MODULE 6 · STUDY NOTES

Adding a Database Layer

How to choose a database, run relational workloads on Amazon RDS and Aurora, scale NoSQL with DynamoDB, match purpose-built databases to the job, and migrate data into AWS

Every application needs somewhere to keep its data, but the era of forcing every workload into one relational database is over. This module trains a cloud architect to work backward from the business need — weighing scalability, storage, data characteristics, durability, and cost — and then pick the database whose data model and access pattern fit. It starts in the relational world with **Amazon RDS**, the managed service behind seven familiar engines, and **Aurora**, the cloud-built engine that delivers commercial-grade performance at roughly one-tenth the cost. It shows how **Multi-AZ** deployments, **read replicas**, **RDS Proxy**, backups, and encryption make that layer available, scalable, and secure. It then crosses into NoSQL with **DynamoDB** — serverless, single-digit-millisecond, keyed by partition and sort keys and extended with secondary indexes and global tables — before surveying the **purpose-built databases** (Redshift, DocumentDB, Keyspaces, MemoryDB, Neptune, Timestream, and QLDB) that each solve one shape of problem well. Finally it covers moving existing data in with **AWS DMS**, and grades the whole design against the **AWS Well-Architected Framework**.

LEARNING OBJECTIVES

- Compare database types and services offered by AWS
- Explain when to use Amazon Relational Database Service (Amazon RDS)
- Describe the advanced features of Amazon RDS
- Explain when to use Amazon DynamoDB
- Identify AWS purpose-built database services
- Describe how to migrate data into AWS databases
- Design and deploy a database server
- Use the AWS Well-Architected Framework principles when designing a database layer

6.1 Database layer considerations

The world has changed, and the one-size-fits-all approach of using a relational database for every application no longer works. Relational databases still matter, but a growing set of **purpose-built** databases — designed from the ground up for social, mobile, IoT, and global-scale access — are rising in popularity. As an architect you will often choose among database types for a given workload, so start with the **business case and work backward** to the components that fit it. Expect the decision to evolve as needs become clearer and as you optimize for cost and performance.

WHAT TO WEIGH BEFORE YOU CHOOSE

- **Scalability** — how much throughput is needed now, and will the solution scale later?
- **Storage requirements** — must the database hold gigabytes, terabytes, or petabytes? Different architectures support different maximums.
- **Data characteristics** — what is the data model (relational, structured/semi-structured, highly connected, timeseries)? What are the access patterns, and do you need low latency?
- **Durability, availability, and recoverability** — how critical is the data, and do data-residency or regulatory obligations apply?
- **Cost** — redundant copies across geographically separated locations raise durability but also cost; balance business need against spend.

Scaling is where the on-premises world hurts most. Scaling a traditional database up can have **unpredictable performance impact and often requires downtime**. **Underprovision** and your applications may stop working; **overprovision** and you pay upfront for resources you do not need — which violates the **cost-optimization** pillar of the AWS Well-Architected Framework. Ideally you choose a database that can handle the needed throughput at launch and still scale up later.

TABLE 6.1 Relational vs. non-relational databases

ASPECT	RELATIONAL (RDBMS / SQL)	NON-RELATIONAL (NOSQL)
Structure	Tabular rows and columns	Key-value, document, graph, in-memory, search
Schema	Strict, well-defined; change-averse	Flexible; each object can differ
Data	Structured	Structured, semi-structured (JSON), unstructured
Transactions	**ACID**: atomic, consistent, isolated, durable	Optimized per access pattern; high throughput
Strengths	**SQL**, data integrity, ease of use, reduced storage	Flexibility, scalability, performance
Best for	**OLTP**; migrating relational workloads	Caching, JSON, single-digit-ms retrieval
Scaling	Vertical (plus read replicas)	Horizontal

Both relational and non-relational databases can scale **horizontally and vertically**. For example, Amazon RDS can add **read replicas** (scaling horizontally), and Redis can scale **vertically** with bigger instance types. AWS offers a main relational option — **Amazon RDS**, a managed service with seven engines including Aurora — and multiple non-relational options such as **DynamoDB** (key-value), **Neptune** (graph), and **ElastiCache** (in-memory).

