**📌 Day 1 – Multi-Region Active-Active Architecture in Azure (6:30 -9:30)**

✅ Learn **Active-Active vs. Active-Passive** design patterns  
✅ Explore **Azure Front Door, Azure Traffic Manager, and Cosmos DB multi-region setup**  
✅ Hands-on: Deploy a simple **multi-region web app** using Azure Front Door

📌 **Outcome:** Understanding of **high availability & disaster recovery strategies**

**📌 Day 2 – Data Strategy: Sharding vs. Replication in Azure SQL & Cosmos DB(3:30 to 6:30)**

✅ Learn **when to use sharding vs. replication**  
✅ Compare **Azure SQL Hyperscale vs. Cosmos DB partitioning**  
✅ Hands-on: Implement **a simple sharded database in Azure SQL**

📌 **Outcome:** Stronger **data architecture skills** for **scalable** solutions

**📌 Day 3 – Enterprise API Security & Gateway Design(12:30 to 3:30)**

✅ Learn **OAuth 2.0, OpenID Connect, API Keys**  
✅ Explore **Azure API Management (APIM) for security**  
✅ Hands-on: Secure an API using **Azure AD Authentication**

📌 **Outcome:** Ability to **design secure APIs** for enterprise applications

**📌 Day 4 – Cloud Migration Strategy for .NET Apps (9:30 to 12:30)**

✅ Study **Lift-and-Shift vs. Replatforming vs. Refactoring**  
✅ Explore **Azure App Service, AKS (Kubernetes), and Azure Functions**  
✅ Hands-on: Migrate a simple .NET app to **Azure App Service**

