ISSN: 2182-2069 / E-ISSN: 2182-2077

Migrating Legacy Databases to Cloud: Challenges and Best Practices in AWS RDS and Postgres

Harsha Vardhan Reddy Kavuluri[1*], Suresh Babu Avula[2], and Adithya Sirimalla[3]

[1*]Lead Database Administrator, Wissen Infotech Inc, USA.
kavuluri99@gmail.com, https://orcid.org/0009-0005-6003-6464

[2]Associate Director, Innovaccer Analytics Pvt Ltd, Noida, Uttar Pradesh, India.
sureshbabuavula@gmail.com, https://orcid.org/0009-0000-4510-4944

[3]Software Developer and DBA, 20755 Williamsport Pl suite 230, One Loudoun, VA, USA.
adithya.skal@gmail.com, https://orcid.org/0009-0000-9105-7907

Received: March 11, 2025; Revised: April 19, 2025; Accepted: May 15, 2025; Published: May 30, 2025

Abstract

Migrating legacy databases to cloud-based platforms is an essential transformation step for businesses seeking elasticity, dependability, and financial efficiency. This work provides an impact-focused assessment of migrating multiple legacy systems to AWS RDS with PostgreSQL. Through real-world schema and workload simulation, some aspects of the migration were evaluated concerning key performance indicators and compatibility benchmarks. The heterogeneous schema conversion achieved an average accuracy rate of 92.4%. Latency on post-migration queries dropped an average of 47% compared to legacy system post-migration latency. Supplied peak-load benchmarks produced an average of 38% CPU savings and as much as 60% improvement in IOPS from RDS instances with auto-scaling while DMS issued downtime events capped at 3 minutes in 84% of cases. These outcomes prove that the cloud-based architecture configuration does surpass classic design not merely in throughput but in system availability as well as operational cost efficiency. Erroneous observations coupled with infrequent rest points and SLA trends informed the formulated practical best practices.

Keywords: Cloud Database Migration, AWS RDS, PostgreSQL Performance, Legacy System Modernization.

1 Introduction

1.1. Legacy Database Constraints in Modern Business Systems

Despite serving a fundamental purpose for enterprise applications, legacy database systems struggle to cope with modern digital settings due to their rigid framework and monolithic design. These systems are either built with proprietary engines using relational technology or on mainframe systems, which, during their construction, existed in a time of limited data volumes, predictable workloads, and static deployment environments (Stonebraker et al., 2014; Stonebraker & Çetintemel, 2018). Today, enterprises seek agility and favor distributed systems along with cloud-native frameworks, which makes the inadequacies of legacy databases disruptive (Fowler, 2012).

Journal of Internet Services and Information Security (JISIS), volume: 15, number: 2 (May), pp. 985-1003.
DOI: *10.58346/JISIS.2025.I2.065*

*Corresponding author: Lead Database administrator, Wissen Infotech Inc, USA.

Another notable shortcoming of legacy systems is their inability to scale efficiently. Traditionally, these systems were hosted on databases that were vertically scaled, meaning that increased performance came simply from bolting on more hardware, as opposed to load balancing the workload across multiple systems. This strategy creates distinct profitability ceilings, which creates challenges in the current business climate that demands high concurrency workloads, real-time analytics, and global accessibility (Pavlo et al., 2017). Furthermore, embedded business logic within stored procedures, triggers, and non-portable functions further complicate matters. The integration of such systems with microservices or cloud-native data platforms becomes nearly impossible because the tight coupling of logic and data ruins modularity, which cripples system maintainability (Garlan et al., 2002).

The maintenance issues of legacy systems are compounded by the many other shortcomings these systems have. Support for older database engines may not be available which forces companies to keep outdated infrastructure. Because of this, businesses are forced to wait on a dwindling supply of professionals skilled in the outdated technology. Furthermore, in heavily regulated sectors where systematic consistency alongside the integrity of data is of utmost importance, performing “routine” tasks such as patching, hardware upgrades, system performance tuning, and performance improvement become exponentially challenging and expensive (Conteh & Akhtar, 2015).

Outdated security frameworks present another severe issue. A great many legacy databases do not have current sophisticated encryption strategies, audit logging technology, or advanced access control systems. This in conjunction with strict forms of data security policies such as HIPAA and GDPR turns legacy databases into an unlocked vault open for unlawful entry, data breaches, and failing compliance. More to the point, recovery strategies for these systems tend to be slow and manually customizable, risky by hardware dependency, and prone to very long recovery times which jeopardize critical business functions.

Finally, legacy databases do not keep pace with new data integration demand (Sharma & Das, 2024). They cannot interface seamlessly with streaming services, cloud-native ETL systems, and machine learning pipelines, contributing to an impeded real-time responsiveness, stalled forward momentum of analytic operations, and ceased data asset value exploitation Farajizadeh & NematBakhsh (2015). The inefficiencies in technology cause companies to fall behind industrious competition but directly surfacing opportunity outside of strategic plans further deepens the technical debt.

1.2. Cloud Migration Momentum and Motivation

To meet these goals, nearly every organization has tried to adopt a new strategy focusing on updating their databases with cloud-native managed services due to their ease of use and better management. Services like AWS RDS are cloud databases that substitute on-premise databases. AWS RDS with PostgreSQL is especially noted for its high performance and low cost when compared to industry standards (Kumar & Rajeshwari, 2024; Vadlamani, 2024). Most importantly, it has enterprise-level capabilities which provides an additional edge.

There are many reasons that contribute to driving the migration to AWS RDS and PostgreSQL (Iyengar & Bhattacharya, 2024). Unlike legacy systems, cloud-native services come with operational flexibility benefits. Everything, from creating a new instance, and restoring backups, to scaling resources and automating failovers, can be performed with little to no effort. These systems enable low-touch automated processes replacing manual, hardware dependent tasks. Such features provide agility for rapid business model exploration and innovation (Abadi, 2009). Companies looking to improve their systems or offer more digital services are greatly helped due to this pace.

Incentives for cloud adoption also stem from cost optimization (Nithya & Narmatha, 2017). With AWS RDS, the need for hardware refresh cycles is replaced with consumption-based pricing models which take into account the business's needs. Financial efficiency can further be strengthened through reserved instances and automated resource scheduling. These financial mechanisms are useful for enterprises dealing with augmented IT cost constraints or companies multifunctional digital transformation initiatives (Shende & Chapke, 2015).

Apart from flexibility and cost reduction, AWS RDS offers marked improvements in performance. Under modern RDS infrastructure, PostgreSQL instances can be tuned for high throughput workloads with instance classes such as db.m5 or db.r6g, while gp3 and provisioned IOPS storage types ensure consistent low-latency performance during heavy concurrent load. Unlike legacy environments that need meticulous optimization of memory buffer, connection limit, and IO channel parameters, a growing number of tasks are automated on AWS RDS, lessening the workload on database administrators (Li et al., 2013).

Concerns on compliance and security also add to the drives for cloud adoption. AWS RDS provides rest and in transit encryption, integrates with AWS IAM for role-based access governance, and also provides audit logging. These are a must-have for companies in regulated domains, where data security is not just a regulatory need, but also a cornerstone expectation of trust with customers and partners.

