AWS Solutions Architect Interview Questions & Answers

Networking & VPC

Q1: Explain the difference between Security Groups and Network ACLs. When would you use each?

Answer:

Security Groups:

- Stateful (return traffic automatically allowed)
- Operate at instance/ENI level
- Support allow rules only
- All rules evaluated before decision
- Can reference other security groups

Network ACLs:

- Stateless (must explicitly allow return traffic)
- Operate at subnet level
- Support both allow and deny rules
- Rules processed in order (lowest number first)
- Cannot reference security groups

When to use:

- Use Security Groups as primary defense for instance-level security
- Use NACLs for subnet-level protection and to block specific IP ranges
- Example: Block a specific malicious IP at NACL level, allow application traffic at Security Group level

---

Q2: Design a highly available and secure VPC architecture for a 3-tier web application.

Answer:

```
Architecture Components:

1. VPC Configuration:
   - CIDR: 10.0.0.0/16
   - 3 Availability Zones
   - Public and Private subnets in each AZ

2. Subnet Design:
   Public Subnets (per AZ):
   - 10.0.1.0/24, 10.0.2.0/24, 10.0.3.0/24
   - For ALB, NAT Gateways, Bastion hosts

   Private App Subnets (per AZ):
   - 10.0.11.0/24, 10.0.12.0/24, 10.0.13.0/24
   - For application servers

   Private DB Subnets (per AZ):
   - 10.0.21.0/24, 10.0.22.0/24, 10.0.23.0/24
   - For RDS instances

3. Routing:
   - Internet Gateway for public subnets
   - NAT Gateway in each AZ for private subnet internet access
   - Route tables: public (IGW), private-app (NAT), private-db (no internet)

4. Security:
   Web Tier SG: Allow 443 from 0.0.0.0/0
   App Tier SG: Allow 8080 from Web Tier SG only
   DB Tier SG: Allow 3306 from App Tier SG only

5. High Availability:
   - Multi-AZ ALB
   - Auto Scaling Groups in multiple AZs
   - RDS Multi-AZ deployment
   - Multi-AZ NAT Gateways
```

Key Design Decisions:

- Separate subnets per tier for security isolation
- Multi-AZ for high availability
- Security groups reference other security groups for maintainability
- No direct internet access to private subnets

Q3: How would you connect multiple VPCs across different regions and accounts?

Answer:

Solution: Transit Gateway with RAM (Resource Access Manager)

Architecture:

```
1. Hub-and-Spoke Model:
   - Central Transit Gateway in primary region
   - Peering connections to Transit Gateways in other regions
   - Spoke VPCs attach to regional Transit Gateways

2. Implementation Steps:
   a) Create Transit Gateway in each region
   b) Share Transit Gateway using AWS RAM to other accounts
   c) Accept RAM invitation in target accounts
   d) Attach VPCs to Transit Gateway
   e) Configure route tables for inter-VPC communication
   f) Create Transit Gateway Peering between regions

3. Benefits:
   - Transitive routing support
   - Centralized network management
   - Scales to thousands of VPCs
   - Reduced complexity vs mesh VPC peering
```

Alternative for Service Access:

- AWS PrivateLink for specific service exposure
- VPC Peering for simple 1-to-1 connections

Cost Considerations:

- Transit Gateway: $0.05/hour + $0.02/GB data transfer
- VPC Peering: $0.01/GB (same region)
- Choose based on number of VPCs and data volume

Compute Services

Q4: When would you choose EC2, ECS, EKS, Lambda, or Fargate? Explain your decision criteria.

