AWS Developer Playbook — Expanded

Audience: Developers, architects and DevOps teams building cloud-native applications for an asset-management / financial-services client

Scope: Practical developer playbook per service (S3 notifications, SQS, SNS, Lambda, DynamoDB, CloudWatch Logs, CloudWatch Alarms, Step Functions, EventBridge, EC2, ECS). Each service includes: overview, architecture patterns, best practices, example IAM policy, CDK (TypeScript) snippet, sample retry/deduplication code, observability, security / regulatory considerations, DR strategies, testing & CI/CD, and a short runbook for common operational failures.

> **How to use this doc:** Copy relevant sections into your internal playbooks, adapt IAM policies & CDK to your account IDs and resource names, and store secrets/keys in Secrets Manager/SSM. This document assumes AWS CDK v2 / TypeScript examples. If you prefer Python CDK or full CloudFormation, tell me and I'll generate it.

Table of contents

---

1. S3 Event Notification

Overview

S3 event notifications can publish `ObjectCreated`, `ObjectRemoved`, and other events to SQS, SNS, or Lambda. Delivery is **at-least-once** and occasionally events can be delayed, duplicated, or (rarely) lost.

Developer playbook & patterns

- Treat S3 events as advisory: the object store is the source-of-truth. Implement periodic reconciliation (list/compare) for completeness.
- Use `versionId` for deduping and to detect updates vs deletes.
- For large objects / multipart uploads use `s3:ObjectCreated:CompleteMultipartUpload` to ensure full object availability.
- For heavy fan-out, prefer S3 -> EventBridge or S3 -> SNS -> SQS fan-out pattern to get durable buffering.
- For sensitive data, use S3 bucket policies + VPC endpoints to restrict access and avoid public egress.

Example IAM (least privilege for S3 to publish to SNS/SQS)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowS3ToSendMessage",
      "Effect": "Allow",
      "Principal": {"Service": "s3.amazonaws.com"},
      "Action": "sts:AssumeRole",
      "Resource": "*"
    }
  ]
}
```

Note: S3 uses a bucket notification configuration; most permission control is done via bucket policy granting `sns:Publish` or `sqs:SendMessage` to `arn:aws:iam::aws:policy/service-role/AmazonS3RoleForReplication` style principals as needed.

CDK snippet (TypeScript): S3 -> SQS notification

```
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as sqs from 'aws-cdk-lib/aws-sqs';
import * as s3n from 'aws-cdk-lib/aws-s3-notifications';

export class S3SqsStack extends Stack {
  constructor(scope: cdk.App, id: string, props?: StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'DataBucket', { versioned: true });
    const queue = new sqs.Queue(this, 'S3EventsQueue', { retentionPeriod:
cdk.Duration.days(14) });
```

```
    bucket.addEventNotification(s3.EventType.OBJECT_CREATED, new
s3n.SqsDestination(queue));
  }
}
```

Sample dedupe/idempotency code (Node.js lambda consumer)

```
// Use DynamoDB for dedupe store - table: s3-event-dedupe (pk: eventId)
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();
const DEDUPE_TABLE = process.env.DEDUPE_TABLE;

exports.handler = async (event) => {
  for (const rec of event.Records) {
    const eventId = rec.eventID; // S3 eventID
    // Try conditional put: if not exists, process
    try {
      await ddb.put({
        TableName: DEDUPE_TABLE,
        Item: { eventId, processedAt: Date.now() },
        ConditionExpression: 'attribute_not_exists(eventId)'
      }).promise();
    } catch (e) {
      if (e.code === 'ConditionalCheckFailedException') {
        console.log('Duplicate event, skipping', eventId);
        continue;
      }
      throw e;
    }