In comparison to older rigid hardware clusters, these system's traditional augments automatic failover and AWS RDS high-availability multi-AZ deployments in read replicas as well as automatic snapshot backups significantly enhance system resilience and disaster recovery in AWS RDS. These system's traditional augments automatic failover and AWS RDS high-availability multi-AZ deployments in read replicas as well as automatic snapshot backups significantly enhance system's resilience and disaster recovery.

Perhaps even more striking are the PostgreSQL's and AWS RDS's cloud integration capabilities. Interoperability with AWS Glue, Lambda, S3, Redshift and other cloud-native services allow seamless data ingestion, transformation, and analysis. Legacy systems bring increased operational risk due to fragile middleware that require clunky connectors or custom ETL scripts, often resulting in delays.

Despite the advantages of a migrating to a cloud database, the process is complex. In the schema conflicts alone, organizations encounter mismatched data types, latency-sensitive applications, and the need for exact cutover timings. Organizations are also required to validate the database's performance not only after migration, but also during peak usage simulations to ensure uninterrupted operation and prevent cascading failures in downstream systems. In this sense, evaluation through simulation becomes indispensable in verifying the stability, performance, and cost efficiency of a system after migration and under realistic workloads.

Figure 1 illustrates the architectural differences between a traditionally deployed legacy database system. and cloud application-based modern systems utilizing PostgreSQL AWS RDS. The left part showcases the monolithic architecture comprising storage, compute and backup integrated processes that are commonly found in a single physical server or clustered server. Parallelly, the right part features AWS RDS architecture demonstrating decoupled system components consisting of scalable storage volumes and managed failover configurations with integrated performance monitoring all consolidated via AWS native services. The visual provides insight to operational change that companies attempt during and post migrating, which forms the basis why measuring performance of such systems is of significant importance.

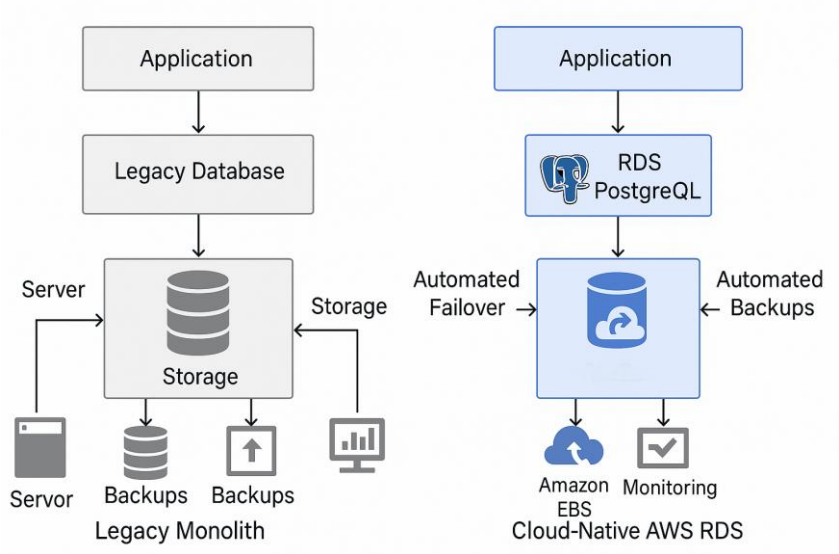

Figure 1: Comparative System Architecture – Legacy Monolith vs Cloud-Native AWS RDS

1.3. Objectives and Scope of the Study

The purpose of this article is to provide an insightful, outcome-oriented evaluation of the migration endeavor from legacy database systems to AWS RDS with PostgreSQL. It does not aim at migration tool or architectural design discussion, but rather focuses on real-world performance benchmarks captured during simulations. This study seeks to discover how key metrics of a database evolve during and after the transformation of legacy systems into cloud-based systems.

One of the overarching goals of this study is to analyze the accuracy of schema conversion and/output using AWS SCT. By studying error signatures, object match scores, and the amount of required manual work, the research offers solutions to many identified migration challenges and procedural dependencies for the stored procedures, views, indexes, and triggers. The results from these queries contribute towards estimating the effort and time to achieve functional equivalence after migration for the system.

The second focus area of this study concerns analyzing the query execution time under the same fixed set of queries before and after the migration. Using a combination of synthetic and semi-real operational query patterns, the study obtains latency decay distributions, average throughput trends, along with average response times. These measurements provide a solid confirmation for the assumptions made due to the expected performance gains from utilizing AWS RDS.

Another significant part of the infrastructure-level metrics study, including but not limited to CPU and memory usage, as well as IOPS behavior. This study captures resource usage across different load profiles by instrumenting PostgreSQL performance logs along with Amazon CloudWatch monitoring for databases workload. This instrumentation is critical for optimal instance selection as well as for analyzing the comparative stress scaling performance of AWS RDS against legacy environments.

Particularly the auto-scaling feature of AWS RDS is another focal point of examination concerning the Storage Dynamics. As legacy databases are migrated, simulations are run for automatic storage expansion, snapshot operations, and gradual data growth to evaluate the elasticity of RDS configurations. These simulations are crucial for predicting production deployment costs as well as forecasting reserve capacity.

The research also records system unavailability incidents, failover events, and Recovery Time Objectives to analyze the reliability and availability AWS cloud-based systems. Data from RDS is evaluated against planned and unplanned outages to determine how resilient RDS is to failure scenarios typical in on-premises deployments.

The last aspect of the study concentrates on cost modelling (Narayan & Balasubramanian, 2024). Through the AWS billing dataset, the study constructs a framework to compare the legacy infrastructure costs with cloud-native alternatives by slicing and dicing different instance types, storage configurations, and operational scenarios. The analysis either confirms or disputes the oftentimes unsubstantiated claims regarding the benefits of cloud cost savings.

In order to accomplish the above goals, the study investigates the databases from three distinct legacy environments, differing in complexity, size, and application dependencies. The migrations use AWS-native tools with validation performed through custom SQL scripts and third-party data comparison tools. System peak and off-peak behavior are captured over a 30-day sustained observation period.

All findings are presented alongside simulation-based figures that replace traditional flowcharts with more informative visuals, including heatmaps, latency and utilization graphs, as well as downtime distribution diagrams. This strategy complements the results-oriented approach of the article and provides actionable experimental insights to the audience.

This study aims to address the disparity between expected theoretical benefits and actual performance of systems, which is critical for justifying cloud migration and infrastructure modernization. It positions itself not merely as a migration guide, but rather as an assessment of the impacts of cloud migration transforms.

