Advanced DevOps / SRE / AWS Questions

Q1. How do you design high-availability architecture across multiple regions?
Answer:

- Multi-region EKS clusters
- Route53 latency-based routing
- DB replication (Aurora Global / DynamoDB Global Tables)
- S3 cross-region replication
- Multi-AZ subnets
- Auto-scaling groups
- Health checks + Failover policies

Q2. Explain how you implemented Disaster Recovery.
Answer:

- Automated backups (RDS/Aurora snapshots)
- DynamoDB cross-region replication
- EKS cluster backups
- Replicated S3 buckets
- Failover routing with Route53
- Weekly DR drills to validate RTO/RPO

Q3. How do you evaluate cost optimization?
Answer:

- Identify idle instances
- Use instance right-sizing
- Switch to Graviton instances
- Use Spot + On-Demand mix
- Clean unused EBS, snapshots, ELBs
- Lifecycle policies for S3
- Enable Compute Savings Plans

Q4. How do you ensure zero downtime deployments?
Answer:

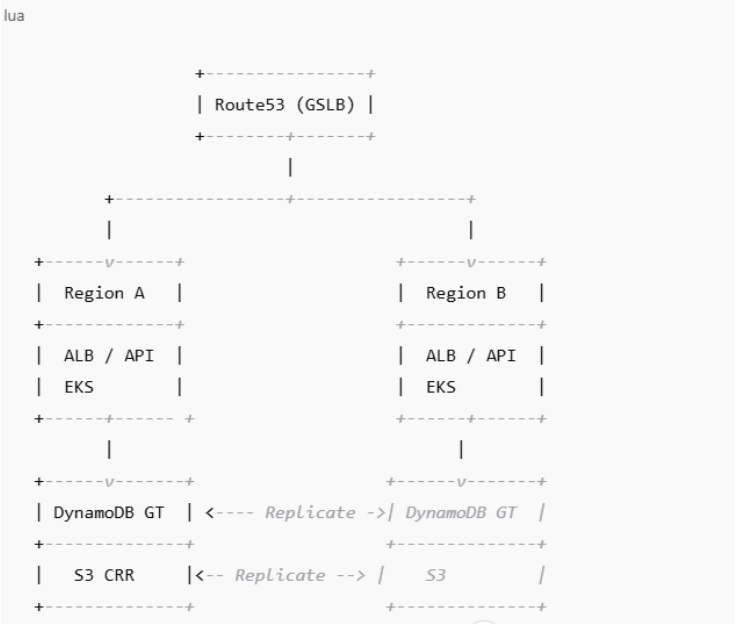
- Canary / Blue-Green
- Readiness + liveness probes
- Auto-rollback in Harness
- Pod disruption budgets
- Rolling updates with controlled maxSurge and maxUnavailable

Q5. How do you secure Kubernetes?
Answer:

- RBAC control
- Pod Security Policies / OPA Gatekeeper
- Image scanning (JFrog/Qualys)
- Secrets encryption using AWS Secrets Manager
- Network policies
- Disable privileged containers

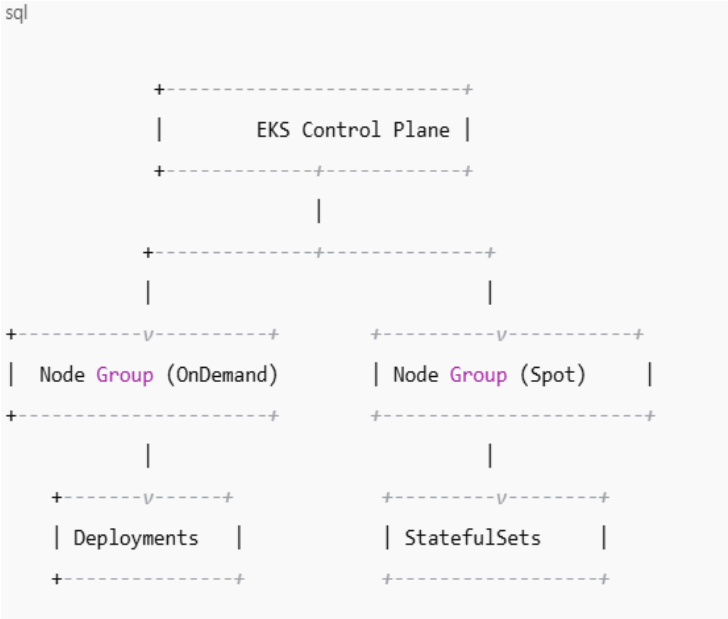
1. Design a Multi-Region, Active-Active Architecture