Answer:

EC2 (Elastic Compute Cloud):

- **Use When:** Full OS control needed, Windows workloads, lift-and-shift migrations
- **Example:** Legacy .NET applications, specific compliance requirements
- **Pros:** Maximum flexibility, any OS/software
- **Cons:** Most operational overhead, patch management

ECS (Elastic Container Service):

- **Use When:** Docker containers, AWS-native solution, simpler than K8s
- **Example:** Microservices architecture without Kubernetes complexity
- **Pros:** Deep AWS integration, simpler than EKS, less overhead
- **Cons:** AWS-specific, limited portability

EKS (Elastic Kubernetes Service):

- **Use When:** Need Kubernetes, multi-cloud strategy, complex orchestration
- **Example:** Organizations with existing K8s expertise, hybrid cloud
- **Pros:** Industry standard, portable, rich ecosystem
- **Cons:** Complex, higher costs, steeper learning curve

Lambda:

- **Use When:** Event-driven, <15 min execution, variable workload
- **Example:** API backends, data processing, scheduled tasks
- **Pros:** No server management, auto-scaling, pay-per-use
- **Cons:** Cold starts, 15-min limit, stateless

Fargate:

- **Use When:** Containers without server management (ECS/EKS)
- **Example:** Microservices with unpredictable traffic
- **Pros:** Serverless containers, no capacity planning
- **Cons:** Higher cost than EC2, less control

Decision Tree:

```
Stateless, event-driven, <15 min? → Lambda
Containers + no server management? → Fargate
Containers + Kubernetes needed? → EKS
Containers + simpler orchestration? → ECS
Full control needed? → EC2
```

Q5: Design an auto-scaling solution for a web application with predictable daily traffic patterns.

Answer:

Solution: Scheduled + Dynamic Scaling

```
1. Traffic Pattern Analysis:
  - Baseline: 10 instances (midnight-6am)
  - Morning ramp: 10→30 instances (6am-9am)
  - Peak: 30-50 instances (9am-6pm)
  - Evening ramp down: 50→10 instances (6pm-midnight)

2. Auto Scaling Configuration:

  a) Scheduled Scaling Actions:
    - 6:00 AM: Set min=20, desired=30, max=50
    - 9:00 AM: Set min=30, desired=30, max=80
    - 6:00 PM: Set min=20, desired=25, max=40
    - 11:00 PM: Set min=10, desired=10, max=30

  b) Dynamic Scaling Policies:
    Target Tracking Policy:
    - Metric: CPU Utilization 70%
    - Scale-out: Add 20% capacity when >70% for 2 min
    - Scale-in: Remove 10% capacity when <50% for 10 min

    Step Scaling (for spikes):
    - 70-80% CPU: +2 instances
    - 80-90% CPU: +4 instances
    - >90% CPU: +8 instances

3. Additional Configuration:
  - Warm-up time: 300 seconds
  - Health check grace period: 180 seconds
  - Scale-in protection during deployments
  - Multiple AZs for availability
  - ALB with slow start mode (30 seconds)

4. Cost Optimization:
  - Use Reserved Instances for baseline (10 instances)
  - On-Demand for predictable scaling (10-30 instances)
  - Spot Instances for burst capacity (above 30)
```

Monitoring:

- CloudWatch dashboards for scaling activities
- Alarms for scale-in/out events
- Lambda for notifications

Q6: How do you handle sudden, unpredictable traffic spikes cost-effectively?

Answer:

Multi-Layer Defense Strategy:

```
1. Edge Layer - CloudFront:
  - Cache static content (images, CSS, JS)
  - Reduce origin load by 60-80%
  - DDoS protection with AWS Shield
  - Lambda@Edge for simple request routing

2. Application Layer - Multiple Options:

  Option A: Lambda Architecture
  - API Gateway + Lambda
  - Automatic scaling to handle spikes
  - Pay only for actual requests
  - Best for: APIs, simple logic

  Option B: Container + Spot Instances
  - ECS/EKS with Fargate Spot
  - Baseline on On-Demand
  - Burst capacity on Spot (70% cost savings)
  - Cluster Autoscaler for EKS

3. Queue-Based Decoupling:
  Internet → ALB → API (small fleet)
         ↓
       SQS Queue
         ↓
  Worker Fleet (Auto Scaling + Spot)

  Benefits:
  - Prevent application overload
  - Process requests asynchronously
  - Queue depth metric triggers scaling

4. Database Layer:
  - Read Replicas for read-heavy workloads
  - ElastiCache (Redis) for frequently accessed data
  - DynamoDB with on-demand capacity
  - Connection pooling (RDS Proxy)

5. Cost Optimization Tactics:
  - Spot Instances for burst capacity
  - Mixed instance types in Auto Scaling
  - Savings Plans for baseline
  - Right-sizing based on actual usage

6. Implementation Example:
  Instance Mix for Web Tier:
```

```
- 20% Reserved (baseline): c5.large
- 10% On-Demand (buffer): c5.large
- 70% Spot (burst): c5.large, c5.xlarge, c6i.large
```