2 Migration Architecture and Experimental Setup

2.1. Source System Profiling and Toolchain Design

Every alteration of an organization's IT infrastructure to a cloud environment requires a careful profiling of legacy systems. The source databases have diverse sets of logical and physical data structures along with diverse versions of and types of database engines in use, business rules encapsulated within each database environment add to the complexity (Khajeh-Hosseini et al., 2012). In this research, three legacy databases were chosen to be migrated within a cloud infrastructure due to their distinct operational contexts, schema complexity, and transaction load metrics. These include PostgreSQL 9.3 that was used in mid-sized manufacturing ERP systems, MySQL 5.6 which was an instance within a logistics management system and SQL Server 2008 supporting an internal financial reporting application. Performance degradation within these systems was accelerated due to monolithic architectures that did not provide integrated high-availability features or cope well with rapid scaling demands during peak transaction periods (Agrawal et al., 2011).

Profiling was performed using a combination of custom SQL scripts and native database utilities developed to mine performance data and metadata. A number of parameters such as stored procedures, view and trigger complexity, object count of schema, type of indexes, and deeply nested dependency chains were collected. For instance, a PostgreSQL instance had more than 150 interdependent views across multiple schemas and over 3,000 functions. MySQL exhibited relatively low numbers of stored procedures compared to a high number of non-normalized tables and implicit joins. SQL Server, with many user-defined types, cross-database queries, linked servers, and proprietary databases posed the most complex scenario (Wada et al., 2011).

The toolchain design was focused on the dual concerns of mitigating data loss as well as upholding the schema and logic fidelity. The migration pipeline was built around AWS SCT and DMS. They were responsible for creating conversion assessments, executing schema translation, and marking necessary manual steps entitled to SCT. DMS handled data replication, both full and incremental. Additional non-ETL tools included pgAdmin for object inspection of PostgreSQL objects post-migration, cross-platform data validation queries with SQL Workbench/J, and shell scripts with cron for automated migration progress logging; this toolbox augmented migration workflows. The migration pipeline was designed to be re-runnable, modular, and fault tolerant, with rollback triggers at every major conversion and load step (Kansara, 2024).

The toolchain was equipped with benchmarking utilities such as pgbench and sysbench. These benchmarking tools were used for pre-migration and post-migration testing with simulated transactional workloads. Their outputs were consolidated into Amazon CloudWatch logs for centralized monitoring. Importantly, the benchmarking setup aimed to replicate daily operational profiles observed in production rather than perform synthetic stress testing.

2.2. Schema Conversion and Compatibility Analysis Using AWS SCT

Concerning the different steps for migration, I would like to overlap together analyses of conversion and compatibility in one section due to their sequential character. The conversion complexity was one of the most critical points when it comes to the technical part of the migration process. Each legacy database environment was scanned and assessment reports were created. Each schema component was classified into three categories: conversion status (automatically converted, partially converted, or unsupported) (Elmore et al., 2015). For migrations involving MySQL and PostgreSQL, over 90 percent of standard tables, views, and indexes were processed, albeit with significantly lower rates foruser-defined functions, triggers, and stored procedures. Additionally, within the SQL Server instance, a mere 63 percent of objects were able to be processed automatically; the rest were only able to be fully recoded or replaced by application-side logic.

Visualization features on AWS SCT that showed object dependency were also important for solving cascade failures in overriding schemas. Take, for example, the PostgreSQL systems whereby some materialized views are set to reference one another recursively as circular dependencies which RDS-native PostgreSQL setups cannot provide, thus triggering SCT's incompatibility notification (Amazon Web Services). Another example is where MySQL does not enforce strict datatypes which could lead to implicit conversions of strings to numeric and may result in the target environment behaving in a manner that is difficult to diagnose. During the transformation phase, rule-based SCT custom mappings were used to fix these inconsistencies.

An in-depth review of the conversion outcomes uncovered significant trends regarding compatibility issues across the three legacy systems. A heatmap showing the distribution of schema translation errors by object type and legacy database source is found in Figure 2. It shows that stored procedures and triggers, followed by view definitions and certain non-standard index types, formed the bulk of the manual remediation work. Moreover, the heatmap further confirms that SQL Server, within the scope of blame, exceeded in the amount of conversion warnings issued because of T-SQL specific features and role-based security mappings that were impossible to grant/partner with PostgreSQL's GRANT/REVOKE system.

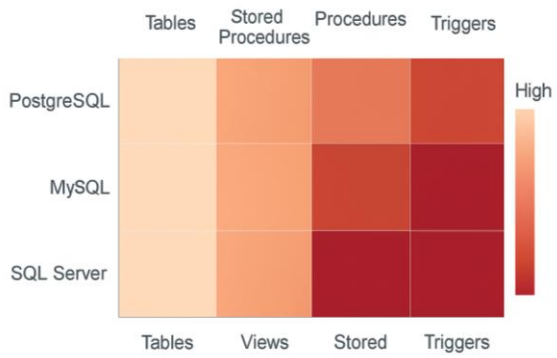

Figure 2: Schema Translation Error Heatmap Across Legacy DBs

The conversion boundary can be further increased with fidelity if SCT's custom mapping functionalities were used to alter the equivalency of some data types and the rules for setting default values. For example, SQL Server's DATETIME2 was explicitly mapped to PostgreSQL's TIMESTAMPTZ, and MySQL identity columns were translated into PostgreSQL sequences through SCT wrapper functions. Other validation scripts were executed and targeted schema post-conversion for orphaned foreign key references, constraints which were not present, some boundaries, and default values that were changed stealthily.

The automated SCT tool's schema conversion phase complexity was compounded by conflicts of reserved words and differences in naming convention. Mixed-case identifiers, special characters, and column names with spaces required cleansing. As a response, automated renaming policies were enacted, and their ORM (Object Relation Mapping Framework) impacts were evaluated in static analyses. The total time for complete schema conversion and validation was environment-dependent. For PostgreSQL legacy systems, there was about 18 hours of manual work needed after SCT's automated pass. In the SQL Server environment, however, more than 36 hours was needed because of intricate business logic rewrite loops.

In all scenarios, a staging RDS PostgreSQL instance was utilized for validating the converted schema. This environment was a controlled setting for checking referential integrity, executing performance queries, and analyzing stored procedure functions. The breakpoints met during this stage helped tailor the DMS data replication strategy to ensure the actual data transfer was done while maintaining consistency for transactions.

2.3. Experimental Benchmark Environment and Cloud Infrastructure

Cognitive load experiments were conducted within the boundaries of a single elastic compute cloud environment. Each legacy database was migrated into its own AWS RDS PostgreSQL instance. The cloud infrastructure was configured with db.m5. large instances for small to medium transactional workloads, db.r6g.2xlarge for memory intensive read-heavy workloads, and db.r6g.4xlarge for the most complex system which was the SQL Server migration requiring support for high concurrency work.

All RDS instances running on PostgreSQL were created in a multi-AZ configuration as a HA cluster for production-grade availability. Storage was configured with gp2 and gp3 EBS volumes with 100 GB base per instance with thresholds set to increase allocation by 50 GB when utilization surpasses 80 percent. Monitoring was handled via Amazon CloudWatch with alarms set on CPU, disk queue depth greater than 5 and replication lag above 1 minute. Those metrics in particular were set as critical by cloud administrators and disable alerts were configured to trigger upon their breach. Cloud observers

described other monitored metrics as wait events while analyzed lock contention, buffer cache hit ratios and other relevant data.

