The American Journal of Applied Sciences
ISSN 2689-0992 | Open Access

TYPE Original Research
PAGE NO. 168-178
DOI 10.37547/tajas/Volume07Issue08-13

Core Data-Management Strategies during Migration to Serverless Aurora Databases

Mykhaylo Kurtikov
Senior Software Developer Austin, United States

OPEN ACCESS

SUBMITED 31 July 2025
ACCEPTED 08 August 2025
PUBLISHED 31 August 2025
VOLUME Vol.07 Issue 08 2025

CITATION
Mykhaylo Kurtikov. (2025). Core Data-Management Strategies during Migration to Serverless Aurora Databases. The American Journal of Applied Sciences, 7(8), 168–178. https://doi.org/10.37547/tajas/Volume07Issue08-13

Abstract: The paper analyses core data-management strategies that ensure a consistent, scalable, and cost-efficient transition from on-premises or monolithic relational databases to Amazon Aurora Serverless. Drawing on recent peer-reviewed research and industry reports, the study first frames serverless Aurora within a microservice-centric architecture, emphasising the "database-per-service" pattern, CAP-theorem trade-offs, and the complementary roles of transactional stores, data lakes, and data warehouses. The second section evaluates mechanisms for maintaining data integrity during and after migration, contrasting ACID guarantees in Aurora with BASE-oriented eventual consistency at the system boundary, and detailing patterns such as sagas and event sourcing for cross-service coordination. The third part (retained in full) offers a practice-oriented synthesis of automation techniques: AWS Database Migration Service for zero-downtime change-data-capture, AWS Schema Conversion Tool for heterogeneous schema conversion, and Infrastructure-as-Code pipelines for repeatable cluster provisioning and continuous delivery. Empirical evidence from large-scale migrations—including multi-billion-row financial and media platforms—is used to quantify benefits (e.g., up to 40 % cost reduction and sub-minute fail-over times) and to highlight common pitfalls. The paper concludes with a set of actionable guidelines that align architectural decisions, consistency requirements, and automation practices, demonstrating that a properly orchestrated move to Aurora Serverless not only preserves, but often enhances enterprise data reliability and agility.

Keywords: *Serverless databases, Amazon Aurora, data migration, data consistency, ACID vs BASE, CAP theorem,* *microservices architecture, AWS DMS, schema conversion.*

Introduction

Modern enterprises are migrating their data and applications to the cloud at an accelerated pace, including mission-critical database systems. Gartner projects that by 2025 up to 85 % of all databases will run on cloud platforms [1]. This transition is driven by demands for elastic scaling, reduced operational overhead, and access to advanced cloud services. A particularly noteworthy innovation is the emergence of serverless database engines, which dynamically allocate resources and charge solely based on actual usage. Amazon Aurora Serverless epitomizes this category: a cloud-native relational database compatible with MySQL and PostgreSQL that can "scale to hundreds of thousands of transactions in a fraction of a second" and automatically adjust its compute capacity to match workload fluctuations without manual intervention [2]. By abstracting infrastructure concerns—such as server provisioning, patch management, and capacity reservation—Aurora Serverless enables teams to focus on business logic while preserving ACID transactional guarantees and robust fault tolerance.

Adopting Aurora Serverless is especially compelling for organizations transitioning from high-throughput monolithic architectures to microservices. Traditional monolithic database deployments (for example, legacy Oracle installations) often require downtime and incur substantial costs when scaling [3], whereas a microservices approach with distributed processing demands a more agile, scalable data backbone. Industry analysts observe that "the next generation of distributed microservices applications requires a centralized cloud database as the backbone for state management and data exchange between services," capable of supporting geo-distributed workloads without sacrificing data integrity [4]. In this context, Aurora Serverless excels, offering virtually unlimited horizontal scalability on a pay-per-use model [2, 5]. Indeed, several large-scale migrations have already been completed successfully: one global platform serving tens of millions of users migrated from a monolithic Oracle database to an Aurora Serverless cluster, achieving faster response times and simplifying future scalability (implementation details are provided in this paper).

This work examines key data management strategies for migrating to Amazon Aurora's serverless architecture. First, it explores architectural patterns for organizing data when moving from monolithic systems to cloud-native microservices, emphasizing the unique characteristics of Aurora Serverless. Next, it analyzes mechanisms for ensuring data consistency and integrity—from ACID transactional properties to eventual consistency and the CAP theorem—within a serverless environment. Finally, it describes automation and orchestration techniques for migration using AWS tools (such as DMS, SCT, and others), supported by examples drawn from the author's practical experience in building serverless infrastructures and optimizing system scalability and resilience.

1. Architectural Approaches to Data Management when Migrating to Aurora Serverless

The transition from a monolithic architecture to microservices entails radical changes in data management strategies. The central principle becomes weak coupling of services and strict isolation of their data. The "Database per Service" pattern recommends that each microservice employ its own data store optimally suited to its needs [6]. This may involve a relational database (for example, Aurora) for transactional services, a NoSQL store (such as DynamoDB or Cassandra) for highly scalable components, or a NewSQL or analytical database for specialized workloads—depending on the characteristics of the data. Independent databases remove bottlenecks and single points of failure, since schema changes or load spikes in one service do not directly impact others [6]. Furthermore, this pattern enhances overall resilience: failure in one data store affects only its corresponding service rather than the entire platform [6].