TABLE 6.2 Your responsibility shrinks with managed services

TASK	ON PREMISES	DATABASE ON EC2	MANAGED SERVICE (RDS/AURORA)
Power, HVAC, network, rack and stack, server maintenance	You	AWS	AWS
OS installation and patching	You	You	AWS
Database installation and patching	You	You	AWS
Backups, high availability, scaling	You	You	AWS
Application and query optimization	You	You	You

Reading the table left to right: on premises the administrator owns everything from hardware to query tuning. On an **EC2 instance**, AWS runs the physical data center and the OS is pre-installed, but you still patch the OS and manage all database software, backups, high availability, and scaling. On a **managed AWS database service**, AWS handles the common, repetitive administration and you are left responsible only for **optimizing your queries and application**.

Database capacity planning rounds out the section: analyze current storage capacity, predict future requirements, and decide whether you need horizontal scaling, vertical scaling, or a combination. **Vertical scaling** expands the resources of the existing server (memory, storage, processing power) — a complex, time-consuming change that usually requires **downtime**. **Horizontal scaling** adds more servers to spread the load; compute is added while the database runs, so it usually happens **without downtime**.

> COMMON PITFALL
>
> Relational databases are often described as scaling vertically and non-relational as scaling horizontally, and that is the typical pattern — but it is not a hard rule. Both types can scale both ways: RDS uses read replicas to scale horizontally, while Redis scales vertically with larger instances. Match the scaling approach to the engine and the workload, not to a slogan.

6.2 Amazon RDS and Aurora

Amazon RDS is a fully managed relational database service you use to set up, operate, and scale relational databases in the cloud. AWS takes over the difficult, tedious work — RDS automates **provisioning, patching, backup, recovery, failure detection, and repair** — so you avoid provisioning infrastructure or maintaining software. RDS uses **Amazon EBS volumes** for database and log storage, and you can scale the allocated storage for a DB instance.

> FOUR BENEFITS OF AMAZON RDS
>
> - **Lower administrative burden** — no infrastructure to provision, no database software to install or maintain; one console and API, with security and monitoring built in.
> - **Highly scalable** — scale compute and storage with a few clicks or an API call, choosing from instance types optimized for different use cases.
> - **Available and durable** — configure automated backups, database snapshots, and automatic host replacement; use **Multi-AZ** for HA and **read replicas** for read-heavy scale-out.
> - **Secure and compliant** — isolate the database in your own **Amazon VPC**, control access with security groups and firewall rules, encrypt at rest (many engines) and in transit (all engines), with compliance readiness including HIPAA eligibility.

Amazon RDS gives you seven engine options: Aurora with MySQL compatibility, Aurora with PostgreSQL compatibility, RDS for MySQL, RDS for MariaDB, RDS for PostgreSQL, RDS for Oracle, and RDS for SQL Server. Behind the scenes RDS manages a specialized EC2 instance that supplies compute. An **RDS instance is an isolated database environment** that can contain multiple user-created databases, each holding your organization's data. Architecturally, an EC2 application tier and the RDS instance both live inside a **VPC**.

Aurora is a relational database engine built for the cloud with full MySQL and PostgreSQL compatibility, managed by Amazon RDS. It runs up to **five times faster than standard MySQL** and **three times faster than standard PostgreSQL**, and provides the security, availability, and reliability of commercial databases at approximately **one-tenth the cost**. Aurora features a distributed, fault-tolerant, **self-healing storage system that auto-scales to 64 TB per database instance**, with up to **15 low-latency read replicas**, point-in-time recovery, continuous backup to **Amazon S3**, and replication across **three Availability Zones**.

INSIDE AN AURORA DB CLUSTER

- A cluster is one or more DB instances plus a **cluster volume** — virtual storage that spans multiple AZs, each AZ holding a copy of the cluster data.
- The single **primary instance** supports read and write operations and performs all data modifications to the cluster volume.
- **Aurora Replicas** (up to 15) connect to the same storage volume, support **read-only** operations, and offload reads from the primary.
- Place replicas in separate AZs for high availability; Aurora **automatically fails over** to a replica if the primary becomes unavailable, and you can set failover priority.

Aurora Serverless is an on-demand, auto-scaling configuration in which the database automatically starts up, shuts down, and scales capacity up or down to match the application — hands-off capacity management with fine-grained scaling. **Aurora Serverless v2** instances can run alongside provisioned instances in the same cluster, and you can convert instances between provisioned and Serverless without creating a new cluster. It suits **variable workloads, new applications, development and testing, and capacity planning**.