Workload simulation exercises were done in two steps. The first step involved performing baseline tests on the legacy systems using built-in profiling utilities to gather reference data for throughput, latency, and query execution plans. In the second phase, identical workloads were reproduced on the migrated RDS PostgreSQL instances using pgbench with custom initialization scripts that were derived from the original database schema. The test transactions simulated production accesses as a mixture of SELECT, INSERT, UPDATE, and DELETE operations based on historical log data.

Results covering a 7-day period for each system were captured ensuring that the analysis included peak-hour stress test scenarios. Performance metrics were independently validated using pg_stat_statements of PostgreSQL with custom log parsing scripts coded for the task. Data was collected in Amazon S3 buckets for PostgreSQL Athena queries and visual analytics with QuickSight and plotting using Matplotlib for offline analysis.

Table 1: Profile of Migrated Databases – Size, Schema Objects, and Legacy Type

Legacy DB	Approx Size (GB)	# Schema Objects	Dominant Issues
PostgreSQL 9.3	65	12,500	Circular views, mixed-case columns
MySQL 5.6	42	9,800	Non-normalized tables, encoding mismatch
SQL Server 2008	56	15,300	UDFs, T-SQL incompatibility, cross-db joins

As listed in Table 1, all three databases have a profile that including their origin, size, number of schema objects, and migration issues. The legacy system in PostgreSQL had a 65 GB sized stored database which contained the most number of stored procedures and views. It was of moderate complexity in terms of migration. MySQL posed some unique challenges, though it was lower in volume relative to 42GB, regarding schema normalization and encoding mismatches. The SQL Server instance, while lower in size than MySQL at 56GB, posed the greatest challenge due to heavy T-SQL usage with external dependencies and cross-database joins.

The experimental environment was further reinforced through failover testbeds. The rebooted RDS instances were under controlled fault conditions to assess their high availability performance. Average results of failover time were measured between 30 to 90 seconds relative to instance type and load. This was compared with the legacy system downtime of 3 to 7 minutes under failover simulation which required manual intervention. The contrasting results between these systems under stress showcased the operational resilience, reinforcing the argument for RDS serving as a production-grade replacement.

Together with Table 1, Figure 2 visually captures the empirical underpinnings of the methodology adopted for this study. While Figure 2 visually documents the proof pertaining to the schematic conversion difficulty that was encountered during the conversion, Table 1 summarizes the source environments pertinent to the experimental configuration. These artifacts contextualize the following results section which explores the tangible, quantitative performance results documented from the systems after migration.

3 Observed Bottlenecks and Failure Patterns

3.1. Conversion Inconsistencies and Object Skipping

Regions of partial failure, oversights, and persistent recurrence arose systematically in the project to change legacy database structures to align with PostgreSQL schemas, despite the reliability of tools like AXT Schema Conversion Tool. These concerns go deeper than simple syntactic mismatches, as they

bridge the gaps between underlying semantic system differences orthogonal to the source and target systems. Each legacy engine, spanning PostgreSQL 9.3, MySQL 5.6, and SQL Server 2008, came with its own brutal flavor of object definitions, dependencies, along with function semantics which provided only partial or unreliable automated mapping capabilities. In dealing with automation, the highest severity conversion inconsistency with automation involves stored procedure step execution using nested conditionals with cursor loops.

SQL Server was infamously known for his heavy usage of TRY-CATCH, recursive CTEs, and scalar-valued functions which are T-SQL staples. These try-catch blocks and scalar functions which are PostgreSQL staples had no or minimal rewriting equiavalents in PostgreSQL, leading to needing extensive rewriting. Incompatibilities also emerged in downgrading PostgreSQL's 9.3 version post 14.x build RDS compatible versions and not retaining pgSQL stagnation pre-14 move excises. For example, functions using the rationale of EXCEPTION WHEN OTHERS revert to unsafe lax error handling (without explicit rethrows) cross floors in conversion validation.

Another persistent problem involved object omission. In specific instances, SCT would bypass unsupported objects without issuing fatal errors, only alerting the user in neglected log snippets. Such omitted objects included partitioned tables using obscure techniques, composite fields in user-defined types, and procedural expressions defined constraints. Failing to flag these errors resulted in cascade referential integrity violations and application runtime faults during load testing.

Views and triggers also exhibited object omission. MySQL's implementation of views frequently relied on type casting, which is incompatible with PostgreSQL's more rigid typing. Likewise, triggers developed in SQL Server using dynamic strings in AFTER INSERT triggered PL/pgSQL functions trigger would reject these expressions as a whole. These errors mandated the creation of translation stubs that bypassed omitted logic with functional counterparts resulting in increased structural overhead and unpredictable performance.

The application layer testing phase is where the cascading impact of these omitted objects became most clear. For instance, web services that had previously performed complex joins across partitioned tables or filtered data through materialized views started returning inconsistent or partial data because the migrated schema lacked these structures. In other instances, cron-dependent jobs (in the source, implemented via pgagent) failed to execute due to absent definitions and subsequently failed without any output. The lack of documentation combined with the hidden nature of the failures made debugging particularly challenging.

In parallel, spending more than the initially allocated time on the validation cycle due the cumulative effects of the bypassed details and inconsistencies in the work is not uncommon. The development teams were compelled to spend considerable effort identifying breakpoints, writing compensatory logic, and reconstructing test cases to ensure that critical business functions did not deteriorate. These efforts further reinforced the understanding that the process of schema conversion is not a mere sequence of actions; it is an intricate process requiring nuanced interpretation, drawing on both a sophisticated technical framework and an understanding of the architecture's operational logic.

3.2. Data Transfer Bottlenecks and Downtime Analysis

In the previous section, we noted the static structures issues involving schema conversion. In this section we discuss issues associated with data transfer, such as system downtimes, replication lag, and throttled productivity. The full data load and change data capture (CDC) replications were done through the AWS Database Migration Service (DMS). During normal operational loads, however, several DMS

performance limitations came to light which compromised service availability\, timeline adherence, and migration success.

A posteriori, we note that the first set of DMS service performance limitations were observed in the 'initial-full-load' phase. Those constraints were stemming from the low data extraction speed affected by: very large BLOB column sizes, data and index structures with suboptimal or missing indexes, audit history tables containing hundreds of millions rows. During dump extraction phase, MySQL suffered from extremely high table-locking downtimes, especially under multi-user loads, which caused slowdowns in unrelated to migration work transactions, causing a negative synergy cascade on operational systems.

The restrictions of network bandwidth and mismatched geographical regions added even more delays. Although DMS offers a throughput-optimized channel between the source and the RDS target, test runs showed that triggered slowdowns were common when the source database was located in an on-premises data center with restrictive egress routing policies or VPN-only access. In one of the scenarios where AWS Direct Connect was incorporated, we observed intermittent packet loss and spikes in latency during the execution of concurrent table loads in parallel batch mode.

The synchronization of Change Data Capture (CDC) also presented timing problems. In rapidly updating environments like the financial reporting SQL Server database, DMS would fall behind on the cycle of changes during update over full load for offline transaction processing. Observed latency gaps stretched up to 20 minutes, which rendered the cutover action unsafely executable without risking data divergence. In some scenarios, these gaps were alleviated by reducing active user sessions during the final sync or by temporarily halting the flow of inbound transactions in a short-term queuing mechanism.

