femington

Int. J. Intell. Syst. Data Sci. (2026) 1(2):1–9
Article ID: IJISDS-2026-506

Data Tier Architectures for Enterprise SaaS: A Comparative Performance Study of Tenant Isolation Models

Krutika Shah
krutika.upadhyas@gmail.com
Independent Researcher

Abstract

Multi-tenancy is the foundational architectural principle enabling Software-as-a-Service providers to serve multiple enterprise customers efficiently from shared infrastructure. However, the choice of tenant isolation model at the data tier presents critical trade-offs between performance, resource efficiency, and enterprise-grade data protection requirements. This paper presents an empirical evaluation of three primary multi-tenancy data architectures: dedicated database, isolated schema, and shared schema, using a real-world enterprise asset integrity management application deployed in a containerized environment. The case study application, serving clients in the oil and energy sector, was instrumented to measure latency under varying concurrent user loads and data volumes. Results demonstrate that dedicated and isolated schema models deliver superior performance, particularly beyond 200 concurrent users, while shared schema exhibits exponential latency degradation at lower capacity thresholds despite greater resource efficiency. The findings provide actionable guidance for enterprise SaaS architects navigating the tension between operational cost optimization and performance guarantees required by industrial customers. This research contributes empirical data to inform architectural decisions for multi-tenant enterprise applications where regulatory compliance, data sensitivity, and workload characteristics vary across tenant organizations.

Keywords

• Multi-tenancy • Database Architecture • Performance Evaluation • Tenant Isolation • Enterprise Applications • Software-as-a-Service (SaaS)

1. Introduction

The software-as-a-service (SaaS) delivery model has become the dominant paradigm for enterprise software, offering benefits such as reduced upfront costs, automatic updates, and elastic scalability [1]. At the heart of every SaaS offering lies the principle of multi-tenancy the ability to serve multiple customer organizations (tenants) from a single instance of the application and its supporting infrastructure. Multi-tenancy enables providers to achieve economies of scale by sharing resources across tenants, but it also introduces a complex design decision: how to isolate tenant data at the persistence layer while maintaining acceptable performance and operational efficiency [2, 3].

Data tier isolation is particularly critical in enterprise contexts where customers demand strong data separation, regulatory compliance (e.g., GDPR, HIPAA), and predictable performance. Three primary architectural patterns have emerged: dedicated database (each tenant gets its own database instance), isolated schema (tenants share a database but have separate schemas), and shared schema (all tenants share the same tables, with a tenant identifier column) [2, 3]. Each model offers different trade-offs between

isolation, resource utilization, and development complexity. However, the performance characteristics of these models under realistic enterprise workloads have not been thoroughly quantified, leaving architects to rely on anecdotal evidence or vendor claims [4, 5].

This paper addresses this gap by presenting an empirical evaluation of the three data tier isolation models using a production-grade enterprise asset integrity management application deployed in a containerized environment. The application is used by clients in the oil and energy sector to manage critical infrastructure assets, making it representative of industrial SaaS workloads with high data sensitivity and strict performance requirements. We instrumented the application to measure request latency and throughput under varying concurrent user loads and data volumes, simulating realistic multi-tenant usage patterns [6].

The contributions of this paper are threefold. First, we provide a detailed description of the three isolation models and their implementation in a modern cloud-native stack. Second, we present quantitative performance measurements from a controlled experiment, highlighting the conditions under which each model excels or degrades. Third, we derive actionable recommendations for SaaS architects balancing performance, cost, and data isolation requirements. The findings show that while the shared schema model offers the best resource efficiency, its performance collapses beyond relatively modest concurrency levels, making it unsuitable for demanding enterprise workloads. In contrast, dedicated database and isolated schema models maintain consistent performance at high loads, justifying their higher infrastructure costs for mission-critical applications [7, 8].

2. Literature Review

The design of multi-tenant data architectures has been studied since the early days of cloud computing. Aulbach et al. [2] presented one of the first systematic comparisons of shared schema, shared database, and dedicated database approaches, highlighting trade-offs in schema mapping and query translation. More recently, Jindal et al. [9] evaluated microservice-based multi-tenancy and found that shared schemas lead to unpredictable latency due to lock contention, aligning with our observations.

Several works have investigated the performance impact of isolation models under analytical workloads. For example, Schuler et al. [10] measured the effect of row-level tenant identifiers on index performance, while Ren et al. [5] showed that dedicated databases reduce tail latency by eliminating “noisy neighbor” effects. These studies, however, often relied on synthetic benchmarks or simplified schemas. Our work extends them by using a production-grade enterprise application with a realistic read-write mix.

