future internet

MDPI

Article

Benchmarking SQL and NoSQL Persistence in Microservices Under Variable Workloads

Nenad Pantelic [1], Ljiljana Matic [2], Lazar Jakovljevic [1], Stefan Eric [1], Milan Eric [1], Miladin Stefanović [1] and Aleksandar Djordjevic [1,*]

[1] Faculty of Engineering, University of Kragujevac, 34000 Kragujevac, Serbia; npantelic.fin@gmail.com (N.P.); lazarjakovljevic8@gmail.com (L.J.); eric.stefan002@gmail.com (S.E.); ericm@kg.ac.rs (M.E.); miladin@kg.ac.rs (M.S.)

[2] Faculty of Economics, University of Kragujevac, 34000 Kragujevac, Serbia; ljiljana.matic@ef.kg.ac.rs

* Correspondence: adjordjevic@kg.ac.rs

Abstract

This paper presents a controlled comparative evaluation of SQL and NoSQL persistence mechanisms in containerized microservice architectures under variable workload conditions. Three persistence configurations—SQL with indexing, SQL without indexing, and a document-oriented NoSQL database, including supplementary hybrid SQL variants used for robustness analysis—are assessed across read-dominant, write-dominant, and mixed workloads, with concurrency levels ranging from low to high contention. The experimental setup is fully containerized and executed in a single-node environment to isolate persistence-layer behavior and ensure reproducibility. System performance is evaluated using multiple metrics, including percentile-based latency (p95), throughput, CPU utilization, and memory consumption. The results reveal distinct performance trade-offs among the evaluated configurations, highlighting the sensitivity of persistence mechanisms to workload composition and concurrency intensity. In particular, indexing strategies significantly affect read-heavy scenarios, while document-oriented persistence demonstrates advantages under write-intensive workloads. The findings emphasize the importance of workload-aware persistence selection in microservice-based systems and support the adoption of polyglot persistence strategies. Rather than providing absolute performance benchmarks, the study focuses on comparative behavioral trends that can inform architectural decision-making in practical microservice deployments.

Keywords: microservices; SQL; NoSQL; performance evaluation; workload concurrency

Check for updates

Academic Editor: Paolo Bellavista

Received: 10 December 2025
Revised: 12 January 2026
Accepted: 13 January 2026
Published: 15 January 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The increasing adoption of microservice architectures and cloud-native design principles has reshaped expectations for data storage systems in modern distributed applications. As software systems evolve toward fine-grained service decomposition, each microservice frequently maintains its own persistence layer, often optimized for the specific requirements of its domain. These trends have accelerated the use of polyglot persistence, in which relational and NoSQL databases coexist within the same ecosystem, allowing services to balance transactional guarantees, schema rigidity, read efficiency, write scalability, and operational flexibility.

Relational databases remain a cornerstone of enterprise systems, offering robust ACID semantics, predictable query optimization, and strong indexing capabilities that make them

highly effective for structured, read-dominant workloads. Indexing, in particular, plays a critical role in improving read latency and reducing CPU load by minimizing full table scans. However, relational systems are known to experience performance degradation under heavy write concurrency, where index maintenance introduces additional latency, locking overhead, and buffer pressure.

NoSQL document stores such as MongoDB are designed to support high-volume writes, flexible schema evolution, and horizontal scalability. These systems typically relax strict consistency guarantees in favor of improved throughput and lower tail latency under write-intensive workloads. Their simplified write paths and memory-mapped storage models enable stable behavior under high concurrency, particularly in write-heavy and mixed workload scenarios, although their performance may be less competitive for structured, read-intensive tasks.

Although the theoretical distinctions between relational and NoSQL systems are well established, empirical comparisons that evaluate their behavior under controlled microservice-style workloads remain relatively limited. In particular, there is insufficient experimental research examining how (i) indexed versus non-indexed relational storage affects tail latency and concurrency scaling, (ii) document-oriented databases compare to relational systems across read-heavy (RH), write-heavy (WH), and mixed (MX) workloads, and (iii) resource metrics such as CPU and RAM usage evolve as concurrency increases from low (1–10 clients) to high (500–1000 clients) levels.

Given the importance of tail latency (p95) in distributed systems—where slow outliers disproportionately affect user-perceived performance—there is a clear need to experimentally evaluate how different persistence models behave across realistic workload patterns in containerized environments. Unlike large-scale production benchmarks, the emphasis of this study is on controlled, application-driven experimentation that isolates persistence-layer behavior under well-defined workload and concurrency conditions.

To address this gap, this study evaluates three persistence configurations—SQL with indexing, SQL without indexing, and MongoDB—within a controlled microservice-based environment deployed using Docker. MongoDB is selected as a representative document-oriented NoSQL system due to its maturity, widespread adoption, and frequent use as a baseline in comparative SQL/NoSQL studies. Standardized RH, WH, and MX workloads are executed across concurrency levels ranging from 1 to 1000 parallel requests, while performance is measured using p95 latency, throughput, CPU usage, and RAM consumption. The workload design prioritizes comparative trend analysis across persistence configurations rather than absolute performance certification against standardized benchmark suites. This multidimensional evaluation offers a comprehensive understanding of how architectural differences between relational and NoSQL storage systems affect system responsiveness, scalability, and resource utilization.

The study is guided by the following research questions: How do relational (indexed and non-indexed) and NoSQL persistence models differ in p95 latency and throughput under RH, WH, and MX workloads? How does increasing concurrency (from 1 to 1000 parallel requests) influence the performance and resource usage of SQL and NoSQL databases? Which persistence configuration provides the most stable behavior across tail latency, throughput, CPU usage, and RAM consumption under mixed workload patterns typical of microservice ecosystems?

Based on architectural characteristics and prior observations from the literature, the following hypotheses are formulated:

- H_1: Indexed SQL achieves significantly lower read latency (p95) than non-indexed SQL and MongoDB under RH workloads due to optimized index access paths.

https://doi.org/10.3390/fi18010053

- H_2: MongoDB achieves superior write throughput and lower tail latency than both SQL configurations under WH workloads, especially at high concurrency levels.
- H_3: MongoDB provides the most stable performance profile across RH, WH, and MX workloads as concurrency increases, exhibiting more predictable CPU and RAM usage compared to relational configurations.

This research contributes to the field by providing: (i) a controlled and reproducible experimental evaluation of SQL (indexed and non-indexed) and MongoDB under diverse workload patterns and concurrency levels; (ii) a detailed empirical comparison using p95 latency, throughput, CPU, and RAM—metrics essential for assessing real-world microservice performance; and (iii) practical insights for system architects on selecting persistence technologies aligned with the workload characteristics of individual microservices, offering evidence in support of selective polyglot persistence in distributed systems.

The remainder of the paper is organized as follows. Section 2 reviews existing literature on SQL and NoSQL systems, indexing strategies, and performance considerations in microservice environments. Section 3 describes the experimental methodology, including workloads, concurrency levels, system architecture, and measurement procedures. Section 4 presents performance results across all metrics and workloads. Section 5 provides a detailed discussion of the findings and their implications, and Section 6 summarizes key conclusions and outlines directions for future research. The study explicitly focuses on controlled comparative behavior rather than absolute production-level performance, and the reported results should be interpreted accordingly.

2. Related Work

Research on microservice architectures, data management patterns, and persistence technologies has expanded over the past decade, reflecting the growing complexity and heterogeneity of modern distributed systems. The literature explores several complementary areas: the evolution of software architectures from monolithic to microservice-based systems, the adoption of polyglot persistence principles, the foundations and performance characteristics of SQL and NoSQL databases, and the impact of indexing strategies and workload structures on database efficiency. These studies collectively highlight that no single persistence model is universally optimal, and that empirical evaluation is necessary to understand how different technologies behave under specific operational conditions such as read-heavy (RH), write-heavy (WH), or mixed (MX) workloads. However, many existing studies focus on isolated database benchmarks or large-scale distributed settings, providing limited insight into application-driven persistence behavior within containerized microservice architectures.

Given the breadth and diversity of prior research, synthesizing key insights across thematic areas is essential for establishing a clear theoretical foundation for the experimental analysis presented in this paper. Table 1 summarizes the most influential works relevant to this study, organized according to their primary contribution domains: microservice architectures, polyglot and hybrid persistence, SQL/NoSQL performance characteristics, indexing behavior, and workload-driven performance considerations.

This synthesized overview enables the reader to understand how each group of studies informs the design of the experimental framework and the interpretation of results. In contrast to survey-oriented or benchmark-centric studies, the present work emphasizes controlled comparative evaluation under uniform experimental conditions.

This synthesized overview provides the conceptual foundation for analyzing microservice data architectures in greater depth, beginning with the evolution of microservices and their implications for distributed storage models.

Table 1. Synthesis of Related Work on Microservices, Polyglot Persistence, SQL/NoSQL Architectures, and Workload Behavior.

Thematic Area	Authors	Main Contribution/Finding	Relevance to This Study
Evolution of Software Architectures & Microservices	Dragoni et al. [1]	Overview of the evolution from monolithic over SOA to microservices; motivations for modularity and scalability.	Explains the need for distributed systems and independent persistence layers.
	Fowler & Lewis [2]	Microservices defined as autonomous units with independent data and versioning.	Supports the concept of decentralized databases in microservices.
	Balalaie et al. [3]	Migration from monolith to microservices; DevOps benefits.	Demonstrates how architecture impacts performance and scalability.
	Villamizar et al. [4]	Comparison of monolith vs. microservices in the cloud; scalability and efficiency.	Highlights performance benefits of distributed systems.
Container Orchestration & Distributed Systems	Burns et al. [5]	Evolution of Borg/Omega → Kubernetes; resource management and latency control.	Justifies the use of Docker/Kubernetes-like environments in experiments.
Data Management in Microservices	Taibi & Lenarduzzi [6]	Patterns: database-per-service, shared DB, CQRS, event sourcing, polyglot persistence.	Provides theoretical foundation for combining SQL and NoSQL.
	dos Santos [7]	Cloud–edge orchestration; environment selection based on latency and resource metrics.	Links data-store performance to quality of service.
Polyglot Persistence & SQL/NoSQL Foundations	Deka [8]	Polyglot persistence; no single model optimal for all workloads.	Motivates empirical comparison of SQL and NoSQL.
	Cattell [9]	ACID vs. BASE differences; limited SQL horizontal scaling; NoSQL linear scaling advantages.	Predicts behavior under RH/WH workloads.
	Pokorný [10]	Categorization of NoSQL models and cloud applications.	Confirms relevance of MongoDB as a representative system.
	Khan et al. [11]	Systematic review of 142 SQL/NoSQL performance studies.	Validates metric selection: latency, throughput, CPU, RAM.
	James & Asagba [12]	Hybrid SQL/NoSQL models; balanced performance under certain conditions.	Supports motivation for evaluating different storage paradigms.
	Bjeladinović [13]	Influence of data structure on hybrid architecture design.	Explains expected variability in SQL vs. NoSQL performance.
	Oussous et al. [14]	NoSQL in big-data stacks for indexing, logging, aggregation.	Supports NoSQL advantages in WH scenarios.
Consistency & Distributed Behavior	Vogels [15]	Definition of eventual consistency (Amazon Dynamo).	Explains NoSQL behavior under WH and high concurrency.
Indexing & Operational Performance	Cattell [9]	Impact of indexing, transactions, and horizontal scaling on performance.	Predicts SQL degradation under WH load.
	Khan et al. [11]	SQL/NoSQL performance considerations and scalability constraints.	Theoretical basis for evaluating RH/WH/MX workloads.