Real-World Scenario: Normal load: 100 req/sec → 10 instances Sudden spike: 5,000 req/sec → Queue buffers, Auto Scaling adds 90 instances over 5 minutes, Spot provides 70% of added capacity

---

Storage & Databases

Q7: Explain different S3 storage classes and design a lifecycle policy for a data lake.

Answer:

S3 Storage Classes:

```
1. S3 Standard:
   - Use: Frequently accessed data
   - Latency: Milliseconds
   - Cost: $0.023/GB/month
   - Availability: 99.99%

2. S3 Intelligent-Tiering:
   - Use: Unknown/changing access patterns
   - Auto-moves between access tiers
   - Cost: $0.023-$0.0125/GB + monitoring fee

3. S3 Standard-IA:
   - Use: Infrequent access (monthly)
   - Cost: $0.0125/GB + retrieval fee
   - Min storage: 30 days

4. S3 One Zone-IA:
   - Use: Recreatable infrequent data
   - Cost: $0.01/GB (20% cheaper than Standard-IA)
   - Single AZ (99.5% availability)

5. S3 Glacier Instant Retrieval:
   - Use: Archive with instant access (quarterly)
   - Cost: $0.004/GB
   - Retrieval: Milliseconds

6. S3 Glacier Flexible Retrieval:
   - Use: Archive (1-2 times/year)
   - Cost: $0.0036/GB
   - Retrieval: Minutes to hours

7. S3 Glacier Deep Archive:
   - Use: Long-term compliance (7-10 years)
   - Cost: $0.00099/GB
   - Retrieval: 12-48 hours
```

Data Lake Lifecycle Policy:

```
json
```

```json
{
 "Rules": [
   {
    "Id": "Raw-Data-Lifecycle",
    "Filter": {"Prefix": "raw-data/"},
    "Status": "Enabled",
    "Transitions": [
      {
       "Days": 30,
       "StorageClass": "STANDARD_IA"
      },
      {
       "Days": 90,
       "StorageClass": "GLACIER_IR"
      },
      {
       "Days": 365,
       "StorageClass": "DEEP_ARCHIVE"
      }
    ]
   },
   {
    "Id": "Processed-Data-Lifecycle",
    "Filter": {"Prefix": "processed/"},
    "Status": "Enabled",
    "Transitions": [
      {
       "Days": 7,
       "StorageClass": "INTELLIGENT_TIERING"
      }
    ]
   },
   {
    "Id": "Temp-Data-Expiration",
    "Filter": {"Prefix": "temp/"},
    "Status": "Enabled",
    "Expiration": {
     "Days": 7
    }
   }
 ]
}
```

Data Lake Architecture:

```
Landing Zone (S3 Standard)
   ↓ (AWS Glue ETL)
Processed Zone (S3 Intelligent-Tiering)
   ↓ (Lifecycle after 30 days)
Archive Zone (S3 Glacier)
```

Q8: How do you choose between RDS, Aurora, DynamoDB, and Redshift?

