecture for model-based and intelligent automation in DevOps", *Journal of Systems and Software*, Vol. 217, 2024, p. 112180.
[32] Díaz-de-Arcaya, J., Torre-Bastida, A. I., Miñón, R., Almeida, A., "Orfeon: An AIOps framework for the goal-driven operationalization of distributed analytical pipelines", *Future Generation Computer Systems*, Vol. 140, 2023, pp. 18–35.
[33] Zhang, L., Jia, T., Jia, M., Wu, Y., Liu, A., Yang, Y., Wu, Z., Hu, X., Yu, P., Li, Y., "A survey of AIOps in the era of large language models", *ACM Computing Surveys*, Vol. 58, No. 2, 2025, p. 44.
[34] Rzig, D. E., Houerbi, A., Chavan, R. G., Hassan, F., "Empirical analysis on CI/CD pipeline evolution in machine learning projects", *arXiv*, 2024.
[35] Hassan, S., Bahsoon, R., Buyya, R., "Systematic scalability analysis for microservices granularity adaptation design decisions", *Software: Practice and Experience*, Vol. 52, No. 6, 2022, pp. 1378–1401.
[36] Assunção, W. K. G., Krüger, J., Mosser, S., Selaoui, S., "How do microservices evolve? An empirical analysis of changes in open-source microservice repositories", *Journal of Systems and Software*, Vol. 204, 2023, p. 111788.
[37] Jani, Y., "Unified Monitoring for Microservices: Implementing Prometheus and Grafana for Scalable Solutions", *Journal of Artificial Intelligence, Machine Learning and Data Science*, Vol. 2, No. 1, 2024, pp. 848–852.
[38] Noor, M., Yahaya, A. S., Ramli, N., Abdullah, M. M. A. B., "Filling missing data using interpolation methods: Study on the effect of fitting distribution", *Key Engineering Materials*, Vol. 594–595, 2013, pp. 889–895.
[39] Luciano, L., Kiss, I., Beardshear, P. W., Chen, J., Smith, A., "WISE: A computer system performance index scoring framework", *Journal of Cloud Computing*, Vol. 10, 2021, p. 8.
[40] SPEC, "SERT 2.0 metric overview", *Standard Performance Evaluation Corporation*, n.d.
[41] Andrew, J., "Balancing speed and scalability: Monolith breakdown strategies with event-driven architectures", *ResearchGate*, 2023.
[42] Wan, X., Guan, X., Wang, T., Bai, G., Choi, B.-Y., "Application deployment using microservice and Docker containers:

© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Framework and optimization", *Journal of Network and Computer Applications*, Vol. 119, 2018, pp. 97–109.

[43] Liu, Y., Yang, B., Wu, Y., Chen, C., Guan, X., "How to share: Balancing layer and chain sharing in industrial microservice deployment", *IEEE Transactions on Services Computing*, Vol. 16, No. 4, 2023, pp. 2685–2698.

[44] Vasireddy, I., Wankar, R., Chillarige, R. R., "Improving scalability, energy efficiency, and cost-effectiveness in Kubernetes clusters using a nonlinear regression-based predictive replica model and ORLE algorithm", *Egyptian Informatics Journal*, Vol. 31, 2025, p. 100732.