2.1. Microservices & Hybrid Cloud–Edge Architectures

The architecture of microservices emerged as a response to the limitations of monolithic and traditional service-oriented systems. Dragoni et al. [1] provide a comprehensive overview of the evolution of software architectures, demonstrating the transition from monolithic applications to object-oriented systems, service-oriented architectures (SOAs), and ultimately microservice-based designs. The authors identify modularity, independent development, and selective scalability as key drivers behind this evolution.

Fowler and Lewis [2] define microservices as a collection of small, independently deployable services organized around business domains. They emphasize principles such as independent data management, decentralized governance, lightweight communication protocols (typically HTTP/REST), and autonomous team structures. This perspective is particularly relevant to persistence-layer design, as it motivates the decentralization of data storage and the coexistence of heterogeneous database technologies. Balalaie et al. [3] further expand on this viewpoint by analyzing the migration of a real-world commercial system from a monolithic architecture to microservices within a DevOps environment. Their findings demonstrate that microservices support faster deployment cycles, partial scaling, and reduced technical debt, albeit at the cost of increased operational complexity.

Villamizar et al. [4] conduct an empirical comparison of monolithic and microservice-based applications deployed in cloud environments, showing that microservices enable more efficient scaling and improved resource utilization under variable workloads. However, they also note that the performance benefits of microservices are closely tied to effective data management strategies, reinforcing the importance of persistence-layer evaluation.

At the infrastructure level, Burns et al. [5] describe the evolution of Google's container management systems—Borg, Omega, and Kubernetes—highlighting mechanisms for resource isolation, latency control, and large-scale orchestration. Kubernetes, as a successor to the Borg/Omega design principles, enables declarative configuration, automated scaling, and container orchestration, thereby facilitating widespread adoption of microservices in industrial settings. While such systems are often evaluated at cluster scale, their foundational principles also motivate controlled, containerized experimentation at smaller scales.

In parallel with architectural evolution, data management has emerged as a central challenge in microservice systems. Taibi and Lenarduzzi [6] analyze data management patterns in microservices, identifying approaches such as database-per-service, shared databases, command query responsibility segregation (CQRS), event sourcing, and polyglot persistence. They conclude that decentralized and polyglot data management is essential, as different services impose distinct functional and performance requirements. These findings provide a theoretical basis for combining SQL and NoSQL databases within a single microservice ecosystem.

Dos Santos [7] extends this discussion by examining microservice management in hybrid cloud–edge environments, where latency constraints and limited resources necessitate adaptive deployment strategies. The author proposes an autonomous microservice deployment system that leverages monitoring metrics, including latency and resource utilization, to select optimal execution environments. This paper underscores the direct relationship between persistence-layer performance and overall quality of service in heterogeneous distributed systems.

Older literature dealing with the microservices ecosystem often focuses on communication patterns, organizational principles, and scalability of applications, while more recent papers—especially in the domain of cloud–edge orchestration—point out the relevance of data as a central resource of the system. Thus, it may be seen that all the mentioned papers identify the relationship of data warehouses to the scalability and reliability of microservices. It is this connection that creates a theoretical bridge between polyglot persistence and database performance.

Consequently, it becomes clear that modern microservice systems are dependent on efficient data management in distributed environments. Different services require different data storage models, different indexes, and different consistency approaches. Cloud–edge architectures further complicate these challenges because resources, latency, and data access vary by location.

2.2. Polyglot Persistence & SQL/NoSQL Foundations

The increasing heterogeneity of data in modern information systems has driven the evolution of storage paradigms and the need to combine multiple database technologies within a single application. This approach, commonly referred to as polyglot persistence, advocates selecting storage models according to data characteristics and workload requirements. Deka [8] provides one of the most comprehensive treatments of this concept, emphasizing that no single database model is optimal across all workloads. Relational databases excel in transactional consistency and structured data processing, whereas NoSQL systems offer greater scalability, schema flexibility, and efficiency for high-volume or unstructured data

Cattell [9] examines the scalability limitations of traditional SQL systems, noting that while ACID transaction guarantees and indexing provide strong consistency and efficient reads, horizontal scaling often requires complex clustering or sharding mechanisms. In contrast, NoSQL systems typically achieve more linear scalability by relaxing consistency guarantees and adopting flexible data models such as document stores, key–value stores, and column-oriented databases. Pokorný [10] further categorizes NoSQL models and discusses their applicability in cloud-based applications, reinforcing their relevance for high-throughput and large-scale data processing.

A systematic review by Khan et al. [11] analyzes over one hundred SQL and NoSQL performance studies, identifying latency, throughput, and resource utilization as the most commonly reported evaluation metrics. Their findings also reveal substantial variability in experimental methodologies, particularly with respect to workload composition and concurrency modeling, which complicates direct comparison across studies. James and Asagba [12] propose hybrid SQL/NoSQL architectures that integrate transactional consistency with scalable data processing, demonstrating that such combinations can yield balanced performance under certain conditions.

Bjeladinović [13] investigates the influence of data structure on the design of hybrid SQL/NoSQL systems, concluding that dataset heterogeneity plays a critical role in persistence selection. Oussous et al. [14] similarly observe that large-scale data processing architectures frequently employ NoSQL technologies for indexing, logging, and aggregation, while relying on relational databases as authoritative sources of truth. This layered approach reflects common practice in microservice and cloud applications, where performance and consistency must be carefully balanced.

Finally, Vogels [15] introduces the concept of eventual consistency as a defining characteristic of many distributed NoSQL systems, such as Amazon Dynamo. Eventual consistency enables high parallelism and low-latency operations but introduces temporal windows of inconsistency. This trade-off is particularly relevant in write-heavy and mixed workloads, where relaxed consistency models can significantly improve throughput and tail-latency behavior.

Collectively, the reviewed literature demonstrates that SQL and NoSQL systems exhibit fundamentally different performance characteristics depending on workload structure, concurrency level, and consistency requirements. Nevertheless, relatively few studies examine indexed versus non-indexed relational configurations alongside document-oriented NoSQL systems within a unified, application-driven microservice context. This observation motivates the experimental design adopted in the present study. Building on these insights, the following section details the experimental methodology adopted to systematically evaluate SQL and NoSQL persistence behavior under controlled microservice workloads.

3. Methodology

The experimental evaluation was conducted in a controlled, containerized microservice environment. The system under study was implemented as a set of REST-based microservices with a dedicated persistence layer, supporting both relational and document-oriented databases. The relational persistence layer was implemented using an SQL database engine, while MongoDB was employed as the representative NoSQL document store. MongoDB was selected as a representative document-oriented NoSQL system due to its maturity, widespread adoption, and frequent use as a baseline in SQL/NoSQL comparative studies. All services and database instances were deployed using Docker, with each component isolated in its own container to ensure reproducibility and to minimize cross-component interference. All experiments were executed in a single-node setup, without replication or sharding, in order to deliberately isolate persistence-layer behavior

and avoid confounding effects introduced by distributed coordination mechanisms. This configuration was intentionally selected to provide a controlled baseline for comparing SQL and NoSQL persistence characteristics in microservice-based architectures.

Hardware and system resources were kept constant throughout all experiments, and no additional background workloads were executed on the host system during measurements. This ensured that observed performance differences originated from the persistence mechanisms themselves rather than from external system noise. Accordingly, the reported results are intended to support comparative analysis of persistence behavior rather than absolute claims about production-scale performance.

3.1. Experimental Design

The experimental design adopted in this study follows methodological principles established in prior work on performance evaluation of distributed systems, hybrid persistence architectures, and containerized microservices. Consistent with controlled benchmarking practices described by AbouShanab [16] and with the resource-contention insights presented by Costa et al. [17], the experiment focuses on isolating the performance characteristics of different persistence mechanisms under varying workload intensities and concurrency levels. This design choice addresses concerns regarding benchmark authority by emphasizing methodological control and internal validity over benchmark standardization.

To enable reproducible and unbiased comparison of persistence backends, all components of the benchmark—the microservice application, workload generator, and database engines—were deployed in a uniform containerized environment. Each of the three database configurations (SQL with indexing, SQL without indexing, and MongoDB as a document store) was executed under identical system conditions. The microservice layer exposed uniform REST endpoints, ensuring that all performance differences originated solely from the persistence layer rather than from variations in service logic or infrastructure.

The experimental protocol was designed to ensure consistency, fairness, and repeatability across all evaluated persistence configurations. For each persistence configuration, the system was initialized in a stable state before measurement. Workloads were then applied in a controlled manner, with concurrency levels increased systematically to observe performance trends under rising contention. Measurements were collected after the system reached steady-state operational behavior, thereby minimizing transient effects and warm-up bias. The protocol explicitly prioritizes comparative trend analysis across configurations instead of absolute performance benchmarking.

The experimental procedure was organized as a structured algorithmic workflow comprising three nested loops. The database loop iterated over the three persistence configurations, reconfiguring the microservice layer to route all requests to the appropriate backend. For each database configuration, the workload loop executed the three defined access patterns: read-heavy (RH), write-heavy (WH), and mixed (MX). These workloads correspond to widely studied categories in SQL/NoSQL research, where read-dominated scenarios emphasize indexing efficiency and query performance, while write-dominated scenarios stress transaction handling, disk I/O, and concurrency control. The concurrency loop systematically increased the number of simultaneous clients (1, 10, 50, 100, 500, and 1000), exposing scalability boundaries and contention effects.

By structuring the experiment as a complete factorial combination of persistence configuration, workload type, and concurrency level, the design ensures comprehensive coverage of the performance space under investigation.

To complement the algorithmic description, the experimental workflow can be formalized using the following mathematical model.

3.1.1. Formal Experimental Mathematical Model

For extended comparative analysis, the persistence configuration set D is augmented with hybrid variants, such that

$$D = D_{\text{base}} \cup D_{\text{hybrid}}, \tag{1}$$

where

$$D_{base} = \{SQL\_indexed, SQL\_no\_index, NoSQL\} \tag{2}$$

and

$$D_{hybrid} = \{Hybrid\_SQL\_indexed, Hybrid\_SQL\_no\_index\} \tag{3}$$

All hybrid configurations follow the same workload definitions, concurrency levels, metric computation, and experimental protocol as the baseline configurations.

$$W = \{\text{RH, WH, MX}\}, \tag{4}$$

the set of workload types, and

$$C = \{1, 10, 50, 100, 500, 1000\}, \tag{5}$$

the set of concurrency levels.