Answer:

Decision Matrix:

Amazon RDS:

- **Use For:** Traditional relational databases, lift-and-shift
- **Engines:** MySQL, PostgreSQL, Oracle, SQL Server, MariaDB
- **Max Size:** 64 TB
- **Performance:** Up to 16,000 IOPS (io1)
- **Best For:** Existing applications, ACID compliance, complex queries
- **Example:** E-commerce order management, CRM systems

Amazon Aurora:

- **Use For:** High-performance relational database
- **Compatible:** MySQL, PostgreSQL
- **Performance:** 5x MySQL, 3x PostgreSQL performance
- **Max Size:** 128 TB (auto-scaling storage)
- **Read Replicas:** Up to 15
- **Best For:** High-throughput OLTP, need for read scalability
- **Global Database:** <1 sec cross-region replication
- **Example:** Gaming leaderboards, financial transactions

Amazon DynamoDB:

- **Use For:** NoSQL key-value/document store
- **Performance:** Single-digit millisecond latency at any scale
- **Scaling:** Automatic, handles millions of requests/sec

- **Data Model:** Key-value, document
- **Best For:** Mobile/web apps, gaming, IoT, real-time
- **Global Tables:** Multi-region, active-active
- **Example:** User profiles, shopping carts, session management

Amazon Redshift:

- **Use For:** Data warehouse, analytics, OLAP
- **Performance:** Columnar storage, MPP architecture
- **Max Size:** Petabyte-scale
- **Best For:** Business intelligence, complex analytical queries
- **Not For:** OLTP, real-time queries
- **Example:** Sales analytics, data marts, reporting

Selection Flow:

```
Need relational + SQL?
├─ Yes → OLTP or OLAP?
│   ├─ OLTP → High performance needed?
│   │   ├─ Yes → Aurora
│   │   └─ No → RDS
│   └─ OLAP → Redshift
└─ No → NoSQL → DynamoDB
```

Hybrid Example:

```
User Profile → DynamoDB (fast K-V lookup)
Transactions → Aurora (ACID, relational)
Analytics → Redshift (aggregations, reporting)
Session Data → ElastiCache (in-memory)
```

Q9: Design a database migration strategy from on-premises Oracle to AWS with minimal downtime.

Answer:

Migration Strategy: AWS DMS with CDC

Phase 1: Assessment (1-2 weeks)

```
1. Database Analysis:
   - Size: 5 TB
   - Transaction rate: 10,000 TPS
   - Dependencies: 50 applications
   - Downtime tolerance: <1 hour

2. Target Selection:
   Option A: RDS for Oracle (license-included)
   Option B: Aurora PostgreSQL (cost-effective)
   Option C: DynamoDB (if NoSQL suitable)

   Decision: Aurora PostgreSQL (70% cost savings)

3. Schema Conversion Tool (SCT):
   - Assess compatibility
   - Generate conversion report
   - Identify manual changes needed
```

Phase 2: Setup (1 week)

```
1. Network Connectivity:
   - VPN or Direct Connect to AWS
   - Security groups: Allow Oracle port from DMS

2. Create Target Database:
   - Aurora PostgreSQL cluster
   - Multi-AZ for HA
   - Appropriate instance size (r6g.4xlarge)

3. DMS Configuration:
   - Replication instance: dms.c5.4xlarge
   - Source endpoint: On-premises Oracle
   - Target endpoint: Aurora PostgreSQL
```

Phase 3: Migration (2-3 weeks)

```
1. Full Load Phase:
   - DMS performs initial bulk copy
   - Estimated time: 72 hours for 5 TB
   - Run during low-traffic period

2. Change Data Capture (CDC):
   - Captures ongoing changes during migration
   - Keeps source and target in sync
   - Monitor replication lag

3. Testing:
   - Parallel run: Applications test against Aurora
   - Performance testing
   - Data validation (row counts, checksums)
   - Application compatibility testing
```

Phase 4: Cutover (1 hour)

```
Cutover Steps:
1. Enable read-only mode on source (T+0 min)
2. Wait for CDC lag to reach zero (T+5 min)
3. Validate data consistency (T+10 min)
4. Update application connection strings (T+15 min)
5. Perform smoke tests (T+20 min)
6. Enable write traffic to Aurora (T+30 min)
7. Monitor for issues (T+30-60 min)

Rollback Plan:
- Keep source database available for 48 hours
- Can switch back connection strings if needed
```

Phase 5: Post-Migration

```
1. Monitoring:
   - Database performance metrics
   - Application error rates
   - Query performance

2. Optimization:
   - Create indexes based on workload
   - Adjust Aurora parameters
   - Enable Performance Insights

3. Decommission:
   - After 2 weeks of stable operation
   - Archive source database
```

Alternative: Zero-Downtime with Bidirectional Replication

- Use Oracle GoldenGate for bi-directional replication
- Allows gradual application cutover
- Higher complexity and cost

Q10: How do you design a globally distributed, low-latency database architecture?