    // process the object
    // ... business logic ...
  }
};
```

Observability

- Enable S3 server access logs and CloudTrail S3 data events for auditing.
- Emit structured logs in downstream processors referencing S3 `objectKey`, `versionId` and eventId.
- Track counts of events, duplicates, failed processings, and reconcilation differences.

Security & regulatory considerations

- Encryption: enforce `SSE-S3` or `SSE-KMS` for buckets with financial data. If KMS, use a CMK with tight key policy and key rotation.
- Bucket policies: restrict `PutObject` and `GetObject` to specific VPC endpoints or IAM principals. Require TLS (`aws:SecureTransport`).
- Data residency: create buckets in approved regions. Use Bucket Lock/Retention for immutable audit artifacts.
- Audit: enable CloudTrail data events for S3 to meet traceability requirements.

DR & Recovery

- Enable Cross-Region Replication (CRR) for critical data; monitor replication lag.
- Keep lifecycle snapshot backups (versioning + lifecycle policies + replication) for RPO.
- Reconciliation job: scheduled process that lists prefixes and compares to processed state (S3 list vs processed table) to find missing events and reprocess.

Runbook (common failures)

- Missing events: check S3 bucket notification configuration, CloudTrail data events and server access logs. Run reconciliation.
- Duplicate processing: verify dedupe table TTL, conditional writes, exceptions during processing that may not have recorded processed marker.

---

2. SQS

Overview

Durable queuing for decoupling components. Standard queues: high throughput, at-least-once, unordered. FIFO queues: ordered and exactly-once semantics (within dedupe window) with lower throughput.

Developer playbook & patterns

- Use FIFO when order matters (trade flows, accounting ledgers) and throughput is compatible.
- Always implement idempotency on the consumer: use message dedupe-id or external dedupe store.
- For long processing: set VisibilityTimeout longer than expected processing time and extend with API if needed.
- Use DLQs to capture poison messages and fix/inspect them manually before reprocessing.
- For large payloads: store payload in S3 and put pointer in SQS (pointer pattern).

Example IAM (consumer role)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "sqs:GetQueueAttributes",
        "sqs:ChangeMessageVisibility"
      ],
      "Resource": "arn:aws:sqs:REGION:ACCOUNT_ID:queue-name"
    }
  ]
}
```

CDK snippet: creating a FIFO queue with DLQ

```
import * as sqs from 'aws-cdk-lib/aws-sqs';

const dlq = new sqs.Queue(this, 'DLQ', { fifo: true, queueName: 'my-
dlq.fifo' });
const queue = new sqs.Queue(this, 'MainQueue', {
  fifo: true,
  contentBasedDeduplication: false,
  queueName: 'my-main-queue.fifo',
  deadLetterQueue: { maxReceiveCount: 5, queue: dlq }
});
```

Sample consumer with retry/backoff and dedupe (Node.js + DynamoDB dedupe)

```
const { SQSClient, ReceiveMessageCommand, DeleteMessageCommand,
ChangeMessageVisibilityCommand } = require('@aws-sdk/client-sqs');
const ddb = new (require('@aws-sdk/client-dynamodb')).DynamoDBClient({});

// Example exponential backoff with jitter
function sleep(ms){return new Promise(r=>setTimeout(r,ms));}
function backoff(attempt){
  const base = 100; // ms
  const max = 10000; // ms
```

```
  const exp = Math.min(max, base * Math.pow(2, attempt));
  return Math.floor(exp / 2 + Math.random() * exp / 2); // decorrelated jitter
}

async function processMessage(msg){
  // dedupe via messageId or body hash
}

// consumer loop (pseudo)
```

Observability

- Metrics: ApproximateNumberOfMessagesVisible, ApproximateNumberOfMessagesNotVisible, ApproximateAgeOfOldestMessage, NumberOfMessagesSent/Received/Deleted, ApproximateNumberOfMessagesDelayed.
- Alert if oldest message age > SLA or visible message count increases beyond threshold.

