DOMAIN 1: DEVELOPMENT WITH AWS SERVICES

Core Competencies

- Develop and deploy serverless applications
- Use AWS SDKs to interact with services
- Implement service integrations (e.g., S3 + Lambda)
- Handle application errors and performance
- Use Step Functions, messaging, and caching efficiently

Essential AWS Docs & References

- AWS Lambda Developer Guide
- DynamoDB Best Practices
- SQS Developer Guide
- Step Functions Developer Guide
- AWS SDK for Python (boto3)
- API Gateway Docs

Summary Cheat Sheet

- Lambda: stateless, event-driven compute; use versions/aliases
- API Gateway: HTTP/REST APIs with throttling, caching, auth
- DynamoDB: NoSQL with on-demand/provisioned modes, indexes
- SQS/SNS: queue-based async messaging; FIFO vs standard
- Step Functions: serverless workflows; handle retries visually
- CloudFront: global CDN with origin failover and caching
- ElastiCache: Redis/Memcached for speed, session, state
- SDKs: boto3 (Python), aws-sdk (Node.js), include retry logic

1.1 Develop Code for Applications Hosted on AWS

- AWS Lambda
  - Runtime support: Python, Node.js, Java, Go, .NET, Ruby
  - Triggers: S3, API Gateway, EventBridge, DynamoDB Streams, SQS
  - Versions & Aliases: enable traffic shifting and deployment strategies
  - Environment Variables: key-value store, can be encrypted with KMS
  - Timeout: maximum 15 minutes
  - Memory Allocation: 128 MB to 10 GB (affects CPU)
  - Best Practices
    - Use provisioned concurrency for low-latency apps
    - Externalize secrets using Secrets Manager or SSM Parameter Store
    - Enable X-Ray for tracing
    - Use layers to include dependencies
- API Gateway + Lambda Integration
  - Proxy mode: passes full request/response (most common)
  - Non-proxy (custom) mode: granular control, requires mapping templates
  - Secure APIs using IAM, Lambda Authorizers, or Cognito User Pools
- Amazon S3
  - SDK actions: PutObject, GetObject, DeleteObject (basic CRUD)
  - Presigned URLs: secure temporary access
  - S3 Event Notifications: trigger Lambda/SNS/SQS on uploads
  - Use S3 Transfer Acceleration for faster uploads globally

1.2 Interact with AWS Services Using SDKs

- AWS SDKs (Python boto3, Node.js aws-sdk)
  - Examples:
  - S3 Upload: s3.put_object(Bucket='mybucket', Key='file.txt', Body=bytes)
  - DynamoDB Get: dynamodb.get_item(TableName='Users', Key={'id': {'S': '123'}})
  - SQS Send: sqs.send_message(QueueUrl=url, MessageBody="msg")
  - Handle retries and exponential backoff (automatic but customizable)

1.3 Integrate Services for Decoupled Architectures

- Amazon SQS (Message Queuing)
  - Standard queue: high throughput, at-least-once delivery
  - FIFO queue: ordered, exactly-once delivery, slower
  - DLQ (Dead-letter queue): for failed messages
  - Visibility Timeout: prevents duplicate processing
  - Polling
    - Short polling: returns immediately
    - Long polling: reduces empty responses and costs
- Amazon SNS (Pub/Sub Messaging)
  - Topics distribute messages to multiple endpoints
  - Subscribers: Lambda, SQS, Email, HTTP/S
  - Message Attributes: filtering logic for subscribers
  - Use SNS → SQS for fan-out with guaranteed delivery
- Step Functions
  - State Machine: visual workflow using JSON/YAML
  - Service Integrations: Lambda, ECS, Batch, DynamoDB
  - Error Handling: catch and retry blocks built in
  - Wait States: delay or poll for result
  - Use Express Workflows for high-volume short-duration steps

1.4 Implement Application Resiliency and Performance Enhancements

- Retry Logic and Backoff
  - SDKs auto-retry 3–4 times by default
  - Customize with exponential backoff plus jitter
  - Handle errors like ThrottlingException, ProvisionedThroughputExceededException
- API Gateway Caching
  - Integration caching: reduces backend calls using TTL and key-based caching
  - Authorization caching: speeds up custom authorizer responses
  - Stage-level caching: can be toggled per stage/environment
- CloudFront
  - CDN for static and dynamic content
  - Works well with S3 + Lambda@Edge
  - Supports signed URLs and geo restrictions
- Amazon ElastiCache (Redis/Memcached)
  - Redis: in-memory store with pub/sub, TTL, persistence
  - Memcached: lightweight, multi-threaded cache
  - Caching strategies: write-through, lazy loading, TTL management
  - Use cases:
    - Caching DB queries
    - Session state for apps
    - Temporary computation storage

1.5 Optimize Application Performance

- DynamoDB Best Practices
  - Partition key design: avoid hot keys by distributing workload
  - LSI/GSIs: enable alternative query patterns
  - Provisioned vs On-Demand: use on-demand for unpredictable traffic
  - DAX: in-memory caching for DynamoDB
  - Batch operations: use BatchGetItem and BatchWriteItem for efficiency
- Service Integration Patterns
  - Event-driven: S3 → Lambda, SQS → Lambda
  - Queue-based decoupling: Frontend → SQS → Worker
  - Fan-out: SNS → SQS, Lambda, HTTP
  - Workflow orchestration: Step Functions → Lambda chain
- Example DVA-C02 Scenarios
  - Avoid multiple retries for same SQS message: use visibility timeout + DLQ
  - Fix slow Lambda first invoke: use provisioned concurrency
  - Send file securely to S3 for 1 hour: use pre-signed URL from SDK
  - Coordinate 3 Lambda steps with error handling: use Step Functions with retry logic
  - Resolve Lambda throttling: increase reserved concurrency or use DLQ

Presented with xmind AI