WORKED EXAMPLE – OLTP AND RIGHT-SIZING AN RDS INSTANCE

Amazon RDS is suited to **online transaction processing (OLTP)**: storing and updating transactional data reliably and efficiently in high volume — banking deposits and withdrawals, for instance, each recorded as a row with a unique transaction ID, date, type, and amount. To power that workload you pick an instance type. **General-purpose** families (T4g, T3, M6g, M5) fit CPU-intensive work and moderate CPU with temporary spikes; **memory-optimized** families (R6g, R5, X2g, X1E) fit query-intensive workloads or high connection counts. When an instance needs an upgrade, first identify the constrained resource: if `db.m6g.large` is short on **CPU**, move up to `db.m6g.xlarge`; if it is short on **memory**, move across to `db.r6g.large`.

TABLE 6.3 Sample RDS instance types and sizing

INSTANCE	FAMILY	MEMORY (GIB)	VCPU
db.m6g.large	General purpose	8	2
db.r6g.large	Memory-optimized	16	2
db.m6g.xlarge	General purpose	16	4
db.r6g.xlarge	Memory-optimized	32	4

AMAZON RDS SECURITY BEST PRACTICES

- Run the DB instance in a **VPC** for the greatest possible network access control.
- Use **IAM policies** to control who can manage RDS resources (create, describe, modify, delete instances).
- Use **security groups** to control which IP addresses and EC2 instances can connect.
- Use **SSL/TLS** connections with engines that support them (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server).
- **Encrypt** instances and snapshots at rest with an **AWS KMS** key (AES-256); logs, backups, and snapshots are then encrypted too.
- Use the **database engine's own security features** to control who can log in.

COMMON PITFALL

Aurora is not a service that competes with Amazon RDS — it is one of RDS's seven **engines**, fully managed by RDS. On a "which service?" question, remember that choosing Aurora still means using Amazon RDS. Security here is also a **shared responsibility**: AWS secures the cloud (the RDS infrastructure), while you secure what is in the cloud (network access, IAM, encryption keys, and engine logins).

6.3 Amazon RDS Proxy, availability, and backups

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon RDS, available for Aurora (MySQL and PostgreSQL), RDS for MariaDB, MySQL, PostgreSQL, and SQL Server, with no infrastructure to provision. It sits **between the application and the database** and makes applications more scalable, more resilient, and more secure.

THREE THINGS RDS PROXY IMPROVES

- **Scalability** — pools and shares database connections. Applications (especially serverless ones) can open thousands of connections, many idle; the proxy detects the gaps and reuses connections so the database receives **fewer** connections and does not exhaust memory or compute.
- **Resilience** — bypasses DNS caches to reduce failover time by up to **66 percent** for RDS Multi-AZ databases, preserving connections and queuing transactions during failover, then routing to the new instance.
- **Security** — enforces **IAM authentication** and stores credentials in **AWS Secrets Manager**, eliminating passwords embedded in application code.

Good candidates for RDS Proxy include any database instance hitting **"too many connections"** errors, applications that open and close large numbers of connections without built-in pooling, and applications (SaaS, ecommerce) that keep many connections open for long periods to minimize latency. During a failover the proxy preserves connections that are not mid-transaction, accepts new connections, queues transactions you send, and passes them along as soon as the standby is available — making failover far more transparent to the application. For authentication, the application requests an IAM token, sends its request to the proxy with that validated token, and the proxy retrieves the mapped secret from Secrets Manager before connecting to the database.

TABLE 6.4 Multi-AZ deployment vs. read replicas

ASPECT	MULTI-AZ DEPLOYMENT	READ REPLICA
Goal	High availability	Read scalability
Replication	Synchronous to a standby	Asynchronous
Failover	Automatic to the standby	Manual promotion to a standalone database
Reads	Standby serves no reads	Offloads read-heavy traffic from the primary
Placement	Standby in a second AZ	Can be created cross-Region for DR and reads near users

TABLE 6.5 Two ways Amazon RDS backs up data

FEATURE	AUTOMATED BACKUPS	DATABASE SNAPSHOTS
Use case	Restore an instance to a specific point in time	Back up an instance in a known state and restore to it
Frequency	Daily during the backup window; transaction logs every 5 minutes	User-initiated, as often as you choose
Retention	Default 7 days, up to 35 days; auto-deleted after the period	Kept until you explicitly delete it
Sharing	Cannot be shared (copy to a manual snapshot first)	Can be shared and copied by other AWS accounts

Automated backups and manual snapshots are stored in **S3 buckets owned and managed by the RDS service**, and any snapshot you copy becomes a manual snapshot. You can copy snapshots **within a Region, across Regions, and across accounts**. For disaster recovery you can configure RDS to replicate snapshots and transaction logs to another Region, and you can create a **cross-Region read replica** to improve DR, scale reads closer to users, and ease migration between Regions.