Figure 1 schematically illustrates an example in which three microservices ("Sales", "Customer", "Compliance") each leverage different database types (Aurora, DynamoDB, RDS for SQL Server) tailored to their functional requirements, interacting exclusively through APIs without direct access to each other's data.

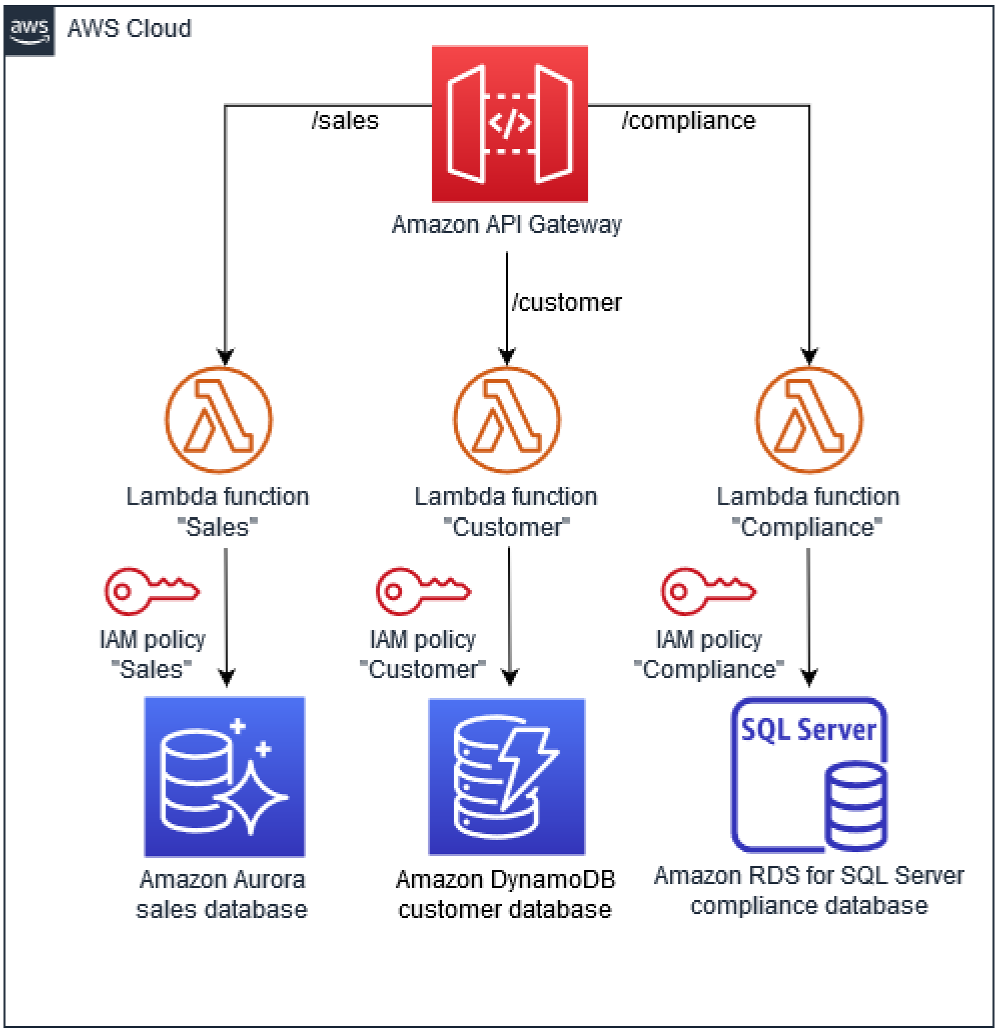
AWS Cloud
/sales
/compliance
Amazon API Gateway
/customer
Lambda function "Sales"
Lambda function "Customer"
Lambda function "Compliance"
IAM policy "Sales"
IAM policy "Customer"
IAM policy "Compliance"
SQL Server
Amazon Aurora sales database
Amazon DynamoDB customer database
Amazon RDS for SQL Server compliance database

Figure 1. The "Database-per-Service" pattern in a microservice architecture: each service uses its own database (Aurora for the Sales service, DynamoDB for the Customer service, Microsoft SQL Server for the Compliance service). Communication occurs via an API Gateway; IAM policies isolate access, ensuring loose coupling of components [6].

In migration practice, it is often necessary to decompose a large monolithic database into multiple smaller databases aligned with distinct domain contexts. A phased approach is employed: identifying modules suitable for extraction as standalone services, and gradually "cutting over" to them (the Strangler Fig pattern). For example, in one initiative a monolithic e-commerce database was divided into "Orders", "Catalog", "Payments", and other services, each of which received its own schema in Aurora or, when required, a separate data store. Temporary dual writes ensured data synchronization between the legacy and new systems during the transition period, and once the new services stabilized, the monolith was decommissioned. This experience underscored that early planning of service boundaries and data models is critical: mistakes in decomposition can lead to inter-service dependencies and complications in data aggregation.