Security & regulatory considerations

- Encryption: SSE-SQS (KMS) for message payload encryption; manage CMKs and key policies.
- Access control: restrict who can send to and receive from queues (cross-account topics restricted).
- Message retention: set retention period according to regulatory retention policies; use DLQ for forensics.

DR & Recovery

- Use cross-region replication patterns: publish events to a central bus or replicate critical data via S3/DynamoDB Global Tables; SQS itself is regional; for cross-region, use SNS/EventBridge or custom replication.
- Periodic snapshot of message pointers not straightforward; design idempotent replays using source-of-truth.

Runbook

- Increasing queue backlog: inspect consumer scaling (Lambda concurrency or EC2/ECS consumers), check for processing errors, check DLQ counts.
- Poison message loop: inspect DLQ, patch code to handle edge cases, then re-inject items safely.

---

3. SNS

Overview

Pub/sub service used for fan-out to multiple subscribers (HTTP/S, email, SMS, SQS, Lambda).

Developer playbook & patterns

- For guaranteed durable delivery to multiple subscribers, use SNS -> SQS for each subscriber (durable buffer) rather than direct HTTP.
- Use message attributes and subscription filtering to reduce recipient load.
- Protect topics with topic policies and require HTTPS endpoints or signed subscriptions for cross-account.

Example IAM policy (publish privilege)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sns:Publish",
      "Resource": "arn:aws:sns:REGION:ACCOUNT_ID:topic-name"
    }
  ]
}
```

CDK snippet: SNS topic with SQS subscriptions

```
import * as sns from 'aws-cdk-lib/aws-sns';
import * as subs from 'aws-cdk-lib/aws-sns-subscriptions';

const topic = new sns.Topic(this, 'Topic');
const queue = new sqs.Queue(this, 'Queue');
topic.addSubscription(new subs.SqsSubscription(queue));
```

Observability

- Monitor `NumberOfMessagesPublished`, `NumberOfNotificationsDelivered`, `NumberOfNotificationsFailed`.
- Track retry counts and use CloudWatch metrics to trigger alerts for failures.

Security & regulatory

- Encryption: use KMS CMKs for encrypting messages.
- Contracts: message schemas should be documented and versioned.
- For SMS/email, respect regulatory consent and opt-out policies.

DR & Recovery

- Use durable subscribers (SQS) with retry and DLQ to enable replay.

---

4. Lambda

Overview

Serverless compute with maximum 15 minute runtime. Supports sync and async invocation models and event sources including S3, SQS, SNS, EventBridge.

Developer playbook & patterns

- Keep Lambdas small and single-purpose; prefer Step Functions for complex orchestration.
- Use versions and aliases for safe deployment; use traffic-shifting for canary/linear deployments.
- For low latency-critical functions use Provisioned Concurrency.
- Use Lambda Layers to share dependencies and reduce cold start size.

Example IAM execution role (minimal)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {"Effect":"Allow","Action":
["logs:CreateLogGroup","logs:CreateLogStream","logs:PutLogEvents"],"Resource":"arn:aws:logs:REGIO
group:/aws/lambda/your-func*"},
    {"Effect":"Allow","Action":
["dynamodb:PutItem","dynamodb:GetItem"],"Resource":"arn:aws:dynamodb:REGION:ACCOUNT_ID:table/
your-table"}
  ]
}
```

CDK snippet: Lambda with SQS event source

```
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as eventSources from 'aws-cdk-lib/aws-lambda-event-sources';

const fn = new lambda.Function(this, 'Processor', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('lambda/processor'),
  timeout: cdk.Duration.seconds(60)
```

```
});

fn.addEventSource(new eventSources.SqsEventSource(queue, { batchSize: 10,
maxBatchingWindow: cdk.Duration.seconds(30) }));
```

Sample retry/backoff pattern (Node.js) and async handling

- For **sync** invocations: return errors deliberately (HTTP 5xx) and rely on caller retry.
- For **async** invocations: use DLQ or Failure Destinations to capture failed events.