lua

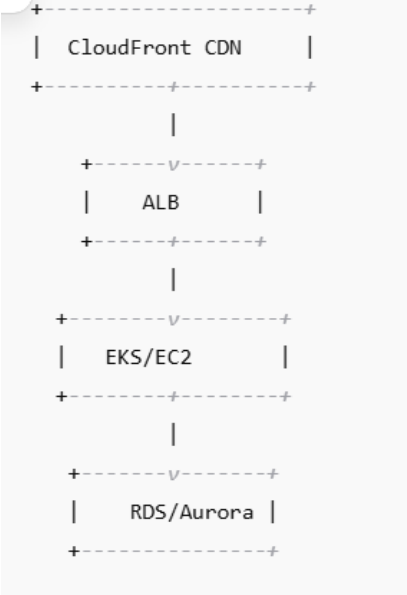


2. Design a Highly Available EKS Cluster

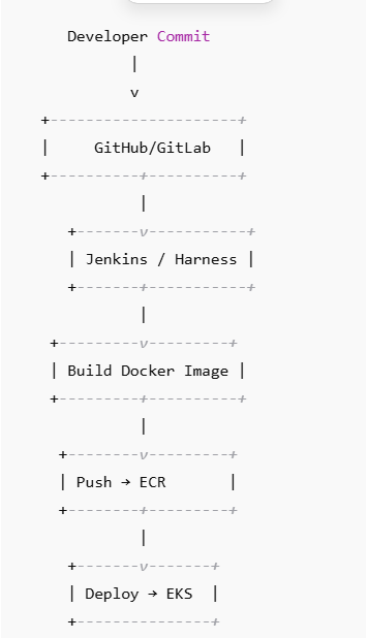
sql



3. **Design a Global CDN-Based Architecture**



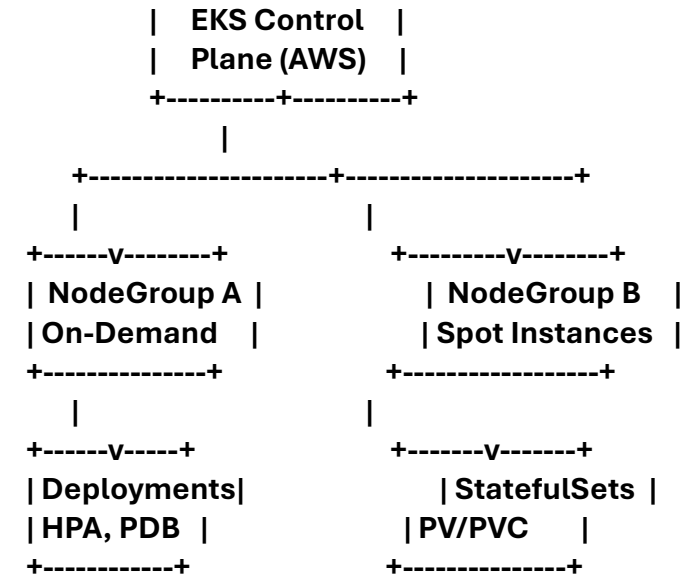
4. **DevOps CI/CD Architecture**



ARCHITECTURE 2 — Highly Available EKS Cluster (Production-Ready)
Goal:
Build a resilient EKS cluster that supports zero downtime, automatic scaling, and safe deployments.

🧩 **Diagram**

+----------+



🛠 Step-by-Step Design

Step 1: Control Plane
Managed by AWS:

- **HA across 3 AZs**
- **Auto-scaled by AWS**
- **Use version lifecycle: EKS 1.27+ recommended**
- **Logging enabled: audit, api, scheduler, authenticator**

Step 2: Node Groups
Create 3 node groups:

1. **On-demand nodes:**
    - **Core services**
    - **System pods**
    - **Ingress controllers**
2. **Spot nodes:**
    - **Non-critical workloads**
    - **Autoscaled**
3. **GPU node group (optional)**
    - **AI/ML workloads**

Step 3: Networking
AWS VPC CNI + Security best practices:

- **Assign IAM roles to service accounts**
- **Pod-level security via:**
    - **Network Policies**
    - **Istio policies**
    - **Calico (optional)**

Step 4: Autoscaling

1. **HPA: CPU/Mem/custom metrics**
2. **Cluster Autoscaler:**
    - **Adds nodes when pods are Pending**
3. **Karpenter (new option):**
    - **More powerful**
    - **Spot instance orchestration**

Step 5: Storage

- **gp3 EBS volumes for PVCs**
- **EFS for shared content**
- **fsx for high-performance workloads**

Step 6: Deployment Safety

Use:

- **Readiness & liveness probes**
- **PodDisruptionBudgets (PDB)**
- **RollingUpdate strategy**
- **Blue-Green via Argo Rollouts / Harness**

Step 7: Security

- **RBAC**
- **OPA Gatekeeper**
- **AWS Secrets Manager**
- **KMS encryption**
- **mTLS via Istio**

Step 8: Monitoring

- **Prometheus + Grafana**
- **CloudWatch Container Insights**
- **Jaeger / X-Ray for tracing**

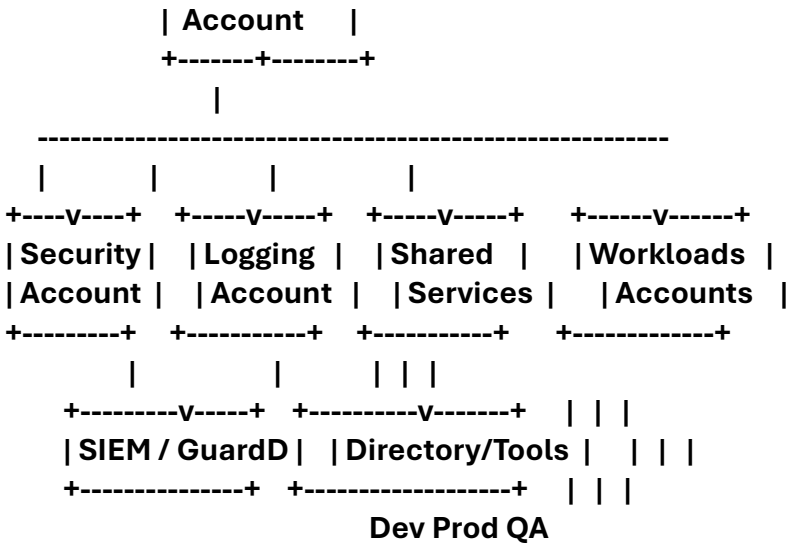
✅ **ARCHITECTURE 6 — Multi-Account AWS Landing Zone (Enterprise-Grade)**

Goal:

A secure enterprise AWS foundation using multi-account isolation, SCP controls, centralized logging, shared services, and secure networking.

🧩 **High-Level Diagram**

```
                AWS ORGANIZATION
               +----------------+
               | Management     |
```



🛠 Step-by-Step Design

Step 1: Set up AWS Organizations

- **Create root organization**
- **Define OU (Organizational Units):**
  - ***Security OU***
  - ***Infrastructure OU***
  - ***Production OU***
  - ***Sandbox OU***

Why?
Logical grouping for applying Service Control Policies (SCPs).

Step 2: Mandatory Accounts

1. Management Account

- **Only for billing and org management**
- **No workloads allowed**

2. Security Account

- **GuardDuty**
- **Security Hub**
- **IAM Access Analyzer**
- **Inspector**
- **Centralized IAM Role creation**

3. Logging Account

- **Centralized CloudTrail**
- **VPC Flow Logs**
- **Lambda function logs**
- **S3 with retention**

4. Shared Services Account

- **Directory Services**

- **DNS**
- **Jenkins/Harness**
- **Jump/Bastion Hosts**
- **Transit Gateway**

5. Workload Accounts

- **Dev**
- **QA**
- **Prod**
- **ML**
- **Analytics**

Why?

Blast-radius reduction and security isolation.