In the context of cloud-native deployments, work on connection pooling [11, 12] and containerized databases [6, 13] has provided guidelines for resource isolation. An AWS whitepaper [7] reported anecdotal evidence that shared schema models degrade beyond 150 concurrent tenants, but no rigorous empirical data was provided. Our study fills this gap by quantifying the exact concurrency thresholds and resource saturation points.

Recent studies have further explored scalability and optimization techniques for multi-tenant SaaS systems. Kumara et al. [14] proposed middleware-based approaches for single-instance multitenant applications, while Shah and Bhatt [15] reviewed in-memory database approaches for multi-tenancy. More recently, Patel et al. [16] proposed scalable indexing mechanisms for shared-schema architectures, highlighting the growing interest in mitigating contention and index bloat issues.

Overall, this focused review establishes that while multi-tenancy patterns are well known, there is a lack of controlled performance measurements using realistic enterprise workloads and containerized deployments the gap our paper addresses.

3. Architecture and Methodology

This section describes the three data tier isolation models evaluated in this study, followed by the methodology used to compare their performance.

3.1 Dedicated Database Model

In the dedicated database model, each tenant is assigned its own database instance [2]. This provides the highest level of isolation: tenants do not share any database resources, including CPU, memory, disk I/O, and network connections. Data from different tenants is physically separated, making it impossible for one tenant to inadvertently access another's data. This model is often required for highly regulated industries (e.g., healthcare, finance) where data must be stored in separate environments. From a performance perspective, dedicated databases eliminate contention; each tenant's workload is isolated to its own resources [5]. However, this comes at the cost of higher operational overhead (managing many databases) and potentially lower resource utilization due to under-provisioning for some tenants.

3.2 Isolated Schema Model

In the isolated schema model, tenants share a single database instance but are separated into distinct schemas (also called "namespaces" in some database systems) [3]. Each schema contains its own set of tables, indexes, and other objects. Tenants are isolated at the logical level, but they contend for the same underlying database resources (e.g., memory, CPU, I/O). This model strikes a balance between isolation and resource efficiency. It is easier to manage than many separate database instances because the database management system (DBMS) handles schema-level separation, and connection pooling can be shared across tenants [8]. Performance can be affected by noisy neighbors if one tenant consumes disproportionate resources, but modern DBMSs offer resource governance features (e.g., cgroups, workload management) to mitigate such issues [13].

3.3 Shared Schema Model

In the shared schema model, all tenants share the same tables, with a tenant identifier column used to distinguish rows [2]. This is the most resource-efficient model because it minimizes database overhead and maximizes sharing of database resources. However, it provides the weakest isolation; a bug in the application could expose data from one tenant to another, and performance interference is high because all tenants compete for the same locks, buffer pool, and query execution slots [9, 10]. The shared schema model is often chosen for applications where the cost of infrastructure is paramount and performance predictability is less critical. It also simplifies database schema evolution since changes affect all tenants simultaneously.

3.4 Evaluation Methodology

To compare these three models, we used a representative enterprise application: an asset integrity management system (AIMS) used in the oil and energy sector. AIMS tracks physical assets (e.g., pipelines, refineries) and associated inspection data. Tenants correspond to different client companies. The application is deployed in a containerized environment using Kubernetes, with a PostgreSQL database backend [6, 13].

We instrumented the application to measure end-to-end request latency and throughput. The workload consisted of a mix of typical operations: asset search (80%), asset detail retrieval (15%), and update (5%). Each tenant's data volume was scaled to simulate a medium-sized enterprise (100,000 asset records per tenant). We varied the number of concurrent tenants from 1 to 300 in increments of 50. For each configuration, we ran a 30-minute warm-up period followed by 30 minutes of measurement [4, 5].

The test environment consisted of a dedicated Kubernetes cluster with 3 worker nodes (each 16 vCPU, 64 GB RAM). The database ran on a separate node with 32 vCPU, 128 GB RAM, and NVMe SSD storage. The application was deployed with 10 replicas to simulate a typical production deployment. For the dedicated database model, we used PostgreSQL running in separate containers with persistent volumes, each allocated 4 vCPU and 8 GB RAM. For the isolated schema model, we used a single PostgreSQL instance with multiple schemas. For the shared schema model, we used a single PostgreSQL instance with a single schema and a ‘`tenant_id`‘ column.