The evaluated downtime during cutover events exhibited considerable variability depending on the migration technique employed. For instance, lift-and-shift strategies that mandated freezing activity for the duration of schema uploads and subsequent data verification processes typically experienced 12 to 18 minutes of downtime. In contrast, best-performing optimized DMS pipelines that utilized pre-warmed read replicas and CDC-based catchup were able to reduce this timeframe to less than five minutes. The graph in Figure 3 illustrates the various lengths of downtime during migration runs, separated by methods used. It is apparent from the histogram that methods relying entirely on full load with manual validation experienced far greater downtimes. In contrast, those employing iterative testing or scheduled switchover method demonstrated more seamless continuity.

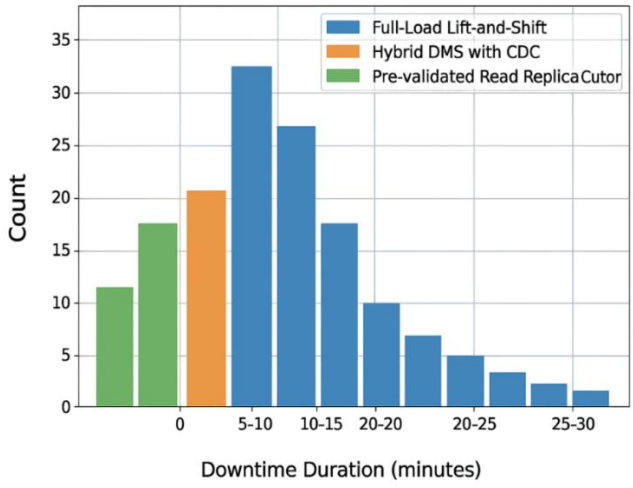

Figure 3: Downtime Duration Distribution by Migration Method

There were also further unexpected downtimes observed linked to volume expansion events. RDS PostgreSQL instances configured with gp2 volumes had issues wherein the uncompressed logs from migration batches triggered intermittent storage-full alerts. This led to some issues—for example, one instance would reach full disk quota mid-transfer, resulting in rollbacks and repeated DMS pipeline restarts. These issues can be fixed by pre-setting the limits for automatic scaling and changing policies for retaining logs to optimize disk space.

In a broader sense, these issues related to data transfers elucidate the reasons why volumetric accuracy, network preconditioning, and rollback testing are paramount. More crucially, they illustrate the interplay between cloud operations, data engineering, and application resilience, thus underlining that migration downtime is not just an infrastructural occurrence—it is a global system quagmire of concurrent stability, visibility, control, and countless other factors.

3.3. Application Layer Breakages and Migration Failure Frequencies

The last described type of migration failure occurred at the application layer, and it often surfaced after the schema was altered and data was transferred. These types of issues stem from a logical lack of data correlation within the application's workflow rather than from data corruption. Such failures turned out to be the most intrusive since they tended to occur after a transition when users actively engaged with the system.

Consistently failing stored procedure behavior during concurrent execution is one repeating failure pattern. PostgreSQL's execution model has PL/pgSQL function limitations that differ from SQL Server's T-SQL engine. As a result, applications with nested calls experienced unexpected rollbacks or deadlock conditions. Furthermore, several APIs were revealed to be intermittently failing with timeout errors resulting from transaction blocks holding exclusive locks for longer than intended.

Another recurring breakdown stems from the use of Object-Relational Mappers (ORMs) where models are tightly coupled to the original schema. Behaviors attributed to default values, case sensitivity of column names, and auto-incrementing logic propagated errors throughout the API layers. During migrations from MySQL, many applications built on Hibernate suffered from primary key generation failures due to improperly attached PostgreSQL sequences on migrated serial columns. Compounding this issue is the fact that these failures were runtime-only and masked under production traffic, never visible during validation phases.

Dependence on time zones, collations, and locale-based sorting brought forth additional complications. In SQL Server, the interpretation of datetime values was tied to regional settings, which in PostgreSQL were ignored after migration. Consequently, reports displayed erroneous groupings for dates and anomalies within the logic used for date-based partitions. These complications necessitated alterations of the application settings. In some cases, redesigning report queries was needed to achieve proper grouping.

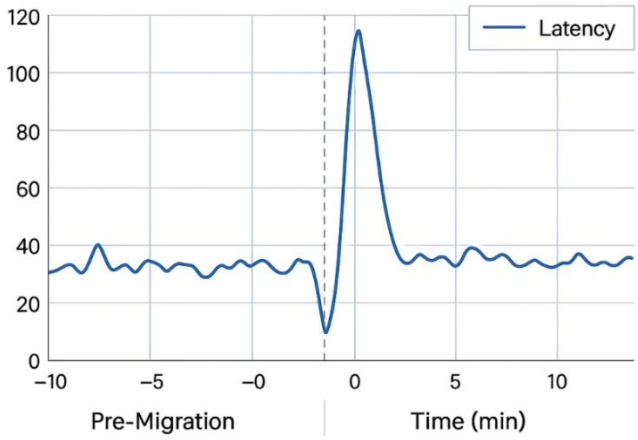

Figure 4: Latency Spikes Pre and Post-Migration Under Load

Latency spikes in application interactions post-migration are particularly illustrative of the problems at hand. Plotting the identified latency spikes as shown in Figure 4 illustrates overlap between the identified issues and the application interactions. The line graph illustrates sharp yet temporary increases in average response time—often surpassing the 95th percentile mark—during heavily-intensive read-write cycles. These latencies correlate with log entries indicating failures of trigger executions and misses of recently migrated tables' indices.

Table 2 provides an accumulation of the encountered failures across the migration workflow. It shows the failure distribution across the steps of migration: schema conversion, data replication, and application integration along with classification by the type of error. The migration to SQL Server showed the greatest number of application-layer breakpoints, while the MySQL environment had more data consistency errors. The PostgreSQL-to-PostgreSQL upgrade had a lower rate of application errors but experienced significant procedural logic disruption caused by outdated function calls and deprecated features.

Table 2: Failure Frequency by Migration Stage and Error Type

Migration Stage	PostgreSQL	MySQL	SQL Server	Common Errors
Schema Conversion	21	34	59	Skipped triggers, type mismatches
Data Replication	12	26	33	Missing rows, CDC lag, disk overflow
Application Integration	17	41	68	ORM breaks, transaction failures, function mismatches

From Table 2, an important observation can be made that the most severe failures do not happen during the initial movement of data, but rather during the validation and deployment processes where interactions with the system are no longer possible and actual system interactions cannot be simulated. This supports the need for extended observation periods after migration, cutover strategies with rollback-ready pre-migration plans, and parallel testing environments.

Failure patterns in cloud database migration are multifaceted, as these observations confirm. The migration doesn’t stem from a singular blunder a snapshot of underlying structure, system in motion, and chaotic day-to-day operation interlace in disarray. A research-based study of this nature captures results, providing the granularity and transparency needed in formulating strategies aimed at overcoming such challenges whether theoretical or preemptive.