When designing data architectures for microservices, it is also essential to consider the constraints imposed by the CAP theorem (Consistency, Availability, Partition Tolerance). In a distributed system—particularly one with geographically dispersed microservices—it is impossible to guarantee strict consistency and availability simultaneously in the presence of network partitions [7]. A trade-off must be made: either the system prioritizes consistency at the expense of availability during a partition (CP mode), or it prioritizes availability at the expense of temporary data inconsistencies (AP mode).

In traditional monolithic DBMS deployments—particularly those running on a single node—this dilemma is less apparent. However, in a microservices environment with isolated databases and no end-to-end transactions, these trade-offs become acutely visible [6]. Under the Database-per-Service pattern, orchestrating distributed transactions that span multiple services poses a significant challenge [6]. Teams typically must either denormalize and duplicate data across services (accepting eventual drift and eventual consistency) or employ saga patterns and other coordination mechanisms in place of a monolithic ACID transaction. The CAP theorem then serves as a design compass: each data store must satisfy at least two of the three CAP properties [6], and architects must decide in advance where the system can temporarily sacrifice consistency for availability and where strict data agreement is non-negotiable.

The Amazon Aurora architecture itself is engineered to minimize CAP compromises within a single cluster. Aurora guarantees strong read-write consistency in its primary availability zone by synchronously replicating data across multiple storage nodes. In an Aurora cluster, a write to the primary (writer) node is acknowledged only after it has been durably stored in at least four of six replicas spread across three availability zones—a 4/6 quorum mechanism [8, 9]. This design delivers high durability and fault tolerance: even the loss of an entire zone or several replicas does not result in data loss. If the primary node fails, Aurora automatically fails over to one of the reader replicas; because all instances share the same distributed storage, the newly promoted writer immediately sees the same committed data state [8, 10]. Thus, within a single region, Aurora achieves consistency and partition tolerance (CP mode) while maximizing availability.

In Figure 2, a simplified Aurora architecture is shown: compute instances (primary and reader) share a distributed storage layer that replicates each data segment sixfold across three availability zones. Writes are synchronously committed to a quorum of replicas to ensure durability, and reads from any replica incur minimal latency. Continuous backups to Amazon S3 further secure long-term data preservation [10].

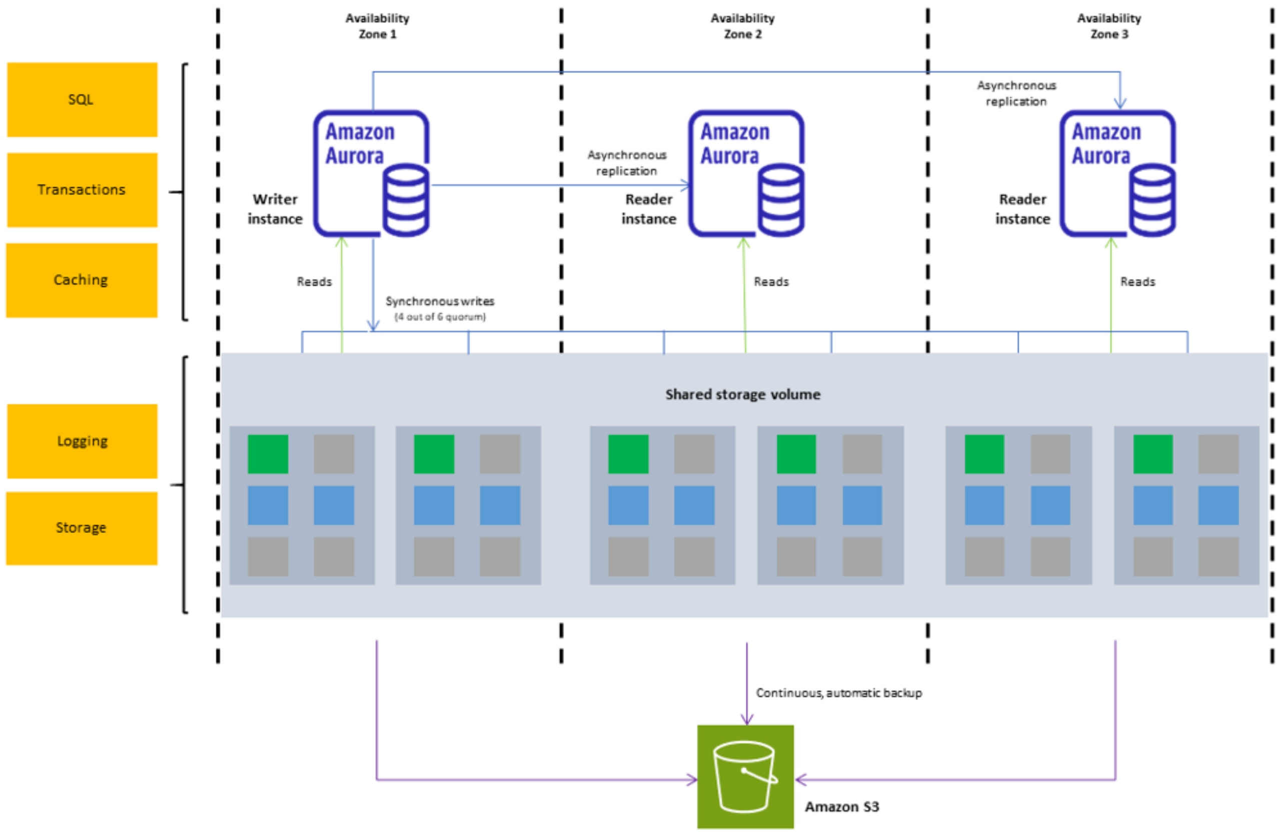

Figure 2. Simplified Amazon Aurora cluster architecture: compute layer separated from distributed storage [10].

