# **Amazon RDS & DynamoDB**

## **1️⃣ Amazon RDS (Relational Database Service)**

### **🔹 What is Amazon RDS?**

Amazon RDS is a managed relational database service supporting multiple engines:

* MySQL, PostgreSQL, MariaDB, Oracle, SQL Server, Aurora
* Handles provisioning, patching, backups, scaling, and failover automatically.

🛠️ Provisioning — Amazon RDS sets up your database server for you. You don’t need to manually install or configure anything.

🩹 Patching — It automatically applies security updates and software fixes, so your database stays safe and up to date without you lifting a finger.

💾 Backups — RDS regularly takes backups of your data so that if something goes wrong, you can restore it easily.

📈 Scaling — If your app grows and needs more power (like more storage or faster performance), RDS can scale up automatically.

🔄 Failover — If something crashes (like the main server), RDS quickly switches to a standby one, so your application keeps running with minimal interruption.

✅ Why Use RDS?  
✔ Fully managed (No infrastructure overhead)  
✔ High availability (Multi-AZ deployments)  
✔ Automated backups & point-in-time recovery  
✔ Security: Encryption at rest & in transit, IAM integration

### **🔹 Creating a DB Instance (Production Best Practices)**

#### **Step 1: Choose the Right Engine & Version**

* PostgreSQL/MySQL (Open-source, cost-effective)
* Aurora (High performance, AWS-optimized)
* Enterprise: Oracle/SQL Server (Licensing included)

📌 Best Practice: Use the latest stable version unless legacy compatibility is needed.

#### **Step 2: Configure Instance Size & Storage**

* Instance Class:
  + General Purpose (db.t3, db.m6g) – Balanced workloads
  + Memory Optimized (db.r6g) – High-performance apps
  + Burstable (db.t3) – Dev/Test (Avoid in production for steady workloads)
* Storage:
  + GP3 (SSD) – Default, scalable IOPS
  + Provisioned IOPS (io1/io2) – High-throughput OLTP

📌 Best Practice:

* Start with GP3 (cost-effective, scalable).
* Monitor CloudWatch metrics (CPU, Memory, Read/Write IOPS) and adjust.

#### **Step 3: High Availability & Backup**

* Multi-AZ Deployment (Automatic failover for production)
* Backup Retention:
  + Automated Backups (7-35 days)
  + Manual Snapshots (Long-term retention)

📌 Best Practice:  
✔ Enable Multi-AZ for production (RTO < 2 min).  
✔ Set backup retention ≥ 7 days (Compliance requirement for most cases).

#### **Step 4: Security & Networking**

* VPC & Security Groups:
  + Place RDS in private subnets (No public access).
  + Restrict access via Security Groups (Allow only app servers).
* Encryption:
  + Enable at-rest encryption (AWS KMS).
  + Enforce SSL/TLS for in-transit encryption.

📌 Best Practice:  
✔ Use IAM Authentication (No password rotation hassle).  
✔ Enable AWS Backup for cross-region disaster recovery.

#### **Step 5: Monitoring & Maintenance**

* CloudWatch Alarms:
  + CPUUtilization > 80%
  + FreeStorageSpace < 20%
  + DatabaseConnections > 80% of max\_connections
* Performance Insights: Enable for query tuning.

📌 Best Practice:  
✔ Set maintenance window during low-traffic periods.

## **2️⃣ Amazon DynamoDB (NoSQL Database)**

### **🔹 What is DynamoDB?**

* Fully managed NoSQL (Key-Value & Document DB)
* Serverless, auto-scaling, single-digit millisecond latency
* Use Cases: High-traffic web apps, gaming, IoT, real-time analytics

✅ Why Use DynamoDB?  
✔ Infinite scalability (No manual sharding)  
✔ Low-latency reads/writes (SSD-backed)  
✔ Built-in replication (Global Tables)  
✔ Pay-per-request pricing (No idle costs)

### **🔹 Getting Started with DynamoDB (Production Best Practices)**

#### **Step 1: Data Modeling (Critical for Performance)**

* Primary Key Design:
  + Partition Key (HASH) – Must distribute evenly (Avoid hot partitions).
  + Composite Key (HASH + RANGE) – For hierarchical queries.