Answer:

Solution: Multi-Region Active-Active Architecture

Option 1: DynamoDB Global Tables

```
Architecture:
- DynamoDB tables in 5 regions (us-east-1, eu-west-1, ap-southeast-1, etc.)
- Multi-region, multi-active replication
- Last-writer-wins conflict resolution

Implementation:
1. Create base table in primary region
2. Enable streams
3. Add replica regions via Global Tables
4. Configure Route 53 geolocation routing

Benefits:
- Single-digit millisecond latency globally
- <1 second replication between regions
- 99.999% availability SLA
- Automatic conflict resolution

User Flow:
User (Tokyo) → Route 53 → API Gateway (ap-northeast-1)
          → Lambda → DynamoDB (ap-northeast-1)
          → Async replication to other regions
```

Option 2: Aurora Global Database

```
Architecture:
- Primary region: Read-write (us-east-1)
- Secondary regions: Read-only (eu-west-1, ap-southeast-1)
- Physical replication (<1 second lag)

Implementation:
1. Create Aurora cluster in primary region
2. Add global database cluster
3. Add secondary regions with read replicas
4. Configure Route 53 latency-based routing

Routing Logic:
- Writes: Always route to primary region
- Reads: Route to nearest region

Disaster Recovery:
- RTO: <1 minute (promote secondary)
- RPO: <1 second

Read Flow:
User (London) → Route 53 (latency routing)
              → ALB (eu-west-1)
              → App Servers
              → Aurora read replica (eu-west-1)

Write Flow:
User (London) → Route 53
              → ALB (us-east-1)
              → App Servers (us-east-1)
              → Aurora primary (us-east-1)
              → Replicates to eu-west-1
```

Option 3: Hybrid with Caching

```
Architecture:
Internet
  ↓
CloudFront (edge locations worldwide)
  ↓
Route 53 (geolocation routing)
  ↓
Regional API Gateways
  ↓
ElastiCache Global Datastore (Redis)
  ↓
Aurora Global Database

Caching Strategy:
- User profiles: Cache 1 hour
- Product catalog: Cache 5 minutes
- Inventory: No cache (real-time)
- Session data: Redis with TTL

Benefits:
- <10ms latency for cached data
- 90% cache hit rate reduces DB load
- Global consistency for writes
```

Comparison:

Feature	DynamoDB Global	Aurora Global	Hybrid
Write Latency	<10ms (local)	<100ms (cross-region)	<10ms (cached)
Read Latency	<10ms	<10ms (local replica)	<5ms (cached)
Conflict Resolution	Automatic	Manual	Application-level
Cost	$$$	$$	$$$$
Complexity	Low	Medium	High

Recommendation:

- **High write throughput, simple K-V:** DynamoDB Global Tables
- **Complex relational queries:** Aurora Global Database + ElastiCache
- **Read-heavy, low latency:** Hybrid with aggressive caching

---

Security & Compliance

Q11: Explain your approach to implementing defense-in-depth security for AWS resources.