Each experiment run can be represented as a tuple

$$(d, w, c) \in D \times W \times C, \tag{6}$$

and the total number of experimental conditions is therefore

$$N_{\text{cond}} = \mid D \mid \cdot \mid W \mid \cdot \mid C \mid = 5 \times 3 \times 6 = 90. \tag{7}$$

For each configuration (d, w, c), the benchmark produces a set of response times

$$\left\{t_1, t_2, \ldots, t_{n(d,w,c)}\right\}, \tag{8}$$

from which the 95th percentile latency is computed as

$$\text{p95}(d, w, c) = Q_{0.95}(t_1, t_2, \ldots, t_{n(d,w,c)}), \tag{9}$$

where $Q_{0.95}$ denotes the empirical 0.95-quantile.

Throughput is defined as the number of successfully completed requests divided by the duration of the experiment:

$$\text{THR}(d, w, c) = \frac{R_{\text{succ}}(d, w, c)}{T_{\text{run}}(d, w, c)} [\text{req/s}]. \tag{10}$$

CPU utilization and RAM consumption are collected as time series at the container level,

$$\{\text{cpu}_1, \ldots, \text{cpu}_k\}, \{\text{ram}_1, \ldots, \text{ram}_k\}, \tag{11}$$

and summarized as arithmetic means:

$$\bar{\text{CPU}}(d, w, c) = \frac{1}{k}\sum_{i=1}^{k}\text{cpu}_i, \bar{\text{RAM}}(d, w, c) = \frac{1}{k}\sum_{i=1}^{k}\text{ram}_i. \tag{12}$$

The overall experimental outcome can thus be modeled as a mapping

$$F : D \times W \times C \to \mathbb{R}^4, \tag{13}$$

where

$$F(d, w, c) = (\text{p95}(d, w, c),\ \text{THR}(d, w, c),\ \bar{\text{CPU}}(d, w, c),\ \bar{\text{RAM}}(d, w, c)), \tag{14}$$

which provides a unified representation of all performance metrics for each persistence configuration, workload type, and concurrency level. The mapping F applies uniformly to both baseline and hybrid persistence configurations.

Within the innermost loop, the workload generator issues parallel HTTP requests to the microservice endpoints, after which the system records four key performance metrics: 95th percentile latency, throughput, CPU utilization, and RAM consumption. The use of p95 latency follows established reasoning in distributed systems research [5,16], which emphasizes tail-latency behavior as a critical indicator of microservice responsiveness under load. All metrics are collected at the container level to ensure technology-neutral and consistent measurement across persistence configurations.

After all combinations of database configuration, workload type, and concurrency level have been executed, the results are aggregated into a unified dataset for statistical analysis. This aggregation step enables systematic comparison of SQL and NoSQL systems across both operational profiles and resource-usage patterns.

3.1.2. Formal Experimental Algorithm

The mathematical model from the previous section formalizes the complete experimental workflow, while Figure 1 provides a visual flowchart representation with blocks (standard steps vs. loop structures) and icons highlighting database operations, workload generation, and metric collection. This dual representation enhances clarity and illustrates how the experimental framework comprehensively explores the performance space of hybrid persistence in microservice environments.

The flowchart in Figure 1 provides a complete conceptual overview of the experimental workflow. The combined representation makes the relationship between configuration loops, workload execution, and metric collection transparent, reinforcing the methodological rigor of the study. This dual presentation ensures that both the logical control structure and the practical execution sequence of the benchmark are well understood before proceeding to the detailed discussion of metrics, environment setup, and results.

3.2. Workload Scenarios (RH/WH/MX)

The workload design was constructed to reflect typical access patterns observed in microservice-based applications, where persistence layers are subject to heterogeneous read and write demands. Rather than relying on synthetic, benchmark-specific query models, the workload was derived from application-level REST interactions executed against the microservices. This application-driven approach improves ecological validity by preserving the same request paths, serialization logic, and persistence access code used by the service layer. Three workload categories were considered: RH, WH, and MX workloads. RH scenarios primarily consisted of data retrieval operations (e.g., REST GET requests), while WH scenarios emphasized data creation and update requests (e.g., REST POST/PUT requests). MX combined read and write operations within the same execution interval, thereby reflecting more realistic application behavior. To ensure comparability, request payload structure, endpoint routing, and response handling were kept consistent across configurations; only the persistence backend differed. All workload types were executed

under varying levels of concurrent client requests to evaluate the impact of increasing contention on persistence-layer performance. These scenarios correspond to typical operational profiles in microservice-based systems and are aligned with the classifications described by Deka [8] for NoSQL workloads and by Cattell [9] and Pokorný [10] for relational and non-relational query behavior. Following common practice in workload-aware evaluation, RH, WH, and MX workloads are defined operationally by the dominance of read vs. write request types within the same execution window (i.e., a read-majority, write-majority, and balanced mix).

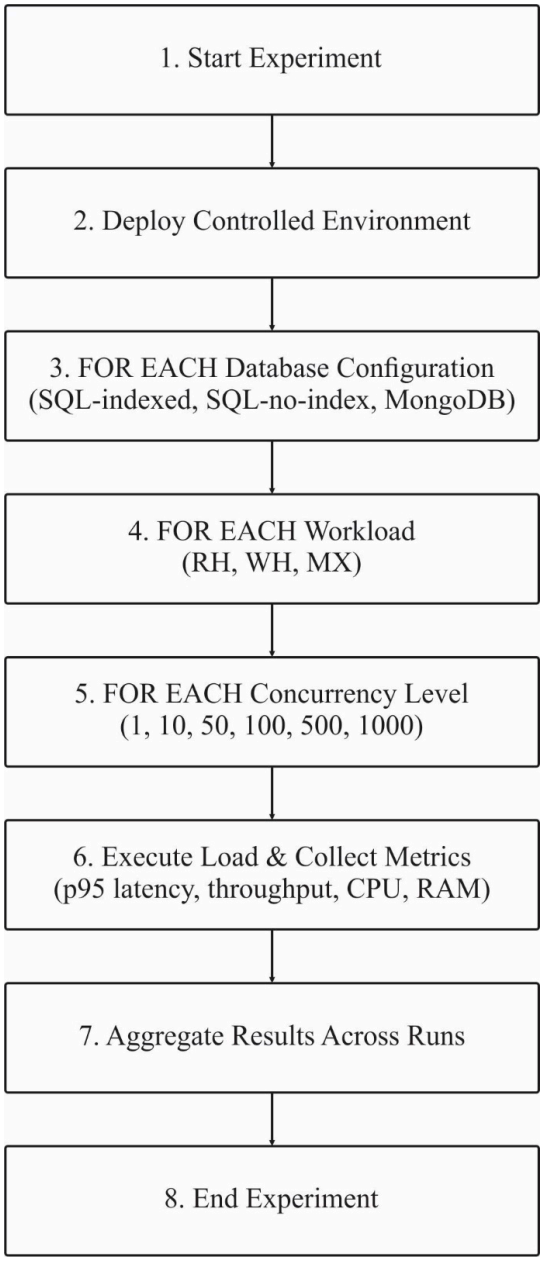


Figure 1. Algorithmic workflow of the experimental procedure, illustrating the initialization of the environment.

Together, the three workload scenarios capture distinct yet complementary operational profiles found in enterprise and cloud-native systems. Their inclusion follows methodological precedence in empirical benchmarking studies [5,16] and in hybrid persistence literature, where multi-dimensional evaluation is required to reveal trade-offs between indexing overhead, concurrency control, schema flexibility, and transactional guarantees.

By structuring the experiment around RH, WH, and MX workloads, the study ensures that its results reflect both isolated and combined performance behaviors, enabling rigorous comparison between SQL and NoSQL persistence under controlled but realistic conditions.

Concurrency levels were systematically increased to observe performance degradation trends, throughput saturation points, and resource utilization behavior under stress. The same workload logic and request sequences were applied consistently across all persistence configurations to ensure fairness of comparison. This approach isolates the influence of the persistence mechanism itself, allowing observed performance differences to be attributed to the underlying database technologies rather than to workload variation. Because the workloads are application-driven, the primary objective is controlled comparative analysis, not certification against standardized benchmark suites; accordingly, the results are interpreted in terms of relative trends across configurations.

3.3. Database Configurations (SQL with Index/SQL Without Index/NoSQL)

The experiment evaluates three distinct database configurations to isolate the performance characteristics of indexing, transactional semantics, and document-oriented storage. These configurations reflect the core architectural differences identified in the literature on relational and non-relational systems [8–11] and are representative of persistence strategies commonly found in microservice-based applications.

Each configuration was deployed as a separate database instance within a controlled, containerized environment, ensuring identical networking, hardware allocation, and execution conditions. Since all other components of the system remained constant—including the workload generator, microservice logic, and container runtime—performance differences can be attributed solely to the characteristics of each persistence model.

The first configuration employs a relational database with indexing enabled on frequently queried fields. In this experiment, enabling indexes represents the optimized relational baseline, reflecting how SQL databases are typically configured in production systems where read performance is a priority. This configuration is expected to perform well under RH workloads but may incur additional overhead under WH workloads due to per-update index maintenance, a phenomenon widely reported in relational performance studies.

The second configuration removes indexing from the relational schema, exposing the raw behavior of SQL storage without optimization structures. This configuration is valuable for isolating the impact of index maintenance and query access paths. It provides insight into how relational systems behave when write operations avoid index updates while read operations increasingly rely on full scans as the dataset grows. Although such a configuration is uncommon in production, it is methodologically useful for controlled comparison because it disentangles indexing effects from other relational characteristics (e.g., ACID semantics and query planning).

The third configuration uses a NoSQL document database (MongoDB), representing a schema-flexible, high-throughput persistence model characteristic of BASE systems [9,14]. NoSQL databases store data as documents rather than normalized relational structures, enabling flexible schemas and efficient write paths under high concurrency. This configuration supports evaluation of whether document-oriented persistence provides measurable advantages under WH and MX workloads and how it compares to indexed and non-indexed relational storage.

The selection of SQL with index, SQL without index, and NoSQL aligns with distinctions emphasized in the literature on hybrid persistence and microservice architectures: (i) indexing vs. no indexing enables direct measurement of optimization structures central to relational performance; (ii) relational vs. document-oriented storage reflects real-world

polyglot persistence patterns (e.g., Taibi and Lenarduzzi [6]); and (iii) transactional integrity vs. flexible consistency mirrors the ACID–BASE contrast that defines SQL/NoSQL trade-offs [8,9]. This controlled triplet of configurations directly addresses reviewer concerns regarding methodological clarity by enabling attribution of observed effects to specific persistence design choices.

In addition to the three primary configurations, the experimental dataset also contains results for two hybrid persistence variants—Hybrid Indexed SQL and Hybrid Non-Indexed SQL—which combine relational persistence with partial NoSQL-style access paths at the service layer. These configurations were not treated as primary experimental factors but were included as supplementary measurements to assess the robustness of the observed trends. Hybrid results are therefore reported selectively in aggregated tables and comparative analyses, without altering the core experimental design.