ENCRYPTION IN AMAZON RDS

- **Data at rest** is encrypted with **AWS KMS** keys (AES-256); all logs, backups, and snapshots of an encrypted instance are encrypted too.
- **Data in transit** uses **SSL/TLS**; RDS installs a certificate at provisioning and you can require encrypted-only connections.
- To protect an **unencrypted** database: take a snapshot of it, copy the snapshot with encryption enabled, then restore that encrypted snapshot to a new instance.

COMMON PITFALL

Do not confuse the two replication features. **Multi-AZ** replicates *synchronously* to a standby whose only job is **automatic failover for high availability** — it never serves reads. **Read replicas** replicate *asynchronously* to scale reads and must be **manually promoted** to stand alone. A scenario that needs both HA and read scaling uses both.

6.4 Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless, NoSQL database supporting both **key-value** and **document** data models. Its **flexible schema** lets each item carry different attributes, so you adapt as requirements change without redefining a table schema. DynamoDB delivers consistent **single-digit-millisecond** response times, automatically scales tables to adjust for capacity with zero administration, encrypts data, and continuously backs it up — making it suited to mission-critical workloads that prioritize speed, scalability, and durability.

> **WHERE DYNAMODB FITS**
>
> - **Software applications** — internet-scale apps with user-content metadata and caches that need high concurrency across millions of users and requests.
> - **Media metadata stores** — throughput and concurrency for media and entertainment (real-time video streaming), with multi-Region replication.
> - **Gaming platforms** — player data, session history, and leaderboards for millions of concurrent users with no operational overhead.

The data model is built from a few pieces. A **table** is a collection of data (unique to an account and Region) holding zero or more items. An **item** is a group of attributes uniquely identifiable among all others — conceptually a row, and the core unit of data. An **attribute** is a fundamental data element, a key-value pair, similar to a field or column. Every item must have a **partition (hash) key**; a **sort key** is optional and determines how items sharing a partition key are ordered (timestamps, version numbers, audit IDs). A partition key alone is a **simple primary key**; a partition key plus sort key is a **composite primary key** that enables rich queries. Beyond the key, an item can have zero or more additional attributes.

KEY TERMS

Table
A collection of items, unique to an AWS account and Region

Item
A uniquely identifiable group of attributes — the core unit of data, like a row

Attribute
A fundamental data element expressed as a key-value pair, like a field or column

Partition key
Required hash key that identifies (and distributes) items; alone it forms a simple primary key

Sort key
Optional key that orders items sharing a partition key; with the partition key forms a composite primary key

To query on attributes other than the primary key without a full table scan, DynamoDB offers **secondary indexes**. A **global secondary index (GSI)** creates a read-only copy of the base table pivoted around **new partition and sort keys** — an alternate schema; data is copied **asynchronously** and is **eventually consistent** (usually within one second), you can create or remove a GSI at any time, it has no data-size limit, is provisioned with **separate capacity**, and allows a maximum of **20** per table. A **local secondary index (LSI)** keeps the **same partition key** but an alternate sort key, lets you choose **strong or eventual consistency**, **must be created with the table** (you cannot add or remove it later), consumes the **base table's** read capacity, and allows a maximum of **5** per table.

TABLE 6.6 GSI vs. LSI at a glance

DIMENSION	GLOBAL SECONDARY INDEX	LOCAL SECONDARY INDEX
Keys	New partition key + optional new sort key	Same partition key, different sort key
Consistency	Eventually consistent reads only	Strong or eventual (you choose)
Created	Any time; add/remove freely	Only with the table; cannot change later
Max per table	20	5
Capacity / size	Separate capacity; no size limit	Uses base-table capacity; size limits apply

By default DynamoDB replicates across multiple AZs in a single Region. **Global tables** extend that to a **multi-Region, multi-active** database: a collection of replica tables owned by one account, each storing the same data, that DynamoDB automatically replicates and keeps conflict-free. Applications read and write to any replica for fast **local** single-digit-ms performance, capacity auto-scales per Region, and the data stays available even if an entire Region is degraded — ideal for a global customer base spread across, say, the US, Europe, and Asia.