In Figure 2, the primary node (Writer) and two reader nodes (Reader) in different AZs all access a single shared storage volume composed of six segment replicas (data fragments are color-coded). A write is acknowledged once at least four of the six copies have been synchronously committed (quorum), after which the data becomes immediately available for reading on all instances. Reader nodes in other AZs receive updates almost instantaneously via the shared storage, providing low-latency reads. Continuous backups to Amazon S3 are also depicted [8–10].

It should be noted that global data distribution (multi-region) introduces an element of eventual consistency: when Aurora is deployed as a Global Database (with read-only replicas in other regions for disaster recovery and local reads), cross-region replication occurs asynchronously. Under normal operation, reads from a remote region may lag the primary write by a few tenths of a second. However, in the event of a failure, a fast cross-region failover mechanism is available—in the latest Aurora versions this completes in under 30 seconds, substantially improving the overall availability of global deployments [11]. Consequently, when designing a data architecture, an organization must decide whether it requires globally distributed replicas (for user-proximate reads and DR) and, if so, account for the slight consistency delay between regions. Overall, Aurora Serverless combines the benefits of vertical scaling (a powerful ACID-compliant cluster) with horizontal distribution (replication across AZs and regions), making it an ideal solution for high-throughput enterprise systems demanding both reliability and scalability.

A further strategic decision during migration is the separation of operational and analytical workloads between the transactional database and dedicated data stores. The concepts of Data Warehouse (DW) and Data Lake (DL) represent two distinct approaches to corporate information storage. A Data Warehouse is a structured repository into which filtered and aggregated data are loaded for specific analytical purposes. In contrast, a Data Lake is a raw pool of heterogeneous data without a predefined schema or target use case [12]. Within the context of an Aurora migration, this means that transactional system data (operations, accounts, orders, etc.) are best retained in Aurora—optimized for Online Transaction Processing and ensuring ACID integrity—while historical records, event logs, and large volumes of unstructured data are offloaded to a Data Lake on object storage (for example, Amazon S3) or into a specialized Data Warehouse (such as Amazon Redshift) for subsequent BI analysis. This bifurcated approach relieves the production Aurora database of heavy analytical queries, preserving its performance for current transactional workloads. Moreover, a Data Lake's ability to store data in its native form is invaluable for machine learning and exploratory analysis. Thus, a strategic integration of Aurora into the enterprise data landscape positions it as the System of Record, complemented by Data Lake/DW platforms for analytics—with appropriate ETL/ELT pipelines between them. Literature emphasizes that DW and DL are complementary: the lake serves as the raw repository, while the warehouse provides a refined, decision-ready view [12]. When managed correctly, this layered architecture achieves an optimal balance between data storage flexibility and analytical efficiency.

To illustrate these architectural choices, consider a generalized case study from practical experience. A large financial institution migrated its monolithic core banking system to a cloud-native microservices platform. The monolithic application—backed by a single Oracle database—was refactored into multiple domain services (customers, accounts, transactions, reporting, etc.). Each service was provisioned with its own data store: transaction processing was moved to Aurora PostgreSQL Serverless (for ACID compliance and scalability), log history was archived in Amazon S3 (Data Lake) with downstream analysis in Redshift, and user session state was managed in Redis (for low-latency access). During migration, data from the legacy database was streamed into the new stores using Kafka for event streams and AWS DMS for initial bulk loading. This enabled the team to incrementally switch each service over to its new database without system downtime. The outcomes confirmed that service-specific databases eliminated resource contention between components, while Aurora Serverless automated capacity scaling to accommodate peak loads (for example, month-end billing cycles)—in one month the cluster's capacity auto-scaled nearly threefold within minutes, with no manual intervention. Furthermore, data separation simplified compliance with security requirements: the customer data service had tightly scoped access, isolated from analytics services, in accordance with the principle of least privilege and GDPR mandates.

2. Ensuring Data Consistency and Integrity in a Serverless DBMS

When migrating mission-critical data to a new platform, preserving both consistency and integrity is paramount. Enterprise systems—Internet banking, telecom billing, e-commerce, and the like—demand that every operation be durably recorded (durability), adhere to business rules (consistency), execute in isolation from concurrent transactions, and never leave partial updates (atomicity). These requirements are captured by the ACID acronym: Atomicity, Consistency, Isolation, Durability. While traditional relational databases (Oracle, PostgreSQL, MySQL, etc.) strive to uphold ACID in full, many NoSQL and distributed stores instead embrace BASE: Basically Available, Soft state, Eventual consistency [13]. Let us explore how these paradigms map onto Aurora Serverless and which strategies ensure data integrity during migration.