3.4. Concurrency Levels (1, 10, 50, 100, 500, 1000)

Evaluating system performance under different levels of concurrent access is fundamental for understanding how persistence technologies behave under increasing operational stress. The concurrency levels used in this study—1, 10, 50, 100, 500, and 1000 simultaneous requests—were selected to reflect progressive escalation from isolated, single-client access to high-load stress-test conditions. This stepwise approach follows methodological principles commonly adopted in microservice performance investigations [16] and multitenancy-oriented evaluations [17], where increasing parallelism exposes bottlenecks in CPU utilization, locking mechanisms, and I/O behavior.

Testing with a single request provides a baseline measurement of raw system responsiveness without contention, isolating intrinsic persistence-layer latency for individual operations. Intermediate concurrency levels introduce measurable contention at the database level. Prior studies [9,11] show that relational systems may experience contention due to transaction boundaries and index maintenance, whereas document stores may exhibit different saturation behavior depending on their write path and internal storage mechanisms. Testing at 50 and 100 concurrent users provides insight into sustained parallel demand and whether resource consumption grows proportionally or exhibits non-linear escalation.

The highest concurrency levels (500 and 1000) represent stress conditions where resource contention dominates system behavior. Research on containerized microservices [16] and distributed systems [5] highlights that tail latency can increase sharply under such loads, making p95 a critical metric. These high-concurrency scenarios are therefore included to reveal scalability boundaries and identify configuration-specific breaking points under uniform experimental conditions.

Employing a wide range of concurrency levels ensures that the experiment evaluates both scalability and resilience of the persistence layer. This methodology aligns with recommendations from empirical studies of hybrid systems, where gradual increases in concurrent demand reveal performance cliffs and resource exhaustion patterns that average-latency metrics alone may conceal.

3.5. Metrics (p95 Latency, Throughput, CPU, RAM)

To comprehensively assess the behavior of different persistence configurations in a microservice environment, multiple performance metrics were collected, capturing both user-perceived performance and system-level resource utilization. The selected metrics reflect commonly adopted indicators in the evaluation of distributed and containerized systems.

The experiment evaluates four primary metrics—p95 latency, throughput, CPU usage, and RAM consumption—to capture complementary dimensions of system behavior: responsiveness (latency), processing capacity (throughput), computational cost (CPU), and

memory efficiency (RAM). These metrics are widely used in empirical studies of distributed systems and containerized microservice environments [5,16] and provide a robust basis for comparing indexed SQL, non-indexed SQL, and document-oriented NoSQL persistence under RH, WH, and MX workloads. Latency was measured at the request level and represents the elapsed time between the initiation of a client request and receipt of the corresponding response. To account for tail-latency effects, percentile-based latency statistics were employed, with particular emphasis on high-percentile values. Throughput was quantified as the number of successfully processed requests per unit of time, reflecting the system's capacity to sustain increasing demand. CPU utilization and memory consumption were monitored throughout all experimental runs to evaluate resource efficiency and identify potential bottlenecks at the persistence layer. These metrics were collected at the container level, ensuring consistent measurement across persistence configurations.

3.5.1. 95th Percentile Latency

Latency analysis in microservice-based systems is particularly sensitive to outliers and tail behavior, especially under increased concurrency and resource contention. For this reason, latency evaluation in this study relies on percentile-oriented statistics rather than average response time alone. High-percentile latency values capture tail effects and more accurately reflect user-perceived performance degradation under load. The 95th percentile latency (p95) was selected as a representative indicator of tail performance, balancing sensitivity to slowdowns with robustness against isolated anomalies. Unlike average latency—which can mask the impact of slow requests—p95 highlights the slowest 5% of responses. In this experiment, p95 latency provides a sensitive indicator of indexing efficiency under RH workloads, index-maintenance overhead under WH scenarios, and overall stability under high concurrency.

3.5.2. Throughput Characteristics

Throughput, measured as the number of successfully processed requests per second, complements latency by capturing sustained processing capacity. Prior comparative studies [8,11] frequently report that relational systems may degrade when locks or index updates accumulate, while document stores can sustain higher throughput under write-dominant loads due to reduced transactional overhead and schema flexibility. In this study, throughput is essential for identifying how each persistence configuration balances read/write pressure, particularly in the mixed workload scenario where competition between operations is highest.

3.5.3. CPU Utilization Characteristics

CPU usage provides insight into the computational overhead associated with each persistence model. As noted in multitenant architecture studies [17], resource contention significantly influences both latency and throughput under parallel access patterns. Similarly, system-level analyses [16] show that CPU saturation correlates strongly with increases in tail latency and request queuing.

3.5.4. RAM Utilization

Memory consumption is measured to evaluate how efficiently each persistence configuration uses system resources under increasing load. These observations are consistent with persistence behavior described in NoSQL and hybrid storage literature [8,14]. Together, the four metrics form a multidimensional performance profile suitable for interpreting the results presented in later sections.

3.6. Hardware & Software Environment

All experiments were executed in a controlled, single-host environment to ensure that performance differences arise solely from the evaluated persistence configurations. The microservice and all three database systems (SQL with index, SQL without index, and MongoDB) were deployed on the same machine using Docker containers, providing isolated, reproducible, and uniform runtime conditions.

The relational database engine used in the experiments was MySQL (version 5.7.36), selected for its maturity, full ACID compliance, and common use in microservice-based systems. Explicitly stating the SQL engine and version addresses concerns regarding objectivity and reproducibility, as SQL performance can vary materially across database implementations. The host machine was configured with an 8-core CPU, 16 GB RAM (to prevent swapping), and 512 GB of local disk storage. A local Docker bridge network was used to ensure stable execution without external interference. Docker Compose orchestrated the microservice and database containers, guaranteeing consistent initialization and identical environment settings across runs.

The application layer used Spring Boot with Spring Data JPA/Hibernate for SQL and Spring Data MongoDB for NoSQL, enabling uniform request logic across persistence models. No OS-level tuning or database-specific optimizations (e.g., caching configuration changes beyond defaults, sharding, or replication) were applied. Before each experimental cycle, containers were restarted and the database state was cleared to ensure reproducibility.

Performance measurements were collected for each workload scenario (RH, WH, MX) across concurrency levels of 1, 10, 50, 100, 500, and 1000. For every configuration, the system recorded p95 latency, throughput, CPU usage, and RAM consumption using both application-level request timing and container-level resource metrics. To mitigate transient effects, each scenario was repeated and the environment was reset between runs. Raw measurements were logged with identifiers for workload type, concurrency level, persistence configuration, and timestamp, enabling direct cross-scenario comparison.

Consistent with the overall methodological positioning of the study, this environment is intended to support controlled comparison rather than to emulate production-grade distributed deployments. Accordingly, the results should be interpreted as comparative indicators of persistence behavior under uniform, reproducible conditions.

3.7. Threats to Validity

As with any empirical evaluation of distributed and microservice-based systems, this study is subject to several threats to validity that should be considered when interpreting the results. Internal validity may be affected by transient system behavior inherent to containerized environments, such as CPU scheduling effects, background daemon activity, and short-term caching phenomena. These risks were mitigated by executing all experiments under identical conditions, restarting containers between runs, clearing database state, and avoiding concurrent background workloads on the host system. Measurements were collected after the system reached steady-state behavior to reduce warm-up bias and transient fluctuations. External validity is limited by the single-node experimental setup, which does not incorporate replication, sharding, distributed consensus protocols, or geographically distributed deployments. Consequently, the results should not be directly generalized to large-scale production systems or cloud-native clusters without further investigation. Instead, the findings are intended to provide comparative insights into persistence-layer behavior under controlled conditions, serving as a baseline for future multi-node or cloud-scale studies. Construct validity is influenced by the application-driven workload design, which prioritizes realistic microservice interactions over standardized benchmark suites. While this choice improves ecological validity, it may limit direct numerical comparability

with benchmark-centric studies. To mitigate this limitation, the study focuses on relative performance trends across identical workloads rather than on absolute performance claims. In conclusion, validity is addressed by emphasizing comparative behavioral patterns instead of isolated measurements. Statistical interpretation relies on percentile-based latency (p95), throughput trends, and consistent cross-configuration comparisons. By avoiding overgeneralization and framing conclusions in terms of observed tendencies rather than universal dominance, the study reduces the risk of overstated or misleading claims.

Taken together, these considerations position the study as a controlled comparative analysis rather than a production benchmarking exercise, and the reported results should be interpreted within this clearly defined methodological scope.

4. Results

The results presented in this section reflect the behavior of the evaluated persistence configurations under different workload types and concurrency levels, following the experimental design described in the previous section. Consistent with the methodological scope defined in Section 3, the analysis focuses on comparative performance trends rather than absolute performance claims. The goal of the analysis is to quantitatively assess how workload characteristics (RH, WH, MX) and underlying data-store architectures influence key performance metrics, including p95 latency, throughput, CPU utilization, and memory consumption.

Before examining each metric in detail, Figure 2 provides an aggregated performance overview across all workloads and persistence configurations.

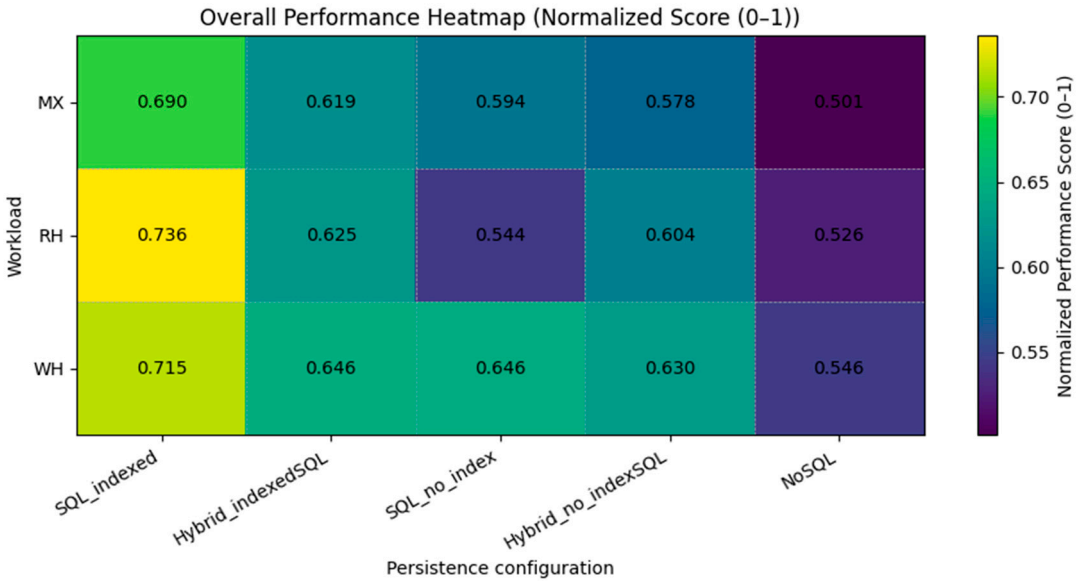


Figure 2. Overall performance heatmap summarizing normalized p95 latency, throughput, CPU utilization, and RAM consumption across workload types (RH, WH, MX) and persistence configurations, including baseline SQL and NoSQL systems as well as hybrid SQL variants. Hybrid configurations are positioned adjacent to their corresponding baselines to highlight the effect of architectural refinements on aggregated performance behavior.