Step 3: Service Control Policies (SCPs)

Examples:

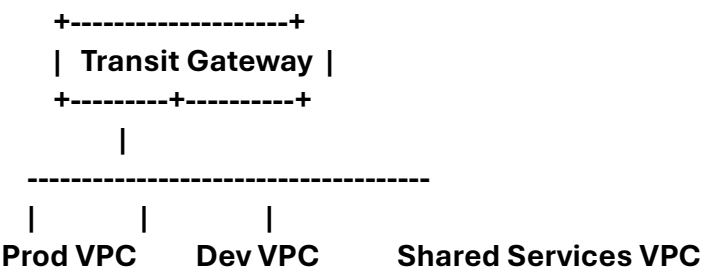
- **Deny disabling CloudTrail**
- **Deny public S3 buckets**
- **Deny IAM policy wildcards**
- **Restrict which regions can be used**

Step 4: Centralized Logging Architecture

Account A --> CloudTrail -->|
Account B --> Flow Logs ---->|--> Logging Account (S3 + OpenSearch)
Account C --> App Logs -----|
All logs are immutable, versioned, and encrypted with KMS.

Step 5: Networking Setup

Use AWS Transit Gateway to connect all VPCs:



Benefits:

- **Dedicated VPC per account**
- **Centralized inspection (firewalls)**
- **No overlapping CIDRs**

Step 6: Identity & Access

- **IAM Identity Center / SSO**
- **RBAC per account**
- **MFA enforced**

- **Cross-account IAM roles**

Step 7: Security Foundation

- **GuardDuty enabled org-wide**
- **Security Hub with CIS Level 1 & 2**
- **AWS Config rules enabled**
- **S3 access logs mandatory**
- **KMS for encryption everywhere**

Step 8: Cost Optimization

- **Consolidated billing**
- **Reserved Instances for shared services**
- **Budgets and anomaly detection**

Use Cases

✓ Fortune-500 companies

✓ Regulated industries

✓ Multi-team large orgs

ARCHITECTURE 8 — Serverless Event-Driven Platform (Kinesis / Lambda)

Goal:

Process millions of events/minute with elastic scaling and minimal ops overhead.

🧩 Diagram

Clients → API Gateway → Kinesis Stream → Lambda Consumers → DynamoDB/S3

|

Firehose → S3 / Redshift

🛠 Steps

Step 1: API Ingestion Layer

Use API Gateway:

- Rate limiting
- AuthZ/A (JWT, IAM)
- Request validation

Step 2: Kinesis Data Stream

- Shards handle scaling
- Partition key determines ordering
- 1MB/sec or 1000 records/sec per shard

Scaling:
Add shards → horizontal scaling
Resharding is online

Step 3: Lambda Consumers
Lambda pulls batches from Kinesis:

- Batch size = 100–500
- Retries enabled
- DLQ = SQS
- Timeout optimized (~1–2 sec)

Step 4: Firehose for Heavy ETL
Use Firehose to deliver:

- Parquet data → S3
- JSON → Redshift
- Transformed results via Lambda

Step 5: Storage Layer

- DynamoDB for low-latency key-value
- S3 for raw data lake
- Redshift/Athena for analytics

Step 6: Observability

- CloudWatch Log Insights
- X-Ray for end-to-end latency
- Lambda concurrency alarms

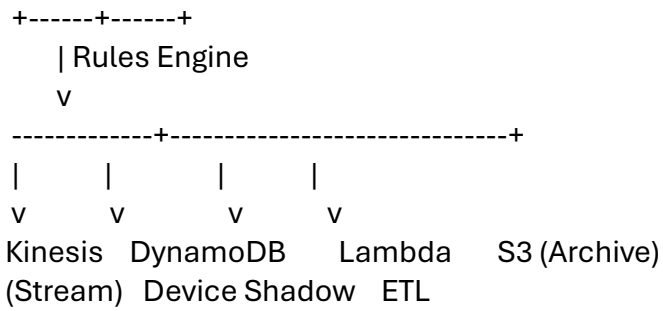
Step 7: Cost Optimization

- Use on-demand Kinesis scaling
- Efficient partition keys
- Compress S3 objects

ARCHITECTURE 12 — IoT Platform Architecture (MQTT + IoT Core + Kinesis + DynamoDB)
Goal:
Support millions of IoT devices sending telemetry in real-time with secure ingestion, processing, device shadows, and two-way communication.

🧩 **Diagram**

```
IoT Devices (MQTT)
     |
     v
+------------+
| AWS IoT Core|
```



🛠 Step-by-Step Design

Step 1: Device Connectivity
Devices connect via **MQTT over TLS 1.2**
Auth:

- X.509 certificates
- Cognito tokens (optional)

All traffic encrypted.

Step 2: Ingestion (IoT Core Broker)
IoT Core broker receives MQTT messages and triggers **Rules Engine**.

Step 3: Rules Engine Routing
Routes incoming messages to:

- DynamoDB (for Device Shadow)
- Kinesis (for streaming analytics)
- Lambda (for processing commands)
- S3 (for cold storage)

Step 4: Device Shadow System
Manages current state of device:

- Desired state
- Reported state
- Delta state

Used for remote control:

- Turn fan ON/OFF
- Update thermostat
- Push firmware

Step 5: Real-time Processing
Kinesis → Lambda

- Aggregates temperature/humidity
- Detects anomalies
- Pushes alerts

DynamoDB Device Shadow Updates

- Low-latency device state
- Fast read/write

Step 6: Batch Analytics
S3 data processed through:

- Athena
- EMR
- Glue
- Redshift

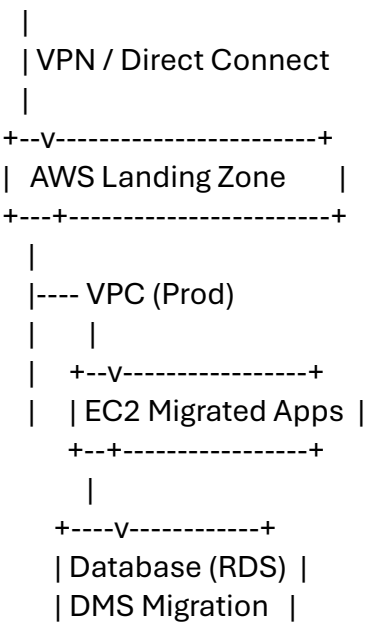
Step 7: Device Commands
Cloud → IoT Core → Device
Via MQTT with QoS.