📌 Best Practice:  
✔ Use UUIDs or composite keys to avoid hotspots.  
✔ GSIs (Global Secondary Indexes) for flexible querying.

#### **Step 2: Capacity Modes**

* Provisioned Mode (Predictable workloads)
  + Set Read/Write Capacity Units (RCU/WCU)
* On-Demand Mode (Spiky workloads)
  + No capacity planning, pay per request

📌 Best Practice:  
✔ Start with On-Demand if traffic is unpredictable.  
✔ Switch to Provisioned + Auto-Scaling for cost savings at steady loads.

#### **Step 3: Global Tables (Multi-Region Replication)**

* Active-Active replication (Low-latency global access)
* Conflict resolution: Last-write-wins (LWW)

📌 Best Practice:  
✔ Enable Global Tables if users are distributed worldwide.

#### **Step 4: Security & Backup**

* Encryption: Enabled by default (AWS KMS)
* IAM Policies: Fine-grained access control
* Backups:
  + Point-in-Time Recovery (PITR) – 35-day rolling backup
  + On-Demand Backups – Manual, retained indefinitely

📌 Best Practice:  
✔ Enable PITR for production (RPO in seconds).

#### **Step 5: Monitoring & Optimization**

* CloudWatch Metrics:
  + ThrottledRequests → Scale up RCU/WCU
  + ConsumedRead/WriteCapacity → Optimize provisioning
* DAX (DynamoDB Accelerator): Microsecond caching

📌 Best Practice:  
✔ Use TTL (Time-to-Live) for automatic data expiry (e.g., sessions).

## **🏁 Summary: RDS vs DynamoDB (When to Use?)**

| Feature | **RDS (SQL)** | **DynamoDB (NoSQL)** |
| --- | --- | --- |
| Use Case | Complex queries, transactions | High-scale, low-latency apps |
| Scalability | Vertical (instance size) | Horizontal (auto-scale) |
| Latency | Sub-ms to ms (depends on config) | Single-digit ms |
| Pricing | Per instance + storage | Per request or provisioned capacity |
| Best For | ERP, CRM, legacy apps | Serverless apps, gaming, IoT |

📌 Final Recommendation:  
✔ Use RDS for structured data with complex joins.  
✔ Use DynamoDB for high-scale, low-latency key-value access.