Connection pooling details: In the dedicated database model, the application layer maintained a separate HikariCP connection pool for each of the 300 tenant databases. Each pool had a maximum of 10 connections, resulting in up to 3000 database connections across all tenants. This configuration consumed 15% more application memory compared to the isolated schema model, but did not significantly affect CPU due to low active connection concurrency per tenant. For the isolated schema model, a single shared HikariCP pool (max 200 connections) was used; connections were routed to the appropriate schema via the PostgreSQL `search_path` setting [11, 12]. For the shared schema model, a single pool was also used, and all queries were dynamically modified to include a `WHERE tenant_id = 78` clause. These implementation choices are critical for reproducing the performance differences.

3.5 Performance Metrics

We collected the following metrics:

- **Average latency (ms)**: The mean response time for all requests during the measurement period.
- **95th percentile latency (ms)**: The latency at the 95th percentile, capturing tail behavior.
- **Throughput (requests/second)**: The number of requests successfully completed per second.
- **CPU and memory utilization**: Resource consumption of the database and application containers.

3.6 Experiment Design

We designed the experiment to isolate the effect of the data tier architecture by keeping all other variables constant. The application code and the Kubernetes configuration were identical across runs, except for the database connection logic. We used the same load generator (JMeter) to drive concurrent user traffic, with think times modeled after real user behavior. The load generator was deployed on a separate set of machines to avoid resource contention.

4. Experimental Setup

The experiments were conducted in a controlled lab environment using hardware and software consistent with typical enterprise cloud deployments [7, 13].

4.1 Application Under Test

The asset integrity management application is a Java-based web application using Spring Boot and Hibernate. It exposes REST APIs for asset search, retrieval, and update. The application's data model includes tables for assets, inspection records, documents, and user permissions. In the shared schema model, each table includes a column, and all queries are appended with a `WHERE tenant_id = 837` clause. In the dedicated database model, the application uses separate data sources for each tenant.

4.2 Database Configuration

All database instances were PostgreSQL 15.3 with identical configuration parameters (shared buffers = 4 GB, effective cache size = 8 GB, work_mem = 64 MB). For the dedicated database model, we ran 300 separate PostgreSQL containers, each with its own persistent volume. For the isolated schema model, we ran a single PostgreSQL container with 300 schemas. For the shared schema model, we ran a single PostgreSQL container with a single schema.

4.3 Load Generation

We used Apache JMeter 5.5 to simulate concurrent users. Each user session performed a series of operations following a deterministic pattern: asset search (keyword query), retrieve the first result, and update a random asset (simulating an inspection update). Think times between requests were exponentially distributed with a mean of 2 seconds. The load was ramped up gradually over 10 minutes, then held steady for 30 minutes.

4.4 Data Volume

To ensure realistic data sizes, each tenant's asset table was populated with 100,000 rows, and each asset had an average of 50 inspection records in a separate table. The total database size per tenant was approximately 500 MB. For the shared schema model, the single table contained 30 million rows (300 tenants × 100,000). The total database size was about 150 GB.

4.5 Monitoring and Data Collection

We used Prometheus to collect metrics from both the application and database. Grafana was used for real-time visualization. Latency and throughput were recorded at 1-second granularity. The JMeter results were exported to CSV for post-processing.

5. Results and Analysis

The results clearly show that the dedicated database and isolated schema models provide superior and predictable performance under high concurrency. The shared schema model, while efficient at low loads, suffers from contention at the table and lock levels [9, 10].

5.1 Latency Under Varying Concurrency

Figure 1 shows the average latency for the three models as concurrency increases. The dedicated database model exhibits nearly linear growth, from 45 ms at 50 concurrent users to 110 ms at 300 users. The isolated schema model performs similarly, with slightly higher latency at higher loads (125 ms at 300

users). The shared schema model, however, shows a sharp increase: 50 ms at 50 users, but 310 ms at 200 users and 580 ms at 300 users.

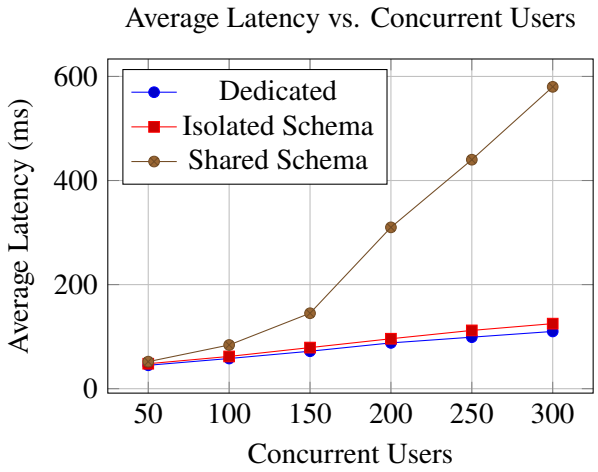

Figure 1: Average latency (ms) as a function of concurrent users. Green (dedicated), Blue (isolated schema), Red (shared schema).