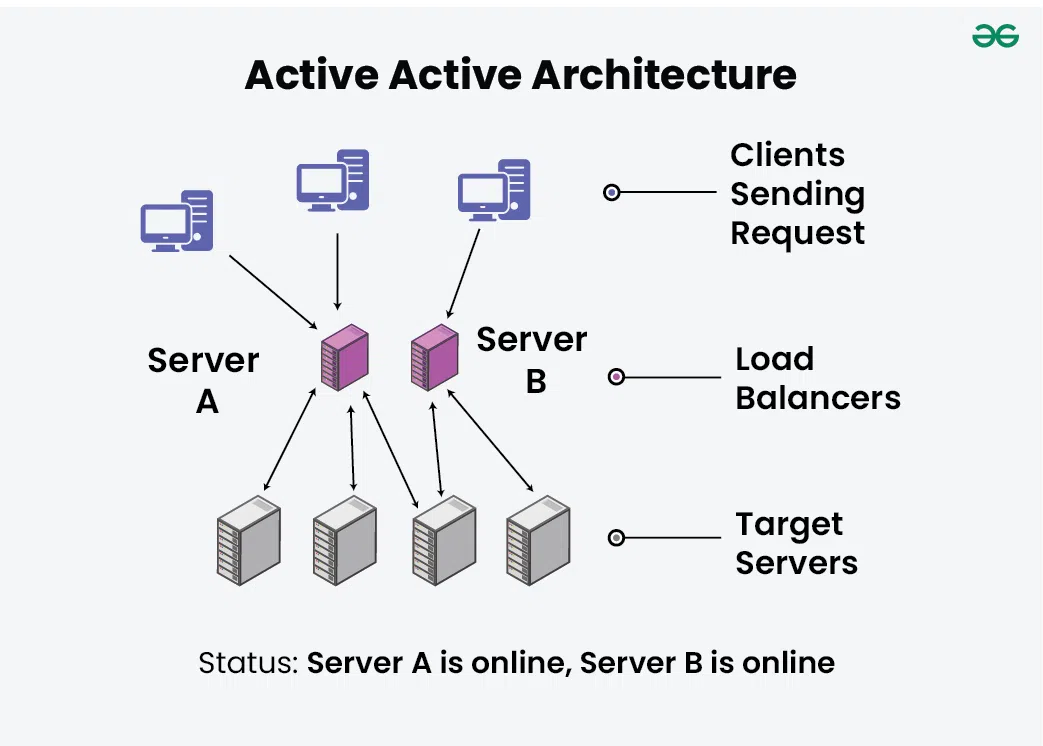
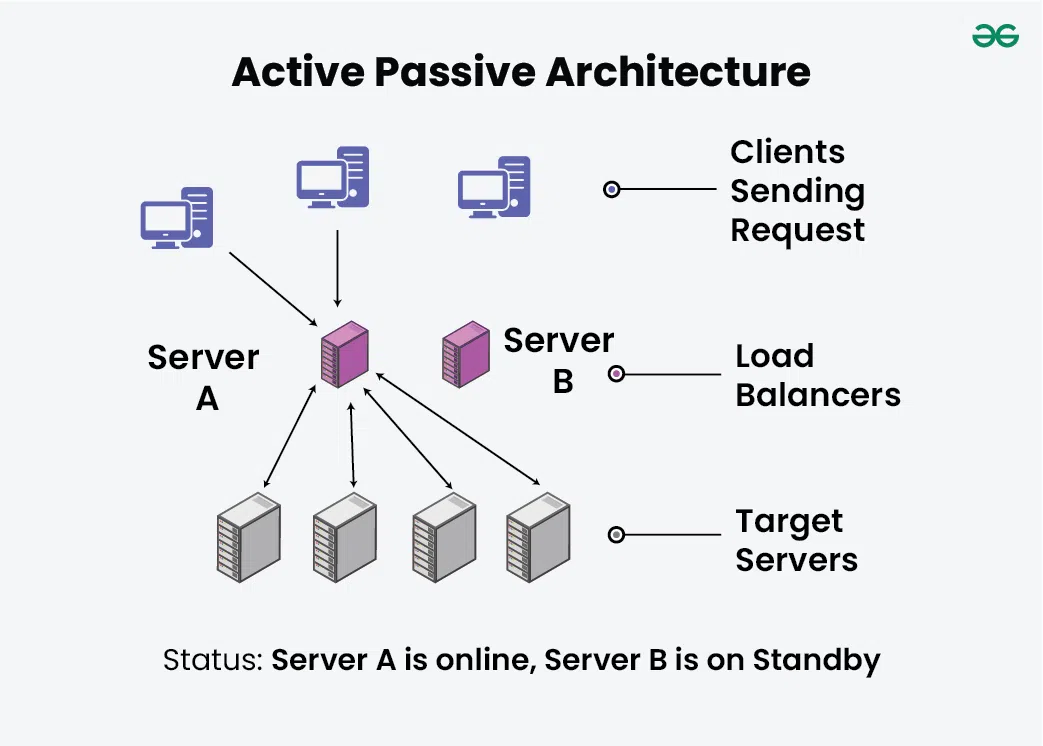
📌 **Outcome:** Understanding of **modernization approaches** for enterprises

**📌 Day 1 – Multi-Region Active-Active Architecture in Azure**

**cost, availability, and failover strategy.**

✅ Learn **Active-Active vs. Active-Passive** design patterns

<https://www.geeksforgeeks.org/active-passive-active-active-architecture-for-high-availability-system/>

  
✅ Explore **Azure Front Door, Azure Traffic Manager, and Cosmos DB multi-region setup**

✅ Hands-on: Deploy a simple **multi-region web app** using Azure Front Door

📌 **Outcome:** Understanding of **high availability & disaster recovery strategies**

**📌 Day 2 – Data Strategy: Sharding vs. Replication in Azure SQL & Cosmos DB**

| **Step** | **Action** |
| --- | --- |
| **1** | **Create 3 separate tables (Table\_1, Table\_2, Table\_3)** |
| **2** | **Implement an Insert Stored Procedure to route records** |
| **3** | **Use a UNION ALL search query for multi-table lookups** |
| **4** | **Add indexes on ID for faster queries** |
| **5** | **Use a RecordLocator table to optimize search performance** |

✅ Learn **when to use sharding vs. replication**

Sharding is a technique used to split large databases into smaller

*Sharding is a database architecture pattern related to*horizontal partitioning *the practice of separating one table’s rows into multiple different tables, known as partitions. Each partition has the same schema and columns, but also entirely different rows. Likewise, the data held in each is unique and independent of the data held in other partitions.*

A diagram of a database

AI-generated content may be incorrect.