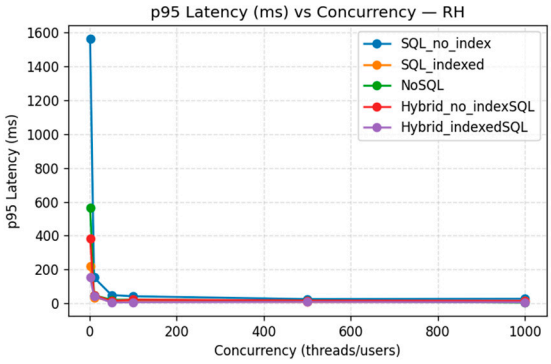
This high-level visualization is intended to support relative comparison and pattern recognition, rather than detailed metric interpretation. It offers a concise comparative summary of the relative efficiency of SQL (indexed and non-indexed) and NoSQL systems and serves as a contextual foundation for the detailed results discussed in the subsections that follow.

The heatmap presents a composite performance score derived from normalized p95 latency, throughput, CPU utilization, and RAM consumption. In addition to baseline configurations (SQL with index, SQL without index, and NoSQL), the evaluation includes hybrid persistence variants (Hybrid SQL with index and Hybrid SQL without index). Hybrid configurations exhibit intermediate performance characteristics, confirming their role as transitional designs between fully relational and document-oriented persistence models. Having established the overall performance landscape, the analysis proceeds with p95 latency, which is treated as the primary indicator of tail behavior and system stability under load, in line with the experimental rationale defined earlier.

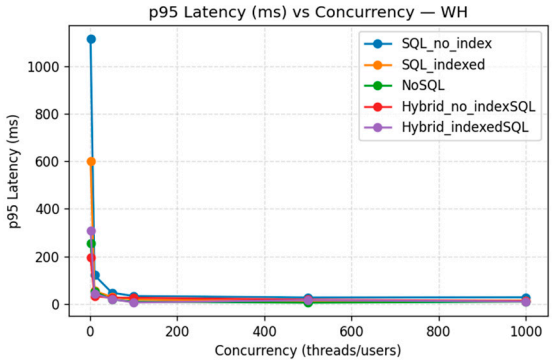
4.1. p95 Latency

The p95 latency results clearly differentiate the three persistence configurations across workload types and concurrency levels. Because p95 captures tail behavior rather than average response time, the discussion emphasizes scalability limits and contention effects rather than isolated low-load performance.

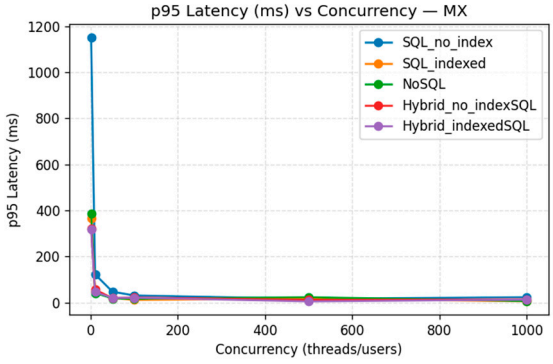
In the RH workload, SQL with index consistently achieves the best tail latency for most concurrency levels (Figure 3a and Table 2). Its average p95 latency over all RH tests is around 52 ms, compared to about 114 ms for NoSQL and more than 310 ms for SQL without index. This result directly reflects the effectiveness of indexing in reducing full table scans and minimizing read-path contention. At very high concurrency (1000 parallel requests), NoSQL briefly achieves the lowest p95 latency; however, this occurs in a regime where overall system saturation is already present, and SQL with index still dominates in terms of throughput. SQL without index remains the worst-performing option in both latency and throughput.



(**a**) p95 latency for read heavy scenario



(**b**) p95 latency for write heavy scenario



(**c**) p95 latency for mixed heavy scenario

Figure 3. P95 latency metrics across different workload scenarios (Read-Heavy, Write-Heavy, Mixed).

Table 2. Average p95 latency (ms) Across Persistence Configurations.

System	RH	WH	MX
SQL_indexed	51.87	120.60	77.51
SQL no index	310.75	228.55	231.57
NoSQL (MongoDB)	113.81	57.80	81.70
Hybrid indexed SQL	38.70	66.86	69.83
Hybrid no index SQL	83.91	51.02	73.70

In the WH workload, the ranking is reversed. NoSQL achieves the lowest p95 latency across all concurrency levels, with an average p95 of roughly 58 ms, compared to about 121 ms for indexed SQL and 229 ms for non-indexed SQL (Figure 3b). This behavior is consistent with the reduced transactional overhead and simpler write paths characteristic of document-oriented persistence. Both relational configurations show higher tail latencies, particularly at low and medium concurrency, reflecting the cost of transactional guarantees and, for the indexed variant, index maintenance. SQL without index performs slightly better than indexed SQL in WH scenarios, but the gap to NoSQL remains substantial.

MX workloads combine reads and writes and therefore expose the systems' ability to handle competing access patterns (Figure 3c). SQL with index and NoSQL exhibit very similar average p95 latency (approximately 78 ms vs. 82 ms), both significantly outperforming SQL without index. At the highest concurrency level (1000 parallel requests), SQL with index achieves the lowest p95 latency, with NoSQL close behind and SQL without index clearly lagging. This indicates that, under balanced read–write pressure, optimized relational storage can match document-oriented systems in tail latency, provided that write pressure does not dominate.

Hybrid SQL configurations follow the same workload-dependent trends as their non-hybrid counterparts, with Hybrid Indexed SQL exhibiting improved tail latency across all workloads, particularly under mixed and write-heavy scenarios (Table 2).

In summary, p95 latency results show that indexed SQL is clearly preferable for RH workloads and competitive with NoSQL in MX workloads, while NoSQL is the best choice for WH workloads. SQL without index is consistently the worst option for tail latency, regardless of workload type.

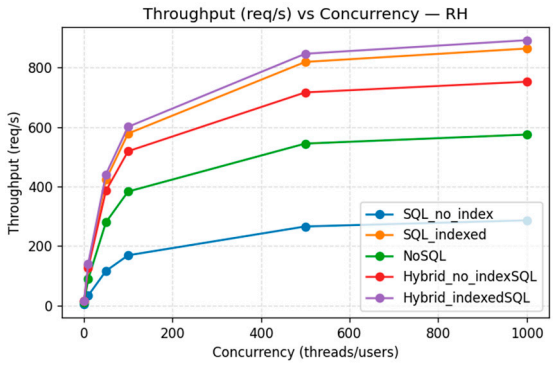
4.2. Throughput

Throughput trends broadly mirror latency results while revealing additional differences in sustained processing capacity. Throughput is interpreted jointly with p95 latency to avoid misleading conclusions based on isolated metrics.

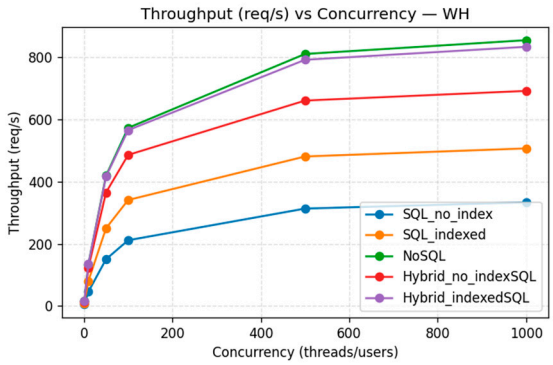
In RH workloads (Figure 4a and Table 3), SQL with index consistently delivers the highest throughput at all concurrency levels, with an average of approximately 472 req/s, compared to 313 req/s for NoSQL and 145 req/s for SQL without index. The severe throughput degradation of the non-indexed configuration highlights the cost of full table scans under concurrent read access.

Table 3. Average Throughput (req/s).

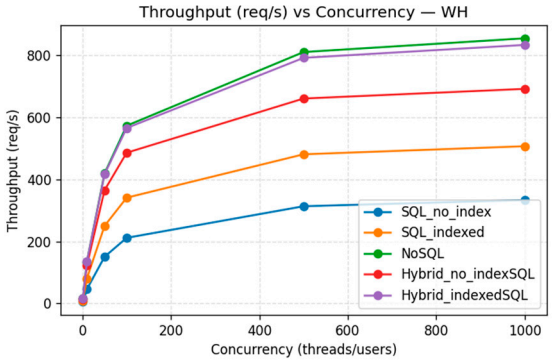
System	RH	WH	MX
SQL_indexed	472.13	277.98	372.79
SQL no index	145.41	176.46	140.22
NoSQL (MongoDB)	313.24	467.93	365.09
Hybrid indexed SQL	489.38	459.76	431.11
Hybrid no index SQL	418.91	390.14	412.24



(**a**) Throughput for read-heavy scenario



(**b**) Throughput for write-heavy scenario



(**c**) Throughput for mixed heavy scenario

Figure 4. Throughput metrics across different workload scenarios (Read-Heavy, Write-Heavy, Mixed).

In WH workloads (Figure 4b), NoSQL clearly dominates, achieving an average throughput of roughly 468 req/s, compared to 278 req/s for indexed SQL and 176 req/s for non-indexed SQL. At higher concurrency levels, NoSQL maintains stable throughput, while relational configurations exhibit diminishing returns due to transactional and index-update overhead.

In MX workloads (Figure 4c), SQL with index and NoSQL again show comparable performance, with indexed SQL slightly ahead. Average throughput is approximately 373 req/s for indexed SQL, 365 req/s for NoSQL, and only 140 req/s for non-indexed SQL. This convergence suggests that, under mixed access patterns, indexing enables relational systems to compete effectively with NoSQL in terms of sustained request processing.

Hybrid SQL configurations follow the same workload-dependent throughput trends as their non-hybrid counterparts, with Hybrid Indexed SQL consistently improving sustained request rates under WH and MX workloads, further reinforcing the robustness of the observed relational vs. NoSQL performance patterns.

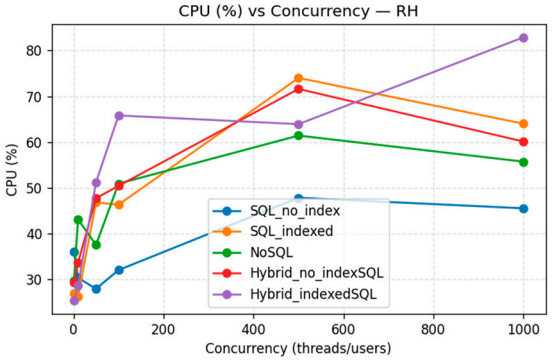
Overall, throughput results confirm that indexed SQL is optimal for RH workloads, NoSQL is best suited for WH workloads, and both approaches perform comparably under MX workloads. Non-indexed SQL consistently exhibits limited scalability and poor throughput under realistic access patterns.

