Benchmarking Multi-Tenant Architectures in PostgreSQL

Patrick K. Erdelt
Berliner Hochschule fuer Technik (BHT)
Berlin, Germany
patrick.erdelt@bht-berlin.de

Tilmann Rabl
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tilmann.rabl@hpi.de

Abstract

Multi-tenancy is an important architectural pattern in cloud-native systems, enabling efficient resource sharing while maintaining isolation between tenants. Multi-tenancy can be implemented on many different levels of modern cloud deployments. Mature database management systems such as PostgreSQL supports different multi-tenancy models, including schema-per-tenant, database-per-tenant, and, in containerized environments, container-per-tenant.

In this paper, we present a comparative study of these multi-tenancy strategies, focusing on performance evaluation and resource efficiency. We evaluate each model in PostgreSQL using industry-standard TPC-C (transactional) and TPC-H (analytical) benchmarks deployed in a dockerized microservice architecture orchestrated by Kubernetes, with each tenant itself encapsulated in a dedicated container. The results highlight the trade-offs among strict isolation, performance, and resource efficiency. For CPU-intensive workloads such as TPC-C, a container-per-tenant approach is the most beneficial. In contrast, for memory-intensive workloads like TPC-H, it is the least efficient.

Keywords

Multi-Tenant, PostgreSQL, Database Management Systems, Performance Evaluation, TPC-C, TPC-H, Benchmarking

1 Introduction

Modern CPUs have increasing numbers of cores and can serve increasingly large workloads. While some database deployments are so large that they require and can fully utilize the increasing compute power, most use cases are smaller and do not utilize complete servers. As a solution, cloud providers host many customers in small VMs on individual servers. While this is sensible for highly diverging use cases and to provide isolation between customers, it results in replication of application code, if many identical VMs are deployed. For database applications, this might also mean a replication of large amounts of static information stored in the database. Mature database management systems already feature use management and isolation, which can also be used for customer isolation on a different level, if the database system is used in a multi-tenant mode.

There exist multiple architectures for multi-tenant applications, primarily distinguished by the degree of tenant isolation. Shared-schema architectures, adopted by SaaS platforms such as Shopify and Salesforce, optimize resource utilization and reduce operational overhead [11, 17]. These systems typically rely on horizontal partitioning (sharding) and metadata-driven customization. Shopify, for example, uses MySQL with Vitess for sharding, where each cluster node hosts multiple shards serving multiple tenants. CockroachDB Serverless implements multi-tenancy via a shared key–value store, with tenants separated by prefixes in the keyspace, and each tenant is served by zero or more compute engine instances deployed as containers in Kubernetes [19]. Such designs combine characteristics of database-per-tenant and instance-per-tenant models, forming a hybrid approach.

Database-per-tenant and instance-per-tenant architectures are explicitly supported by Oracle Multitenant, Google Cloud Spanner, Microsoft Azure SQL Database, and Amazon Aurora [1, 9, 12, 14]. These models provide strong isolation by provisioning each tenant with a dedicated logical database or compute instance. They enable fine-grained control over performance, security, and lifecycle management at the cost of increased resource overhead. Instance-level isolation can be achieved using dedicated virtual machines, with provider-specific implementations. Kubernetes has become a common foundation for container-based isolation across applications. For databases, container-per-tenant offers a lightweight alternative to VM-based isolation, but its performance implications under multi-tenant workloads remain unclear. This makes container-per-tenant particularly interesting to compare against database-per-tenant in terms of performance trade-offs. Private-schema architectures allow tenants to maintain isolated schemas within a shared database instance. This approach is common in smaller-scale systems, such as Rails-based SaaS applications on PostgreSQL [16]. However, the performance implications of schema-per-tenant compared to database-per-tenant remain largely unexplored.

Commercial cloud providers offer PostgreSQL as a fully managed service, automating provisioning, backups, patching, scaling, and monitoring. AWS RDS for PostgreSQL, Azure Database for PostgreSQL, and Google Cloud SQL for PostgreSQL run the standard PostgreSQL engine on network-attached storage tied to individual instances. In contrast, AWS Aurora and Google AlloyDB are cloud-native, PostgreSQL-compatible systems with distributed storage decoupled from compute. Tenant isolation is primarily achieved through instance- or cluster-level separation, complemented by network, storage, and access-control mechanisms, while schema-level multi-tenancy remains optional.

Depending on the workload, different aspects of a DBMS become critical and can strongly affect performance, making optimization challenging. Transactional performance depends on the interplay between storage, computation, and coordination. Durability requirements make write throughput sensitive to storage performance, CPU resources are required for transaction processing, and concurrency control introduces synchronization overheads that may limit scalability in the presence of contention. In contrast, analytical workloads rely heavily on the ability to parallelize queries and on available RAM for large in-memory operations. Previous studies have primarily focused on analytical workloads [10, 13, 20, 23] or microbenchmarks such as YCSB [7, 8]. To the best of our knowledge, we are the first to evaluate both transactional and analytical workloads while comparing three multi-tenancy models in PostgreSQL. We systematically explore overheads across different levels, including durability settings, storage types (local disk, distributed storage, and fast RAM disk), and CPU policies. Our evaluation considers multiple

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.46

parallel tenants in a single instance, multiple parallel instances, and fully isolated instances in separate containers, analyzing the resulting performance and overheads. Our contributions are as follows:

- We present different multi-tenancy models in PostgreSQL-based setups.
- We evaluate all presented setups using both TPC-C and TPC-H from 1 up to 10 tenants.
- We propose a set of metrics specifically designed for this experimental setup.
- Our analysis shows that, depending on the storage system and durability settings, the containerized setup offers favorable scalability and efficiency at the cost of higher memory overhead than the other configurations.

The rest of this paper is structured as follows. In the Section 2, we discuss related work. Section 3 presents the models of multi-tenancy that can be supported with PostgreSQL. In Section 4, we give an overview of the evaluation framework. We present our evaluation in Section 5 and summarize our findings in Section 6, before concluding the paper with directions of future work in Section 7.

2 Related Work

TPC benchmarks have a long history of being used in the context of multi-tenancy. Kiefer et al. investigate a multi-tenant setup using a database-per-tenant approach in three benchmarks [10]: *Scalability* (increasing the number of tenants), *Fairness* (measuring differences in run time per tenant), and *Isolation* (all tenants execute a steady, moderate workload, and at a certain point, a single tenant starts executing a heavy workload). The workload consists of individual TPC-H queries executed against up to eight tenants, each with TPC-H data (SF = 0.5) stored in MySQL. While the authors also mention the schema-per-tenant and VM-per-tenant patterns, no comparative analysis is provided. Neugebauer et al. extend TPC-H to make it suitable for benchmarking a shared-schema setup [13]. This architecture differs fundamentally from others, as tenants must share a common table design. In contrast, our work considers only architectures where tenants are independent and free to define their own table schemas.

Göbel examines a multi-tenant setup using the database-per-tenant model in two benchmarks [7, 8]: *Scalability* (increasing the number of tenants) and *Isolation* (measuring differences in run time per tenant). The workload consists of YCSB queries executed against up to ten tenants, each with 600 MB of data in MySQL. The authors also mention the shared-schema and schema-per-tenant patterns but do not provide a comparative evaluation. However, unlike their work, we do not modify OLTP-Bench (BenchBase), but take the official version without any modification.

Recently, benchmarks of the TPC have been used again to inspect multi-tenant services. Van Rennen and Leis take a customer's perspective [20]. The authors compare the characteristics of the Snowflake Dataset (Snowset, [21, 22]) with those of TPC-H and TPC-DS. Based on their observations, they propose a new benchmark for analytical cloud services. The metrics include duration, CPU time, and data size. A key aspect of the benchmark is the modeling of arrival times, in contrast to the typical back-to-back execution model. Another important aspect is multi-tenancy. Each tenant operates its own TPC-H database. In the experiments, the authors compare Snowflake with System X. One experiment evaluates a shared cluster used by all tenants (private databases). A second experiment compares shared clusters to per-tenant clusters. Yin et al. take a provider's perspective [23]. The authors analyze factors such as table placement strategies, Service Level Agreements (SLAs), and pricing models. Their benchmark is based on TPC-DS. As a baseline, they evaluate a single tenant executing workload tests of varying sizes—small, medium, and large. In their experiments, the authors deploy six tenants on the distributed PostgreSQL derivative, Postgres-XL, across 11 nodes of the Grid'5000 platform. They also compare the system under test (SUT) to Apache Spark. Both studies simulate scenarios where multiple tenants execute workloads of varying intensity to evaluate the system's ability to manage fluctuations in demand. In contrast, we focus on assessing the system's performance under stress conditions.