**Sharding types**

**Sharding by Key Range**

<https://www.youtube.com/playlist?list=PLmCXsOwg2OK5qlKgSrGm3pMX0WgPayK3d>

**Step-by-Step Guide for Database Sharding in SQL Server**

**📌 Scenario**

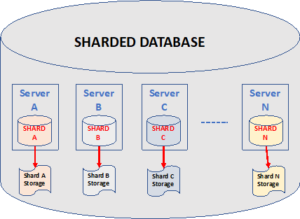
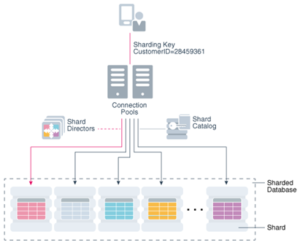
* You have **3 million records** and want to distribute them across **three tables** dynamically:
  + **Table\_1:** Stores records **1 to 1,000,000**
  + **Table\_2:** Stores records **1,000,001 to 2,000,000**
  + **Table\_3:** Stores records **2,000,001 to 3,000,000**
* **New inserts should go into the appropriate table based on record count.**
* **Read queries should efficiently fetch records from the correct table.**

**Step 1: Create the Sharded Tables**

|  |
| --- |
| **CREATE TABLE Table\_1 (**  **ID BIGINT PRIMARY KEY,**  **Name NVARCHAR(255),**  **CreatedDate DATETIME DEFAULT GETDATE()**  **);**  **CREATE TABLE Table\_2 (**  **ID BIGINT PRIMARY KEY,**  **Name NVARCHAR(255),**  **CreatedDate DATETIME DEFAULT GETDATE()**  **);**  **CREATE TABLE Table\_3 (**  **ID BIGINT PRIMARY KEY,**  **Name NVARCHAR(255),**  **CreatedDate DATETIME DEFAULT GETDATE()**  **);** |
| **Step 2: Create a Partitioning Function (Alternative)**  **-** **Instead of manually moving records, you can use partitioning, but for now, we will use separate tables.** |
| **Step 3: Implement an Insert Logic with Dynamic Table Selection**  **Since SQL Server doesn’t support automatic dynamic table selection, we implement it in Stored Procedure:**  CREATE PROCEDURE InsertRecord  @ID BIGINT,  @Name NVARCHAR(255)  AS  BEGIN  IF @ID <= 1000000  INSERT INTO Table\_1 (ID, Name) VALUES (@ID, @Name);  ELSE IF @ID > 1000000 AND @ID <= 2000000  INSERT INTO Table\_2 (ID, Name) VALUES (@ID, @Name);  ELSE  INSERT INTO Table\_3 (ID, Name) VALUES (@ID, @Name);  END;  ✅ Now, when inserting data, SQL Server will route records to the right table based on ID. |
| **Step 4: Searching for Records Across All Shards**  **Since data is spread across multiple tables, searching efficiently requires a UNION ALL query.**  **SELECT \* FROM Table\_1 WHERE ID = @SearchID**  **UNION ALL**  **SELECT \* FROM Table\_2 WHERE ID = @SearchID**  **UNION ALL**  **SELECT \* FROM Table\_3 WHERE ID = @SearchID;** |
| **Step 5: Indexing Strategy for Efficient Search**  **CREATE INDEX IDX\_Table\_1\_ID ON Table\_1 (ID);**  **CREATE INDEX IDX\_Table\_2\_ID ON Table\_2 (ID);**  **CREATE INDEX IDX\_Table\_3\_ID ON Table\_3 (ID);** |

A diagram of a number of shard

AI-generated content may be incorrect.

Other way:

**Advanced Database Sharding Using Partitioning in SQL Server**

Instead of **multiple tables (Table\_1, Table\_2, Table\_3)**, we can use **Table Partitioning** to improve performance and reduce complexity.