4.3. CPU Utilization

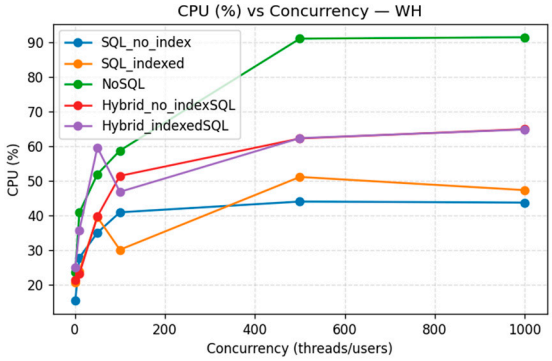
CPU utilization results reveal how much computational effort each configuration requires to deliver its observed latency and throughput. CPU metrics are interpreted as a measure of efficiency rather than as an objective to be minimized in isolation.

In RH workloads (Figure 5a and Table 4), all three configurations show increasing CPU usage with concurrency, but there is no single dominant winner. NoSQL and indexed SQL exhibit similar average CPU usage (around 46–47%), while non-indexed SQL uses

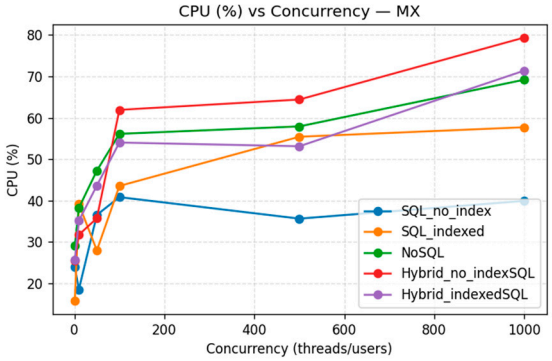
slightly less CPU (~37%) but at the cost of much poorer latency and throughput. The lower CPU usage of non-indexed SQL in RH workloads therefore reflects underutilization and inefficiency rather than better optimization. However, this lower CPU usage coincides with significantly worse latency and throughput, indicating inefficiency rather than superior optimization.



(**a**) CPU utilization for read-heavy scenario



(**b**) CPU utilization for write-heavy scenario



(**c**) CPU utilization for mixed heavy scenario

Figure 5. CPU utilization metrics across different workload scenarios (Read-Heavy, Write-Heavy, Mixed).

Table 4. Average CPU Utilization (%).

System	RH	WH	MX
SQL_indexed	47.37	35.53	39.88
SQL no index	36.58	34.57	32.53
NoSQL (MongoDB)	46.37	59.68	49.60
Hybrid indexed SQL	52.95	49.15	47.15
Hybrid no index SQL	48.78	43.87	49.77

In WH workloads (Figure 5b), CPU patterns diverge more strongly. NoSQL consumes the most CPU on average (about 60%), while both relational configurations use considerably less (approximately 36% for indexed SQL and 35% for non-indexed SQL). This indicates that NoSQL aggressively uses available CPU resources to sustain high write throughput, while relational systems trade lower CPU usage for weaker performance under write-heavy load. This reflects NoSQL's strategy of aggressively utilizing available CPU resources to sustain high write throughput.

In MX workloads (Figure 5c), NoSQL again records the highest average CPU usage (~50%), indexed SQL uses a moderate amount (~40%), and non-indexed SQL the least (~33%). As in RH workloads, the lower CPU usage of non-indexed SQL coincides with

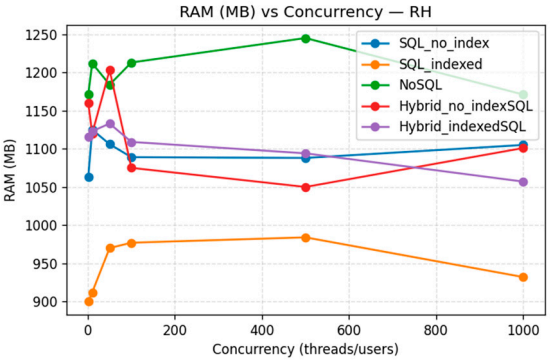
clearly inferior latency and throughput, meaning that its resources are not effectively converted into useful work. In contrast, NoSQL consistently couples higher CPU usage with strong throughput and competitive tail latency. Across all workloads, higher CPU usage in NoSQL correlates with better performance rather than wasted computation.

Hybrid SQL variants exhibit CPU utilization profiles that closely mirror those of their base SQL configurations, while partially mitigating peak CPU saturation under mixed workloads, indicating that the primary resource-usage trends remain stable under hybrid persistence designs.

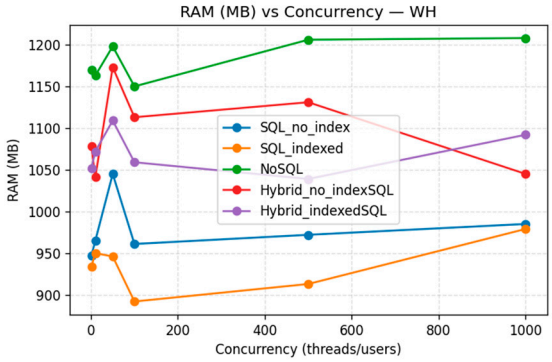
These findings suggest that NoSQL is the most CPU-intensive option but uses CPU resources effectively to maximize throughput, particularly for WH and MX workloads. SQL with an index achieves good performance with moderate CPU cost, while SQL without an index often appears CPU-efficient since it fails to process as many requests in the same time.

4.4. RAM Consumption

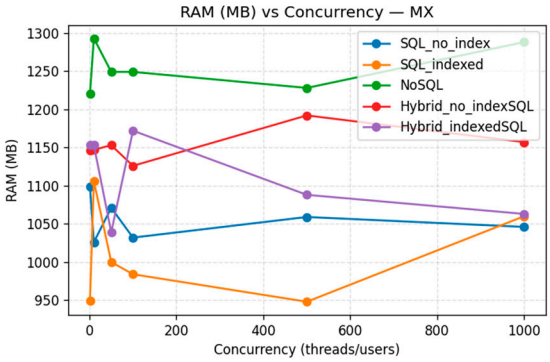
RAM usage patterns complement the CPU results and highlight important trade-offs in memory behavior. Across all workloads (Figure 6a–c and Table 5), NoSQL consistently consumes the most RAM. Its average memory footprint ranges from roughly 1180 MB (WH) to 1255 MB (MX), reflecting the cost of memory-mapped files and internal caching mechanisms. SQL without index uses intermediate amounts of RAM (about 979–1096 MB), while indexed SQL maintains the smallest memory footprint in all three workloads, with averages between 936 MB (WH) and 1008 MB (MX). This behavior is consistent with memory-mapped storage mechanisms and internal caching strategies typical of document-oriented databases.



(**a**) RAM consumption for read-heavy scenario



(**b**) RAM consumption for write-heavy scenario



(**c**) RAM consumption for mixed heavy scenario

Figure 6. RAM consumption metrics across different workload scenarios (Read-Heavy, Write-Heavy, Mixed).

Table 5. Average RAM Consumption (MB).

System	RH	WH	MX
SQL_indexed	945.83	935.67	1007.83
SQL no index	1096.00	979.17	1055.50
NoSQL (MongoDB)	1199.33	1182.50	1254.50
Hybrid indexed SQL	1105.33	1070.50	1111.33
Hybrid no index SQL	1118.17	1096.83	1153.50

In RH workloads, indexed SQL achieves superior latency and throughput while also using the least RAM, indicating that its indexing structures and buffer management are relatively memory-efficient compared to NoSQL and the non-indexed relational configuration. In WH and MX workloads, NoSQL maintains higher throughput and better write performance but at the cost of significantly increased memory usage. Non-indexed SQL occupies a middle position, with moderate RAM usage but generally inferior performance. Non-indexed SQL again fails to present a favorable trade-off between resource usage and performance.

Memory consumption patterns of Hybrid SQL configurations remain consistent with relational storage behavior, showing moderate RAM overhead compared to MongoDB while preserving the same relative ordering across workload types, which confirms that hybridization does not alter the fundamental memory–efficiency trade-offs.

Taken together, the resource metrics show that NoSQL offers high performance at the cost of higher CPU and RAM consumption, while indexed SQL achieves strong read and mixed performance with the lowest memory footprint. Non-indexed SQL rarely presents an attractive trade-off, as it combines weaker performance with only modest resource savings.

4.5. Integrated Performance Patterns

The combined analysis of p95 latency, throughput, CPU utilization, and RAM consumption yields (Figure 7) a coherent picture of the three persistence configurations:

- SQL with index is clearly the best option for read-heavy workloads, offering the lowest p95 latency and highest throughput while using the least RAM. In mixed workloads, it remains competitive with NoSQL, often slightly outperforming it in both latency and throughput at high concurrency, again with a smaller memory footprint. Its main weakness is write-heavy load, where index maintenance inflates tail latency and limits throughput.
- SQL without index consistently performs worst in latency and throughput for read-heavy and mixed workloads, due to full table scans. It improves in write-heavy workloads compared to indexed SQL, but still significantly trails NoSQL. Its lower CPU and moderate RAM usage generally reflect reduced effective work rather than better efficiency. Its apparent resource efficiency largely reflects reduced effective work rather than superior scalability.
- NoSQL (MongoDB) is the most robust choice for write-heavy workloads and performs competitively in mixed workloads, achieving the highest or near-highest throughput and low p95 latency, especially at high concurrency. However, this comes at the cost of higher CPU and RAM consumption in all workload types. In purely read-heavy scenarios, it is outperformed by indexed SQL but still significantly better than non-indexed SQL. These advantages are achieved at the cost of higher CPU and RAM consumption, which may be acceptable in environments where resources are not the primary constraint.

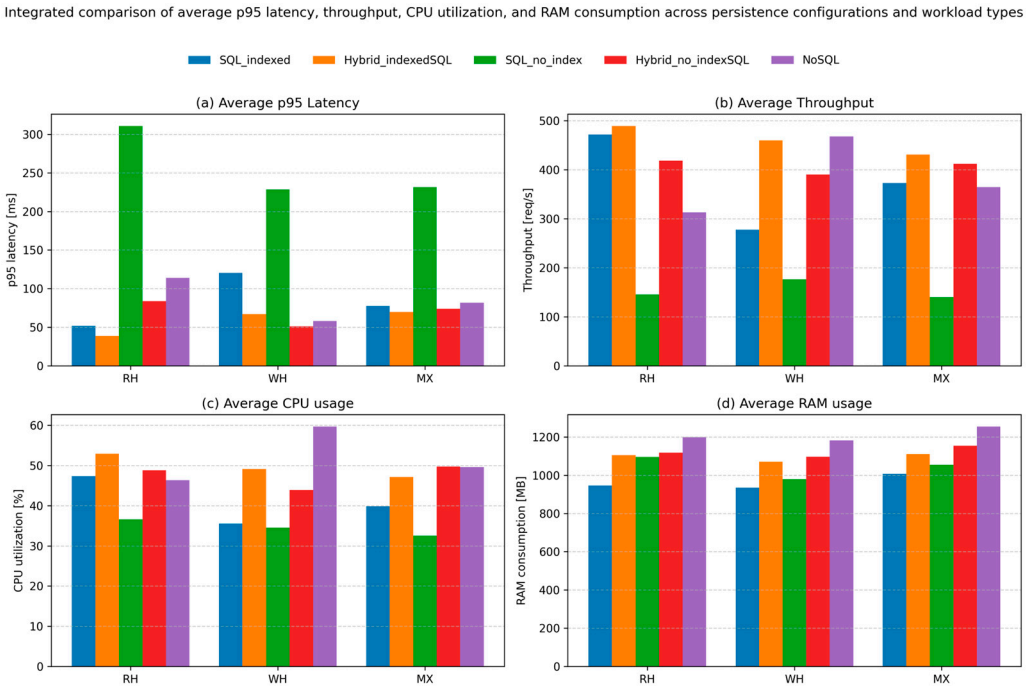


Figure 7. Integrated comparison of average p95 latency, throughput, CPU utilization, and RAM consumption across persistence configurations. Hybrid SQL variants are included to illustrate their relative position with respect to baseline SQL and NoSQL designs under aggregated workload conditions.