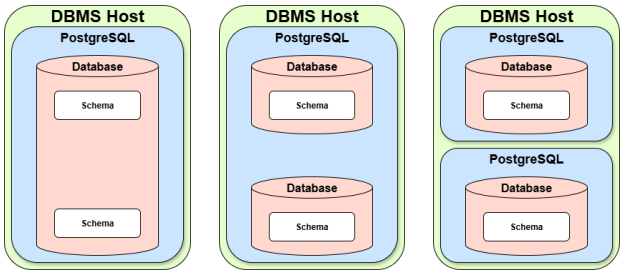


Figure 1: multi-tenant models for two tenants

3 Multi-Tenancy

In the following, we present different levels of multi-tenancy.

A *tenant* is a group of users that shares access to a software application. Each tenant has its own, distinct and (at least logically) isolated set of data. A *multi-tenant architecture* is a design pattern in which a single instance of an application serves multiple tenants. Multi-tenancy can be implemented at different levels:

Shared-schema. In this table-level multi-tenancy model, all tenants share the same schema and tables, with a tenant identifier (e.g., `tenant_id`) used to isolate data. This model presumes that all tenants adhere to an identical table structure, which simplifies schema governance but limits per-tenant customization.

Private Schema. In the schema-per-tenant model, all tenants share a single database, but each tenant has a separate schema. In the following, we refer to this as the *schema model*. This model allows per-tenant schema evolution, supporting heterogeneous requirements, but increases metadata overhead and complicates provisioning, patching, and migration at scale.

Private Database. In the database-per-tenant model, all tenants share a single DBMS instance, but each tenant has a separate database. In the following, we refer to this as the *database model*. Compared to private-schema, the private-database model provides stronger isolation and per-tenant configuration, but incurs higher metadata and operational overhead, as each database must be managed, patched, and migrated individually.

Private Container. In the container-per-tenant model, all tenants share a host OS, but each tenant has a separate instance of the DBMS as a container. In the following, we refer to this as

the *container model*. Compared to private-database, the private-container model provides stronger isolation and deployment flexibility, as each tenant runs in an independent DBMS instance, but it increases memory and CPU overhead and complicates lifecycle management across many containers.

Private VM. In the VM-per-tenant mode, there is a single machine, but each tenant has a separate instance of the DBMS as a VM. Compared to private-container, the private-VM model offers stronger isolation with dedicated OS and DBMS resources per tenant, at the cost of higher resource overhead and reduced density.

These architectures differ in their operational implications, including aspects such as backup management, tenant migration, and deployment complexity. However, in this work, we focus specifically on performance and efficiency dimensions.

The performance of a shared-schema multitenant model depends critically on how the `tenant_id` attribute is introduced and used. Using `tenant_id` as a physical partitioning key further introduces several alternative strategies with significant performance implications. No standard benchmark exists, and constructing a new benchmark would require careful methodological choices, and is beyond our scope; instead, we rely on existing benchmarks.

Likewise, the performance of a VM-per-tenant model depends on how virtualization is managed. To constrain the space of design parameters, we fix the virtualization platform to a single Docker engine and compare only the remaining three models. Our canonical procedure for introducing a new tenant involves creating a schema, creating a database, and starting a container of a publicly available DBMS.

In distributed DBMSs, data distribution constitutes another fundamental design dimension. Approaches span load-driven rebalancing and data migration to modern leader-election mechanisms among replicated shards. These choices can significantly influence performance and introduce substantial architectural and operational complexity. We argue that a solid understanding of single-host effects is essential before analyzing multi-host deployments.

To reduce the number of interacting factors, we focus exclusively on single-host performance and rely on well-established benchmarks. Overall, partitioning choices in shared-schema systems, virtualization management in VM-per-tenant systems, and distribution strategies in distributed DBMSs all have substantial impact on performance. Each entails distinct trade-offs that merit systematic investigation beyond the scope of this work.

3.1 PostgreSQL

PostgreSQL is an open-source, standards-compliant, and extensible relational DBMS, offering features such as ACID transactions, MVCC, rich indexing, full-text search, JSON/JSONB support, and geospatial capabilities via PostGIS. Its extensible architecture allows for custom data types, operators, and procedural languages, supporting a broad spectrum of workloads from transactional to analytical. The active and large community provides continuous development, a robust ecosystem of tools, and a wide range of extensions. Several cloud and commercial products build on or are compatible with PostgreSQL, including Amazon Aurora, Azure Database for PostgreSQL Hyperscale, Greenplum, Citus, TimescaleDB, CockroachDB, and YugabyteDB. In PostgreSQL, several critical subsystems are shared across an entire server instance (cluster). These include the Lock Manager, which provides concurrency control at various granularities; the Write-Ahead Log (WAL), which is maintained globally under `pg_wal/` and ensures durability and crash recovery for all databases; and the Buffer Manager, which caches disk pages in a shared memory pool accessible by all backend processes. Updates from shared buffers to disk may be flushed by background processes such as the checkpointer, the background writer, or directly by connection backends when triggered by client operations. Because these components are shared at the cluster level, the practical degree of isolation between databases is not fundamentally different from that achieved by separate schemas within a single database. Both arrangements share WAL, lock management, and buffer pools, meaning heavy workloads in one database can influence latency and recovery characteristics in others.

In contrast, some resources are scoped per database. For example, many system catalogs (such as `pg_class` and `pg_attribute`) exist separately in each database, and connection backends are bound to a single database for the duration of a session. Transaction IDs are allocated per database, so high write activity in any schema accelerates XID consumption and may trigger earlier autovacuum freeze, underscoring that only separate databases—not schemas—provide isolation at this level. Nevertheless, even per-database components interact with the shared subsystems described above; therefore, true isolation in PostgreSQL can only be achieved through separate instances, where only the underlying host infrastructure is shared.

4 Evaluation Framework

In this section, we discuss the scope of our evaluation and the evaluation framework.

4.1 Benchmark Architecture

Our objective is to compare the *schema*, *database*, and *container* models under load. To this end, we execute both a transactional and an analytical benchmark. We measure key performance metrics on a per-tenant basis and compute system-wide metrics to capture the overall performance. In addition, we monitor hardware-level metrics throughout the experiments to provide a comprehensive view of system behavior. We increase the number of tenants to better understand their impact on system behavior (*Scalability*). We aim to investigate how much resource capacity the provider must allocate in order to guarantee performance (*Efficiency*). Finally, we aim to examine how the performance experienced by a tenant is influenced by the presence and activity of other tenants (*Fairness*).

4.2 Framework and Tools

We identify four distinct phases in the execution of an experiment. In the setup phase, the databases and schemas are prepared. During the ingestion phase, the tenant inserts data into the system. This is followed by the post-ingestion phase, in which the system reacts to the newly inserted data. Finally, in the workload execution phase, the tenant operates on its dataset. These phases are executed individually for each tenant. We aim to synchronize the phases across all tenants such that all tenants transition to the next phase simultaneously. This ensures that the system under test (SUT) is subjected to the intended workload conditions in a coordinated and controlled manner. Each tenant queries its own dataset and is represented by a dedicated Docker container that

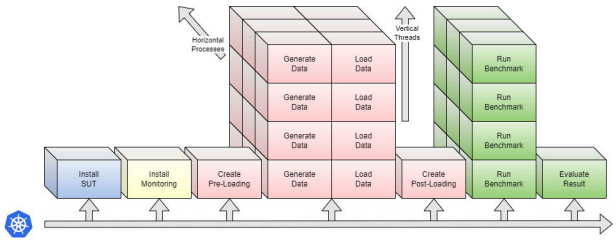


Figure 2: Components of a Benchmarking Experiment [5]

encapsulates the user's application runtime, configuration, and data access context. Each tenant connects via JDBC (or `psql`), and the server, database, and schema are selected directly as part of the connection process. The experiments are managed using *Bexhoma*[1], a framework for automated benchmarking and workload orchestration [3]. The architecture can be summarized as follows [4]: the benchmarking process is decomposed into reusable components, which are treated as microservices. As many components as possible are containerized and orchestrated using Kubernetes. This approach is referred to as *cloud-native benchmarking* [5], as it decomposes the traditionally monolithic benchmarking process according to cloud-native principles. Wherever possible, execution is shifted to the cloud, thereby simulating the structure and behavior of a cloud-native application. For observability, container logs and hardware metrics are collected using a dedicated stack consisting of cAdvisor[2] and Prometheus[3]. Hardware metrics are collected at 30-second intervals.