**📌 Why Use Partitioning Instead of Multiple Tables?**

✅ **Better Performance:** SQL Server optimizes partition reads instead of scanning all tables.  
✅ **Easier Maintenance:** No need to manage multiple tables manually.  
✅ **Automatic Partition Selection:** SQL Server automatically directs inserts to the correct partition.

**🚀 Step-by-Step Guide for Table Partitioning in SQL Server**

**Step 1: Create a Partition Function**

Partitioning works by defining a **range** for each partition.  
We will create partitions for:

* **Partition 1:** 1 - 1,000,000
* **Partition 2:** 1,000,001 - 2,000,000
* **Partition 3:** 2,000,001 - 3,000,000

CREATE PARTITION FUNCTION pf\_RecordRange (BIGINT)

AS RANGE RIGHT FOR VALUES (1000000, 2000000);

🔹 This function **divides the data** into different partitions based on ID.

**Step 2: Create a Partition Scheme**

A **Partition Scheme** maps partitions to filegroups (for performance tuning).

CREATE PARTITION SCHEME ps\_RecordScheme

AS PARTITION pf\_RecordRange ALL TO ([PRIMARY]);

🔹 This means **all partitions** will be stored in the **PRIMARY filegroup** (you can create separate filegroups if needed).

**Step 3: Create a Partitioned Table**

Now, we create a **single table** but partitioned based on ID.

CREATE TABLE Records (

ID BIGINT NOT NULL,

Name NVARCHAR(255),

CreatedDate DATETIME DEFAULT GETDATE(),

PRIMARY KEY (ID)

) ON ps\_RecordScheme(ID);

🔹 **Now, SQL Server will store records in the correct partition automatically!**

**Step 4: Insert Data**

Now, we can **insert records normally**, and SQL Server **automatically places them in the correct partition**.

sql

CopyEdit

INSERT INTO Records (ID, Name) VALUES (500000, 'John Doe'); -- Goes to Partition 1

INSERT INTO Records (ID, Name) VALUES (1500000, 'Jane Doe'); -- Goes to Partition 2

INSERT INTO Records (ID, Name) VALUES (2500000, 'Alice'); -- Goes to Partition 3

✅ **No need for manual table selection!**

**Step 5: Searching for Records**

You can query normally:

SELECT \* FROM Records WHERE ID = 2500000;

🔹 **Will SQL Server scan all partitions?**  
👉 **No!** SQL Server will directly go to **Partition 3** (where ID 2500000 is stored) instead of scanning all data.

**Step 6: Indexing for Partitioned Tables**

To optimize search performance, **create an index aligned with partitions**:

sql

CopyEdit

CREATE UNIQUE CLUSTERED INDEX IDX\_Records ON Records(ID) ON ps\_RecordScheme(ID);

✅ Now, SQL Server **automatically looks in the right partition** when searching for a record.

**📌 Comparing Partitioning vs. Multiple Tables**

| **Feature** | **Partitioning (Recommended)** | **Multiple Tables (Manual Sharding)** |
| --- | --- | --- |
| **Performance** | 🚀 **Optimized partition scanning** | ❌ **UNION ALL needed (slower)** |
| **Insert Logic** | ✅ **Automatic partition selection** | ❌ **Requires stored procedure** |
| **Search Efficiency** | ✅ **SQL Server selects partition automatically** | ❌ **Searches all tables** |
| **Maintenance** | ✅ **Single table, easy to manage** | ❌ **Multiple tables, complex maintenance** |

**🔥 Final Takeaways**

✔ If using **SQL Server Enterprise Edition**, use **Partitioning** for best performance.  
✔ If on **Standard Edition**, manual sharding with **multiple tables** is the alternative.  
✔ Indexing is **crucial** to ensure efficient queries.