Step 8: Security

- Mutual TLS
- IAM policies per certificate
- Device revocation list
- JITR/JITP certificate provisioning

ARCHITECTURE 14 — Lift & Shift Migration Architecture (On-Prem → AWS)
Goal:
Migrate legacy workloads with minimal redesign using VPN/Direct Connect, Database Migration Service (DMS), and VM imports.

🧩 **Diagram**



+-----------------+

🛠 Step-by-Step Migration Strategy

Step 1: Network Connectivity
Options:

- Site-to-Site VPN (quick)
- Direct Connect (stable, low latency)

Setup:

- Transit Gateway to attach multiple VPCs
- Route filtering
- BGP for dynamic routing (DX)

Step 2: Assessment
Tools:

- AWS Application Discovery Service
- Migration Evaluator

Identify:

- CPU usage
- Storage needs
- Licensing
- Dependencies

Step 3: Server Migration
Use **AWS Application Migration Service (MGN)**
Steps:

1. Install agent on on-prem servers
2. Continuous replication to AWS
3. Launch test instances
4. Cutover window selected
5. Switch DNS

Step 4: Database Migration
Use **AWS DMS**:

- Minimal downtime
- Continuous replication
- Schema conversion via SCT

Step 5: Storage Migration
Use:

- AWS Snowball for large TB/PB
- AWS DataSync for continuous sync

Step 6: User Identity Integration

- Connect AD → AWS AD Connector
- Single Sign-On

Step 7: DR Setup

- Backups
- Cross-region replication
- RPO/RTO definitions

Step 8: Optimization

Post-migration:

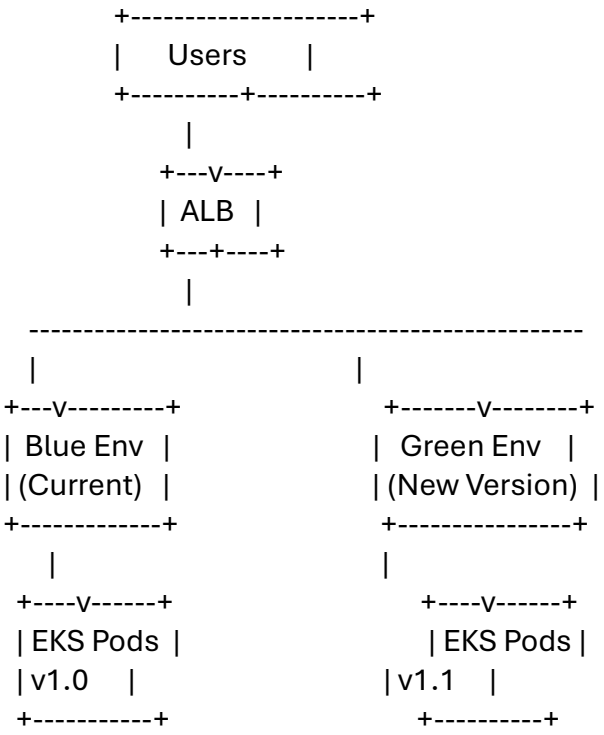
- Convert EC2 → ECS/EKS
- Move DB to Aurora
- Add auto-scaling
- Add caching layer

ARCHITECTURE 16 — Zero-Downtime Blue/Green Deployment Architecture (EKS + ALB + CI/CD)

Goal:

Release applications with **zero downtime**, instant rollbacks, and low-risk deployments.

🧩 **High-Level Diagram**



🛠 **Step-by-Step Design**

Step 1: Two Identical Environments

- **Blue environment** = running production version
- **Green environment** = new version being deployed

Both deployed in:

- Separate EKS namespaces
- Or separate EKS deployments
- Or separate clusters (large enterprises)

Step 2: ALB Target Groups

ALB configured with:

- **TG-Blue** → Blue pods
- **TG-Green** → Green pods

Routing = 100% to Blue initially.

Step 3: CI/CD Pipeline

Pipeline steps:

1. Build Docker image
2. Security scan (SNYK/Trivy)
3. Deploy to Green environment
4. Run smoke tests
5. Validate logs, error rates, latencies

Step 4: Traffic Switch

Once green environment validated:

- Shift ALB routing from Blue → Green
- Begins with partial:
    - 10% traffic
    - 25%
    - 50%
    - 100%

Or immediate full cutover.

Step 5: Instant Rollback

If errors or latency spike:

- Route back to **Blue**
- Green scaled down but not removed
- Debug safely

Rollback time: **< 5 seconds**

Step 6: Cleanup

Once stable:

- Destroy Blue
- Mark Green as new Blue
- Next deployment → new Green environment

Step 7: Monitoring & Observability

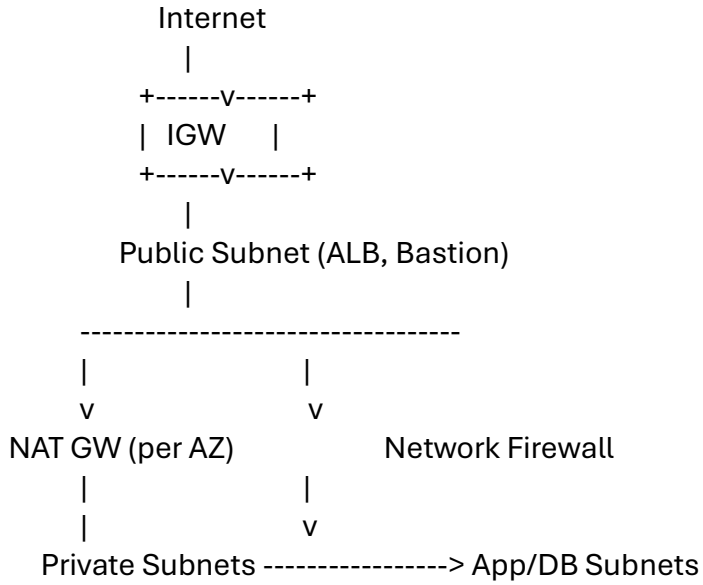
- Compare both versions side-by-side
- Key metrics:
  - 5xx errors
  - p95 latency
  - CPU/memory for pods
  - Logging anomalies

Step 8: Best Practices

- Don't share DB write schemas
- Use read replicas for migration testing
- Keep both environments identical
- Test failure injection on green

ARCHITECTURE 18 — VPC with Public/Private Subnets + NAT + Bastion + Firewalls
Goal:
A secure AWS network where private servers have no public exposure, outbound internet is controlled, and access is tightly limited.