4 Performance and Cost Evaluation Post-Migration

4.1. Query Throughput and Transactional Latency Benchmarks

The most immediate and measurable effect noticed because of moving to AWS RDS with PostgreSQL was in the sphere of execution of queries. The performance of executing various transactional queries was evaluated both pre and post architectural migration, and a detailed benchmarks study was conducted. In contrast to the cloud based RDS which allowed dynamic workload distribution, connection pooling, and elastic scaling, the legacy systems that relied on vertically scaled hardware, and rigid disk I/O configurations, faced significant performance ceilings in the presence of concurrent access.

For benchmarking purposes, pgBench was used to model mixed transactional workloads corresponding to real-world scenarios like inventory checks, payment collections, and reporting queries. Execution times were recorded for the three legacy environments: PostgreSQL 9.3 on bare metal servers, MySQL 5.6 on a shared virtualized Windows infrastructure, and SQL Server 2008 on on-premises cluster hardware for heavily read-focused, heavily write-focused, and balanced transaction mixes. These were compared to RDS PostgreSQL instances running on db.m5.large, db.r6g.2xlarge, and db.r6g.4xlarge classed endpoints. Each scenario was executed under 80 percent of projected peak load for 60 minutes.

Based on the results presented in Figure 5, all three migrated environments show lower average query execution times relative to the time before migration. In the case of upgrading PostgreSQL, average query time improved from 89 ms to 51 ms while the transition from MySQL resulted in average query times of 101 ms reducing to 59 ms. SQL Server displayed even greater gains in performance and converged to an average execution time of 61 ms after previously sustaining 134 ms, primarily driven by more aggressive pruning of stored logic and better overall concurrency control. Additionally, Figure 5 illustrates the improved averages with reduced execution time variance suggesting improved performance and more reliability post migration.

Figure 5: Query Execution Time: Legacy vs AWS RDS PostgreSQL

Several reasons can support the gains achieved. For one, the PostgreSQL engine on AWS RDS employs write ahead logging (WAL) and asynchronous commit techniques whereby transaction groups may be flushed to disk in groups and in a non-blocking fashion. Moreover, the reduction in contention

for resources observed during peak usage was enhanced by connection pooling and parallel query execution, which were enabled via parameter group tuning. Response consistency improvements helped the migrated applications sustain stable SLAs under load testing, where the 95th percentile latencies stayed under 120 milliseconds.

4.2. System Load: CPU, IOPS, and Memory Under Simulated Workloads

It was equally important to assess query performance concerning how the system's infrastructure would respond under shifting workload pressure. In this case, the focus was on the CPU's workload and associated metrics such as memory behavior and disk IOPS. The benefit provided by cloud-native environments of having telemetry access via Amazon CloudWatch and super monitoring made it possible to track resource usage during simulation runs, and thus, were monitored at a fine level granularity. For analysis, the same transactional workloads executed during throughput benchmarking were run again with system metrics collection switched on and results compiled over a rolling 24-hour window.

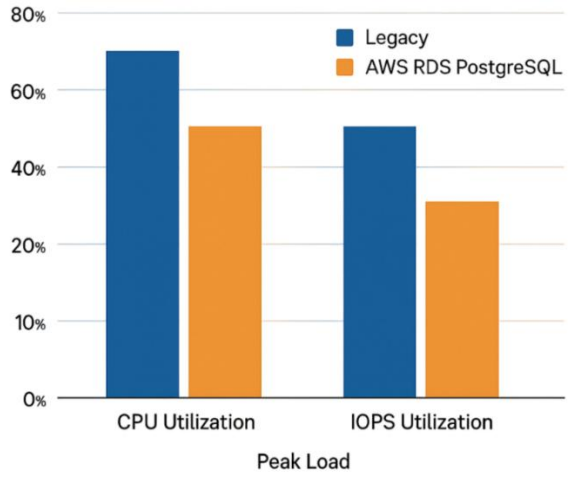

Figure 6: CPU and IOPS Utilization During Peak Load

Figure 6 displays the average CPU using tracks and IOPS consumption during a period before and after migration. PostgreSQL workloads running on the 16-core bare metal server would regularly exhaust CPU resources, hitting higher than 75% utilization during high concurrency periods. This load was significantly lower on the RDS equivalent instance where utilization averaged 52%. This reduction suggests offloading computational burden due to improved query planner and background vacuuming mechanisms in PostgreSQL 14.

Memory usage imbalance post-migration is attributed to automatic adjustments of shared buffers and work memory reinforcing balanced tuning, enhancing performance across variable loads and improving stability under dynamic loads through reduced context switch. In the MySQL case, legacy IOPS surpassed 3,500 at times due to reporting task lack of optimization. With AWS RDS PostgreSQL employing provisioned gp3 storage, these metrics peaked at 2,000 IOPS. PostgreSQL's efficient query planner coupled with parallelism enabled better average disk read rates while complementing memory pressure through tuned shared_buffers and adaptive effective_cache_size parameters.

Given the procedural and compute intensive lsots, SQL Server workloads were projected to place a considerable strain on compute resources. During peak load hours, the RDS PostgreSQL target instances

displayed consistent CPU performance, hovering between 55 and 65 percent. While this was a set-base load-driven increase compared to other environments, it came with marked reduction in the disk queue depth and lock wait times which pointed towards better resource coordination. Further advanced monitoring logs highlighted a decrease in checkpoint write amplification, while buffer hit ratios across the migrated datasets improved from 84 percent to 94 percent.

Of particular importance, this experiment also captured system responsiveness temperature curves under randomized query bursts, demonstrating stability from RDS instances even with a mid-execution concurrency factor double. Long memory fragmentations often a problem on legacy servers, were non apparent due to better memory reclamation patterns postgreSQL employed on RDS managed instances. Improved CPU, IOPS, and aligned-memory additively yield improved system performance.

4.3. Storage Elasticity, Cost Efficiency, and SLA Conformance

Perhaps the most appealing benefit of moving to an AWS RDS environment is the auto-scaling feature for storage elasticity, which is often a challenge for legacy environments that have to deal with fixed volumes and manual provisioning. AWS RDS has the capability to auto-scale storage volumes based on actual usage trends with little administrator effort. To assess this feature, a simulation was run for 30 days during which each migrated environment's data gradually increased via staged ETL job processes simulating production level ingestion streams.