Aurora, as a MySQL- and PostgreSQL-compatible engine, is firmly ACID-compliant. It guarantees transaction atomicity, enforces defined consistency constraints, isolates concurrent operations (offering isolation levels up to Serializable), and ensures the durability of committed results. For example, in a banking application, a funds transfer between accounts on Aurora will either debit one account and credit the other in its entirety or roll back entirely—no half-measures. Under the hood, write-ahead logging (WAL) combined with multi-node replication delivers durability: once a transaction commits, its data persist even if a storage node or entire data center fails. Crucially, running Aurora in Serverless mode does not compromise these guarantees; autoscaling happens transparently, and the same logging and locking mechanisms continue to protect transactional integrity. Transaction isolation in Aurora mirrors that of standard RDS MySQL/PostgreSQL—MySQL defaults to REPEATABLE READ with MVCC, while PostgreSQL operates at READ COMMITTED or higher—preventing dirty reads and related anomalies. In practice, this means that moving to Aurora Serverless imposes no new transactional model on developers: ACID guarantees remain intact, underpinning stable business operations.

That said, it is important to account for scaling behavior. Aurora Serverless v2 can adjust its compute capacity units (ACUs) on the fly—without pausing active transactions—thanks to implementation optimizations that avoid the need for “quiet points” (unlike v1) [14]. While consistency remains uncompromised, very long-running transactions can still pose challenges: they may delay change propagation to replicas or tie up compute resources. As a rule of thumb during migration, review your application workflows for excessively long transactions and, where possible, break them into shorter steps or adopt asynchronous processing patterns.

By contrast, the BASE model—widespread in NoSQL systems and large-scale web applications—prioritizes availability over strict consistency. BASE stands for “Basically Available, Soft state, Eventual consistency,” meaning the system strives to remain responsive (avoiding operation blocking) but allows temporary data discrepancies without instant consistency guarantees [13]. Under eventual consistency, when updates cease, all replicas will converge to the same state over time. A familiar example is distributed caching or loosely synchronized replication: if User A updates their profile, User B might see the old data briefly until the update has rippled through every node. Although Aurora Serverless itself enforces ACID, eventual consistency can surface at external layers—caches like Amazon ElastiCache being a prime case. If cache invalidation lags behind Aurora writes, stale data may be served to readers. To safeguard integrity, you must implement synchronization measures: disable caching for critical operations, employ write-through protocols, or set very short TTLs so any inconsistency window remains measured in milliseconds. In sum, even within an ACID database, architects must pinpoint zones of potential eventual consistency—typically in integrations with other systems, asynchronous message queues, caches, and similar components.

As already noted, Aurora implements a CP model within a region—favoring consistency over availability in the event of a network partition. In practice, this means that if fewer than four of the six replicas acknowledge a write, the system will pause processing (waiting for quorum restoration) rather than accept divergent data. From the CAP theorem perspective, this represents a deliberate choice in favor of strict consistency and partition tolerance—appropriate for financial and other mission-critical workloads where incorrect data are unacceptable. The alternative AP mode (always responding, even with stale data) is characteristic of some NoSQL systems—for example, Cassandra can be configured to accept writes with a lower quorum, increasing availability but allowing “split-brain” data. During migration, requirements must be defined clearly: if the business cannot tolerate any anomalies, Aurora’s CP behavior is the correct solution. If temporary divergence is acceptable for uninterrupted operation (for instance, social-media like counters that may briefly differ between users), other technologies or architectural patterns can be layered atop the database. In sum, data consistency is not only a technical property of the DBMS but also an architectural decision that must align with application needs and user expectations. [6]

A separate challenge is ensuring data consistency throughout the migration process to Aurora. Migration typically proceeds in several stages: export/copy, verification, change data capture (CDC), and final cutover. At each step, it is critical not to lose or corrupt data. The following strategy is recommended:

- Initial load: perform a full data export from the legacy system and load it into Aurora (manually or using AWS DMS). Thereafter, compare checksums or record counts on key tables between source and target—for example, order counts or balance totals—to verify that all data have been transferred and match exactly.

- Change replication (CDC): enable change-data capture on the source database (via binlog for MySQL/Oracle or transactional logs). AWS Database Migration Service supports a full-load + CDC mode, continuing to replicate new transactions to the target Aurora instance in real time. This ensures both databases remain synchronized until migration completes. In Samsung's experience migrating 1.1 billion account records, this approach allowed uninterrupted service—DMS migrated approximately 2–4 TB of data over several days while keeping the source up to date without downtime. [3]

- Data validation: before cutting over traffic to the new database, conduct spot checks to verify consistency. For example, execute a suite of business operations in a test Aurora environment and compare results against expectations, or use built-in validation tools (AWS DMS offers a data validation feature for post-migration comparison).