Node.js partial sample: extend SQS visibility timeout while processing

```
const AWS = require('aws-sdk');
const sqs = new AWS.SQS();

async function extendVisibility(queueUrl, receiptHandle, seconds) {
  await sqs.changeMessageVisibility({ QueueUrl: queueUrl, ReceiptHandle:
receiptHandle, VisibilityTimeout: seconds }).promise();
}

exports.handler = async (event) => {
  for (const r of event.Records) {
    const handle = r.receiptHandle; // attempt to extend as you near time limit
    // periodically call extendVisibility(queueUrl, handle, 60);
  }
}
```

Observability

- Use structured logs and X-Ray for distributed tracing. Correlate logs by requestId and trace id.
- Monitor `Invocations`, `Errors`, `Duration`, `Throttles`, and `ConcurrentExecutions`.

Security & regulatory

- Avoid embedding secrets in environment variables unless encrypted; use Secrets Manager & IAM to access secrets at runtime.
- VPC placement: Lambda in VPC requires ENIs; avoid unless necessary. If in VPC, ensure access to necessary endpoints via VPC endpoints.
- For financial workloads, enable CloudTrail data events for Lambda to capture invocation/audit data.

DR & Recovery

- Use multiple regions for critical functions; deploy Canary aliases and traffic-shifting to test failover.
- Use DLQs for async failures and event archive (EventBridge) to replay events.

Runbook

- High error rate: check code errors in CloudWatch logs, check dependent services' availability, increase concurrency or add retries with jitter.

---

5. DynamoDB

Overview

NoSQL key-value and document store with single-digit ms latency. Design must be access-pattern driven.

Developer playbook & patterns

- Model for queries: collect all access patterns first, then design PK/SK and GSIs.
- Single-table design can reduce complexity and latency — document patterns and keep items small.
- Protect against hot partitions by salting keys or redesigning to distribute workload.

Example IAM policy for app to access table

```
{
  "Version": "2012-10-17",
  "Statement": [{"Effect":"Allow","Action":
["dynamodb:GetItem","dynamodb:PutItem","dynamodb:Query","dynamodb:UpdateItem"],"Resource":"arn:aw
transactions"}]
}
```

CDK snippet: table + stream + global secondary index

```
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';

const table = new dynamodb.Table(this, 'TxTable', {
  partitionKey: { name: 'pk', type: dynamodb.AttributeType.STRING },
  sortKey: { name: 'sk', type: dynamodb.AttributeType.STRING },
  billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,
  stream: dynamodb.StreamViewType.NEW_AND_OLD_IMAGES
});

table.addGlobalSecondaryIndex({ indexName: 'GSI1', partitionKey: { name:
'gsi1pk', type: dynamodb.AttributeType.STRING }, projectionType:
dynamodb.ProjectionType.ALL });
```

Sample conditional write / idempotency (Node.js, AWS SDK v3)

```
import { DynamoDBClient, PutItemCommand } from '@aws-sdk/client-dynamodb';
const client = new DynamoDBClient({});

async function putIfNotExists(tableName, pk, sk, payload) {
  const params = {
    TableName: tableName,
    Item: { pk: { S: pk }, sk: { S: sk }, payload: { S:
JSON.stringify(payload) } },
    ConditionExpression: 'attribute_not_exists(pk)'
  };
  try { await client.send(new PutItemCommand(params)); return true; }
  catch (e) { if (e.name === 'ConditionalCheckFailedException') return false;
throw e; }
}
```

Observability

- Monitor throttled requests, read/write latency, consumed capacity. Set alarms on ThrottledRequests and high latency.
- Use CloudWatch Contributor Insights to spot hot keys.

Security & regulatory

- Use encryption at rest (AWS-managed or CMK). If CMK, control key usage via IAM and key policies.
- Use VPC endpoints for DynamoDB to avoid egress.
- For audit trails, enable CloudTrail for table-level API calls and Streams for CDC.