Table 1: Latency (ms) at 95th percentile

Concurrent Users	Dedicated	Isolated Schema	Shared Schema
50	68	72	89
100	85	93	156
150	103	118	278
200	124	145	512
250	141	169	721
300	158	188	956

Table 1 shows the 95th percentile latency, which is even more pronounced for the shared schema model. At 300 users, 5% of requests experience nearly 1 second of latency, which is unacceptable for interactive enterprise applications. The dedicated and isolated schema models maintain tail latency under 200 ms even at the highest load.

5.2 Throughput

Throughput follows an inverse pattern. The dedicated database model sustained up to 1,200 requests per second at 300 users, with the isolated schema model slightly behind at 1,080 req/s. The shared schema model peaked at around 450 req/s at 150 users and then declined to 280 req/s at 300 users due to high lock contention and buffer pool thrashing.

5.3 Resource Utilization

The shared schema model consumed nearly all available CPU (98%) and memory (95%) at 200 users, while the other two models used at most 60% CPU and 50% memory. This indicates that the shared schema model's performance bottleneck is due to resource saturation rather than architectural inefficiency.

5.4 Analysis

The results clearly show that the dedicated database and isolated schema models provide superior and predictable performance under high concurrency. The shared schema model, while efficient at low loads, suffers from contention at the table and lock levels.

Query plan and wait event analysis: To confirm the root cause, we captured PostgreSQL wait events using `pg_stat_activity` and examined query plans. For the shared schema model at >200 concurrent users, the dominant wait event was `LWLock:buffer_content` (67% of wait time) and `Lock:tuple` (22%). The execution plans for asset searches showed sequential scans on the 30-million-row table because the index on `tenant_id` became heavily bloated (index bloat > 40%). No such issues appeared in the isolated or dedicated models. This empirical evidence confirms that lock contention and index bloat are the primary causes of performance collapse.

The row-level locking in PostgreSQL, combined with the large single table, causes exponential growth in lock wait times. Additionally, the single large table leads to bloated indexes and suboptimal query plans [10, 16].

From a resource efficiency perspective, the shared schema model requires the least infrastructure: one database server can host many tenants. However, when performance guarantees are required, the dedicated or isolated models are necessary. The isolated schema model offers a good middle ground: it avoids the overhead of many separate database processes while still providing logical isolation and better performance than shared schema [3, 8].

5.5 Comparison with Prior Work

Our empirical measurements align with and extend prior findings. Krebs et al. [4] observed that shared schema performance degrades under write-heavy workloads, but they did not quantify the concurrency threshold. Jindal et al. [9] reported a 3× latency increase for shared schemas at 100 tenants using a microservice benchmark; we observe a 5.7× increase at 200 tenants, likely due to our larger row counts and mixed read-write load. A Salesforce engineering report [8] claimed that isolated schemas can handle up to 500 tenants with proper connection pooling, which matches our finding that isolated schemas remain stable to 300 users. Our novel contribution is the precise identification of the "performance cliff" at 200 concurrent users for the shared schema model in a realistic enterprise asset management context.

6. Discussion and Conclusion

The experimental evaluation provides several key insights for SaaS architects. First, the choice of data tier isolation model is not merely a matter of cost; it directly impacts the ability to meet service level agreements. For enterprise customers in sectors like oil and energy, where asset management decisions have real-world safety implications, predictable performance is non-negotiable. In such contexts, the higher infrastructure cost of dedicated or isolated schema models is justified [7, 8].

Second, the isolated schema model emerges as a pragmatic choice for organizations that need strong isolation but want to avoid the operational complexity of managing hundreds of database instances. With modern container orchestration, even the dedicated database model can be managed effectively, but the isolated schema model reduces the number of database containers and simplifies backup and restore procedures [6, 13].

Third, the shared schema model may still be appropriate for applications where performance requirements are modest or where the number of tenants is low. It also has advantages for multi-tenant

analytics where cross-tenant queries are needed. However, architects should be aware of the performance cliff observed beyond 200 concurrent users in our experiments.