MORE DYNAMODB FEATURES AND SECURITY

- **DynamoDB Streams** — change data capture that records a time-ordered sequence of every item-level change in near-real time; ideal for **event-driven** architectures, with before/after views.
- **Point-in-time recovery (PITR)** — continuous backups protecting against accidental writes/deletes; restore to any second in the preceding **35 days**.
- **Encryption at rest by default** — all user data in tables, indexes, streams, and backups is encrypted with **AWS KMS** keys.
- **Fine-grained access control** — no usernames or passwords; **IAM** policies and conditions can restrict read/write down to specific items and attributes.
- **Detective controls** — monitor operations and KMS key usage with **AWS CloudTrail**, and track configuration and compliance with **AWS Config** and Config rules.

COMMON PITFALL

The consistency and lifecycle rules are prime exam targets. **GSI reads are eventually consistent only**, and a GSI can be added or removed at will. An **LSI must be created when the table is created** and offers strongly consistent reads. Mixing these up — for example expecting strong consistency from a GSI or adding an LSI to an existing table — is a classic wrong answer.

6.5 Purpose-built databases

Application architectures have repeatedly split apart to improve scaling and resiliency, and data storage evolved in parallel. **Relational** databases replaced hierarchical ones that struggled to define relationships. **Data warehouses** — still relational, but **OLAP** systems optimized for read-heavy reporting — separated analytics from the transactional database. As the internet produced more varied, less structured data, the first **non-relational** databases emerged. Finally the cloud's freedom to scale on actual usage, combined with **microservices**, made it attractive to connect each component to a **purpose-built** store rather than one multipurpose database.

TABLE 6.7 The evolution of purpose-built databases

OPPORTUNITY	DATABASE EVOLUTION	AWS EXAMPLE
Improve on hierarchical databases' limited ability to relate data	Relational	Amazon RDS
Separate read-heavy reporting from the transactional database	Data warehouse / OLAP	Amazon Redshift
Analyze the more varied data generated on the internet	Non-relational	Amazon DynamoDB
Scale stores on actual usage and connect microservices to fit-for-purpose data	Purpose-built (document, wide-column, in-memory, graph, timeseries, ledger)	DocumentDB, Keyspaces, MemoryDB, Neptune, Timestream, QLDB

Amazon Redshift is a fully managed, cloud-based **data warehousing** service for **petabyte-scale analytics (OLAP)**. It achieves efficient storage and optimum query performance through **massively parallel processing, columnar storage, and targeted compression** — columnar storage dramatically reduces disk I/O. Redshift is an enterprise-class relational query and management system that connects to BI, reporting, and analytics tools, and **Amazon Redshift Serverless** adjusts capacity in seconds for unpredictable workloads. Use Redshift when you primarily need to store and analyze massive amounts of data quickly.

Choosing a purpose-built database aligns with the **performance efficiency** pillar — select the right tool for the job. For each candidate, the module examines four elements: **suitable workloads**, the **data model**, **features and benefits**, and **common use cases**. Start from the workload requirements and data model, evaluate the options, then use the chosen database's features and configurations (and reference architectures) to optimize for your use case.

TABLE 6.8 Six purpose-built databases (plus the warehouse)

SERVICE	DATA MODEL	SUITABLE WORKLOADS	COMMON USE CASES
Amazon DocumentDB	Document (MongoDB-compatible)	Flexible schema for fast iterative dev; items with differing attributes	Content management, customer profiles, operational data stores
Amazon Keyspaces	Wide-column (Cassandra, CQL)	Fast querying of high volumes; heavy or balanced write loads	Industrial equipment maintenance, trade monitoring, route optimization
Amazon MemoryDB	In-memory (Redis), durable	Latency-sensitive, high request rate and throughput, no data loss	Caching, game leaderboards, banking transactions
Amazon Neptune	Graph (nodes and edges)	Find connections/paths; navigate highly connected data	Recommendation engines, fraud detection, knowledge graphs, drug discovery
Amazon Timestream	Timeseries	Identify patterns and trends over time; efficient analytics	IoT trend analysis, real-time web-traffic performance
Amazon QLDB	Ledger (immutable)	Accurate, verifiable history of application data	Financial transactions, claim history, supply-chain tracking

A few defining traits are worth memorizing. **DocumentDB** stores and queries JSON-like documents in the same model developers use in code, and integrates with AWS DMS to migrate MongoDB workloads with virtually no downtime.