🧩 **Diagram**



🛠 **Step-by-Step Design**

Step 1: VPC & Subnet Layout
Create:

- 3 AZs
- 3 public subnets

- 3 private subnets
- 3 DB subnets

CIDR example:

- VPC: 10.0.0.0/16
- public: 10.0.1.0/24
- private: 10.0.2.0/24
- db: 10.0.3.0/24

Step 2: Internet Gateway + Public Subnets

Public subnets host:

- Bastion host
- ALB
- NAT Gateway

Routes:

0.0.0.0/0 → IGW

Step 3: NAT Gateway in Each AZ

Private subnets route outbound via NAT:

0.0.0.0/0 → NAT-A

NAT is **highly available** when placed per-AZ.

Step 4: Private Subnets

Private workloads:

- EKS worker nodes
- ECS tasks
- EC2
- Lambda in VPC

No public internet exposure.

Step 5: Database Subnets

For RDS/Elasticache:

- No outbound
- No inbound except from app subnets
- Encrypted in transit

Step 6: Bastion Host

Access via:

- SSH session manager (preferred)
- MFA authentication
- IAM role

Security:

- Restricted IP
- Logging via Systems Manager

Step 7: Network Firewall
Use **AWS Network Firewall**:

- Egress filtering
- Domain filtering
- Block malicious outbound traffic

Step 8: VPC Endpoints
For S3, DynamoDB, ECR:

- Reduce NAT cost
- Increase security
- No public internet required

ARCHITECTURE 19 — Hybrid Cloud Architecture (On-Prem ↔ AWS)
Goal:
Connect on-prem data centers with AWS securely to form a hybrid environment for workload extension.

🧩 **Diagram**

```
On-Prem DC
  |
  | IPSec VPN / Direct Connect
  |
+--v--------------------+
| AWS Transit Gateway   |
+--+--------------------+
  |
  |----- VPC (Prod)
  |----- VPC (Shared)
  |----- VPC (Analytics)
```

🛠 **Step-by-Step Design**

Step 1: Network Connectivity
Choose:

- **VPN** (cheap, fast to configure)
- **Direct Connect** (stable, dedicated)

Best: Use **DX with VPN backup**.

Step 2: Transit Gateway
Connects:

- On-prem routers
- Multiple VPCs
- Virtual appliances

- Shared services

Route domain isolation:

- Prod traffic
- Dev traffic
- External traffic

Step 3: Identity Integration

Options:

- AWS Managed AD
- AD Connector
- SAML Federation for SSO
- IAM Identity Center

Step 4: Data Sync

Use:

- AWS DataSync for file servers
- AWS Transfer for SFTP
- RDS Migration Service
- Database replication from on-prem

Step 5: Application Integration

Hybrid workloads:

- On-prem DB + AWS app tier
- On-prem MQ + AWS microservices
- AWS analytics processed from on-prem sources

Step 6: Security

- IPSec encryption
- Route filtering
- Security Group + NACL boundaries
- DDoS protection via AWS

Step 7: DR

On-prem → AWS DR:

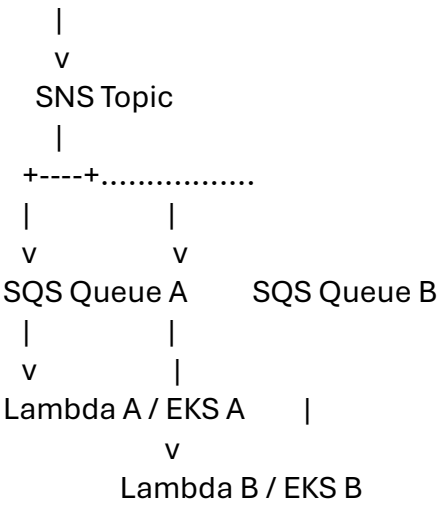
- S3 as off-site backup
- EBS snapshots
- RDS cross-region replication

ARCHITECTURE 20 — Event-Driven Microservices Architecture (SNS + SQS + Lambda/EKS)

Goal:

Loosely coupled microservices communicating via events instead of synchronous APIs.

🧩 **Diagram**



🛠 **Step-by-Step Design**

Step 1: Producer Service

Service A publishes events to **SNS Topic**.

Step 2: SNS → SQS Fan-Out

Multiple microservices can subscribe:

- SQS Queue A
- SQS Queue B
- Lambda
- HTTPS endpoints

Benefits:

- Decoupling
- Parallel processing
- Scalability

Step 3: SQS Queue Consumers

Two options:

Lambda

- Auto-scales
- Event-driven
- Low cost

EKS/ECS Services

- Always running
- Suitable for heavy tasks

Step 4: Retry & DLQ

SQS → DLQ policy:

- 3 retry attempts
- After that → DLQ
- Alerts triggered

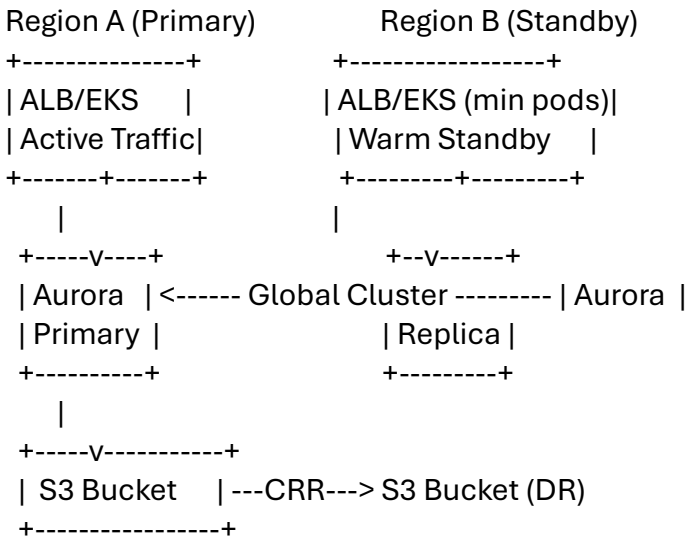
Step 5: Ordering
SQS FIFO Queue ensures:
- Exactly-once processing
- Message order maintained

Step 6: Idempotency
Consumer services must:
- Generate unique message IDs
- Avoid reprocessing repeated events

Step 7: Monitoring
Tools:
- CloudWatch Metrics
- SQS queue depth alarms
- Lambda concurrency alarms
- SNS failure notifications

ARCHITECTURE 22 — Multi-Region Active-Passive Architecture (Warm Standby)
Goal:
Provide fast DR with optimized cost by maintaining a **warm standby** in another region.