5 Experiments

All tenant workloads (i.e., the benchmark drivers) are executed on a single physical node equipped with two Intel® Xeon® E5-2630 CPUs (40 cores total) and 750 GB of RAM. All PostgreSQL instances are deployed on another single physical node equipped with an AMD Opteron 6378 processor (64 cores) and 500 GB of RAM. The nodes are part of a Kubernetes cluster. Each instance runs as a Docker container, with its primary data directory initially located on a local ephemeral disk, which is cleared upon container termination. Consequently, any data is lost upon container shutdown or restart. To ensure persistence, each PostgreSQL instance is assigned a dedicated persistent volume backed by a distributed file system (CephFS), and is deployed such that its data directory resides on this volume. This configuration enhances storage-level isolation, reducing interference between instances and enabling independent management and recovery. Persisting data on distributed storage rather than node-local disks is a standard practice in containerized deployments, as it ensures durability across restarts and supports flexible rescheduling independent of the physical host.

We use the official PostgreSQL 16.1 container image. The size is about 150 MB. The PostgreSQL configuration is tuned for high concurrency, extensive parallelism, and large memory usage, targeting a high-performance server environment (e.g., `max_connections`, `max_worker_processes`). Key parallelism parameters are maximized to leverage many CPU cores, while memory settings allocate substantial resources to caching and query operations (e.g., `shared_buffers`, `work_mem`). Write-ahead logging and checkpoint settings prioritize throughput by minimizing durability guarantees, with WAL level set to minimal and synchronous commit disabled (e.g., `wal_level`, `synchronous_commit`, `max_wal_size`, `checkpoint_timeout`). Autovacuum is turned off to avoid background interference during tests (`autovacuum`). A high random page cost (`random_page_cost`) reflects slow or remote storage.

5.1 TPC-C

In our first experiment, we use the TPC-C benchmark. The tenant's container runs Benchbase[4] (based on OLTP-Bench [2]). Each tenant is configured with 10 warehouses, and is served by 100 clients that issue queries to the tenant's database without keying or thinking time. This setup imposes high stress on the system, even at smaller scales. The setup phase runs `CREATE DATABASE` or `CREATE SCHEMA` and the post-ingestion phase runs `VACUUM ANALYZE` on all tables. The workload is run for 10 minutes and is executed twice. For each metric, we report the less favorable outcome—specifically, lower throughput, higher latency, or increased resource consumption, unless stated otherwise. BenchBase reports goodput, defined as the throughput of successfully completed queries. Throughput and hardware resource usage metrics are aggregated across tenants by summation. For average latencies, we compute the macro average, i.e., the mean of the per-tenant averages. The durations of ingestion and vacuum are aggregated across tenants by maximum. These aggregated figures represent the total workload imposed on the system under test (SUT), i.e., the provider's perspective.

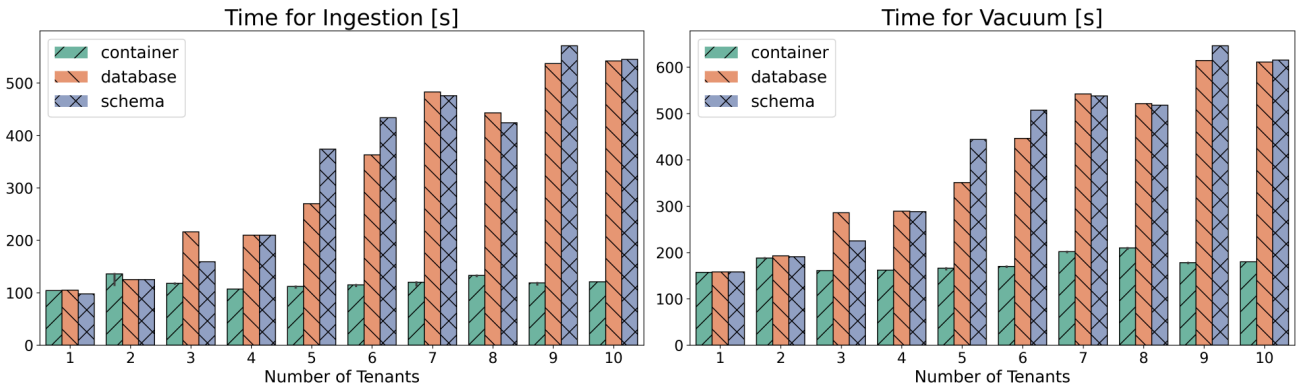
5.1.1 Evaluation from the Provider's Perspective. The container model demonstrates significantly better scalability for both ingestion and vacuuming operations, as shown in Fig. 3. This is attributed to the fact that, in this model only, each tenant is assigned a dedicated persistent volume, and the underlying CephFS file system scales efficiently under these conditions. To isolate the effect of the filesystem, we rerun the experiment with the databases stored in the container's ephemeral storage instead of on a distributed filesystem. Fig. 4 shows that the scaling behavior can, in fact, be attributed to the use of the distributed filesystem. When data is stored on a shared local disk, loading times exhibit greater variability. While the container model continues to perform best, its advantage is less pronounced. As all other metrics remain largely unchanged, we report results based on the distributed filesystem configuration for the remainder of the paper, since it represents the common deployment setting.

Under the schema and database models, the system becomes nearly saturated with just a single tenant, as illustrated in Fig. 5. Adding more tenants results in only marginal improvements in throughput, while latency increases dramatically. For the container model, the behavior is notably different. Throughput continues to increase with the number of tenants, albeit sub-linearly, while latency rises only gradually up to four tenants. This trend is also evident in CPU utilization, as shown in Fig. 6, which presents the results from the first run. The container model exhibits higher CPU usage, whereas the schema and database models do not fully utilize the available CPU resources. The efficiency metrics are defined as follows:

$$E_{\text{Tpx}} := \frac{\text{Goodput} \times d}{\text{CPUs}} \quad (1)$$

$$E_{\text{Lat}} := \frac{1}{\sqrt{\text{Latency} \times \text{CPUs}}} \quad (2)$$

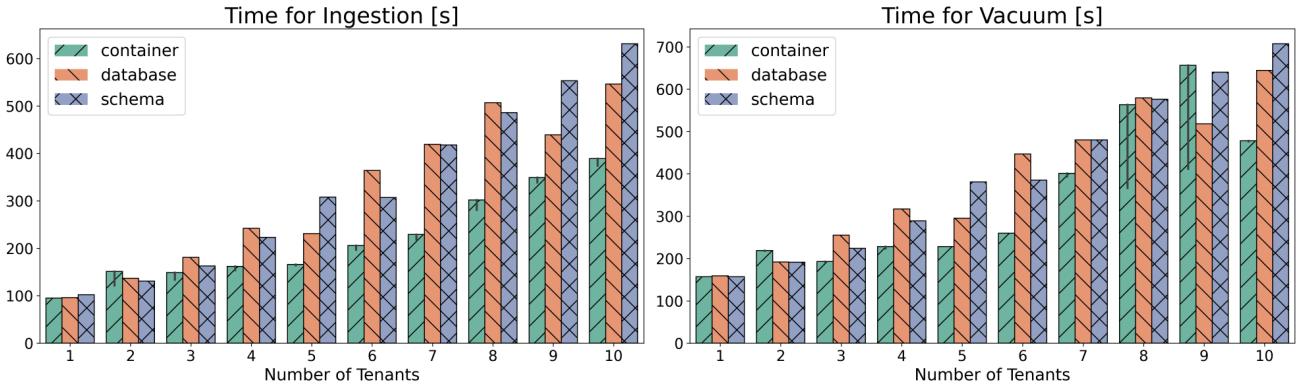
[1] https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager
[2] https://github.com/google/cadvisor
[3] https://prometheus.io/
[4] https://github.com/cmu-db/benchbase