Keyspaces is a two-dimensional key-value (wide-column) store queried with **CQL**, strong on heavy writes. **MemoryDB** keeps the entire dataset in memory but adds a distributed transactional log for **durability**, unlike a pure cache. **Neptune** supports Gremlin, SPARQL, and openCypher and up to 15 read replicas. **Timestream** is serverless with built-in timeseries functions (smoothing, interpolation) queried in SQL. **QLDB** provides a transparent, immutable, cryptographically verifiable transaction log queried with **PartiQL**.

> **COMMON PITFALL**
>
> "Purpose-built" means each database trades generality for a strength — so do not force a workload onto the wrong one. A **ledger** database like QLDB is optimized for an immutable, verifiable history and is often **slower**, so it is not ideal for complex or fast reads and writes. Likewise, use ElastiCache or MemoryDB for caching and low latency, Neptune for relationships, and Redshift for analytics — not DynamoDB or RDS stretched to cover all of them.

6.6 Migrating data into AWS databases

AWS Database Migration Service (AWS DMS) is a managed migration and replication service that helps move existing database and analytics workloads **to and within AWS**. It supports most widely used commercial and open-source databases and can replicate data **on demand or on a schedule**. The two data stores are called **endpoints**, and **one endpoint must be on an AWS service**. Migrations are **homogeneous** (same source and target engine) or **heterogeneous** (different engines), and the **source database stays fully operational** during migration to minimize downtime.

> **WHAT AWS DMS CAN (AND CANNOT) DO**
>
> - Migrate between the **same** engine (homogeneous) or **different** engines (heterogeneous).
> - Continuously **replicate with low latency** from any supported source to any supported target — for example into **Amazon S3** to build a data lake, or into **Amazon Redshift** to consolidate a warehouse.
> - Supported **sources** include Oracle, SQL Server, MySQL, MariaDB, PostgreSQL, Db2 LUW, SAP, MongoDB, and Aurora; **targets** include Oracle, SQL Server, PostgreSQL, MySQL, Redshift, SAP ASE, S3, and DynamoDB.
> - **Constraint:** you cannot use DMS to migrate from one on-premises database to another on-premises database.

A **homogeneous** migration moves a self-managed, on-premises, or cloud database to the **equivalent engine** on Amazon RDS or Aurora — for example on-premises PostgreSQL to RDS for PostgreSQL or Aurora PostgreSQL. It is **serverless** (DMS auto-scales the resources) and uses **native database tools** for high-performing like-to-like transfers, operating through instance profiles, data providers, and replication projects.

A **heterogeneous** migration changes engines and needs schema conversion. DMS provides three tools to automate assessment and conversion at scale; objects that cannot be converted automatically are flagged as action items with instructions for manual conversion. A replication runs on an **AWS DMS replication instance** — a managed EC2 instance that hosts one or more replication tasks between the source and target endpoints.

TABLE 6.9 Tools for heterogeneous migrations

TOOL	ROLE	WHAT IT DOES
AWS DMS Fleet Advisor	Discovery	Automatically inventories and assesses an on-premises database and analytics fleet and recommends engine and instance options, in hours
AWS Schema Conversion Tool (AWS SCT)	Schema conversion	Standalone app you download; converts the source schema and SQL code into an equivalent target schema and code
AWS DMS Schema Conversion	Schema conversion	A centrally managed schema assessment and conversion service available within AWS DMS workflows

Migration is not only lift-and-shift. A common pattern replicates data **from a database into a data lake**: for example, a higher-education Student Information System on premises is connected via a DMS source endpoint, a replication task moves the data, and a destination endpoint lands it in an **Amazon S3** raw bucket where analytics and visualization services derive insight. In the course's café scenario, this is architecture **Version 3** — separating the web and database layers by migrating the café database to Amazon RDS on a private subnet to cut maintenance effort and secure the data.

TABLE 6.10 The evolving café architecture

VERSION	BUSINESS REASON	ARCHITECTURE UPDATE
V1	Create a static website for a small business	Host the website on Amazon S3
V2	Add online ordering	Deploy the web application and database on Amazon EC2
V3	Reduce database maintenance and secure the data	Separate the web and database layers; migrate the database to Amazon RDS on a private subnet
V4	Enhance web application security	Use Amazon VPC features to configure public and private subnets
V5	Create role-based access	Add IAM groups, attach resource policies, add users to groups by role
V6	Handle an expected traffic increase	Add a load balancer, auto scaling, and two-AZ distribution of compute and database

> **COMMON PITFALL**
>
> AWS DMS always needs AWS on one side — **one endpoint must be an AWS service**, so it cannot migrate on-premises to on-premises. Also keep the two conversion tools straight: **AWS SCT** is a standalone application you download to your local drive, while **AWS DMS Schema Conversion** is the fully managed equivalent inside the DMS console.