- Cutover: schedule during a low-usage window. First quiesce the application; then confirm that all source-DB changes have been applied to Aurora (DMS provides replication-lag metrics). Once data currency is validated, reconfigure the application to point at Aurora and resume operations. Ideally, downtime is measured in seconds or minutes. Samsung's full regional migrations took around 22 weeks, yet each cutover incurred minimal downtime—users barely noticed the transition. [3]

Aurora supports all the standard integrity mechanisms: primary and foreign keys, unique constraints, CHECK constraints, triggers, and stored procedures. During migration, it is essential to verify that every business rule is enforced either at the Aurora database level or within the application. For instance, if the source database employed cascading constraints, these must be reproduced in the target system. In heterogeneous migrations (e.g. Oracle → Aurora PostgreSQL), the AWS Schema Conversion Tool will translate the schema automatically and flag any incompatibilities for manual remediation. It is not uncommon for legacy systems to disable certain constraints for performance reasons, relying instead on application logic to maintain integrity; in such cases, enabling those constraints in Aurora—whose performance often accommodates them without degradation—yields a clear consistency benefit.

Finally, once the system has been refactored into microservices with separate databases, it is crucial to design interservice communication so that global data consistency is preserved. Achieving full atomicity across services is impractical—distributed transactions are overly complex and do not scale—so compensating transactions and event-driven reconciliation are employed. For example, in a ticketing system the "Booking" and "Payments" services operate independently: booking immediately reserves a seat, and payment may complete seconds later. If the payment fails, the Payments service emits a cancelation event, and the Booking service releases the seat. This achieves eventual consistency: data may temporarily conflict (a seat held without payment), but the system ultimately converges on the correct state. When architecting on Aurora, such scenarios must be anticipated by embedding coordination patterns (e.g. sagas). Here the BASE principle—"soft state" and eventual consistency—applies [13]. However, leveraging Aurora's reliable WAL and guaranteed event delivery via SQS, EventBridge, or similar services ensures that no events are lost and that the system attains consistency within a short window.

In summary, an integrity strategy for migrating to Aurora Serverless entails preserving transactional rigor where it is critical (ACID within each Aurora-backed service) and managing asynchronous processes thoughtfully where synchronous guarantees are infeasible (interservice interactions, integrations, caches). Aurora provides a robust foundation—"data are always consistent and durable within the cluster"—while architectural patterns and mature migration tools address the rest. When executed correctly, moving to serverless Aurora not only maintains consistency but enhances overall reliability by eliminating human error (through automated scaling and fault tolerance) and employing modern data governance mechanisms.

3. Automation and Orchestration of Migration to Aurora Using AWS Tools

Migrating an enterprise database is a multifaceted endeavor whose success hinges on the level of automation applied. Hand-rolling schema definitions, data transfers, and business-logic migration is fraught with risk and can lead to lengthy downtime. As a result, modern practice leans heavily on purpose-built tools and managed services to streamline this transition. Within the AWS ecosystem, a powerful toolset is available: AWS Database Migration Service (DMS), AWS Schema Conversion Tool (SCT), AWS Data Pipeline, AWS Snowball (for offline transfer of large datasets), and infrastructure-as-code frameworks (CloudFormation, Terraform) for repeatable environment provisioning. This section explores best practices for orchestrating a migration to Aurora Serverless, drawing on the capabilities of these services.

The linchpin for data movement is AWS DMS—a managed, serverless service that can replicate data between source and target with virtually no application downtime. DMS accommodates both homogeneous migrations (for example, MySQL → Aurora MySQL) and heterogeneous migrations (Oracle → Aurora PostgreSQL), automatically transforming data types and applying ongoing changes in real time. When migrating to Aurora, DMS is invaluable for minimizing outage windows: in Full Load + CDC mode, it first performs a complete bulk load of the existing dataset and then switches to continuous change-data-capture, streaming all new transactions to the target Aurora cluster [15]. This approach keeps the source database fully operational while ensuring that every write is mirrored into Aurora without interruption.

In Figure 3, a typical DMS-based migration architecture is illustrated: an on-premises MySQL instance connects to AWS over a secure channel (Direct Connect or VPN), where DMS—running serverlessly in a private subnet—reads the source database logs and applies each change to an Amazon Aurora cluster. This serverless migration pattern enables background replication with minimal impact on application availability.