Overall, the results confirm that no single persistence configuration is universally optimal. Performance outcomes arise from the interaction between workload composition, concurrency intensity, and internal persistence mechanisms such as indexing strategies, transaction handling, and storage models. The choice of storage technology should be guided by workload profile and resource constraints: indexed SQL is preferable for strongly read-dominated services or when memory efficiency is critical; NoSQL is better suited for write-intensive and high-concurrency environments where resource consumption is acceptable; and non-indexed SQL should be avoided in most practical scenarios except as a controlled baseline for isolating indexing effects. These empirically grounded insights provide a solid basis for the architectural recommendations and design guidelines discussed in the next section. Hybrid SQL configurations consistently position themselves between their corresponding baseline SQL variants and MongoDB, confirming that hybridization refines but does not overturn the dominant workload-driven performance trends.

To ensure traceability between the research objectives, hypotheses, and empirical findings, the following subsection explicitly maps the experimental results to the research questions and hypotheses defined in Section 1.

4.6. Mapping Results to Research Questions and Hypotheses

This subsection explicitly maps the empirical findings presented in Sections 4.1–4.5 to the research questions and hypotheses defined in Section 1. The objective of this mapping is to ensure traceability between the study objectives, experimental design, and observed outcomes, and to clarify how the reported results address the stated research questions and validate (or qualify) the proposed hypotheses.

4.6.1. Mapping to Research Questions

RQ1: How do relational (indexed and non-indexed) and NoSQL persistence models differ in p95 latency and throughput under RH, WH, and MX workloads?

The results demonstrate clear, workload-dependent performance differentiation among the evaluated persistence configurations. Under RH workloads, indexed SQL consistently achieves the lowest p95 latency and the highest throughput across almost all concurrency levels. This behavior reflects the effectiveness of indexing structures in minimizing full table scans and optimizing query execution paths. Non-indexed SQL exhibits substantially higher tail latency and significantly reduced throughput, confirming the critical role of indexing in relational read performance. MongoDB performs better than non-indexed SQL but remains inferior to indexed SQL in both latency and throughput under RH conditions. Under WH workloads, MongoDB clearly outperforms both relational configurations. It achieves the lowest p95 latency and the highest throughput at all concurrency levels, particularly under medium and high contention. Indexed SQL performs worst in this scenario due to index maintenance overhead, while non-indexed SQL improves relative to indexed SQL but remains significantly behind MongoDB. Under MX workloads, indexed SQL and MongoDB exhibit comparable performance. Both achieve low p95 latency and high throughput across increasing concurrency levels, with indexed SQL often maintaining a slight advantage at high concurrency. Non-indexed SQL remains consistently inferior due to inefficient read handling. The results confirm that persistence performance is strongly workload-dependent. Indexed SQL is optimal for read-dominant workloads, MongoDB is superior for write-dominant workloads, and both indexed SQL and MongoDB provide competitive performance under mixed workloads, while non-indexed SQL is not suitable for most practical scenarios.

RQ2: How does increasing concurrency (from 1 to 1000 parallel requests) influence the performance and resource usage of SQL and NoSQL databases?

Increasing concurrency systematically amplifies architectural differences between persistence models. Indexed SQL scales efficiently under RH workloads but exhibits sharp increases in p95 latency and declining throughput under high write concurrency due to index update overhead and transactional contention. Non-indexed SQL degrades rapidly under high read concurrency because full table scans scale poorly as parallelism increases. MongoDB demonstrates comparatively stable p95 latency and sustained throughput as concurrency increases, particularly in WH and MX workloads. Even at extreme concurrency levels (500–1000 parallel requests), MongoDB maintains consistent performance trends, indicating strong robustness under contention. Resource utilization metrics corroborate these observations. MongoDB consistently consumes more CPU and RAM, reflecting aggressive resource utilization to sustain throughput. Indexed SQL achieves competitive performance with lower memory consumption, especially in RH and MX workloads. Non-indexed SQL often appears resource-efficient only because it processes fewer requests effectively, rather than due to superior optimization. Concurrency acts as a stress multiplier that exposes scalability limits in relational configurations while highlighting the robustness of document-oriented persistence under high parallel load.

RQ3: Which persistence configuration provides the most stable behavior across tail latency, throughput, CPU usage, and RAM consumption under mixed workload patterns typical of microservice ecosystems?

Under mixed workloads, MongoDB exhibits the most stable p95 latency and throughput across increasing concurrency levels, demonstrating robustness under competing read and write demands. However, this stability is achieved at the cost of higher CPU and RAM consumption. Indexed SQL shows comparable stability in MX workloads while maintaining a significantly smaller memory footprint. Although its performance degrades under extreme write pressure, it remains highly competitive under balanced access patterns. Non-indexed SQL does not provide stable behavior across any performance dimension and exhibits poor scalability in mixed workloads. Stability depends on the performance

dimension considered. MongoDB offers the most stable latency and throughput under concurrency, while indexed SQL provides a more balanced trade-off between performance stability and resource efficiency. This reinforces the importance of workload-aware persistence selection in microservice ecosystems.

4.6.2. Mapping to Hypotheses

H1. *Indexed SQL achieves significantly lower read latency (p95) than non-indexed SQL and MongoDB under RH workloads due to optimized index access paths.*

Outcome: Supported.

Indexed SQL consistently exhibits the lowest p95 latency and highest throughput under RH workloads. Non-indexed SQL suffers from severe tail-latency degradation due to full table scans, while MongoDB remains consistently slower than indexed SQL for structured read operations.

H2. *MongoDB achieves superior write throughput and lower tail latency than both SQL configurations under WH workloads, especially at high concurrency levels.*

Outcome: Supported.

MongoDB achieves the lowest p95 latency and highest throughput across all concurrency levels in WH workloads. Both relational configurations are penalized by transactional overhead, with indexed SQL performing worst due to index maintenance costs.

H3. *MongoDB provides the most stable performance profile across RH, WH, and MX workloads as concurrency increases, exhibiting more predictable CPU and RAM usage compared to relational configurations.*

Outcome: Partially supported.

MongoDB provides the most stable latency and throughput trends under increasing concurrency, particularly in WH and MX workloads. However, this stability is accompanied by consistently higher CPU and RAM consumption compared to indexed SQL. Indexed SQL offers comparable stability in MX workloads with superior memory efficiency.

Summary of Hypothesis Evaluation

Overall, H1 and H2 are fully supported by the experimental results, while H3 is supported with qualification. MongoDB offers robust performance stability under concurrency, but this comes at the cost of higher resource utilization. These findings emphasize that no single persistence configuration is universally optimal and that performance outcomes emerge from the interaction between workload composition, concurrency intensity, and persistence architecture. Hybrid SQL variants further reinforce these conclusions by demonstrating that the same workload-dependent performance trends persist when relational persistence is combined with hybrid access patterns, indicating that the observed behaviors are robust beyond strictly isolated baseline configurations. This explicit mapping provides a clear bridge between the empirical results and the research objectives, and serves as a foundation for the deeper interpretative discussion presented in the following section.

These results establish a clear empirical link between the stated research questions, hypotheses, and observed system behavior, providing a structured foundation for the interpretative discussion that follows.

5. Discussion

The observed performance differences can be interpreted through the lens of architectural trade-offs inherent to relational and document-oriented persistence models under varying workload and concurrency conditions. These differences align closely with architectural principles while also providing new empirical insights into their behavior under high concurrency.

5.1. Read-Heavy (RH) Behavior

The results confirm that indexing is the decisive factor for read-intensive workloads. SQL with index achieves the lowest p95 latency and the highest throughput across all concurrency levels, while also maintaining relatively low CPU and RAM usage. In contrast, SQL without index performs the worst in every RH metric due to full table scans, which inflate latency and CPU overhead as concurrency increases. MongoDB performs moderately well, better than non-indexed SQL but consistently behind indexed SQL. These observations validate the core relational expectation that properly designed indexes dramatically improve read performance.

5.2. Write-Heavy (WH) Behavior

In WH workloads, the performance hierarchy reverses. MongoDB achieves the lowest p95 latency and highest throughput across almost all concurrency levels, supported by efficient document-level writes and lightweight transactional semantics. SQL without index performs better than indexed SQL because it avoids index-maintenance overhead. Indexed SQL experiences the steepest latency and throughput degradation as concurrency increases, reflecting the cumulative cost of maintaining B-tree indexes under heavy write activity. These results confirm that NoSQL systems provide distinct advantages for write-intensive microservices.

5.3. Mixed (MX) Behavior

MX workloads emphasize the balance between read and write performance. SQL with index and MongoDB exhibit highly similar average p95 latency and throughput, with SQL slightly outperforming MongoDB at the highest concurrency levels. MongoDB, however, maintains more stable resource usage, particularly RAM, across the entire concurrency range. SQL without index again underperforms both alternatives, illustrating that eliminating indexes provides write benefits but severely penalizes read performance in mixed-access scenarios. These results demonstrate that MX workloads expose the most balanced strengths of SQL with index and NoSQL.

5.4. Scalability and Resource Behavior

Concurrency amplifies the inherent architectural characteristics of each system. Indexed SQL scales well for reads but poorly for writes; non-indexed SQL scales poorly for reads but moderately for writes; and MongoDB scales smoothly across all workload types, with predictable increases in CPU and RAM usage. Notably, NoSQL consistently consumes more RAM than both SQL configurations, reflecting memory-mapped storage internals, but converts that memory usage into higher throughput and lower tail latency. CPU patterns similarly align with expectations: indexed SQL is CPU-efficient during reads but CPU-expensive during writes, while MongoDB combines high CPU utilization with superior throughput in WH and MX workloads.