6.7 Applying Well-Architected Framework principles to the database layer

The **AWS Well-Architected Framework** has six pillars, each with best practices and questions to consider. For the database layer, three pillars stand out — **performance efficiency**, **security**, and **cost optimization** — echoing the architect's two overarching goals: evaluate the available database options so you can optimize performance, and secure the infrastructure so data is durable and safe from threats.

THREE PILLARS FOR THE DATABASE LAYER

- **Performance efficiency** — architecture selection: use a **data-driven approach**, evaluate how trade-offs impact customers and efficiency, and factor cost into the choice.
- **Security** — data protection at rest: **implement secure key management** and **enforce encryption at rest**.
- **Cost optimization** — cost-effective resources: **select the resource type, size, and number based on data** about the workload.

For **architecture selection**, the optimal database varies by availability, consistency, partition tolerance, latency, durability, scalability, and query needs — and many systems use different databases for different sub-systems. Understand your data characteristics, evaluate the available options, and choose storage based on **access patterns**. Document anticipated data and traffic growth, run **load tests** to find key metrics and bottlenecks, and know when to use read replicas, global tables, partitioning, and caching rather than simply increasing instance size. This module's toolkit maps directly onto that: RDS for relational data, Redshift for a warehouse, DynamoDB for low-latency key-value at scale, or another purpose-built database — tuned with instance sizing, Multi-AZ, read replicas, RDS Proxy, secondary indexes, and global tables.

For **data protection**, encryption preserves confidentiality if storage is accessed without authorization. Define an encryption approach covering key **storage, rotation, and access control** — **AWS KMS** provides durable, secure, redundant key storage and integrates with many services — and ensure the only way to store data is encrypted. In this module, both **Amazon RDS** and **DynamoDB** use KMS: DynamoDB encrypts all user data at rest (tables, indexes, streams, backups) by default, and RDS encrypts the underlying storage plus all logs, backups, and snapshots, decrypting transparently with minimal performance impact.

For **cost-effective resources**, selecting the best type, size, and number of resources meets the technical requirement at the **lowest cost**. **Right-sizing** considers every resource and attribute and is **iterative** — revisited as usage patterns change or as AWS drops prices or releases new resource types. Base the selection on data about the workload (compute, memory, throughput, write intensity). The module's examples fit: **Aurora** offers better performance at lower cost than standard engines, and **Aurora Serverless** provides on-demand scaling so you avoid overprovisioning.

WORKED EXAMPLE – THE MODULE'S SAMPLE EXAM QUESTION

An application requires a highly available relational database with an initial storage capacity of 8 TB, growing 8 GB per day, and needs at least eight read replicas to handle reads. Which option meets these requirements? Isolate the key words: **highly available relational database**, **grows daily**, **read replicas**. The answer is **Amazon Aurora** — a fully managed relational database that supports **up to 15 read replicas** and whose storage **auto-scales** with the cluster volume. Why not the distractors? **DynamoDB** is non-relational; **Neptune** is a purpose-built graph database; and **Amazon Redshift** does not support read replicas.

COMMON PITFALL

The sample question's fastest eliminations come from remembering two facts: a **read replica requirement** rules out **Redshift** (it has none), and a **relational requirement** rules out **DynamoDB** (key-value) and **Neptune** (graph). Read-replica scale plus relational HA points to **Aurora** almost every time.