(a) Total time for ingestion

(b) Total time for vacuum

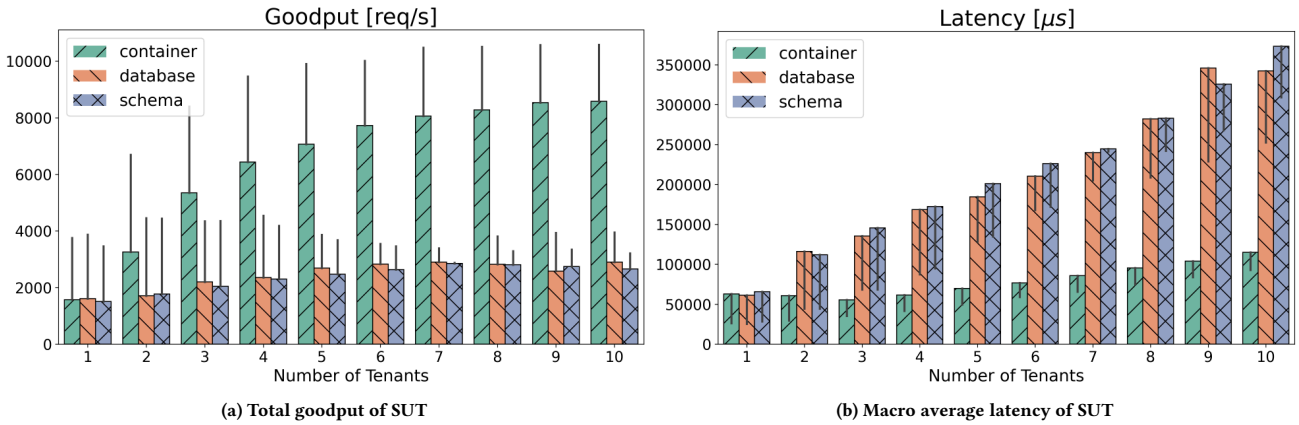
Figure 3: TPC-C - data loading into distributed storage



(a) Total time for ingestion

(b) Total time for vacuum

Figure 4: TPC-C - data loading into local ephemeral storage

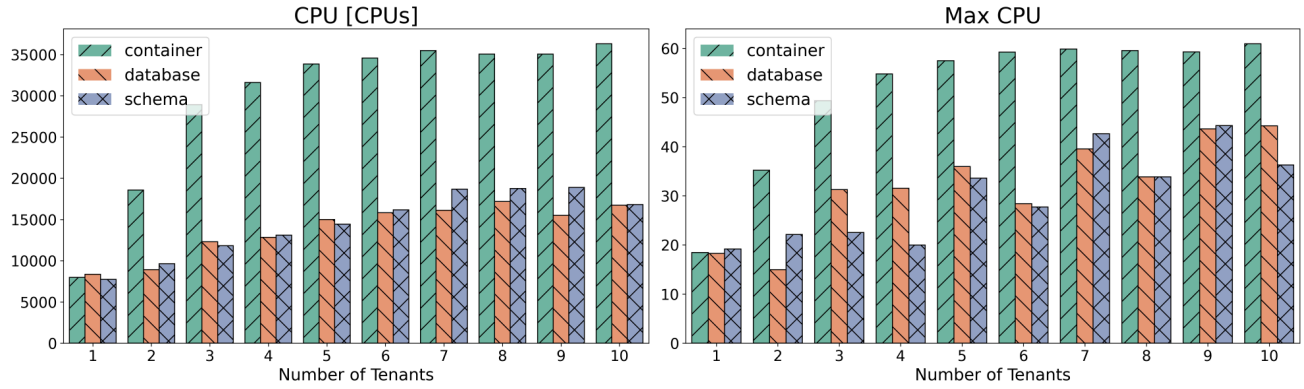


(a) Total goodput of SUT

(b) Macro average latency of SUT

Figure 5: TPC-C - performance

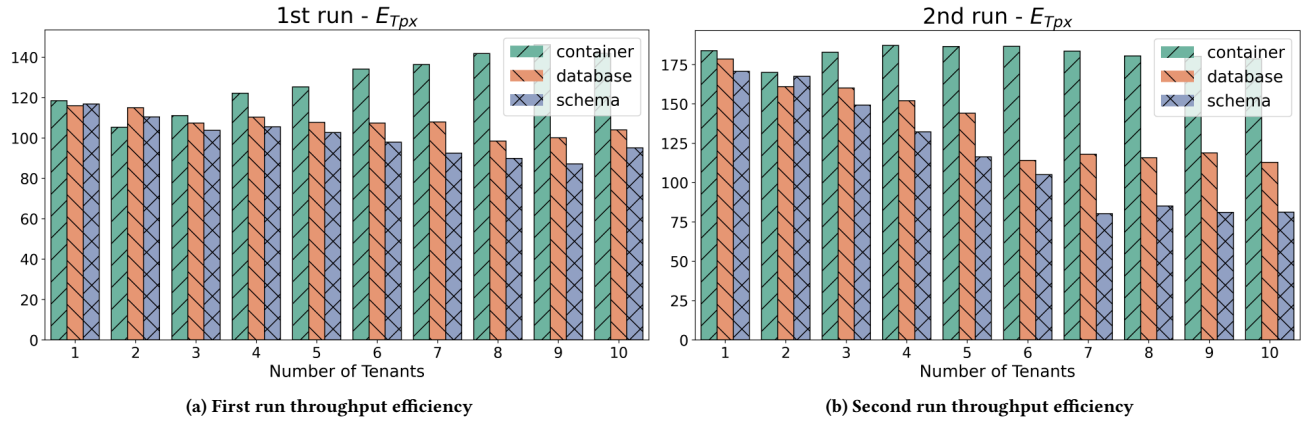
where $d = 600$ denotes the execution duration in seconds, *Goodput* is measured in requests per second, and *Latency* is the average request latency in seconds. The efficiency metric E_{Tpx} quantifies throughput normalized by CPU usage, representing the number of successful queries executed per second of CPU time. This formulation of E_{Lat} balances responsiveness and resource consumption by penalizing systems with high latency or high CPU usage, using the geometric mean to treat both factors symmetrically.



(a) Total CPU time

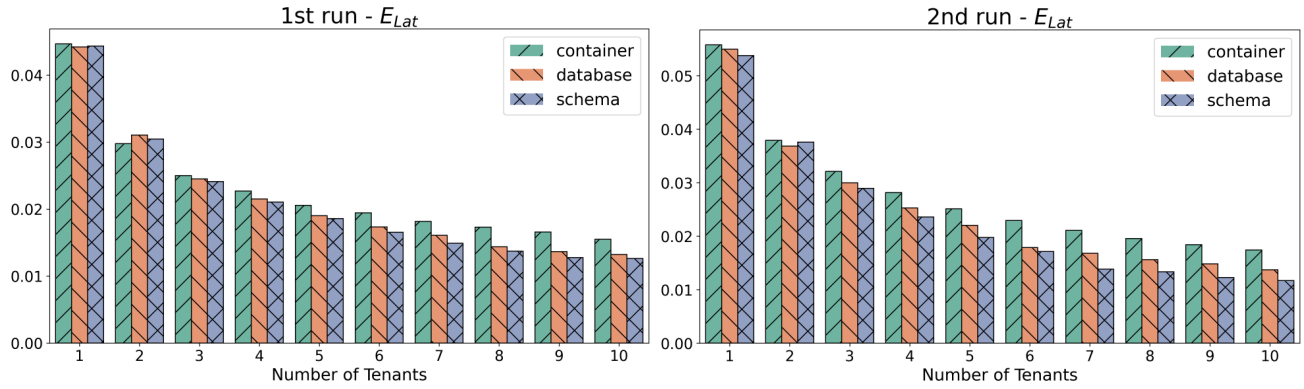
(b) Max CPU in total

Figure 6: TPC-C - CPU utilization



(a) First run throughput efficiency

(b) Second run throughput efficiency



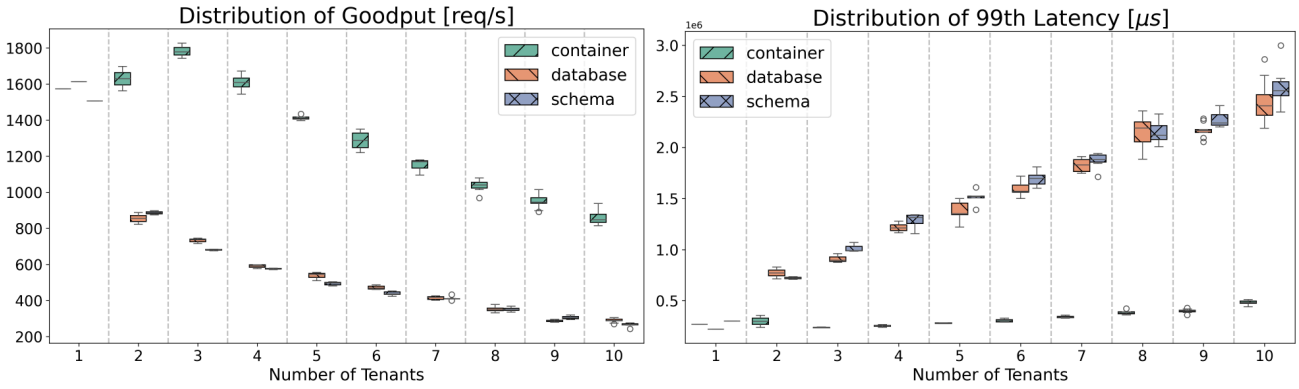
(c) First run latency efficiency

(d) Second run latency efficiency

Figure 7: TPC-C - efficiencies

As shown in Fig. 7, the container model is the most efficient for throughput. In a single-tenant configuration, all models utilize approximately the same amount of CPU. However, as the number of tenants increases, the efficiency of the container model improves, while that of the schema and database models degrades—most notably in the schema model, which performs the worst overall. In the second run, the system has stabilized. There is no observable difference in the efficiency of the container model with respect to the number of tenants. The schema model continues to exhibit the lowest efficiency. When evaluating latency efficiency, the results are less conclusive. All models exhibit comparable behavior, with the container-based model showing a slight advantage and the schema-based model performing the least efficiently. However, the observed differences across models are marginal.

