Advanced questions on Reliability and scalability in AWS

Q1

Describe the architectural trade-offs between a Warm Standby and a Multi-Site Active/Active Disaster Recovery (DR) strategy on AWS in terms of RTO, RPO, and Cost.

Warm Standby:
Cost-effective because only minimal infrastructure runs in the DR region. However, it has a **higher RTO** (you must scale up DR infra during failover) and **higher RPO** (possible replication lag and data catch-up).

Multi-Site Active/Active:
Provides **near-zero RTO** and **near-zero RPO** because both regions actively serve traffic with continuous replication. Trade-off: **highest cost** due to fully duplicated infrastructure and multi-region data transfer.

Q2

How would you achieve a five-nines (99.999%) availability target for a critical application's data layer using Amazon RDS? Detail the specific features and trade-offs.

Use **RDS Multi-AZ** for synchronous replication and automatic failover within a region.
Add **Cross-Region Read Replicas** for DR.
Trade-offs: high cost for duplicated infrastructure and additional complexity in application failover logic.

Q3

Explain how AWS Global Accelerator contributes to high availability and reliability, contrasting its function with that of a global Amazon CloudFront distribution.

Global Accelerator uses the AWS global network and static IPs to route users to the **nearest healthy regional endpoint**with constant health checks—improving reliability and multi-region failover.
CloudFront focuses on **content caching and latency reduction**, not multi-region failover based on application health.

Q4

A microservice deployment requires zero downtime blue/green deployments. Which AWS services and strategy would you use to ensure traffic is seamlessly shifted, and why?

Use **Route 53 weighted routing**, **ALB with two target groups (Blue/Green)**, and **AWS CodeDeploy hooks** (optionally Lambda) to gradually shift traffic.
For ECS/EKS, use **App Mesh** or ALB listener rules for fine-grained traffic shifting.

Section 2: Scalability and Performance Optimization

Q5

Detail the difference between Target Tracking, Step, and Scheduled scaling policies in EC2 Auto Scaling, and give a complex use case requiring all three.

- **Target Tracking:** Keeps a metric (CPU, etc.) at a target.
- **Step Scaling:** Scales more aggressively based on alarm severity.
- **Scheduled Scaling:** Pre-adjusts capacity based on predictable patterns.

Use Case: An e-commerce site uses
Target Tracking for daily load,
Step Scaling for flash-sale spikes,
Scheduled Scaling for known traffic peaks (e.g., 9 AM login surge).

Q6

Your application suffers from database connection pooling saturation. How does RDS Proxy solve this?

RDS Proxy maintains a **shared pool of database connections**, reducing DB overhead and preventing saturation. It protects the DB from connection storms and improves resilience.

Q7

How does S3 key structure impact performance, and what are the best practices for high-throughput workloads?

S3 performance scales with **prefixes**. Heavy access to one prefix can cause throttling. Best practice: distribute keys across multiple prefixes using **hash-based or random key prefixing** for high throughput.

Q8

How would you use Metric Math to create an Auto Scaling policy based on the % of

healthy ALB requests?

Create a metric math expression:
(HealthyRequests / (HealthyRequests + UnhealthyRequests)) × 100
Use the computed metric as the scaling target.

Section 3: Reliability, Observability, and Cost

Q9

Contrast CloudWatch Logs Insights vs Contributor Insights when diagnosing a latency spike.

- **Logs Insights:** Query raw log data to find errors, slow requests, patterns.
- **Contributor Insights:** Identifies the **top contributors** (IPs, users, hosts) causing latency or errors.

Q10

What is the purpose of a Saga pattern in microservices, and how does Step Functions help?

Saga coordinates long-running transactions using **local transactions + compensating actions**.
Step Functions orchestrate each step, track progress, and automatically run compensations on failure.

Q11

Your platform ingests massive IoT data. What is the most cost-effective long-term storage and query solution?

Store data in **S3 Standard-IA / Glacier** and query via **Athena** or **Redshift Spectrum**.
Use **S3 partitioning** and **Intelligent-Tiering** to reduce scan cost and storage cost.
Use **partition projection** to scan only needed data.

Q12

Explain Idempotency Keys and which AWS services support them.

An Idempotency Key ensures an action is executed **only once**, even if retried (prevents double-charges, duplicate writes).

Supported in:

- API Gateway
- Step Functions
- AWS CLI/SDK `ClientToken` parameters (RunInstances, CreateVolume, etc.)

Section 4: Advanced Services and Design Patterns

Q13

What is shuffle sharding, and how can Route 53 implement it?

Shuffle sharding assigns users to a small, random subset of resources. Failures affect only that shard.
Implement via **Route 53 Resolver Endpoints**, assigning different endpoint subsets to different business units/applications to limit blast radius.

Q14

What mechanisms does AWS AppConfig provide to ensure safe configuration deployments?

1. **Validation** (Lambda/schema)
2. **Deployment strategies** — linear/canary
3. **Automatic rollback** triggered by CloudWatch alarms

Q15

How do AWS Network Firewall and VPC Endpoint Services provide a secure hub-and-spoke architecture?