DR & Recovery

- Global Tables for multi-region active-active replication for low RTO/RPO.
- Enable Point-in-Time Recovery (PITR) for recovery to any second in 35 days.
- On-demand backups for retention beyond PITR windows.

Runbook

- Throttling: increase capacity (if provisioned) or enable autoscaling / move heavy reads to eventually-consistent reads or cache (DAX/Redis).
- Lost writes: check conditional write failures and transaction logs.

---

6. CloudWatch Logs

Overview

Central log store. Supports subscription filters, insights queries, and metric extraction.

Developer playbook & patterns

- Ship structured JSON logs with correlation IDs.
- Use log group naming convention: `/aws/<service>/<app>/<env>`.
- Limit verbose logs in production (set log level via environment variables and remote configuration).

Example IAM to push logs

```
{
  "Effect":"Allow",
  "Action":["logs:CreateLogGroup","logs:CreateLogStream","logs:PutLogEvents"],
  "Resource":"arn:aws:logs:REGION:ACCOUNT_ID:log-group:/aws/lambda/*"
}
```

CDK snippet: log group + retention

```
import * as logs from 'aws-cdk-lib/aws-logs';

new logs.LogGroup(this, 'AppLogGroup', { logGroupName: '/aws/app/prod',
retention: logs.RetentionDays.ONE_YEAR, removalPolicy:
cdk.RemovalPolicy.RETAIN });
```

Observability & metric filters

- Create metric filters for critical events (authentication failures, exceptions) and route to CloudWatch Alarms.
- Use CloudWatch Logs Insights for ad-hoc investigations.

Security & regulatory

- Enable encryption for log groups if logs contain sensitive data.
- Ensure log retention policies comply with regulatory retention requirements; export to secure archival (S3 with immutability) if required.

DR & Recovery

- Export critical logs to S3 (lifecycle + Glacier) for long-term retention and legal hold.

---

7. CloudWatch Alarms

Overview

Alerting on metric thresholds and composite conditions.

Developer playbook & patterns

- Use multiple datapoints and evaluation periods to avoid noise.
- Use composite alarms to combine signals (error rate AND latency) to reduce false positives.
- Route alarm notifications through SNS to integrate with paging systems.

CDK snippet: alarm on Lambda error rate

```
import * as cw from 'aws-cdk-lib/aws-cloudwatch';

const metric = fn.metricErrors({ period: cdk.Duration.minutes(1) });
new cw.Alarm(this, 'LambdaErrorAlarm', { metric, threshold: 5,
evaluationPeriods: 3, alarmDescription: 'High lambda error rate' });
```

Security & regulatory

- Keep alarm history and attach runbooks. For financial services, store incidents and resolution steps for audits.

DR & Recovery

- Automate safe remediation (e.g., restart service) but gate production-critical automation with manual approval or progressive escalation.

---

8. Step Functions

Overview

Serverless orchestration. Standard (durable, audit-able) vs Express (high throughput, lower cost for short-lived flows).

Developer playbook & patterns

- Use Step Functions for transactional flows that need retries and clear state. Use explicit `Retry` and `Catch` clauses.
- Keep state small (avoid embedding large objects in state); use S3 pointers.
- Prefer integrated service actions (AWS SDK integration) to call AWS services directly from state machine (avoids Lambda shim).

CDK snippet: simple state machine

```
import * as sfn from 'aws-cdk-lib/aws-stepfunctions';
import * as tasks from 'aws-cdk-lib/aws-stepfunctions-tasks';

const hello = new tasks.LambdaInvoke(this, 'CallLambda', { lambdaFunction: fn,
payloadResponseOnly: true });
const sm = new sfn.StateMachine(this, 'StateMachine', { definition: hello,
stateMachineType: sfn.StateMachineType.STANDARD });
```

Observability

- Execution history is available in console; enable X-Ray for end-to-end tracing where needed.