✅ Compare **Azure SQL Hyperscale vs. Cosmos DB partitioning**  
✅ Hands-on: Implement **a simple sharded database in Azure SQL**

📌 **Outcome:** Stronger **data architecture skills** for **scalable** solutions

**📌 Day 4 – Cloud Migration Strategy for .NET Apps (9:30 to 12:30)**

✅ Study **Lift-and-Shift vs. Replatforming vs. Refactoring**  
✅ Explore **Azure App Service, AKS (Kubernetes), and Azure Functions**  
✅ Hands-on: Migrate a simple .NET app to **Azure App Service**

📌 **Outcome:** Understanding of **modernization approaches** for enterprises

**Summary of Cost Considerations:**

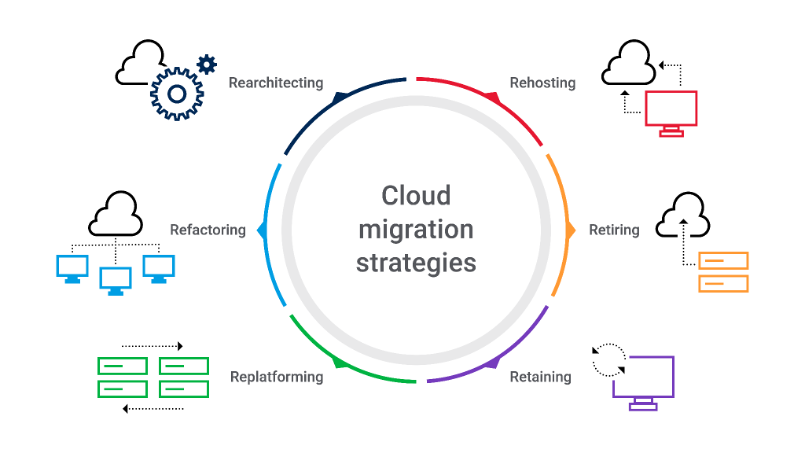
* **Low Cost: Retiring applications, Serverless (if optimized).**
* **Medium Cost: Replatforming (PaaS), Hybrid setup (Retaining).**
* **High Cost: Rehosting (VMs), Rearchitecting (Microservices), Refactoring (Code-level changes).**
* Here’s an updated table that includes cost considerations for each migration strategy:

| **Scenario** | **Strategy** | **Explanation** | **Azure Services** | **Cost Impact** |
| --- | --- | --- | --- | --- |
| Applications to be deployed on **Azure VMs** | **Rehosting** | Lift and shift existing applications to Azure VMs with minimal changes. | Azure Virtual Machines, Azure Load Balancer | **High Cost** (VMs require ongoing management, licensing, and maintenance) |
| Applications to be deployed on **Azure App Services (PaaS)** | **Replatforming** | Move to Azure App Services with slight modifications (e.g., remove dependencies on local storage, IIS configurations). | Azure App Service, Azure SQL Database | **Medium Cost** (Lower management overhead, but App Services pricing varies by usage) |
| Applications to be modernized into **Microservices** | **Rearchitecting** | Break monoliths into microservices for scalability and performance improvement. | Azure Kubernetes Service (AKS), Azure Functions, Azure API Management, Azure SQL Database, Cosmos DB | **High Cost Initially** (Development effort is high but operational efficiency improves over time) |
| Applications that are **slow or need a major overhaul** | **Refactoring** | Modify code significantly to optimize performance, possibly moving to serverless or containerized workloads. | Azure Functions, Azure Kubernetes Service (AKS), Azure Service Bus | **Medium to High Cost** (Development cost is high, but operational costs may reduce with optimization) |
| Applications that are **not in use or no longer needed** | **Retiring** | Identify and decommission non-critical applications to reduce costs. | N/A (Shut down) | **Low Cost** (Cost savings by shutting down unused resources) |
| Applications that will **remain on-premise for now** | **Retaining** | Keep these applications on-premise due to security, compliance, or business reasons but plan for future migration. | Hybrid connectivity (Azure Arc, VPN Gateway, ExpressRoute) | **Medium Cost** (On-prem infrastructure costs remain, but hybrid options may increase costs) |