🧩 **Diagram**



🛠 **Step-by-Step Architecture Design**

Step 1: Primary Region Setup
Components:

- EKS cluster (full capacity)
- ALB for routing
- Aurora/MySQL PostgreSQL primary cluster
- S3 for storage
- Redis/ElastiCache

Step 2: Standby Region Setup

- EKS with minimal nodes
- Aurora read replica
- S3 CRR
- Offline Redis replica (optional)

Warm but not fully active.

Step 3: Data Replication
Database:

- Aurora Global DB for sub-second replication
- Failover time: **< 1 minute**

Storage:

- S3 CRR replicates objects asynchronously
- DynamoDB Global Tables (optional)

Step 4: DNS Failover
Using **Route53 failover routing:**

- Primary ALB = Active
- Standby ALB = Passive
- Health checks every 30 sec

If Region A fails:
→ Route53 shifts traffic to **Standby Region B**.

Step 5: Autoscaling for DR
On failover:

- EKS CA & HPA triggered
- Nodegroups scale from 1 → required size
- Redis cluster spins up via automation

Step 6: Recovery Runbook

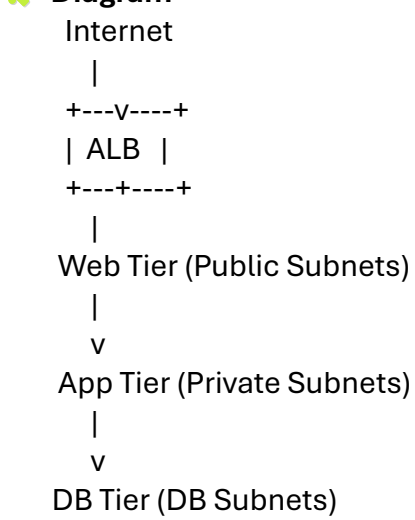
1. Declare failover event
2. Route53 moves traffic
3. Promote Aurora read replica
4. Redeploy pods on standby region
5. Validate endpoints
6. Announce DR-safe state

Step 7: Cost Optimization

- Minimal nodes in standby
- Aurora replica costs only read capacity
- Use spot nodes in DR cluster
- S3 CRR compress objects

ARCHITECTURE 23 — Secure Multi-Tier Architecture (Web → App → DB)
Goal:
Highly compliant, layered architecture where each tier has strict access boundaries.

🧩 **Diagram**



🛠 **Step-by-Step Architecture Design**

Step 1: 3-Tier Network Segmentation

- **Tier 1: Web Tier**
  - ALB
  - Reverse proxies
  - API gateways
- **Tier 2: App Tier**
  - EKS apps
  - Business logic
- **Tier 3: Database Tier**
  - RDS
  - Redis
  - No public routes

Each tier in different subnets.

Step 2: Security Group Chaining

- Web SG → App SG only

- App SG → DB SG only
- DB SG → no outbound

Example rules:
ALB-SG → Allow 80/443 from public
APP-SG → Allow 8080 from ALB-SG
DB-SG → Allow 5432 from APP-SG only

Step 3: Network Access Controls
NACLs:

- Stateless
- Extra protection
- Block unknown IPs

Step 4: WAF & Shield
Deploy WAF for:

- SQL injection
- XSS
- Bot protection

Shield for DDoS.

Step 5: App Tier Architecture
EKS with:

- Pod security policies
- mTLS
- Resource quotas
- HPA

Step 6: DB Tier

- RDS Multi-AZ
- Automated backups
- Encryption at rest & transit
- Parameter group tuning

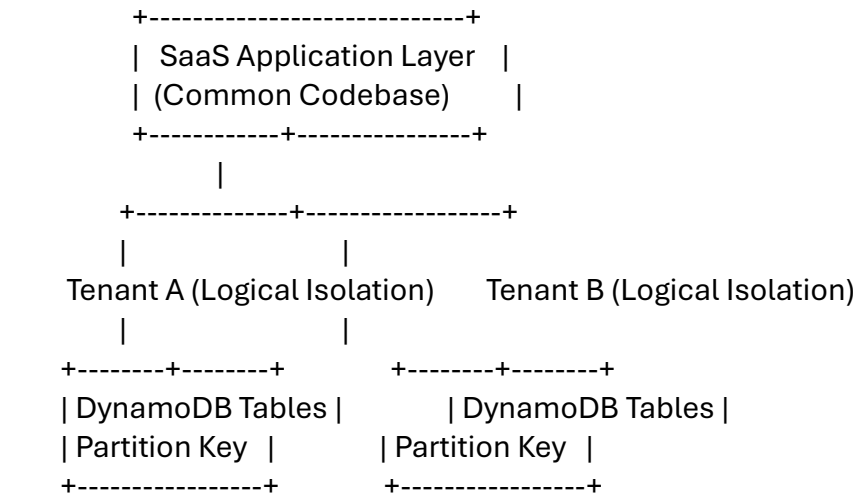
Step 7: Observability

- ALB access logs
- VPC Flow logs
- X-Ray
- CloudWatch metrics

✅ **ARCHITECTURE 24 — SaaS Multi-Tenant Architecture (Tenant Isolation + Single Codebase)**
Goal:

Host multiple customers (tenants) on the same application while maintaining data isolation.