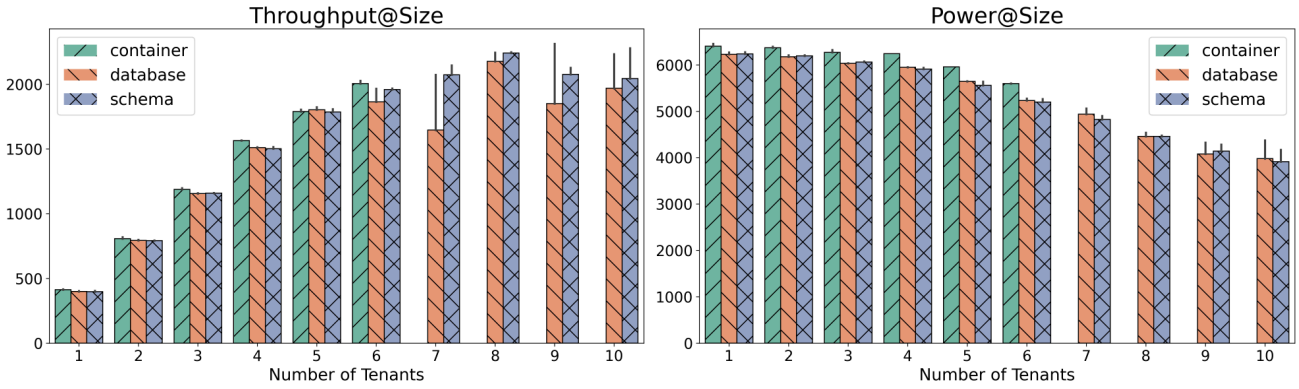
5.1.2 Evaluation from the Tenant's Perspective. Since tenants are modeled as independent benchmark drivers, the system enables the collection and reporting of performance metrics at the



(a) Goodput for each tenant

(b) 99th latencies for each tenant

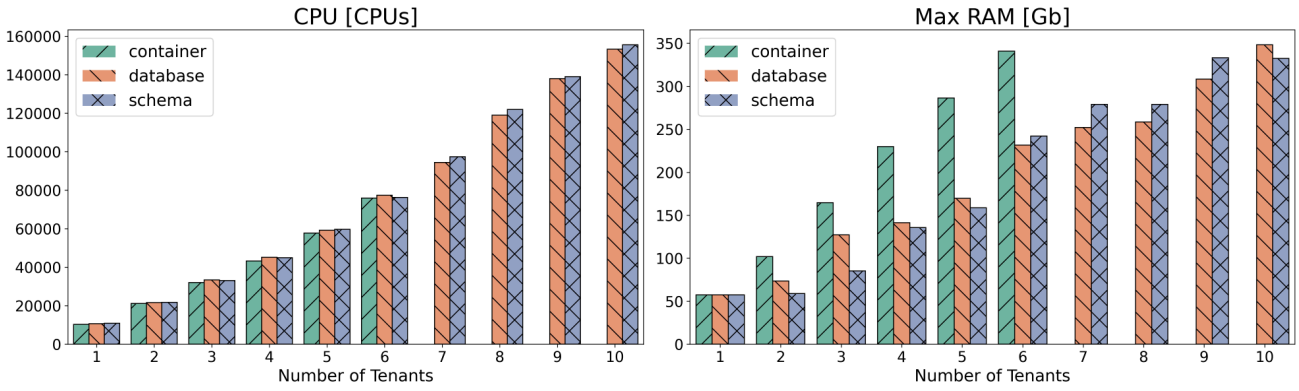
Figure 8: TPC-C - tenant's perspective



(a) Total throughput of SUT

(b) Total power of SUT

Figure 9: TPC-H - performance

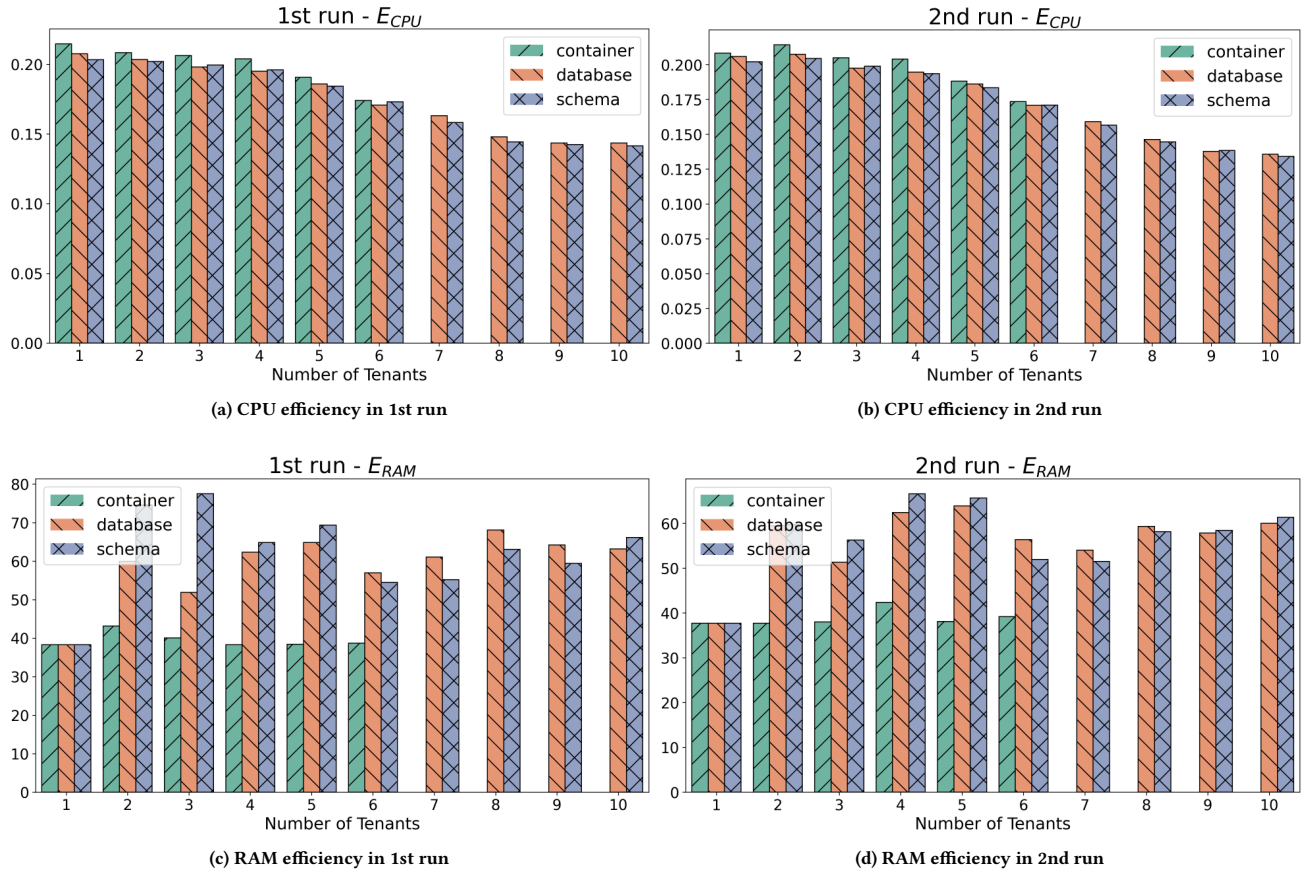


(a) Total CPU time

(b) Max RAM in total

Figure 10: TPC-H - hardware utilization

granularity of individual tenants. This allows us to quantify the performance experienced by each tenant. As shown in Fig. 8, there is little variation in performance between tenants when the total number of tenants remains constant. The container model exhibits slightly greater stability in terms of latency, thereby providing fairer performance across tenants. At the same time, it is marginally the least stable with respect to goodput. The container model achieves the best overall performance and is capable of maintaining the 99th percentile latency below 0.5 seconds, thereby providing strong latency guarantees.