**Cloud migration strategies: the 6 Rs**

[**https://www.akamai.com/blog/cloud/cloud-migration-strategy**](https://www.akamai.com/blog/cloud/cloud-migration-strategy)

1. [Rehosting](https://www.akamai.com/blog/cloud/cloud-migration-strategy#rehosting)
2. [Retiring](https://www.akamai.com/blog/cloud/cloud-migration-strategy#retiring)
3. [Retaining](https://www.akamai.com/blog/cloud/cloud-migration-strategy#retaining)
4. [Replatforming](https://www.akamai.com/blog/cloud/cloud-migration-strategy#replatforming)
5. [Refactoring](https://www.akamai.com/blog/cloud/cloud-migration-strategy#refactoring)
6. [Rearchitecting](https://www.akamai.com/blog/cloud/cloud-migration-strategy#rearchitecting)

[](javascript:void(0);)

**Challenges of migrating to the cloud**

Even with guidelines and concrete steps to help you toward a successful cloud migration, your organization should be clear-eyed about the challenges common to such an effort. These challenges include:

* **Security and compliance.** Ensuring data security and meeting regulatory requirements can be challenging, especially in [multicloud](https://www.akamai.com/blog/cloud/securing-applications-in-a-multicloud-world)and hybrid cloud environments. Implement robust security controls and work with your cloud service provider to maintain compliance.
* **IT support and skill set.** Migrating to the cloud requires new skills and expertise in areas including DevOps, virtual machines (VMs), and containerization. Invest in training existing staff or hire cloud experts to support the migration process and ongoing management.
* **Budgeting and cost management.**Although the cloud can provide cost savings, unexpected expenses can arise during the migration or because of improper resource management. Develop a comprehensive budget and track expenses to avoid cost overruns.
* **Infrastructure compatibility.**Not all applications and systems are compatible with cloud environments; incompatibility may require modifications or replacements. Assess compatibility early and plan for necessary changes.
* **Data migration and integrity.**A high-volume data transfer can be time-consuming and prone to errors. Use reliable data migration tools and ensure data integrity through validation processes.
* **Downtime and business continuity.** Minimize downtime by scheduling migration during periods of low demand and having a rollback plan in case of issues. Implement disaster recovery and backup strategies for business continuity.
* **Change management and organizational resistance.** Cloud migration can impact workflows, roles, and responsibilities. Address potential resistance through clear communication, training, and stakeholder involvement.

;’]]=-

**How Containerization Improves Application Speed**

| **Factor** | **How It Improves Speed** | **Comparison with Traditional Deployments** |
| --- | --- | --- |
| **Faster Startup & Deployment** | Containers start in **seconds** compared to VMs that take **minutes** | Traditional VMs have a full OS boot process, making them slower |
| **Optimized Resource Utilization** | Containers share the OS kernel, reducing overhead and improving efficiency | VMs have their own OS, consuming more CPU & memory |
| **Improved Scalability** | Can scale **each microservice independently** based on load | Traditional monoliths scale as a whole, leading to inefficiency |
| **Consistent Performance Across Environments** | Containers **eliminate “it works on my machine” issues** by packaging dependencies | App Services and VMs may have version conflicts, causing slowdowns |
| **Auto-healing & Load Balancing** | AKS (Kubernetes) detects slow/unhealthy instances and replaces them automatically | In VMs or App Services, failure recovery is slower |
| **Efficient Caching & API Gateway Integration** | Containers work well with **Redis, API Gateway, and Sidecar patterns**, reducing response times | Traditional deployments may require heavy middleware for caching |
| **Parallel Processing with Multiple Containers** | Workloads can be split into smaller **parallelized tasks** | Traditional monoliths process sequentially, increasing delays |

**When Does Containerization Improve Speed the Most?**

✅ **High-traffic APIs** – Containers help with load balancing and autoscaling.  
✅ **CPU/Memory-intensive applications** – Containers can be optimized for resource usage.  
✅ **Applications with frequent deployments** – Containers reduce downtime with rolling updates.  
✅ **Microservices-based architectures** – Independent scaling improves response times.