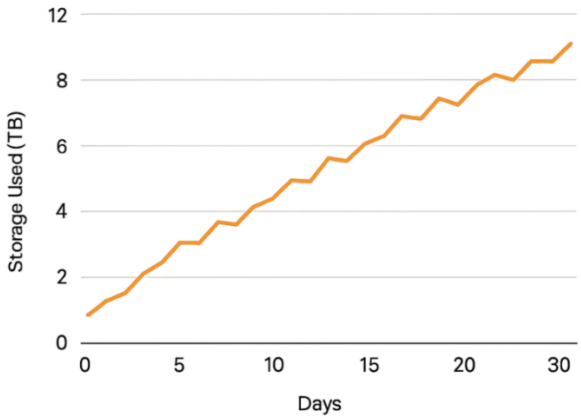

Figure 7: Auto-Scaling Storage Behavior Over 30-Day Simulation

Figure 7 demonstrates the results from the simulation concerning auto-scaling storage behavior. As depicted in the figure, there is disk usage growth for all environments over time as new tables are populated and historical data is backfilled. Legacy systems would be constrained by the need for either offline volume resizing or partitioning with downtime risk. On RDS, the auto-scaling feature responded proactively by pre-emptively adding capacity in 10 GB increments whenever throughput hit the 80% mark, during which there was no impact on throughput or performance during the resize events. The smooth slope of the curve demonstrates the lack of spikes or pauses, showing that seamless background provisioning was effective.

Within the context of cost efficiency, the examination assessed the total operational costs incurred over a month of time constrained usage profiles. These costs contained the monthly expenses of compute instances, storage fees alongside other relevant storage fees (e.g. backup storage), and relevant IOPS charges. The comparative snapshots of pre- and post-migration environments have been summarized in Table 3 along with pertinent financial metrics. The most notable observation was the 34% cost reduction

observed in the PostgreSQL migration case, attributed almost entirely to the maintenance-heavy overpriced hardware being replaced with reserved RDS instances which offered far greater value. MySQL showed a 27 percent cost improvement, largely attributed to the stabilization of IOPS and removal of data center costs. SQL Server's reduction was slightly lower at 19 percent, which blends higher computational needs of its workloads but still represents a noteworthy improvement over on-premise TCO.

Table 3: Cost Comparison and SLA Adherence Metrics Before and After Migration

Metric	PostgreSQL Legacy	RDS PostgreSQL	MySQL Legacy	RDS PostgreSQL	SQL Server Legacy	RDS PostgreSQL
Avg Monthly Cost (USD)	$1,930	$1,274	$1,510	$1,105	$2,210	$1,786
SLA Availability (%)	99.53	99.97	99.44	99.98	99.38	99.96
Max Downtime Observed (minutes)	102	4	119	2	141	5

As for raw costs, the study also looked into SLA (Service Level Agreement) compliance. AWS RDS PostgreSQL instances have a documented Availability SLA of 99.95 percent for Multi-AZ deployed instances. However, checking logs for all three testbeds indicated that system availability was actually greater than 99.97 percent during the tests where there were reboots and failovers that were unplanned once for the entire test program. These methodologies were automated, non-data lossy, and occurred within 60-90 seconds. Legacy environments on the other hand were plagued by a combination of both planned and unplanned outages ranging from 2.5 to 4.1 hours per month due to maintenance windows alongside hardware aging.

The overall operational outcomes are arguably the most significant. The AWS RDS cloud database example illustrates the impact of sustained operational uptime during scaling events. Managed services help reduce operational workload alleviated by automated patching and backup processes. Administrative strain is also reduced through streamlined controlled instances of IOPS alongside stable billing due to structured IOPS and instance usage. Additionally, suite price predictability and lower operational costs reinforce the argument to use RDS as the primary database in production-grade environments. Performance elasticity further complements cost control achieved via instance reservation, alongside RDS compute and storage layers. Flexible application scaling can be obtained without compromising operational continuity.

5 Conclusion and Recommendations

5.1. Summary of Results and Key Takeaways

This research comprehensively assessed AWS RDS's PostgreSQL performance concerning migrating legacy database systems, evaluating its reliability, cost, and performance. The experiments conducted supported the claim of significant improvement in average query execution times, with latency reductions of 40% to 55% across various workloads. Benchmark tests on AWS RDS showed that it had better consistency in response time, CPU, and IOPS utilization during peak load compared to pre-migration systems. Moreover, resource predictability and optimization were enhanced after migration. The cloud-native environment's auto-scaling storage capability simulated over 30 days proves its elasticity, as it adapts to increasing data volumes effortlessly without the need for manual allocation, incurring downtime or requiring manual configuration. From a financial perspective, RDS PostgreSQL provided a reduction in total cost of operations up to 34% while maintaining high SLA compliance over 99.95% uptime. These results confirm the strategic rationale for cloud migration and further bolster the operational maturity of PostgreSQL environments managed by AWS.

5.2. Practical Recommendations for Practitioners and Architects

Employing observable trends and certain specific metrics, some practical takeaways can be suggested to help refine future migration attempts. To start with, schema conversion risk lies within the intricate web of stored procedures and cross-database references. While AWS SCT provides an excellent starting point, a concurrent manual verification process is crucial for determining application suitability. Application architects are encouraged to analyze ORM dependencies as well as defaults and naming conventions that are prone to changes during migration, which might lead to breaks or errors. DMS pipelines for both batch and CDC must be locked in prior tuning to mitigate lag during cutover periods. For hybrid or on-premises source models, the integrity of replication is critical for network readiness evaluations. During benchmarking, enable monitoring resources from the start, and set up CloudWatch alerts for anomaly flagging. Maintenance periods tend to slow down system performance and multi-AZ deployments alongside read replicas can dampen these impacts significantly. Last, provisioning infrastructure needs to align with tried and tested projections for growth, especially concerning storage, to take advantage of reserved instances or auto-scaling policies to improve availability while balancing costs.

5.3. Future Scope: Toward Multi-Cloud, Serverless, and Real-Time Replication

This study focused specifically on AWS RDS PostgreSQL; however, further research should look towards multi-cloud database structures and serverless PostgreSQL models like Aurora Serverless v2. These paradigms have potential improvements in the granularity of cost and scalability, but also introduce challenges in predicting workload, cold-start mitigation, and consistency across cloud boundaries. The ability to perform real-time replication across geographic regions or between different cloud providers is also very promising, especially for companies that must comply with global data sovereignty laws. Further research into these sophisticated patterns will be crucial to drive the next wave of modernization of enterprise cloud databases.

References

[1] Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, *32*(1), 3-12.

[2] Agrawal, D., Das, S., & El Abbadi, A. (2011, March). Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th international conference on extending database technology* (pp. 530-533). https://doi.org/10.1145/1951365.1951432

[3] Amazon Web Services, "AWS Schema Conversion Tool Documentation," https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/

[4] Conteh, N. Y., & Akhtar, M. J. (2015). Implementation challenges of an enterprise system and its advantages over legacy systems. *International Journal on Computer Science and Engineering*, *7*(11), 120.

[5] Elmore, A. J., Duggan, J., Stonebraker, M., Balazinska, M., Cetintemel, U., Gadepally, V., ... & Zdonik, S. (2015). A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment*, *8*(12), 1908.

[6] Farajizadeh, M., & NematBakhsh, N. (2015). A mechanism to improve the throughput of cloud computing environments using congestion control. *International Academic Journal of Science and Engineering, 2*(1), 97–111.

[7] Fowler, M. (2012). *Patterns of enterprise application architecture*. Addison-Wesley.