(a) CPU efficiency in 1st run

(b) CPU efficiency in 2nd run

(c) RAM efficiency in 1st run

(d) RAM efficiency in 2nd run

Figure 11: TPC-H - efficiency

5.2 TPC-H

Here, the pre-ingestion phase also runs `CREATE TABLE` and the post-ingestion phase runs `CREATE INDEX`. The latter also creates constraints and performs `ANALYZE`. The tenant's container for ingestion basically contains `psql`. It loads data generated by the TPC-H[5] toolkit. The tenant's container for execution is based on DBMSBenchmarker[6] [3, 6]. Each tenant is configured to execute TPC-H read-only queries sequentially at a scale factor of 10, using identical datasets. However, the query parameters and the order of execution are varied across tenants. To increase system load and enable more robust measurements, each query is executed 10 times. Following the TPC-H specification, we define the *Throughput@Size* metric as:

$$\text{Throughput@Size} := \frac{n \times S \times 22 \times 3600}{T_s} \times \text{SF} \tag{3}$$

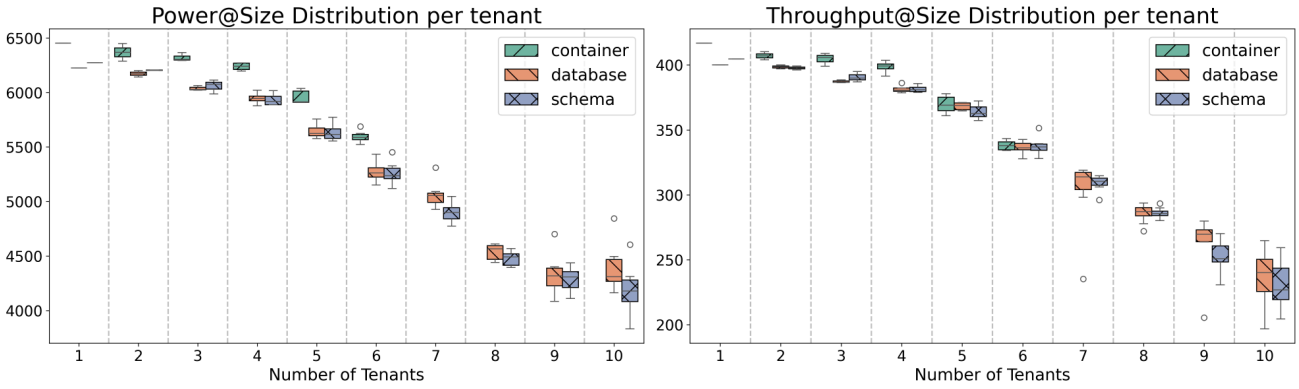
where S denotes the number of query streams, T_s is the total execution time in seconds measured from the start of the first query to the completion of the last, and SF is the scale factor. In our setting, we interpret each tenant as an individual query stream, thereby aggregating throughput across all tenants. An additional factor $n = 10$ is included, as each of the 22 TPC-H queries is executed ten times per stream. In the same spirit, we define *Power@Size* based on the geometric mean T_i of the execution times of the 220 queries within a single stream i, yielding a per-tenant performance metric. To aggregate across S tenants, we compute the geometric mean of the S per-stream geometric means. This is equivalent to computing the geometric mean over all $220 \times S$ query executions. The resulting metric is defined as:

$$\text{Power@Size} := \frac{\text{SF} \times 3600}{\sqrt[S]{\Pi_i T_i}} \tag{4}$$

In the single-tenant case $S = 1$ and for $n = 1$, these metrics coincide with those defined by the TPC specification without update streams.

5.2.1 Evaluation from the Provider's Perspective. The container model supports a maximum of six concurrently active tenants. When a seventh tenant is introduced, Kubernetes begins terminating containers due to out-of-memory (OOM) conditions. To mitigate this limitation, we assign a per-container memory limit of $\frac{480}{S}$ GB, where S denotes the number of active PostgreSQL containers. This configuration is designed to enable Kubernetes to reclaim memory by evicting page cache as containers approach their memory limits. However, this strategy proves ineffective in practice, suggesting suboptimal memory utilization within the container model. A comparable limitation is observed in the database model, which begins to encounter OOM errors at nine tenants. In this case, however, the memory reclamation mechanism proves more effective, allowing stable operation with up to ten concurrent tenants. For data loading and indexing, we observe behavior similar to that of the TPC-C benchmark and therefore refrain from presenting additional plots. The models do not exhibit significant differences in performance, as shown

[5]https://www.tpc.org/tpch/
[6]https://github.com/Beuth-Erdelt/DBMS-Benchmarker



(a) Power for each tenant

(b) Throughput for each tenant

Figure 12: TPC-H - tenant's perspective

in Figure 9. Similarly, CPU utilization remains comparable across models, as illustrated in Figure 10. However, with respect to memory consumption, the container-based model clearly demonstrates the highest RAM usage. The efficiency metrics are defined as follows:

$$E_{\text{CPU}} := \frac{\text{Throughput@Size} \times T_s}{3600 \times \text{CPUs}} = \frac{n \times S \times 22 \times \text{SF}}{\text{CPUs}} \quad (5)$$

$$E_{\text{RAM}} := \frac{\text{Throughput@Size} \times T_s}{3600 \times \text{max. RAM}} = \frac{n \times S \times 22 \times \text{SF}}{\text{max. RAM}} \quad (6)$$

where E_{CPU} denotes the CPU efficiency metric (queries completed per second of CPU time) and E_{RAM} denotes the RAM efficiency metric (queries completed per GB of RAM - peak usage). We show both metrics for both runs in Fig 11. Again, the models exhibit only minor differences in terms of CPU usage. System efficiency degrades as the number of tenants increases. Moreover, it is evident that the container-based model utilizes RAM less efficiently than the other models.

5.2.2 Evaluation from the Tenant's Perspective. Although Figure 12 shows only minor differences, the container model exhibits slightly better Power@Size performance when viewed on a per-tenant basis.

6 Discussion

We scaled the system from 1 to 10 tenants. In TPC-C, the system's resources are saturated: consuming 36,000 CPU-seconds in ten minutes corresponds to a sustained average load of 60 CPU cores. For the container model in Fig. 7, the system approaches this limit beginning at approximately five tenants. We also observe that performance ceases to improve in Fig. 5, a trend that becomes even more pronounced in the remaining models. In TPC-H, throughput stops improving at around six tenants, and the power metric begins to decline. This behavior is expected to continue until memory is exhausted, which occurs for the container model already at seven tenants.

In an idealized setting, all tenants in the container model would exhibit uniform performance characteristics, as they execute identical workloads. Furthermore, PostgreSQL itself introduces no cross-tenant contention, and any observed contention in the container model arises solely at the host level. In contrast, in the database model and the schema model, PostgreSQL introduces additional sources of contention.