Answer:

Multi-Layer Security Strategy:

```
Layer 1: Edge Protection
- CloudFront with AWS WAF
- AWS Shield (DDoS protection)
- Geo-blocking for restricted regions
- Rate limiting

Layer 2: Network Security
- VPC with private subnets
- NACLs for subnet-level filtering
- Security Groups (stateful firewall)
- VPC Flow Logs for monitoring
- AWS Network Firewall for deep packet inspection

Layer 3: Identity & Access
- IAM roles with least privilege
- MFA enforcement
- AWS SSO for centralized access
- Service Control Policies (SCPs)
- Regular access reviews

Layer 4: Application Security
- ALB with WAF rules (OWASP Top 10)
- API Gateway with throttling
- Secrets Manager for credentials
- Certificate Manager for TLS
- Inspector for vulnerability scanning

Layer 5: Data Security
- Encryption at rest (KMS)
- Encryption in transit (TLS 1.2+)
- S3 bucket policies
- RDS encryption
- Macie for sensitive data discovery

Layer 6: Monitoring & Response
- CloudTrail (API logging)
- GuardDuty (threat detection)
- Security Hub (centralized view)
- Config (compliance monitoring)
- EventBridge + Lambda for automated response
```

Implementation Example:

```
Web Application Security Stack:

1. Perimeter:
   CloudFront → WAF Rules:
   - SQL injection protection
   - XSS filtering
   - Bot control
   - Rate limiting: 2000 req/5min per IP

2. Network:
   Public Subnet SG:
   - Allow 443 from CloudFront IPs only

   Private Subnet SG:
   - Allow 8080 from ALB SG only

   Database SG:
   - Allow 3306 from App SG only

3. Access Control:
   EC2 Instance Role:
   - Read from S3 bucket (specific bucket)
   - Write to CloudWatch Logs
   - Read from Secrets Manager (specific secret)

4. Data Protection:
   S3 Bucket Policy:
   - Deny non-HTTPS access
   - Deny public access
   - Require encryption
   - Versioning enabled

   RDS:
   - Encryption at rest (KMS CMK)
   - Automated backups encrypted
   - SSL/TLS enforced

5. Monitoring:
   CloudWatch Alarms:
   - Failed login attempts >5 in 5 min
   - Root account usage
   - Security group changes
   - IAM policy changes
```

```
GuardDuty Findings:
- Lambda auto-isolates compromised instances
```

Q12: How do you implement encryption for data at rest and in transit across AWS services?

Answer:

Encryption Strategy:

At Rest Encryption:

```
1. S3:
  - SSE-S3: AWS-managed keys
  - SSE-KMS: Customer managed keys (CMK)
  - SSE-C: Customer-provided keys

  Recommended: SSE-KMS with CMK

  {
    "Version": "2012-10-17",
    "Statement": [{
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::my-bucket/*",
      "Condition": {
        "StringNotEquals": {
          "s3:x-amz-server-side-encryption": "aws:kms"
        }
      }
    }]
  }

2. EBS:
  - Enable encryption by default (regional setting)
  - Uses KMS CMK
  - Snapshots automatically encrypted

  AWS CLI:
  aws ec2 enable-ebs-encryption-by-default --region us-east-1

3. RDS/Aurora:
  - Enable at creation (cannot add later)
  - Uses KMS CMK
  - Automated backups, snapshots encrypted
  - Read replicas must use same encryption

4. DynamoDB:
  - Encryption enabled by default
  - AWS owned, AWS managed, or Customer managed KMS keys
  - No performance impact

5. EFS:
  - Enable encryption at creation
  - Uses KMS CMK
  - Applies to data and metadata
```

In Transit Encryption:

```
1. Application Load Balancer:
   - HTTPS listener with ACM certificate
   - SSL/TLS termination at ALB
   - Backend encryption optional

   Configuration:
   - Protocol: HTTPS
   - Security Policy: ELBSecurityPolicy-TLS-1-2-2017-01
   - Certificate: ACM or imported

2. API Gateway:
   - Enforces HTTPS by default
   - Custom domain with ACM certificate
   - Regional or edge-optimized endpoints

3. RDS:
   - Force SSL connections

   MySQL:
   REQUIRE SSL;

   PostgreSQL:
   rds.force_ssl = 1

4. S3:
   - Bucket policy to deny non-HTTPS

   {
     "Effect": "Deny",
     "Principal": "*",
     "Action": "s3:*",
     "Resource": "arn:aws:s3:::my-bucket/*",
     "Condition": {
       "Bool": {"aws:SecureTransport": "false"}
     }
   }

5. VPC:
   - VPN with IPSec
   - Direct Connect with MACsec
   - PrivateLink for service access
```