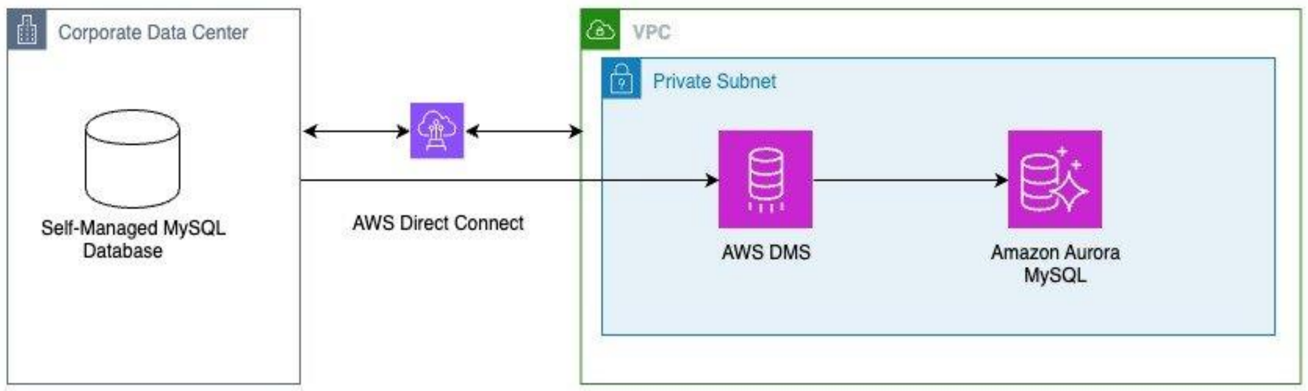

Figure 3. Database migration architecture using AWS DMS: an on-premise MySQL instance is linked over a secure channel (Direct Connect/VPN) to AWS. AWS DMS reads changes from the source database and replicates them to a target Amazon Aurora cluster (MySQL-compatible) [15].

Use of DMS provides several advantages. Firstly, migration becomes safer – DMS maintains a progress log and, in the event of failure, can resume from the point of interruption without retransmitting already migrated data. Secondly, DMS includes optimizations for rapid initial loading (for example, parallel table exports and the use of mydumper/myloader for MySQL) [15], and it automatically configures the target database by creating tables and indexes. Thirdly, DMS is serverless and scales to match workload demands, relieving the team of managing intermediate replication instances. As noted in Samsung's migration of 1.1 billion account records, AWS DMS replicated a heterogeneous 2–4 TB dataset in 3–4 days per region while keeping the source database available to users [3]. Upon completion of the initial load, DMS facilitated a smooth traffic cutover to Aurora by intercepting requests to the legacy database and routing them to the new cluster [3]. Consequently, such a large-scale migration proceeded with minimal user impact—a testament in part to the robustness of AWS tooling.

If the migration is heterogeneous (i.e., the source and target engines differ, such as Oracle or SQL Server → Aurora PostgreSQL), schema and code conversion (stored procedures, triggers) become paramount. The AWS Schema Conversion Tool (SCT) automates much of this work: it analyzes the source schema, generates an equivalent schema for Aurora, and flags incompatibilities. According to AWS, SCT automatically converts roughly 80 % of database code; the remaining cases require manual intervention, but the tool provides detailed reports and even code-fix examples. In this way, SCT significantly reduces the effort required by DBAs and developers when migrating to Aurora. Moreover, SCT can migrate data warehouses—e.g., Oracle or Teradata to Amazon Redshift—which, although beyond our immediate scope, demonstrates the tool's versatility.

It is worth mentioning AWS's Database Freedom initiative (formerly "Project Aurora"), designed to help customers move from commercial engines to managed services like Aurora. This program offers not only the necessary tools (DMS, SCT) but also methodology and expertise. AWS identifies three pillars of success: innovative migration technologies (DMS, SCT) to automate manual tasks; expert partners for architectural guidance and training; and risk-mitigation programs (workshops, proofs of concept). In our context, adhering to these recommendations means treating an Aurora migration not as a one-off administrative task but as a full project encompassing planning, testing, developer involvement, and maximal automation.

Beyond data transfer, automating the provisioning and configuration of the Aurora environment and its associated services is essential. Infrastructure-as-Code tools—AWS CloudFormation, Terraform, and AWS CDK—allow teams to describe the desired Aurora cluster (engine type, Serverless mode, autoscaling parameters, subnet/VPC settings, user accounts, etc.) in declarative templates. These templates can then be deployed consistently and repeatably across environments (Dev, Staging, Prod), eliminating human error in database setup and simplifying environment recovery in case of failure.

In a CI/CD context, it makes sense to integrate Aurora schema updates into application delivery pipelines. Because Aurora is compatible with popular engines, teams can use schema-migration tools such as Liquibase or Flyway, or native AWS mechanisms (for example, managing schema changes via CodePipeline and the AWS CLI). This ensures controlled schema evolution, which is critical in a microservice architecture where each service update may require corresponding database alterations.

Large migrations often involve orchestrating multiple steps—backing up the source database, provisioning the Aurora cluster, launching DMS tasks, monitoring their completion, switching application endpoints, running live tests, and performing post-migration cleanup. To automate such workflows, AWS offers Step Functions, and third-party tools like Jenkins pipelines or Airflow may also be used. For example, one might configure a Step Function that: (1) deploys the Aurora cluster via CloudFormation; (2) triggers the DMS task; (3) monitors DMS replication-lag metrics in CloudWatch; (4) when lag reaches zero, invokes a Lambda function to update application environment variables to point to the new database; (5) executes integration tests; and (6) shuts down the legacy database. This seamless orchestration minimizes manual intervention and documents the entire process clearly.

Automation, however, cannot succeed without continuous monitoring. AWS provides metrics for Aurora (ACU utilization, connection counts, replication lag), for DMS (copy progress, CDC lag, errors), and comprehensive logging. By configuring CloudWatch Alarms on critical thresholds—such as migration throughput drops or data-conflict errors—the team can respond rapidly to issues. It is also important to profile Aurora performance at early stages (for example, during a test migration run), since tuning parameters (transaction-log size, Serverless v1 auto-pause settings, etc.) may be required to achieve optimal results.