6.1 Monitoring

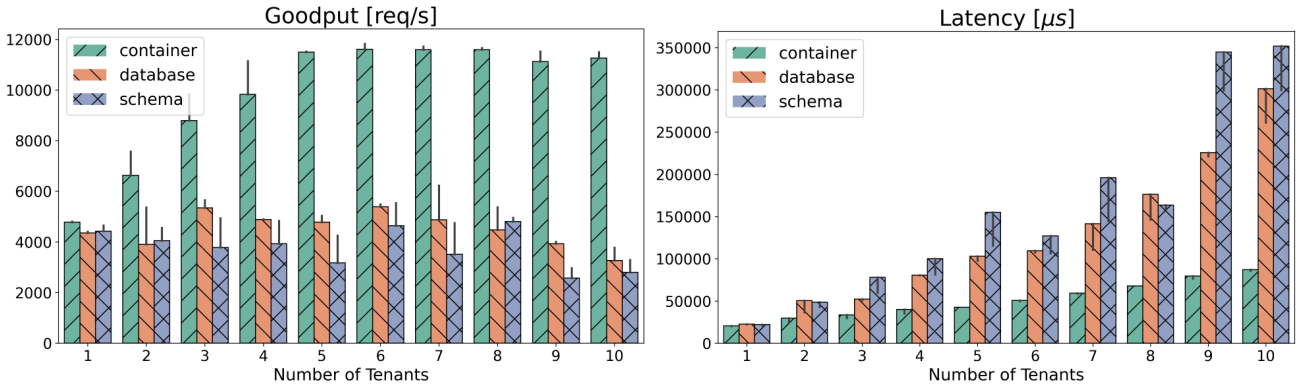
In general, monitoring data has limited explanatory power. It provides a general understanding of system behavior and offers some level of quantification. However, metrics are collected only every 30 seconds. As a result, it is possible for a tenant to experience unfavorable conditions that go unobserved if they occur within the final 29 seconds of an execution window. Despite this limitation, we have retained the corresponding results, as they do not appear to be significantly influenced by randomness. All tenants start simultaneously, and the TPC-C workload runs steadily for a fixed duration of 10 minutes. Under this setup, the maximum possible error caused by a missing measurement is limited to 5%, which we deem acceptable for reliable evaluation.

6.2 Transactional Workload

For transactional workloads, the container model proves to be clearly the most suitable. It achieves higher throughput and lower latency, both at the system level and for individual tenants. This advantage correlates with higher and more efficient utilization of CPU resources. The performance gap becomes increasingly pronounced as the number of tenants increases.

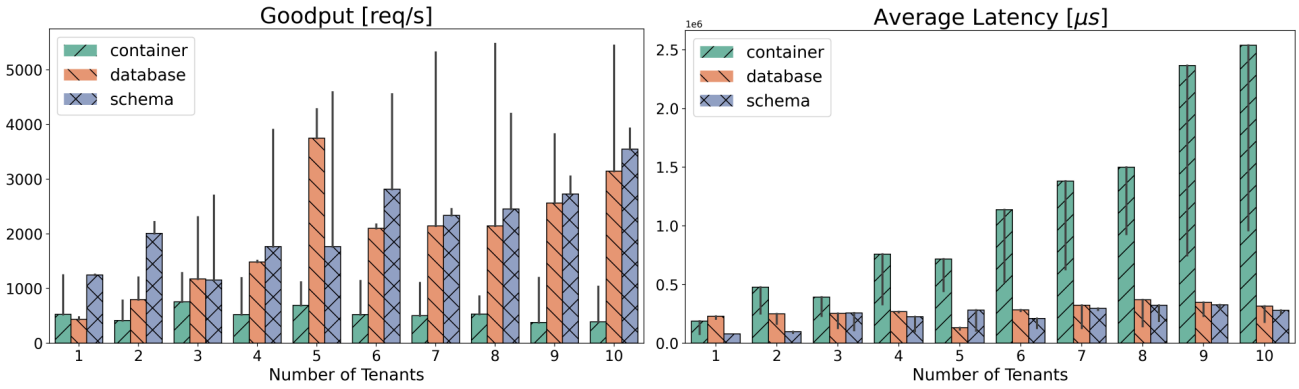
We attribute the observed degradation in E_{Lat} with increasing tenant count to host-level resource contention, such as increased context switching. For a fixed number of tenants, all deployment models exhibit similar E_{Lat} values, indicating comparable host system contention. This aligns with PostgreSQL's process-per-connection design. In both the database- and schema-level models, we observe higher latency despite comparable efficiency. This behavior suggests either internal contention within PostgreSQL, arising from insufficient CPU utilization, or host-level contention caused by a disk shared among all tenants. Despite the absence of cross-database or cross-schema transactions, the system spends a substantial amount of time in wait states without clear external causes. This behavior is slightly more pronounced in the schema-based model, which offers the weakest isolation among the evaluated approaches.

The observed decrease in throughput efficiency (E_{Tpx}) with an increasing number of tenants in both the database and schema models can likewise be attributed to internal contention within a single PostgreSQL instance or to additional host-level contention by a shared disk. As the tenant count grows, PostgreSQL's internal subsystems—such as the buffer manager, lock manager, and



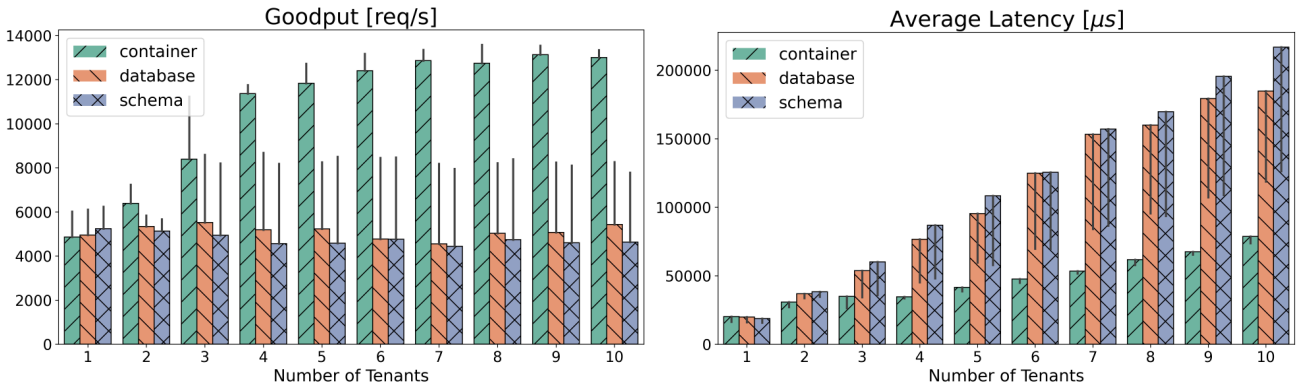
(a) Total goodput of SUT (b) Macro average latency of SUT

Figure 13: TPC-C - CPUManager Policy -distribute-cpus-across-cores- performance



(a) Total goodput of SUT (b) Macro average latency of SUT

Figure 14: TPC-C - databases on shared local drive - performance



(a) Total goodput of SUT (b) Macro average latency of SUT

Figure 15: TPC-C - databases on a RAM disk - performance

background workers—are increasingly stressed. This leads to a shift in CPU usage from query execution toward coordination and maintenance tasks, ultimately reducing throughput per unit of computational resource. In contrast, the container model exhibits an increase in throughput efficiency as the number of tenants grows, up to a saturation point at approximately 175 queries per second of CPU time consumed. This indicates that the containerized deployment achieves significantly better parallelism. As the number of tenants rises, each individual PostgreSQL instance contributes a smaller share to total system throughput. This parallelization appears to offset the host-level overhead introduced

by the additional containers, mitigating the effects of contention and enabling higher overall efficiency.

To further examine this effect, we used the Kubernetes CPU-Manager policy *Distribute CPUs Across Cores* [18], which distributes processes across CPU cores instead of colocating related processes. This policy indeed has an impact on performance (Fig. 13): with Distribute CPUs Across Cores enabled, throughput is generally higher and more stable, and latency tends to be lower. However, the policy does not affect the observed superiority of the container-based execution model. This suggests that the effect is not primarily related to the CPU scheduler, but rather arises from internal mechanisms within PostgreSQL or from the storage layer.

6.3 Disk

The container model exhibits advantages in this regard because each tenant is assigned a dedicated disk, and CephFS scales very effectively. It should be noted that PostgreSQL supports the concept of tablespaces, which would theoretically allow for a dedicated disk per tenant in the other models as well. However, this approach would introduce substantial administrative overhead and undermine the benefits of those models: Kubernetes pods are immutable with respect to their volumes, so adding a new tenant would require taking all existing tenants offline. For this reason, we chose not to pursue the tablespace-based approach. Setting `synchronous_commit = off` returns success to the client as soon as the change has been written to the WAL, regardless of whether the WAL has been flushed to disk. This reduces the dependency on the storage subsystem (at the cost of durability) and results in a significant increase in performance. In comparative experiments, we evaluated configurations with high-durability settings, i.e., with heavier write operations. In a shared local disk setting, the container model exhibits the lowest performance (Fig 14). Multiple independent PostgreSQL instances operating uncoordinatedly on the same physical disk substantially harms throughput and latency. When the database is stored on a RAM disk, the container model again achieves the best performance (Fig 15). In the configuration presented here, the container model benefits from using independent volumes, which is necessary to maintain high performance. However, this alone does not account for all observed effects, as PostgreSQL's internal subsystems also contribute significantly to the performance behavior.