Key Management:

```
KMS Best Practices:

1. Key Hierarchy:
   - Separate CMKs per environment (dev/prod)
   - Separate CMKs per data classification
   - Automatic key rotation enabled (yearly)

2. Key Policy:
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "Enable IAM policies",
         "Effect": "Allow",
         "Principal": {"AWS": "arn:aws:iam::123456789012:root"},
         "Action": "kms:*",
         "Resource": "*"
       },
       {
         "Sid": "Allow S3 to use key",
         "Effect": "Allow",
         "Principal": {"Service": "s3.amazonaws.com"},
         "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
         "Resource": "*"
       }
     ]
   }

3. Monitoring:
   - CloudTrail logs all KMS API calls
   - CloudWatch alarms for key usage
   - Audit key policies regularly
```

High Availability & Disaster Recovery

Q13: Explain RTO and RPO. Design solutions for different recovery requirements.

Answer:

Definitions:

- **RTO (Recovery Time Objective):** Maximum acceptable downtime
- **RPO (Recovery Point Objective):** Maximum acceptable data loss

DR Strategies (Cost ↔ Recovery Time Trade-off):

1. Backup and Restore (High RTO/RPO - Lowest Cost)

```
RTO: Hours to days
RPO: Hours
Cost: $

Architecture:
- Regular snapshots to S3
- CloudFormation templates for infrastructure
- Automated backups

Use Case: Non-critical applications, cost-sensitive

Implementation:
Primary Region:
- EC2 with daily AMI snapshots
- RDS with automated backups (7-day retention)
- S3 with versioning

DR Steps:
1. Create VPC from CloudFormation (30 min)
2. Launch EC2 from latest AMI (15 min)
3. Restore RDS from snapshot (45 min)
4. Update DNS (5 min)
Total RTO: ~2 hours
```

2. Pilot Light (Medium RTO/RPO - Low Cost)

```
RTO: 10 minutes to 1 hour
RPO: Minutes
Cost: $$

Architecture:
- Minimal resources always running in DR region
- Database replication active
- Application servers off/minimal

Use Case: Business-critical applications

Implementation:
DR Region (Always Running):
- VPC and subnets configured
- RDS read replica (continuous replication)
- AMIs updated weekly
- Data replication via S3 Cross-Region Replication

Failover Steps:
1. Promote RDS read replica (5 min)
2. Launch EC2 from AMIs via Auto Scaling (5 min)
3. Update Route 53 (2 min)
4. Warm up application (3 min)
Total RTO: 15-20 minutes
RPO: ~5 minutes (replication lag)
```

3. Warm Standby (Low RTO/RPO - Medium Cost)

```
RTO: Minutes
RPO: Seconds to minutes
Cost: $$$

Architecture:
- Scaled-down version running in DR region
- Continuous data replication
- Can handle some load immediately

Implementation:
DR Region (Running):
- Auto Scaling: min=2, max=10 (vs prod: min=10, max=50)
- Smaller instance types (t3.medium vs c5.xlarge)
- Aurora Global Database (replication lag <1 sec)
- Route 53 weighted routing (90% primary, 10% DR)

Failover Steps:
1. Increase Auto Scaling capacity (2 min)
2. Update Route 53 weights to 0% primary, 100% DR (1 min)
Total RTO: 5 minutes
RPO: <1 minute
```

4. Multi-Site Active-Active (Minimal RTO/RPO - Highest Cost)

```
RTO: Automatic (zero downtime)
RPO: Zero to seconds
Cost: $$$$

Architecture:
- Full capacity in multiple regions
- Active traffic in all regions
- Real-time data replication

Implementation:
Global Setup:
- DynamoDB Global Tables (multi-region, multi-active)
- Route 53 latency/geo routing
- CloudFront for global edge caching
- Identical infrastructure in all regions

Each Region:
- Full production capacity
- Auto Scaling based on traffic
-
```