Conclusion

Migrating to the serverless Amazon Aurora database is a complex, multifaceted process requiring both robust technology and a carefully considered data management strategy. This study has examined the key aspects of the challenge. First, in terms of architectural approaches: moving from a monolithic architecture to microservices with isolated databases enables independent scalability and enhanced system resilience. Aurora Serverless integrates seamlessly into such an architecture, providing a cloud-native database capable of adapting to dynamic workloads while supporting ACID transactions at the service level. Second, regarding data consistency and integrity: Aurora preserves traditional integrity guarantees (ACID), while distributed-system patterns (such as sagas and event sourcing) deliver eventual consistency where needed. We have discussed CAP theorem trade-offs and demonstrated that within a single region Aurora prioritizes consistency and partition tolerance, ensuring data coherence even under failure conditions. Third, in terms of migration automation: AWS tools (DMS, SCT, etc.) markedly reduce both risk and duration of migration by enabling data transfer with minimal downtime and operational overhead. Practical examples—including migrations of over one billion records—show that, when leveraged effectively, these tools allow large-scale enterprise systems to migrate to Aurora while improving performance and reducing costs.

Particular attention should be paid to assessing which strategies apply: there is no universal formula, so each organization must consider its own requirements—consistency guarantees, acceptable downtime, application compatibility with a new DBMS, and team expertise. However, general recommendations can be formulated as follows:

(1) Plan and design the data architecture in advance by defining domain services and selecting optimal storage models (Aurora for transactional workloads, Data Lake/Warehouse for analytics, etc.), ensuring scalability for the future.

(2) Ensure integrity at every step by enforcing ACID properties where required and carefully managing zones of eventual consistency; verify that constraints, triggers, and business-logic rules are migrated intact.

(3) Automate the process to the greatest extent possible by using DMS/SCT for data migration, IaC for environment provisioning, and orchestration tools to execute steps error-free; this accelerates migration and reduces team workload.

(4) Test and monitor by running trial migrations, measuring Aurora's performance under load, and configuring monitoring and alerts during and after migration to detect bottlenecks swiftly.

Migration to Aurora Serverless often becomes a catalyst for further improvements—refactoring legacy code, optimizing queries, and adopting modern DevOps practices. Experts note that organizations gain not only cost savings (no licensing fees, pay-as-you-go pricing) but also accelerated innovation: developers can deliver new features faster without managing database scaling manually. Thus, migrating to serverless Aurora represents a step toward a more agile and resilient data architecture capable of meeting the demands of high-throughput systems in the cloud era.

References

1. Gartner (2021). Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences. Press Release. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences
2. Deloitte & AWS (2021). Accelerating your Database Modernization journey with Deloitte on AWS. Retrieved from https://d1.awsstatic.com/partner-network/AWSDatabase_Modernization.pdf
3. AWS (2020). Samsung Migrates 1.1 Billion Users across Three Continents from Oracle to Amazon Aurora with AWS Database Migration Service. Retrieved from https://aws.amazon.com/ru/solutions/case-studies/samsung-migrates-off-oracle-to-amazon-aurora/
4. Lawton, G. (2020). How to carefully plan a database migration to the cloud. techtarget. Retrieved from https://www.techtarget.com/searchcloudcomputing/feature/How-to-carefully-plan-a-database-migration-to-the-cloud
5. Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44-54. DOI: 10.1145/3368454
6. Amazon Web Services. Database-per-service pattern. Retrieved from https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/database-per-service.html
7. Amazon Web Services. CAP theorem. Retrieved from https://docs.aws.amazon.com/whitepapers/latest/availability-and-beyond-improving-resilience/cap-theorem.html
8. Amazon Web Services. Amazon Aurora storage. Retrieved from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.StorageReliability.html
9. Barnhart, B., Brooker, M., Chinenkov, D., Hooper, T., Im, J., Jha, P. C., ... & Yan, J. (2024). Resource Management in Aurora Serverless. Proceedings of the VLDB Endowment, 17(12), 4038-4050.
10. Amazon Web Services. Amazon Aurora DB clusters. Retrieved from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.html
11. Amazon Web Services. Amazon Aurora reduces cross-Region Global Database Switchover time to

typically under 30 seconds. Retrieved from https://aws.amazon.com/ru/about-aws/whats-new/2025/05/amazon-aurora-cross-region-global-database-switchover-time-under-30-seconds/

12. Nambiar, A., & Mundra, D. (2022). An overview of data warehouse and data lake in modern enterprise data management. Big data and cognitive computing, 6(4), 132.
13. Amazon Web Services. What's the Difference Between an ACID and a BASE Database?. Retrieved from https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/?nc1=h_ls
14. Amazon Web Services. Migrating to Aurora Serverless v2. Retrieved from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.upgrade.html
15. AWS Database Blog (2025). Srivastava, A. et al. Migrate a self-managed MySQL database to Amazon Aurora MySQL using AWS DMS homogeneous data migrations. Retrieved from https://aws.amazon.com/blogs/database/migrate-a-self-managed-mysql-database-to-amazon-aurora-mysql-using-aws-dms-homogeneous-data-migrations/