6.4 PostgreSQL

PostgreSQL exposes numerous configuration parameters that govern disk, memory, and CPU usage, and thereby influence performance through their interplay. We optimized the configuration for high-load scenarios without evaluating the effect of each individual parameter in isolation. We assume that this primarily benefits the *schema* and *database* models. A "standard" PostgreSQL setup would likely be unable to serve ten tenants simultaneously, which would otherwise further support the container-per-tenant approach. A high `random_page_cost` causes PostgreSQL to favor sequential scans over index usage. This configuration can make the container model perform more effectively. As the buffer is sufficiently large to hold all data, the reduced efficiency of the other models may be attributable to buffer fragmentation.

When a RAM disk is used for persistence (Fig 15), both the database-per-tenant and schema-per-tenant models can become saturated under a single tenant's load, indicating a subsystem limit. PostgreSQL relies on shared memory for critical subsystems such as the buffer cache, lock manager, and WAL buffers, all coordinated via lightweight locks (LWLocks) to maintain consistency. PostgreSQL's LWLocks use atomic memory operations for fast uncontended acquisition, but when a lock is contended, the waiting process blocks on a system semaphore rather than busy-waits, which can lead to a heavy-weight OS-level context switch [15]. Under high concurrency, contention and race conditions on these shared-memory structures can exhaust a CPU core, limiting throughput and causing latency spikes. We thus conclude that the subsystems have reached their performance limits.

6.5 Analytical Workload

As the number of tenants increases, CPU efficiency (E_{CPU}) declines across all deployment models, reflecting growing contention for shared host resources. PostgreSQL appears to parallelize analytical workloads effectively. For a fixed number of tenants, all three models exhibit similar behavior, indicating minimal internal contention within PostgreSQL. The effectiveness of parallelization is also reflected in memory efficiency (E_{RAM}). Utilizing a single PostgreSQL instance for multiple tenants is significantly more memory-efficient than deploying separate containers. In this workload, performance does not depend on the speed of writing to disk. Moreover, as we use only one connection per tenant executing a complex query, the overhead effects observed in the transactional workload do not apply here.

7 Conclusion

This study evaluated three multi-tenancy architectures for PostgreSQL – schema-per-tenant, database-per-tenant, and container-per-tenant – using TPC-C and TPC-H benchmarks. Experiments were conducted with 1 to 10 tenants, focusing on scalability, efficiency, and fairness. The results indicate that the container model reduced internal contention and utilized system resources more effectively under transactional workloads. Independent containers can benefit from scalable storage subsystems, particularly for write-heavy workloads. On the other hand, slow underlying storage can degrade their performance disproportionately. For analytical workloads, all models achieved comparable performance, with the schema and database models showing advantages in memory utilization. These findings highlight trade-offs between isolation, resource sharing, and workload characteristics, offering practical insights for designing scalable and efficient multi-tenant PostgreSQL deployments.

Future work should explore the underlying causes of contention and develop strategies for its mitigation. Another interesting direction is the analysis of hybrid multi-tenancy strategies. We also hope that the presented benchmark-based methodology serves as a useful foundation for further research into multi-tenancy performance in relational database systems. This work has focuses on PostgreSQL, as one of the most widely used database management systems, evaluating other systems will lead to a broader picture of multi-tenancy performance.

Acknowledgments

The authors thank Peter Tröger and Paul Grundmann for their administration of the Kubernetes cluster utilized in this study. This work was partially funded by the German Research Foundation (ref. 414984028 and ref. 556566056) and SAP.

8 Artifacts

All scripts and configurations required to reproduce the results and evaluations presented in this paper are publicly available in our GitHub repository[7].

References

[1] Anum Jang Sher and Peter Fein. 2022. Build scalable multi-tenant databases with Amazon Aurora. https://d1.awsstatic.com/events/Summits/reinvent2022/DAT318_Build-scalable-multi-tenant-databases-with-Amazon-Aurora.pdf AWS re:Invent 2022, Session DAT318. Online; accessed 2026-02-16.

[2] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288.

[3] Patrick K. Erdelt. 2021. A Framework for Supporting Repetition and Evaluation in the Process of Cloud-Based DBMS Performance Benchmarking. In *Performance Evaluation and Benchmarking*. 75–92.

[4] Patrick K. Erdelt. 2022. Orchestrating DBMS Benchmarking in the Cloud with Kubernetes. In *Performance Evaluation and Benchmarking*. 81–97.

[5] Patrick K. Erdelt. 2024. A Cloud-Native Adoption of Classical DBMS Performance Benchmarks and Tools. In *Performance Evaluation and Benchmarking*. 124–142.

[6] Patrick K. Erdelt and Jascha Jestel. 2022. DBMS-Benchmarker: Benchmark and Evaluate DBMS in Python. *Journal of Open Source Software* 7, 79 (2022), 4628.

[7] Andreas Göbel. 2014. MuTeBench: Turning OLTP-Bench into a Multi-Tenancy Database Benchmark Framework. In *CLOUD COMPUTING 2014, The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*. 84–87.

[8] Andreas Göbel. 2024. MuTeBench. https://github.com/MuTeBench/MuTeBench Online; accessed 2026-02-16.

[9] Google Cloud. 2025. Implement multi-tenancy in Spanner | Google Cloud Documentation. https://docs.cloud.google.com/spanner/docs/implement-multi-tenancy Online; accessed 2026-02-16.

[10] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. 2013. MulTe: A Multi-Tenancy Database Benchmark Framework. In *4th TPC Technology Conference - Selected Topics in Performance Evaluation and Benchmarking*. 92–107.

[11] Paarth Madan and Xiaoli Liang. 2021. Shard Balancing: Moving Shops Confidently with Zero-Downtime at Terabyte-Scale. https://shopify.engineering/mysql-database-shard-balancing-terabyte-scale Online; accessed 2026-02-16.

[12] Microsoft Azure. [n. d.]. Multitenancy and Azure SQL Database - Azure Architecture Center | Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/sql-database Online; accessed 2026-02-16.

[13] Philipp Neugebauer, Christian Maier, and Alexander Bumann. 2017. Benchmark Proposal for Multi-Tenancy in the Database Layer. In *Proceedings of the 9th Central European Workshop on Services and their Composition (ZEUS 2017), Lugano, Switzerland, February 13-14, 2017 (CEUR Workshop Proceedings, Vol. 1826)*. 71–78.

[14] Oracle Corporation. [n. d.]. Oracle Multitenant. https://www.oracle.com/database/multitenant/ Online; accessed 2026-02-16.

[15] PostgreSQL Global Development Group. 2025. Lock Manager Internals (README). https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README. Online; accessed 2026-02-16.

[16] Rails on Services. 2025. Apartment: Multi-Tenancy for Rails. https://github.com/rails-on-services/apartment. Online; accessed 2026-02-16.

[17] Salesforce Architects. 2022. Platform Multitenant Architecture. https://architect.salesforce.com/fundamentals/platform-multitenant-architecture. Online; accessed 2026-02-16.

[18] Jiaxin Shan. 2024. Kubernetes v1.31: New Kubernetes CPUManager Static Policy: Distribute CPUs Across Cores | Kubernetes. https://kubernetes.io/blog/2024/08/22/cpumanager-static-policy-distributed-cpu-across-cores/ Online; accessed 2026-02-16.

[19] Jeff Swenson, Andy Kimball, Raphael 'kena' Poss, Rebecca Taft, Jay Lim, Adam Storm, Sumeer Bhola, Paul Bulkley-Logston, Pj Tatlow, Rachael Harding, Rafi Shamim, Aditya Maru, and Irfan Sharif. 2025. CockroachDB Serverless: Sub-second Scaling from Zero with Multi-region Cluster Virtualization. In *Companion of the 2025 International Conference on Management of Data*. 648–661.

[20] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.

[21] Midhul Vuppalapati. 2025. Snowflake dataset containing statistics for 70 million queries over 14 day period. https://github.com/resource-disaggregation/snowset Online; accessed 2026-02-16.

[22] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 449–462.

[23] Shaoyi Yin, Franck Morvan, Jorge Martinez-Gil, and Abdelkader Hameurlain. 2025. MTD-DS: An SLA-Aware Decision Support Benchmark for Multi-Tenant Parallel DBMSs. *IEEE Transactions on Knowledge and Data Engineering* 37, 5 (2025), 2743–2755.

[7] https://github.com/BHT-Math/Benchmarking-Multi-Tenant-Architectures-in-PostgreSQL