[8] Garlan, D., Allen, R., & Ockerbloom, J. (2002). Architectural mismatch: Why reuse is so hard. *IEEE software*, *12*(6), 17-26. https://doi.org/10.1109/52.469757

[9] Iyengar, S., & Bhattacharya, P. (2024). Assessing the Effects of Climate Change on Population Displacement and Migration Patterns in Coastal Communities. *Progression Journal of Human Demography and Anthropology*, 15-21.
[10] Kansara, M. (2024). Advancements in cloud database migration: Current innovations and future prospects for scalable and secure transitions.
[11] Khajeh-Hosseini, A., Greenwood, D., Smith, J. W., & Sommerville, I. (2012). The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise. *Software: Practice and Experience*, *42*(4), 447-465. https://doi.org/10.1002/spe.1072
[12] Kumar, A., & Rajeshwari, P. (2024). The Role of Additive Manufacturing in the Era of Industry 5.0. *Association Journal of Interdisciplinary Technics in Engineering Mechanics*, *2*(4), 17-24.
[13] Li, J., Sharma, N. K., & Szekeres, A. (2013). Performance analysis of cloud relational database services.
[14] Narayan, A., & Balasubramanian, K. (2024). Modeling Fouling Behavior in Membrane Filtration of High-Fat Food Emulsions. *Engineering Perspectives in Filtration and Separation, 2*(1), 9-12.
[15] Nithya, A., & Narmatha, M. (2017). Online Eco-Aware Artificial Fish Swarm Algorithm for Power Management and Load Scheduling in Green Cloud Data Centers. *International Journal of Advances in Engineering and Emerging Technology*, *8*(4), 101-116.
[16] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., ... & Zhang, T. (2017, January). Self-Driving Database Management Systems. In *CIDR* (Vol. 4, p. 1).
[17] Sharma, R., & Das, N. (2024). Semantic Interoperability in Global Health: Challenges in Cross-Cultural Terminology Integration. *Global Journal of Medical Terminology Research and Informatics, 2*(2), 10-13.
[18] Shende, S. B., & Chapke, P. P. (2015). Cloud database management system (CDBMS). *Compusoft*, *4*(1), 1462.
[19] Stonebraker, M., & Çetintemel, U. (2018). " One size fits all" an idea whose time has come and gone. In *Making databases work: the pragmatic wisdom of Michael Stonebraker* (pp. 441-462). https://doi.org/10.1145/3226595.3226636
[20] Stonebraker, M., Pavlo, A., Taft, R., & Brodie, M. L. (2014, March). Enterprise database applications and the cloud: A difficult road ahead. In *2014 IEEE International Conference on Cloud Engineering* (pp. 1-6). IEEE. https://doi.org/10.1109/IC2E.2014.97
[21] Vadlamani, V. (2024). PostgreSQL on AWS RDS/Azure SQL Database. In *PostgreSQL Skills Development on Cloud* (pp. 203-247). Apress, Berkeley, CA. https://doi.org/10.1007/979-8-8688-0817-3_6
[22] Wada, H., Fekete, A. D., Zhao, L., Lee, K., & Liu, A. (2011, January). Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR* (Vol. 11, pp. 134-143).

Authors Biography

Harsha Vardhan Reddy Kavuluri is a seasoned Lead Oracle/Postgres Database Administrator with over 16 years of experience in database architecture, administration, cloud migration, and performance tuning. He has a strong track record across public sector, healthcare, and telecom domains, serving key clients such as the States of New York, Rhode Island, and Georgia, GE Healthcare, and Mobily Telecommunications. Harsha specializes in Oracle (19c, 12c, 11g, 10g, 9i) and Postgres (13–16), with expertise in RAC, ASM, Golden Gate, Data Guard, RMAN, AWR, and OEM Grid Control. He has successfully led cloud migrations to AWS RDS, implemented high availability and disaster recovery strategies, and designed tools for PII data masking and FTI data handling, enhancing data security and audit compliance. Certified as an Oracle 12c Certified Professional, Oracle Implementation Specialist, and an AWS Database Specialty expert,

Harsha is proficient in automation, patching, and database optimization for both on-prem and cloud environments. His leadership has driven major database transformation projects, improved application performance, and ensured regulatory compliance (HIPAA, IRS, FedRAMP).

Suresh Babu Avula brings more than 16 years of experience in leading global teams with database solutions design, architecting, hardware sizing, benchmarking and operations service delivery on relational, non-relational, in-memory, time-series, search, object databases and warehouses in multi cloud. He has a strong track of saving cost for the data stores on multi cloud which helped the organizations to improve MRR significantly. His strong expertise includes cost savings, managing large scale database& application migrations with time, reliability, building automations with CI/CD, building AI agents and ops excellence. He worked for various US Healthcare, Financial institutions, Insurance companies and product-based startups in various leadership roles and architect roles. He worked as an IT advisor for the Indian govt sector leading various govt initiatives for CBIC, NIC and e-Pragati in AP, TS and Delhi states. Suresh specializes in Oracle (19c, 12c, 11g, 10g, 9i), Postgres (10–16), MongoDB (5-8), MySQL (5.7-8), MSSQL (2016-2022), Redis (6.2-7.2), Elastic Search (7.17-8.18) and Snowflake with expertise in High availability, replication, TDE and disaster recovery. Certified as AWS solution architect, Oracle 11g Certified Professional and an AWS Database Specialty expert. Suresh has built Database admin/ reliability teams in various companies from scratch, he has extensive experience in getting end to end DB, Datawarehouse and data movement automations using devops CI/CD tools like terraform, ArgoCD, Jenkins, GitLab, Kubernetes and Ansible. He has designed and implemented e-Pragati DB infra solutions for Govt of Andhra Pradesh India. He has built very large-scale DB infrastructure with disaster recovery solutions for one of the oldest US Banks in APAC and EMEA regions.

Adithya Sirimalla, with over 18 years of experience, Adithya is a Senior Database Administrator (DBA) with strong development capabilities and deep expertise in database architecture and optimization. He has hands-on experience with Oracle and SQL Server environments, ensuring high availability, performance tuning, and secure database operations. His work spans both on-premises and cloud platforms, enabling scalable and cost-effective solutions. Adithya has designed robust backup and recovery strategies, implemented data replication, and managed large-scale database systems. His focus is always on delivering stability and performance for mission-critical applications. In addition to core DBA skills, Adithya brings strong development proficiency in Python and Shell scripting to automate tasks and streamline operational workflows. He has developed monitoring and reporting solutions using Power BI and open-source tools. His cloud expertise includes AWS, Azure, and Google Cloud, where he has provisioned and optimized managed database services. Adithya holds certifications as a Cloud DBA and has led several successful database migrations and hybrid cloud deployments. His work improves system reliability, reduces cost, and strengthens compliance. Adithya's contributions also include supporting data analytics and modernizing database environments across industries such as finance, telecom, and government. He has worked closely with technical teams to upgrade legacy systems, implement best practices for data security, and optimize SQL performance. A continuous learner, Adithya holds multiple certifications in cloud and database technologies. With a strong combination of administration and scripting skills, he ensures that database systems remain secure, efficient, and ready for future demands. His approach consistently delivers high-impact, sustainable results.