🧩 **Diagram**



🛠 **Step-by-Step Design**

Option 1 — Pool Model (Logical Isolation, Single DB)

Single DynamoDB table with:

- Partition key = tenant_id
- IAM conditions enforce tenant-level access
- Cheaper and simpler

Option 2 — Bridge Model (Mixed)

- One shared DB
- Some resources isolated
- Ideal for mid-sized SaaS

Option 3 — Silo Model (Physical Isolation)

Separate:

- VPC
- DB
- S3 Buckets

Used for:

- High security tenants
- Financial/healthcare

Step 3: Tenant Routing

Use:

- Tenant-specific subdomain
    - customer1.myapp.com

    - customer2.myapp.com
- JWT token includes tenant_id

Step 4: Authentication
Cognito / IAM-based:

- Multi-tenant user pool
- Tenant attributes in token

Step 5: Storage Isolation
DynamoDB isolation:

- Condition expression: tenant_id = ${token.tenant_id}

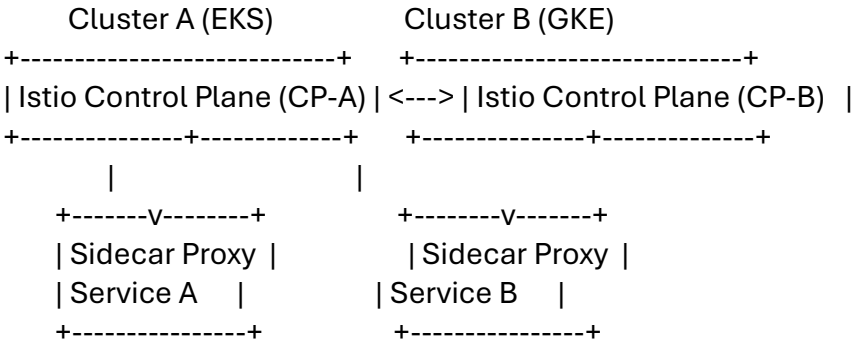
S3 isolation:
bucket/tenant_id/*

Step 6: Observability

- Per-tenant dashboards
- Per-tenant CloudWatch filters
- Billing by tenant usage

ARCHITECTURE 30 — Service Mesh Across Multiple Clusters (Istio Multi-Cluster)
Goal:
Provide **mTLS, traffic control, cross-cluster service discovery, observability** across multiple Kubernetes clusters.

🧩 **Diagram**



🛠 **Step-by-Step Design**

Step 1: Multi-Cluster Topology Choice
Choose mode:
1. Primary-Remote
One control plane, multiple remote clusters
(Simple, centralized)
2. Primary-Primary

Each cluster has its own control plane
More complex but more resilient

Step 2: Cross-Cluster Communication
Use:

- East-West Gateway
- mTLS
- Shared root certificate authority

Service A in Cluster A can call Service B in Cluster B **securely**.

Step 3: Service Discovery
Istio replicates service registry:

- If service removed → sync to other clusters
- Load balances requests cross-cluster

Step 4: Traffic Scheduling
Use DestinationRules to:

- Split traffic across clusters
- Fail over to another cluster
- Apply locality-aware routing

Example:

- 90% to local cluster
- 10% to remote cluster

Step 5: Observability
Unified telemetry across clusters:

- Kiali for mesh topology
- Prometheus for metrics
- Jaeger for tracing
- Grafana dashboards

Step 6: Zero-Trust Security

- mTLS cluster-wide
- RBAC for services
- JWT validation at ingress

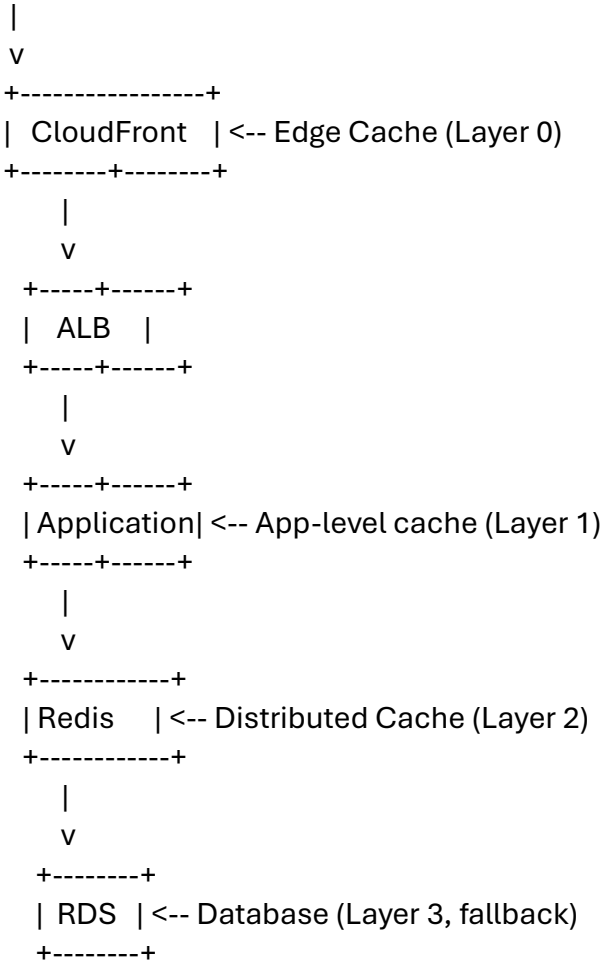
Step 7: Failover
If Cluster A fails:

- Traffic routed to Cluster B automatically
- Control plane detects healthy endpoints
- East-West gateway handles routing

ARCHITECTURE 31 — High-Performance Caching Layer (Redis/ElastiCache + CDN + App Cache)

Goal:
Reduce database load, improve API latency, and scale read-heavy applications using multi-layer caching.

🧩 **Diagram**



🛠 **Step-by-Step Architecture**

Step 1: Layer-0 (CDN Edge Cache)
Use CloudFront for:

- Static files
- Cached API responses
- Signed URLs
- Cache invalidation

Benefits:
✓ Lowest latency
✓ Zero backend compute usage

Step 2: Layer-1 (Application Cache inside Pod/EC2)

Examples:

- In-memory dictionary
- LRU cache
- Local file cache

Benefits:

✓ Nanosecond access

✓ No network roundtrip

✗ Limited by pod lifecycle (evicted on restart)

Step 3: Layer-2 (Redis / ElastiCache Cluster)

Modes:

- Redis Cluster (sharded, high scale)
- Redis Primary/Replica
- Redis with Multi-AZ

Use cases:

- User sessions
- Token verification
- Product catalog
- Leaderboards
- Distributed locks

Key Design Considerations

✓ Eviction Policies

- LRU
- LFU
- Volatile TTL

✓ TTL Strategy

- Aggressive TTL for fast-changing data
- Longer TTL for static data

Step 4: Layer-3 (RDS / Primary Store)

Fallback to DB only when Redis misses.