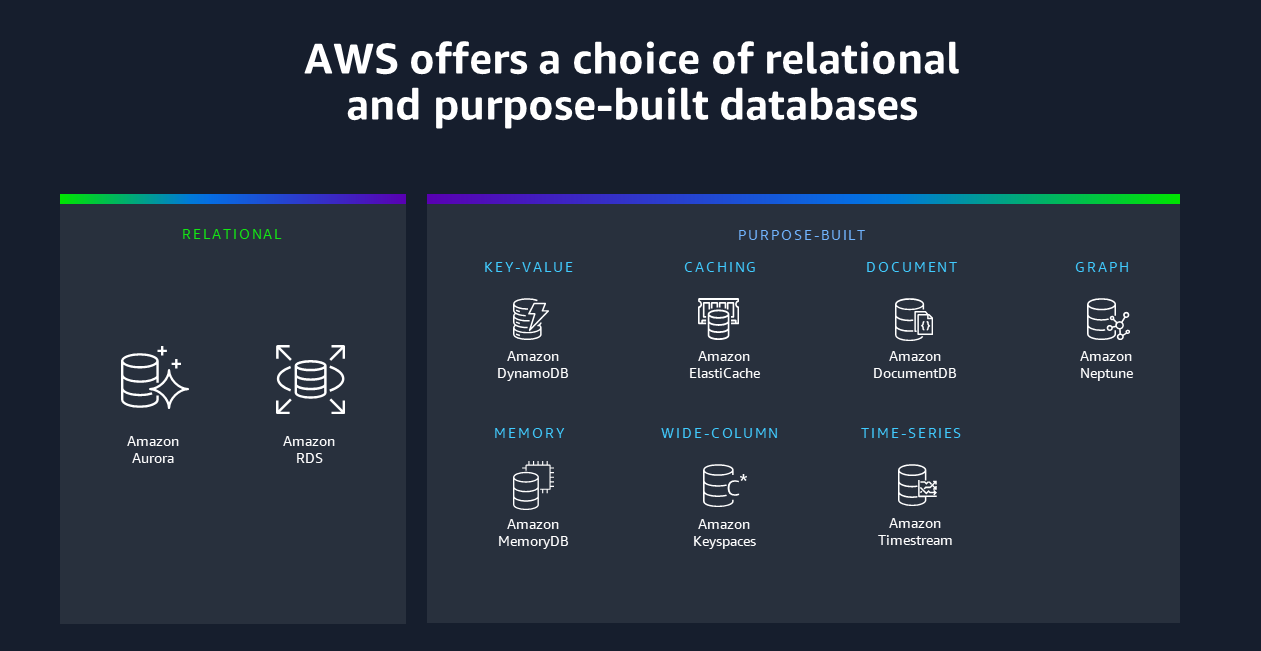
## **🚀 Production Checklist**

### **For RDS:**

✅ Enable Multi-AZ for HA  
✅ Configure automated backups & PITR  
✅ Set CloudWatch alarms for performance  
✅ Restrict access via Security Groups & IAM

### **For DynamoDB:**

✅ Choose On-Demand or Provisioned + Auto-Scaling  
✅ Design efficient partition keys  
✅ Enable Global Tables if multi-region  
✅ Use DAX for read-heavy workloads



| Data model | When would you use it? | What is it optimized for? | Related database engines or services |
| --- | --- | --- | --- |
| **Relational** | Use when you're migrating or modernizing an on- premises relational workload, or if your workload has less predictable query patterns. | Optimized for structured data that is stored in tables, rows, and columns. Relational databases support complex queries through joins. | [Amazon Aurora](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html)  [Amazon RDS](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html) |
| **Key-value** | Use for workloads such as session stores or shopping carts. Key-value databases can scale to large amounts of data and extremely high throughput of requests, while servicing millions of simultaneous users through distributed processing and storage. | Optimized to provide a serverless, NoSQL, fully managed database with single-digit millisecond performance at any scale. | [Amazon DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html) |
| **In-memory** | Use Amazon ElastiCache when you need a caching layer to improve read performance. Use Amazon MemoryDB when you need full data persistence, but still need sub-millisecond read latencies. | Optimized to support microsecond reads and sub-millisecond writes. MemoryDB supports microsecond reads and single-digit millisecond writes. ElastiCache is an ephemeral cache, while MemoryDB is an in-memory database. | [Amazon ElastiCache](https://docs.aws.amazon.com/elasticache/)  [Amazon MemoryDB](https://docs.aws.amazon.com/memorydb/latest/devguide/what-is-memorydb-for-redis.html) |
| **Document** | Use when you want to store JSON-like documents with rich querying abilities across the fields of the documents. | Optimized for storing semi-structured data as documents with multilayered attributes. | [Amazon DocumentDB (with MongoDB compatibility)](https://docs.aws.amazon.com/documentdb/latest/developerguide/what-is.html) |
| **Wide-column** | Use when you need to migrate your on-premises Cassandra workloads, or when you need to process data at high speeds for applications that require single-digit millisecond latency. | Optimized for workloads that require heavy reads/writes and high throughput, coupled with low latency and linear scalability. | [Amazon Keyspaces (for Apache Cassandra)](https://docs.aws.amazon.com/keyspaces/latest/devguide/what-is-keyspaces.html) |
| **Graph** | Use when you have to model complex networks of objects, such as social networks, fraud detection, and recommendation engine use cases. | Optimized for traversing and evaluating large numbers of relationships, and identifying patterns with minimal latency. | [Amazon Neptune](https://docs.aws.amazon.com/neptune/) |
| **Time series** | Use when you have a large amount of time series data, potentially from a number of sources, such as Internet of Things (IoT) data, application metrics, and asset tracking. | Optimized for storing and querying data that is associated with timestamps and trend lines. | [Amazon Timestream](https://docs.aws.amazon.com/timestream/) |

📚 Official Docs:

* [Amazon RDS](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html)
* [Amazon DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html)