Network Firewall lives in a **central inspection VPC**.
PrivateLink provides **private connectivity** from spoke VPCs to the hub, ensuring all traffic is inspected while avoiding public Internet exposure.

Q16

Explain eventual consistency in DynamoDB and how to compensate for stale reads.

Writes may not be immediately visible.
Compensations include:

- Use **Strongly Consistent Reads** for critical operations
- Use **versioning/timestamps** to avoid acting on stale data

---

Section 5: Serverless and Container Reliability

Q17

How does EKS maintain a minimum number of running pods during node failures?

Use a **Pod Disruption Budget (PDB)** to limit voluntary disruptions.
Use **Cluster Autoscaler** to add/remove EC2 nodes to meet pod scheduling needs.

Q18

How to manage environment-specific Lambda configurations securely?

- **Inferior:** Lambda environment variables (requires redeploy, weak secrecy)
- **Superior:** Store configs in **SSM Parameter Store** or **Secrets Manager**, fetched at runtime for reliability and security.

Q19

How do you handle persistent SQS message failures to avoid system blockage?

Use a **Dead-Letter Queue (DLQ)** with a redrive policy. Poison messages are moved to the DLQ after max retries.

Q20

How does EC2 Fleet with mixed Spot + On-Demand improve cost and reliability?

Spot provides **massive cost savings**, while a baseline of On-Demand ensures **minimum guaranteed capacity**.
If Spot instances are interrupted, Fleet replaces them automatically.

Q21

How does the Cell-Based Architecture pattern improve the reliability of large-scale applications, and how can this be implemented using AWS services?
A cell-based architecture divides an application into *multiple isolated cells*, each running a full copy of the stack. A failure in one cell only impacts users routed to that cell.
Implementation: Deploy each cell in its own VPC + ALB + RDS cluster. Use **Route 53 latency routing or shuffle sharding** to distribute users across cells. This offers fault isolation, independent scaling, and prevents "blast radius" expansion during failures.

Q22

Explain how DynamoDB adaptive capacity contributes to reliability and cost optimization in uneven workload distributions.
Adaptive capacity automatically shifts unused RCU/WCU from underutilized partitions to "hot" partitions. This prevents partition-level throttling, improves reliability during unpredictable spikes, and eliminates the need to over-provision table capacity.

Q23

Describe how AWS FIS (Fault Injection Simulator) improves reliability and what kinds of failures you should test in a real-world system.
AWS FIS injects controlled failures to validate resilience and autoscaling behavior.
You should test:

- EC2 termination
- AZ outages (disable subnets)
- RDS failover delays
- Latency injection on microservice calls
- Throttling on downstream services
  This validates circuit breakers, retries, timeouts, and fallback paths.

Q24

How can you design a resilient, globally available API using Amazon API Gateway across multiple AWS regions?
Deploy identical API Gateway + Lambda stacks in two regions. Use **Route 53 latency-based routing** with health checks. Store data in DynamoDB Global Tables for multi-region writes. This ensures near-zero RTO and reduces latency for global users.

Q25

Explain quorum-based writes and reads in distributed systems like DynamoDB and

how they affect consistency and reliability.
Quorum means a majority of replicas must agree (e.g., W=2 out of 3 replicas).

- **High write quorum (W=2, R=1)** → Stronger consistency
- **High read quorum (R=2)** → Avoid stale reads
  Quorums increase reliability during node failures but reduce throughput due to more replication coordination.

Q26

What is the "circuit breaker" pattern and how can you implement it using AWS services?
A circuit breaker stops calls to a failing downstream service after repeated failures, preventing cascading failure.
Implementation:

- API Gateway + Lambda authorizer to block requests
- App Mesh or ALB to configure retries & max connection errors
- Step Functions Retry+Catch blocks to trigger fallback flows

---

Q27

How do you design a multi-region Kafka-like streaming pipeline using Amazon MSK that survives AZ failures?
Use:

- **MSK Multi-AZ** cluster with replication factor ≥ 3
- **ACKS=all** for producer reliability
- Consumer offset replication to another region using **MirrorMaker2**
  This architecture survives broker failures and zone outages with minimal message loss.

Q28

Why does placing compute and data in different regions usually reduce reliability, and how can you avoid this issue?
Cross-region traffic introduces high latency and intermittent failures.
Fix: Keep compute and data in the same region or use **DynamoDB Global Tables**, **Aurora Global Database**, or **S3 CRR** to keep data close to compute.

Q29

Explain how AWS Aurora Global Database offers near-zero RPO and very low RTO

for global workloads.
Aurora Global Database replicates storage changes asynchronously using dedicated high-speed physical storage replication (<1 sec).
RTO is low because a secondary region can promote the read replica to writer in 30–60 seconds.

Q30

How do you use read/write separation to improve reliability for a highly read-heavy RDS database?
Use RDS Read Replicas for read-heavy traffic. Application uses:

- **writes → primary**
- **reads → read replicas**
  This reduces load on primary, preventing overload and increasing uptime.