Module summary

- The one-size-fits-all relational database is obsolete; choose by working backward from the business need and weighing scalability, storage, data characteristics and access patterns, durability, and cost.
- Relational databases use strict schemas, SQL, and ACID integrity and suit OLTP; non-relational (NoSQL) databases use flexible schemas and purpose-built models (key-value, document, graph, in-memory, wide-column) for high throughput. Both can scale vertically and horizontally.
- Amazon RDS is a managed relational service with seven engines (including Aurora) that automates provisioning, patching, backup, recovery, and failover; on managed services you are responsible only for optimizing your application and queries.
- Aurora is a cloud-built, MySQL/PostgreSQL-compatible engine at about one-tenth the cost, with self-healing storage auto-scaling to 64 TB across three AZs and up to 15 read replicas; Aurora Serverless auto-scales capacity on demand.
- Multi-AZ deployments replicate synchronously to a standby for high availability with automatic failover; read replicas replicate asynchronously for read scaling and are manually promoted. RDS Proxy pools connections, cuts failover time up to 66%, and enforces IAM auth via Secrets Manager.
- RDS backups come as automated backups (point-in-time, 7–35 day retention) and user-initiated snapshots (shareable, kept until deleted); RDS encrypts at rest with KMS and in transit with SSL/TLS, and supports cross-Region copies and read replicas.
- DynamoDB is a fully managed, serverless NoSQL database (key-value + document) with single-digit-ms performance, keyed by a partition key plus optional sort key; GSIs use new keys and are eventually consistent (add anytime, max 20), while LSIs share the partition key, allow strong reads, and are created with the table (max 5).
- DynamoDB adds global tables (multi-Region, multi-active), Streams (change data capture), point-in-time recovery, default encryption at rest, and fine-grained IAM access control.
- Purpose-built databases match a data shape to a service: Redshift (OLAP warehouse), DocumentDB (document), Keyspaces (wide-column), MemoryDB (durable in-memory), Neptune (graph), Timestream (timeseries), and QLDB (ledger).
- AWS DMS migrates data between endpoints (one must be on AWS) via homogeneous (same engine) or heterogeneous (different engine) migrations; AWS SCT and DMS Schema Conversion convert schema and code, and Fleet Advisor discovers on-premises fleets — but DMS cannot migrate on-premises to on-premises.
- Apply the Well-Architected Framework to the database layer: performance efficiency (select based on data characteristics and access patterns), security (secure key management and encryption at rest with KMS), and cost optimization (right-size resource type, size, and number based on data).
- Sample-exam pattern: a highly available relational database needing many read replicas and growing storage points to Amazon Aurora — not DynamoDB (non-relational), Neptune (graph), or Redshift (no read replicas).

Review questions

1. You need a highly available relational database with automatic failover. Which RDS feature provides it, and how does its replication differ from a read replica's?

Answer: A Multi-AZ deployment — it synchronously replicates to a standby in another AZ and fails over automatically for high availability. Read replicas replicate asynchronously for read scaling and must be manually promoted to become standalone databases.

2. Contrast a DynamoDB GSI and LSI on keys, consistency, and when each can be created.

Answer: A GSI uses a new partition key (and optional new sort key), supports only eventually consistent reads, and can be added or removed anytime (max 20). An LSI keeps the same partition key with a different sort key, allows strong or eventual consistency, and must be created with the table (max 5).

3. An ecommerce order system stores structured, transactional data and needs ACID guarantees. Which database category and AWS service fit?

Answer: The relational (SQL) category — Amazon RDS or Aurora, which are suited to OLTP and provide ACID transactions and data integrity.

4. Match each need to a purpose-built database: (a) highly connected data with recommendations, (b) immutable, verifiable financial history, (c) petabyte-scale analytics and reporting, (d) IoT timeseries data.

Answer: (a) Amazon Neptune (graph); (b) Amazon QLDB (ledger); (c) Amazon Redshift (data warehouse / OLAP); (d) Amazon Timestream (timeseries).

5. What is the difference between a homogeneous and a heterogeneous AWS DMS migration, and which tool converts schema and code for the heterogeneous case?

Answer: Homogeneous migrations keep the same source and target engine (for example on-premises PostgreSQL to RDS for PostgreSQL); heterogeneous migrations change engines and require AWS SCT (or AWS DMS Schema Conversion) to convert the schema and SQL code.

6. Name three benefits RDS Proxy provides, and describe what decreases at the database as a result.

Answer: Connection pooling for scalability, faster failover (up to 66% for Multi-AZ) for resilience, and IAM authentication with AWS Secrets Manager for security; the database receives fewer open connections, so it does not exhaust memory or compute.

7. Which three Well-Architected Framework pillars did the module highlight for the database layer, with one best practice each?

Answer: Performance efficiency — select the database based on data characteristics and access patterns; Security — implement secure key management and enforce encryption at rest with AWS KMS; Cost optimization — select resource type, size, and number based on data.

8. An application needs a highly available relational database with 8 TB of storage growing 8 GB per day and at least eight read replicas. Which AWS database fits, and why not DynamoDB, Neptune, or Redshift?

Answer: Amazon Aurora — it is relational and highly available, supports up to 15 read replicas, and auto-scales storage. DynamoDB is non-relational, Neptune is a graph database, and Redshift does not support read replicas.