Security & regulatory

- Control who can start/stop executions; log execution inputs/outputs carefully (avoid PII in plain text unless encrypted and audited).

DR & Recovery

- Replay capability for reprocessing fragments: if state machine supports idempotency and S3 pointers, you can re-run executions or run compensating transactions.

---

9. EventBridge

Overview

Central event bus for event-driven architectures. Supports schema registry, archive & replay, cross-account event delivery.

Developer playbook & patterns

- Define event schemas and version them; include `source`, `detailType`, `detail` JSON and correlation IDs.
- Use EventBridge archives to enable replay for recovery and testing.
- Use rule-level filtering to avoid fan-out storms.

CDK snippet: rule and target

```
import * as events from 'aws-cdk-lib/aws-events';
import * as targets from 'aws-cdk-lib/aws-events-targets';

const rule = new events.Rule(this, 'Rule', {
  eventPattern: { source: ['com.mycompany.orders'] }
});
rule.addTarget(new targets.LambdaFunction(fn));
```

Observability

- Monitor `Invocations`, `FailedInvocations`, `MatchedEvents` for rules, and check archive/replay counts.

Security & regulatory

- Use cross-account restrictions and resource policies to tightly control which principals can put events.
- Archive events to S3 for auditing and evidence in legal/regulatory events.

DR & Recovery

- Use EventBridge archive + replay to reprocess events into targets in another region/account after failover.

---

10. EC2

Overview

Traditional VM compute; used for workloads requiring full OS control, specialized drivers (GPUs), or custom networking.

Developer playbook & patterns

- Use AMI baking (Packer/Image Builder) to create golden images with security baseline.
- Use IMDSv2 enforced and instance roles for obtaining temporary credentials; do not store secrets on disk.
- Use Auto Scaling Groups with health checks and graceful termination hooks.

CDK snippet: simple auto-scaling group

```
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as autoscaling from 'aws-cdk-lib/aws-autoscaling';

const vpc = ec2.Vpc.fromLookup(this, 'VPC', { isDefault: true });
const asg = new autoscaling.AutoScalingGroup(this, 'ASG', { vpc, instanceType:
new ec2.InstanceType('t3.medium'), machineImage:
ec2.MachineImage.latestAmazonLinux() });
```

Observability

- Install CloudWatch agent to gather system-level metrics (memory, disk) not available by default.

Security & regulatory

- Encryption of EBS, snapshots, strict IAM roles. Use VPC endpoints and private subnets for sensitive workloads.
- Patch management with SSM Patch Manager.

DR & Recovery

- Use AMI automation and ASG to recover compute quickly. For AZ/region level failures, deploy multi-region with DNS failover.

---

11. ECS (EC2 & Fargate)

Overview

Container orchestration — allow running containers on Fargate (serverless) or EC2 (self-managed host).

Developer playbook & patterns

- Use awsvpc network mode for task level networking and predictable security groups.
- For secrets, use Secrets Manager/SSM and reference in Task Definitions.

- Use health checks, deployment circuit breakers and ALB target group draining to ensure safe rollouts.

CDK snippet: Fargate service behind ALB

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecs_patterns from 'aws-cdk-lib/aws-ecs-patterns';