5.5. Cross-Metric Performance Trends and Architectural Implications

Interpreting p95 latency, throughput, CPU utilization, and RAM consumption jointly reveals architectural performance trends that transcend individual metrics and clarify the underlying trade-offs among the evaluated persistence configurations. Rather than being driven by a single dominant factor, observed performance outcomes emerge from the interaction between workload composition, concurrency intensity, and the internal mechanisms of each persistence engine. The behavior of hybrid SQL configurations further supports these interpretations. Hybrid Indexed SQL consistently improves both latency and throughput relative to its non-hybrid counterpart, particularly in write-heavy and mixed

workloads, suggesting that partial decoupling of persistence access paths can mitigate some of the traditional indexing overheads. Importantly, these improvements do not alter the fundamental workload-dependent hierarchy observed among SQL and NoSQL systems, but rather confirm their stability under extended architectural variations.

5.5.1. Impact of Workload Type on Persistence Selection

The experimental results clearly indicate that workload type is the dominant factor shaping optimal persistence choice in microservice-based systems.

Indexed SQL consistently proves optimal for read-heavy workloads, delivering the lowest p95 latency and the highest throughput while maintaining the smallest memory footprint. This behavior reflects the effectiveness of indexing structures in optimizing query execution and minimizing unnecessary data access. In contrast, NoSQL systems, while competitive, remain inferior to indexed SQL for structured, read-dominant access patterns.

Under write-heavy workloads, NoSQL demonstrates the strongest performance characteristics. It achieves superior throughput and lower tail latency, particularly under high concurrency, confirming the advantages of document-oriented storage and relaxed consistency models for write-intensive scenarios. SQL without index performs better than indexed SQL in these cases, but remains clearly behind NoSQL.

For mixed workloads, which closely resemble realistic microservice access patterns, NoSQL remains highly competitive and indexed SQL often achieves comparable or slightly superior latency and throughput, especially at higher concurrency levels. SQL without index, however, remains significantly penalized due to inefficient read handling.

5.5.2. Concurrency as an Amplifier of Architectural Strengths and Weaknesses

Concurrency acts as a critical stress factor that amplifies the inherent architectural characteristics of each persistence model.

Indexed SQL exhibits sharp performance degradation under high write concurrency, primarily due to index-maintenance overhead and transactional contention. Conversely, SQL without index collapses under high read concurrency as full table scans scale poorly with increasing parallelism.

MongoDB scales more gracefully across all workload types, maintaining stable performance even at extreme concurrency levels (500–1000 concurrent requests). This robustness under contention highlights the suitability of document-oriented persistence for high-parallelism environments, albeit at increased resource cost.

5.5.3. Resource Utilization as a Complementary Performance Indicator

Resource utilization patterns corroborate and contextualize the observed latency and throughput results.

Indexed SQL maintains low CPU usage in read-heavy scenarios, but incurs substantial CPU and RAM overhead under intensive write activity. SQL without index exhibits high CPU usage during read-heavy workloads due to inefficient scans, while showing moderate resource consumption during write-dominated access patterns.

MongoDB demonstrates predictable CPU and RAM utilization across workloads, but consistently allocates more memory than relational configurations. This behavior reflects its memory-mapped storage model and aggressive caching strategies, which enable stable performance at the expense of higher resource consumption.

These findings indicate that lower resource utilization does not necessarily correspond to better performance; rather, effective conversion of resources into throughput and low tail latency is the decisive factor.

5.5.4. Tail Latency as a Critical Indicator Under Stress

Across all experiments, tail latency (p95) emerges as the most sensitive indicator of system behavior under stress.

Indexed SQL exhibits steep p95 increases in write-heavy scenarios, particularly beyond 100 concurrent clients, reflecting index update overhead. SQL without index shows the greatest p95 degradation in read-heavy workloads due to full table scans. MongoDB maintains the most stable p95 latency in mixed and write-heavy workloads, even under extreme concurrency, reinforcing its robustness under contention.

5.5.5. Implications for Persistence Design in Microservice Architectures

Indexed SQL is best suited for read-intensive microservices or environments where memory efficiency is a priority. SQL without index offers limited benefits only in narrowly defined write-dominated scenarios and should generally be avoided in practical deployments. MongoDB emerges as the most robust general-purpose solution, delivering strong write performance, competitive mixed-workload behavior, and stable performance under high concurrency, albeit at the cost of increased CPU and RAM consumption.

These discussion-level insights provide a natural bridge from the empirical results to the high-level conclusions and architectural recommendations presented in the following section.

5.6. Implications for Microservice Architectures

The findings have direct architectural implications. Read-oriented microservices (e.g., configuration lookup, authentication, catalog services) benefit most from indexed relational storage. Write-intensive services (e.g., audit logs, telemetry ingestion, event streams) are better served by MongoDB. Mixed workloads, common in transactional microservices, perform well with either indexed SQL or MongoDB, depending on whether the system prioritizes memory efficiency (favor SQL) or balanced scaling under concurrency (favor MongoDB). These results empirically support the adoption of polyglot persistence in distributed systems where workload heterogeneity is the norm.

5.7. Summary of Key Insights

The experimental findings discussed in this section highlight that persistence performance in microservice-based systems is fundamentally shaped by the interaction between workload composition, concurrency intensity, and database architecture. Across all evaluated scenarios, no single persistence configuration consistently outperforms the others, confirming that workload-aware selection is essential for achieving balanced system behavior.

Indexed SQL demonstrates clear advantages in read-heavy workloads, where it achieves superior tail latency and throughput while maintaining efficient CPU and memory utilization. These characteristics make it particularly suitable for microservices dominated by structured read access patterns. In contrast, MongoDB exhibits the strongest performance under write-heavy workloads and remains highly competitive in mixed scenarios, especially as concurrency increases. Its ability to sustain low p95 latency and high throughput under contention reflects the benefits of document-oriented storage and relaxed consistency mechanisms, albeit at the cost of increased CPU and RAM consumption.

SQL without indexing consistently underperforms in read-heavy and mixed workloads due to inefficient access paths and poor scalability under concurrency. While it offers limited benefits in narrowly defined write-dominated scenarios, its overall performance profile restricts its applicability in realistic microservice deployments.

Across all configurations, tail latency (p95) emerges as the most sensitive indicator of system behavior under stress, revealing scalability limits and degradation patterns more clearly than average-based metrics. Together, these insights provide a coherent empirical

foundation for understanding persistence trade-offs in microservice architectures and set the stage for the high-level conclusions and design recommendations presented in the following section.

6. Conclusions

This study presented a systematic empirical evaluation of three persistence configurations—SQL with indexing, SQL without indexing, and MongoDB—under RH, Wh, and MX workloads, across concurrency levels ranging from 1 to 1000 parallel requests. By jointly analyzing p95 latency, throughput, CPU utilization, and RAM consumption, the study provides a comprehensive and internally consistent characterization of persistence-layer behavior in a controlled microservice-based environment.

The results confirm that no single persistence mechanism is universally optimal. Indexed SQL is the most effective choice for RH workloads, offering the lowest tail latency and highest throughput with modest resource requirements. SQL without indexing provides acceptable performance only in narrowly defined write-dominated scenarios and scales poorly otherwise. MongoDB exhibits the most balanced performance profile, performing best under WH workloads and remaining competitive under mixed access patterns, while trading higher memory consumption for stability under high concurrency.

These findings reinforce established principles from database systems theory while providing new quantitative insights tailored to containerized microservice architectures. From a practical standpoint, the results strongly support the adoption of polyglot persistence strategies, in which microservices select storage technologies according to their dominant workload characteristics. Relational databases with appropriate indexing are well suited for read-oriented services, document-oriented NoSQL systems are advantageous for write-intensive and high-concurrency workloads, and hybrid approaches are appropriate for complex systems with heterogeneous data access patterns. Supplementary results obtained from hybrid SQL configurations further confirm that the reported workload-dependent performance trends remain stable under extended persistence designs, reinforcing the robustness—rather than expanding the scope—of the primary findings.

Future research may extend this work to distributed and production-oriented settings, including replicated and sharded deployments, distributed transaction management, caching layers, additional NoSQL engines, and cloud-native autoscaling environments. Nevertheless, the present study offers actionable guidance: aligning persistence technologies with dominant workload profiles leads to measurable improvements in performance, scalability, and resource efficiency in modern microservice ecosystems.

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

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

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

References

1. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Springer International Publishing: Cham, Switzerland, 2017. [CrossRef]

2. Fowler, M.; Lewis, J. Microservices: A Definition of This New Architectural Term. Available online: https://martinfowler.com/articles/microservices.html (accessed on 30 October 2025).
3. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to Cloud-Native Architectures. *IEEE Softw.* **2016**, *33*, 42–52. [CrossRef]
4. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. In Proceedings of the 2015 10th Computing Colombian Conference (10CCC), Bogotá, Colombia, 23–25 September 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 583–590.
5. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, Omega, and Kubernetes. *Commun. ACM* **2016**, *59*, 50–57. [CrossRef]
6. Taibi, D.; Lenarduzzi, V.; Pahl, C. Microservices Anti-Patterns: A Taxonomy. In *Microservices*; Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M., Eds.; Springer: Cham, Switzerland, 2020; pp. 111–128. [CrossRef]
7. dos Santos, O.P.N. Towards an Automatic Microservices Manager for Hybrid Cloud Edge Environments. Master's Thesis, NOVA University Lisbon, Lisbon, Portugal, 2022.
8. Deka, G.C. NoSQL Polyglot Persistence. *Adv. Comput.* **2018**, *109*, 357–390. [CrossRef]
9. Cattell, R. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Rec.* **2021**, *39*, 12–27. [CrossRef]
10. Pokorný, J. NoSQL Databases: A Step to Database Scalability in Web and Cloud Applications. In Proceedings of the 13th International Conference on Information Integration and Web-based Applications & Services, Ho Chi Minh City, Vietnam, 5–7 December 2011. [CrossRef]
11. Khan, W.; Kumar, T.; Zhang, C.; Raj, K.; Roy, A.M.; Luo, B. SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review. *Big Data Cogn. Comput.* **2023**, *7*, 97. [CrossRef]
12. James, B.E.; Asagba, P.O. Hybrid Database System for Big Data Storage and Management. *Int. J. Comput. Sci. Eng. Appl.* **2017**, *7*, 13–25. [CrossRef]
13. Bjeladinović, S. A Fresh Approach for Hybrid SQL/NoSQL Database Design Based on Data Structuredness. *Enterp. Inf. Syst.* **2018**, *13*, 1202–1220. [CrossRef]
14. Oussous, A.; Benjelloun, F.-Z.; Lahcen, A.A.; Belfkih, S. Big Data Technologies: A Survey. *J. King Saud Univ.-Comput. Inf. Sci.* **2018**, *30*, 431–448. [CrossRef]
15. Vogels, W. Eventually Consistent. *Commun. ACM* **2009**, *52*, 40–44. [CrossRef]
16. AbouShanab, Z. Optimizing Containerized Spring Boot Microservices in Kubernetes: Development, Experimentation, and Performance Analysis. Master's Thesis, Eötvös Loránd University, Budapest, Hungary, 2024.
17. Costa, J.G.; Gonçalves, L.M.G.; Xavier-de-Souza, S. Multitenancy in Single Instance of Data Persistence Service for Supporting Low-Code Platforms. *IEEE Access* **2025**, *13*, 13427–13439. [CrossRef]

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