Section 2: Autoscaling, Performance Scaling & Load Management

Q31

Explain how predictive scaling in EC2 works and when it is more reliable than target tracking scaling.
Predictive scaling uses ML to analyze historical load patterns and scales ahead of time. If your application experiences predictable traffic peaks (e.g., every morning at 9 AM), predictive scaling is more reliable than reactive target tracking.

Q32

Your application has unpredictable spiky loads. How do you use queue-based load leveling to improve reliability?
Use SQS between frontend and backend. Backend consumers scale independently. This prevents overload on compute during spikes, absorbing bursts in the queue and avoiding cascading failures.

Q33

How does DynamoDB On-Demand mode ensure reliability during sudden, massive traffic spikes?
On-Demand instantly scales RCU/WCU without provisioning. It handles > 4x sudden traffic spikes without throttling, making it ideal for unpredictable workloads.

Q34

Explain how you can use Kinesis Enhanced Fan-Out to prevent consumer lag during peak loads.
Enhanced Fan-Out gives each consumer its own 2MB/sec pipe, avoiding the shared 2MB/sec limit. This reduces consumer throttling and ensures real-time processing even during high ingestion.

Q35

How does CloudFront improve both reliability and scalability for dynamic applications?

- Global edge network reduces latency
- Origin Shield protects against origin overload
- Auto scalable by design
- Integrated failover between multiple origins
  This prevents overload on backend APIs and provides global resilience.

Q36

Why is idempotent retry logic crucial in autoscaling environments?
During retries (due to throttling or transient failures), multiple instances may perform the same action. Idempotency ensures only one action succeeds, preventing data corruption, double writes, or double billing.

Q37

How does Amazon ElastiCache help improve scalability and reduce database failures?
Caching frequently accessed read traffic offloads database stress, reducing connections and preventing overload failure. Redis supports replication and cluster sharding for high availability and horizontal scaling.

Q38

What is connection draining and how does it increase reliability during deployments?
Connection draining (ALB deregistration delay) allows in-flight requests to complete before terminating or replacing an instance. This prevents user-visible failures during deployments.

Q39

Explain the difference between concurrency and throughput scaling in Lambda, and how Provisioned Concurrency improves reliability.
Concurrency = number of parallel executions
Throughput = executions/sec
Provisioned Concurrency pre-warms Lambdas, eliminating cold starts and ensuring predictable latency for critical workloads.

Q40

Why is horizontal scaling more reliable than vertical scaling in cloud environments?
Horizontal scaling distributes load across multiple instances—no single point of failure. Vertical scaling increases capacity of one node, making failure impact larger.

Section 3: Observability, Resilience & Operational Excellence

Q41

Describe how you would implement end-to-end distributed tracing for a microservices architecture on AWS.
Use **AWS X-Ray**, integrated with API Gateway, Lambda, ECS/EKS, and SDKs. X-Ray captures traces, segments, and annotations across service boundaries to visualize latency bottlenecks and failures.

Q42

What is "back pressure" and how do you design systems to avoid it on AWS?
Back pressure happens when downstream systems can’t keep up with upstream traffic. Solutions:

- SQS buffering
- Kinesis with enhanced fan-out
- Autoscaling consumers
- Circuit breakers
- Graceful degradation via feature flags

Q43

How do you use CloudWatch Anomaly Detection to improve reliability?
Anomaly Detection applies machine learning to identify abnormal spikes or drops. This triggers alarms early, helping teams resolve issues before customer impact.

Q44

How do you design for reliability using the Bulkhead pattern in AWS?
Bulkheads isolate resources. On AWS:

- Separate ASGs per microservice
- Separate RDS clusters per tenant
- Separate queues for independent workflows
  This prevents one failing workload from impacting others.

Q45

What AWS features allow you to detect partial AZ failures and route traffic away automatically?

- Route 53 health checks
- ALB/NLB multi-AZ health probing
- RDS Multi-AZ automatic failover
- EKS node health & PDB protections
  Together, they provide automatic removal of unhealthy AZs.

Q46

Explain Chaos Engineering and provide one AWS-native way to run chaos experiments.
Chaos Engineering introduces controlled failures to test resilience.
AWS-native tool: **AWS FIS (Fault Injection Simulator)** – injects instance terminations, network blackholes, latency spikes, and AZ outages.

Q47

How does DynamoDB Conditional Writes prevent race conditions?
Conditional writes only succeed if an attribute matches a condition (e.g., version=5).
Prevents lost updates and ensures strong concurrency control.

Q48

How do you implement automated failback after a disaster recovery event in AWS?
Use:

- Route 53 health checks to restore primary routing

- Data sync back to primary (DynamoDB Global Tables, Aurora Global database)
- Automated scripts to restore primary ALB/ASG/EKS endpoints

Q49

How do you prevent cascading retries from amplifying failures in a distributed system?
Use:

- Exponential backoff with jitter
- Circuit breakers
- Client-side throttling
- Queue buffering
  This prevents retry storms and cascading failures.

Q50

How do you design a throttling-resistant workload using API Gateway + Lambda + DynamoDB?
Use:

- API Gateway throttling (global + per key)
- Lambda reserved concurrency to avoid downstream overload
- DynamoDB On-Demand or adaptive capacity
  This ensures no single component overwhelms another and maintains reliability under bursts.