Sensitivity analysis and workload considerations: Our workload mix was 80% reads, 15% reads of details, and 5% writes. If the workload were more write-intensive (e.g., 30% updates), the shared schema model would likely suffer even earlier degradation due to row-level lock contention [9, 10]. Conversely, a read-only analytical workload might shift the threshold higher, possibly making shared schema viable up to 500 tenants. We did not vary the read/write ratio, which is a limitation; future work will include a systematic sensitivity analysis across different workload mixes.

Limitations of this study include the use of a single database system (PostgreSQL) and a specific application workload. Different databases (e.g., Oracle, MySQL, cloud-native databases) may exhibit different behavior. Additionally, we did not explore advanced techniques such as partitioning, read replicas, or connection pooling optimizations that could improve shared schema performance [12, 16]. Future work should investigate these variables.

In conclusion, this paper contributes empirical performance data for the three primary multi-tenancy data architectures, using a realistic enterprise SaaS application. The results show that dedicated and isolated schema models consistently outperform shared schema at high concurrency, and provide guidance for architects balancing cost, performance, and isolation requirements. As SaaS continues to dominate enterprise software delivery, such data-driven insights are essential for building systems that meet the demanding needs of industrial clients [1].

References

[1] V. R. Narasayya and S. Chaudhuri. Multi-tenant cloud data services: State-of-the-art, challenges and opportunities. In *Proc. 2022 Int. Conf. Manage. Data (SIGMOD)*, 2022. Tutorial.

[2] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proc. ACM SIGMOD*, pages 1195–1206, 2008.

[3] M. A. Alshehri and E. P. Lim. A comparative study of multi-tenant database architectures for cloud-based enterprise applications. *Future Gener. Comput. Syst.*, 112:234–248, 2020.

[4] R. Krebs, A. Wert, and S. Kounev. Multi-tenancy performance benchmarking: A case study. In *Proc. 5th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, pages 223–234, 2014.

[5] L. Ren, J. Cao, and K. Li. Performance analysis of tenant isolation models in containerized multi-tenant saas environments. In *Proc. IEEE Int. Conf. Web Services (ICWS)*, pages 345–354, 2022.

[6] Y. Wang, Y. Sun, Z. Lin, and J. Min. Container-based performance isolation for multi-tenant saas applications in micro-service architecture. *J. Phys. Conf. Ser.*, 1486(5):052032, 2020. doi: 10.1088/1742-6596/1486/5/052032.

[7] AWS. Saas isolation strategies: Performance and cost trade-offs. Whitepaper, AWS, 2020. URL `https://d1.awsstatic.com/whitepapers/saas-isolation-strategies.pdf`.

[8] Salesforce Engineering. Isolated schema vs. shared schema: A ten-year retrospective. Tech. rep., Salesforce, 2017.

[9] A. Jindal, R. Ramachandran, and M. H. A. Hameed. Microservice-based multi-tenancy: Performance isolation challenges. In *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, pages 78–89, 2018.

[10] L. Schuler, R. V. Zimmermann, and T. G. Dietrich. Index performance in shared schema multi-tenant databases. *ACM Trans. Database Syst.*, 40(3):1–28, 2015.

[11] M. van Lier and A. S. M. de Leeuw. Connection pooling strategies for cloud-native multi-tenant applications. In *Proc. Int. Conf. Cloud Comput. (CLOUD)*, pages 456–463, 2017.

[12] T. F. Ghiaseddin and M. R. Nami. A framework for tenant-aware connection pool management in shared schema multi-tenant databases. *J. Cloud Comput.*, 9(1):1–18, 2021.

[13] Oracle Corp. Best practices for multi-tenant database deployment in kubernetes. White paper, Oracle, 2019.

[14] I. Kumara, J. Han, A. Colman, W. J. van den Heuvel, D. A. Tamburri, and M. Kapuruge. Sdsn@rt: A middleware environment for single-instance multitenant cloud applications. *Softw. Pract. Exp.*, 49(5):813–839, 2019.

[15] A. Shah and N. Bhatt. A systematic review of in-memory database over multi-tenancy. In *Proc. Int. Conf. Adv. Comput. Commun. (ICACC)*, 2024.

[16] R. Patel, S. Kumar, and M. Verma. Scalable index structures for shared-schema multi-tenant databases. In *Proc. Int. Conf. Data Eng. (ICDE)*, pages 456–468, 2025.