Step 5: Write-Through vs Write-Back vs Cache-Aside

Cache-aside (most common):

- Read → miss → fetch DB → write cache

Write-through:

- Write to cache → write to DB

Write-back:

- Write to cache only
- Async flush to DB

Step 6: Scaling Redis

- Read replicas
- Cluster mode enabled
- Redis I/O optimized nodes
- Auto-failover

Step 7: Security

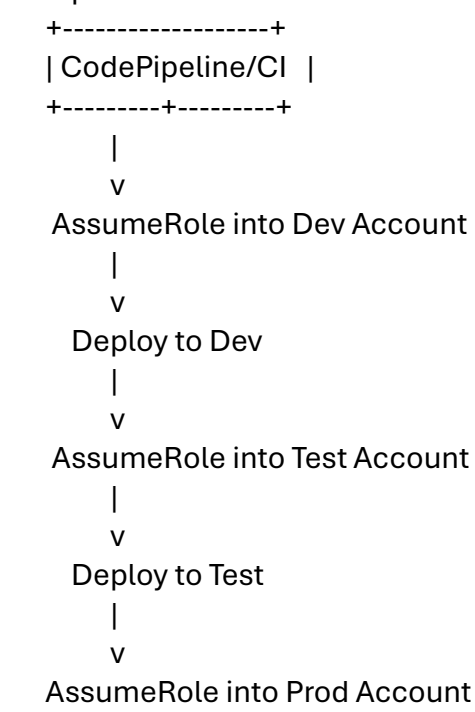
- At-rest encryption
- In-transit encryption
- SG restrictions
- IAM-auth (Redis 7+)

Step 8: Monitoring

- Cache hit ratio
- Evictions
- Memory fragmentation
- Latency of GET/SET

ARCHITECTURE 39 — Cross-Account CI/CD (Pipeline Account → Dev/Test/Prod)
Goal:
Centralized pipeline deploys workloads into multiple accounts securely using IAM Roles and cross-account resource access.

🧩 **Diagram**



|
v
Deploy to Prod

🛠 Step-by-Step Architecture

Step 1: Account Structure

- CI/CD Account
- Dev Account
- Test Account
- Prod Account

Enforced by AWS Organizations.

Step 2: Cross-Account Roles

In each target account:
Role: DeployRole
Trust: Pipeline Account
Pipeline uses:
sts:AssumeRole

Step 3: CI Stage

Runs:

- CodeBuild
- Unit tests
- Linting
- Image build
- Security scanning
- Upload artifacts to S3/ECR

Step 4: CD Stage

Pipeline executes:

- Deploy to dev
- Integration tests
- Deploy to test
- UAT tests
- Manual approval
- Deploy to prod

Step 5: Separation of Duties

Pipeline cannot write to prod unless:

- IAM approval
- Manual stage approval
- Multi-level checks

Step 6: Auditing
CloudTrail logs capture:

- Role Assume events
- Pipeline changes
- Deployment logs

Step 7: Security

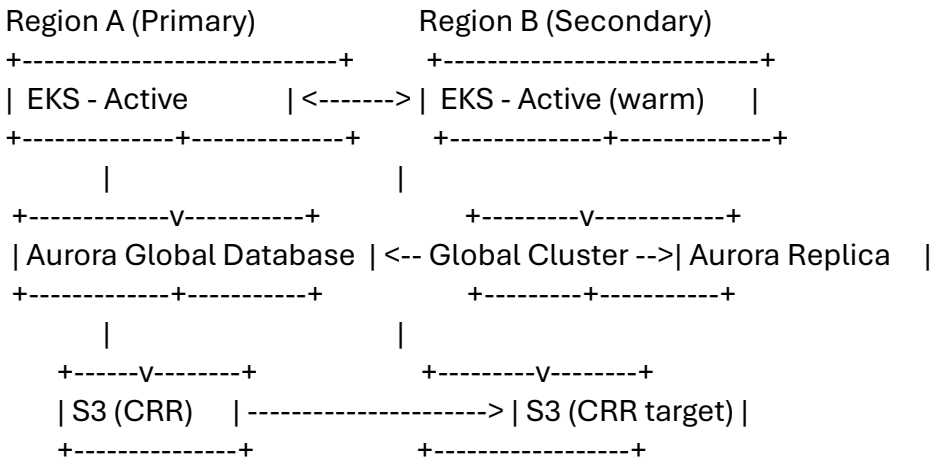
- KMS encryption everywhere
- No long-lived credentials
- IAM boundary policies

Step 8: Pipeline Optimizations

- Parallel stages
- Caching layers
- Canary deployments
- Blue/green for production

✅ **ARCHITECTURE 40 — Large-Scale Disaster Recovery Architecture (RPO 0, RTO < 5 Minutes)**
Goal:
Achieve near-zero data loss and ultra-fast recovery across two AWS regions.

🧩 **Diagram**



🛠 **Step-by-Step Architecture**

Step 1: Compute Layer — Active/Active or Active/Warm
EKS clusters running in both regions:

- Primary cluster handles traffic
- Secondary cluster scaled to minimum

- Autoscaling kicks in during disaster

Step 2: Database Layer — Aurora Global DB
Benefits:
- Sub-second replication
- Fast failover (<15s)
- One primary writer, multiple read replicas

Step 3: Storage Layer — S3 CRR
Cross-Region Replication:
- Versioning enabled
- KMS keys replicated
- Ensures no data loss

Step 4: DNS — Global Failover
Route 53 routing policies:
- Health checks
- Failover routing
- Latency-based routing

Failover time:
- 20–40 seconds DNS propagation
- Combined with Aurora failover = <5 minutes RTO

Step 5: Stateless Microservices
Containers store no state → easy failover.
State stored in:
- Aurora
- Redis Global Datastore
- S3

Step 6: Message Queue Replication
Options:
- Cross-region SQS
- Kafka cluster replication
- Kinesis with enhanced fan-out

Step 7: Automation
DR scripts perform:
- Failover Aurora
- Update environment variables
- Scale EKS nodegroups
- Shift DNS weights

Step 8: Testing

Regular DR drills:

- Chaos engineering
- Simulate region outage
- Validate RTO and RPO