const cluster = new ecs.Cluster(this, 'Cluster', { vpc });
new ecs_patterns.ApplicationLoadBalancedFargateService(this, 'Service', {
cluster, taskImageOptions: { image:
ecs.ContainerImage.fromRegistry('myimage:latest') } });
```

Observability

- Collect container logs via awslogs or FireLens to central store; track task restarts and exit codes.

Security & regulatory

- Image scanning with ECR, use immutable tags, image signing where required.
- Isolate prod cluster in separate account or VPC for tenancy and compliance.

DR & Recovery

- Use multiple AZs for tasks. For region-level failover, deploy to alternate region and shift traffic via Route53.

Cross-cutting: Regulatory & Financial Services Considerations

Key controls to implement

- **Encryption**: all PII and financial data encrypted at rest (S3, EBS, DynamoDB, RDS) and in transit (TLS 1.2+). Use KMS CMKs when auditability & access control required. Rotate keys as per policy.
- **Auditability**: enable CloudTrail (management & data events), and log all API access. Keep immutable audit logs in a separate account and S3 bucket with object lock for legal hold.
- **Least privilege**: strict IAM policies, role separation (developer vs operator vs admin), and no long-lived credentials.
- **Data residency & sovereignty**: deploy to approved AWS regions; enforce via SCPs (AWS Organizations) and guardrails.
- **Retention & deletion**: implement data retention policies and secure deletion; document for compliance (SEC/FINRA etc.).

- **Change control**: use CI/CD pipelines with approvals for prod changes and maintain infrastructure-as-code history (Git).
- **Incident response / forensics**: centralize logs, enable packet captures where necessary, maintain runbooks and RACI.

Regulatory specifics to consider (asset management)

- Records retention (SEC/FINRA) — maintain transaction/audit records for mandated durations.
- Trade surveillance — ensure event logging has required fields and timestamps.
- Encryption & key management controls — proof of access controls and rotation.
- Segregation of environments — production isolation and mandatory change approvals.

Cross-cutting: DR strategies & RTO/RPO recommendations

1. **Define RTO/RPO** per workload. Classify services: critical (RTO < 1 hour), important (RTO < 4 hours) and best-effort.
2. **Multi-AZ first**: use AZ-resilient services by default (DynamoDB single-region multi-AZ, S3 multi-AZ durability, ECS multi-AZ tasks).
3. **Multi-region** options:
4. Use Global Tables (DynamoDB) and Cross-Region Replication (S3 CRR) for near-active-active or warm-standby.
5. For compute/lambda, deploy to multiple regions and use Route53 weighted failover or Route53 health checks.
6. **Event replay**: enable event archives (EventBridge) or store events in durable S3 + pointer patterns enabling replay.
7. **Backups**: enable PITR (DynamoDB), automated snapshots (EBS), periodic AMI & RDS snapshots (if RDS used) and export logs to S3.
8. **Runbooks & drills**: maintain runbooks and perform scheduled DR drills (failover + failback) with stakeholders.

Cross-cutting: Testing, CI/CD & Automation

- Use IaC (CDK, Terraform) for all infra — avoid manual console changes.
- Build automated smoke tests and integration tests as part of pipelines; include replay tests for event-driven flows.
- Use Canary deployments for production changes and automated rollbacks.
- Add infrastructure tests (cfn-nag, cdk-nag, tfsec) to the pipeline.
- Maintain separate AWS accounts for prod, staging, dev and centralize shared infra (logging/monitoring) in a separate account.

Example runbook template (use and adapt per integration)

1. **Title:** S3→SQS→Lambda processing failure
2. **Symptoms:** Backlog in SQS, Lambda errors or DLQ fills.
3. **Quick checks:**
4. Check SQS ApproximateNumberOfMessagesVisible and AgeOfOldestMessage.
5. Check Lambda CloudWatch logs for stack traces and throttles.
6. Check DLQ for poison messages.
7. **Remediation:**
8. If code error: fix and redeploy via pipeline; reprocess DLQ items manually after verification.
9. If throttling: scale consumers (Lambda concurrency, ECS task count) and/or increase visibility timeout.
10. If messages duplicated: verify dedupe table TTL and error handling.
11. **Postmortem:** record root cause, duration, and steps to prevent recurrence; update alarm thresholds if needed.

Attachments & next steps

- If you want, I can generate:
- CDK repo skeleton (TypeScript) with setup pipeline and per-service examples.
- Detailed runbook PDFs per integration.
- IAM policy matrix for roles and responsibilities.

End of playbook.