Serverless and Application Services

Architecture Deep Dive - Event-Driven Architecture

Types of architecture:

- MONOLOTHIC: everything is built together and coupled; everything is always running and incurring charges even if not being used; if one part fails, whole thing fails
- TIERED: monolith broken apart into tiers that can be on same server or different servers. Tiers still coupled as they each connect to endpoint of next tier. Benefits are that tiers can be vertically scaled independently (server size of each tier can be increased). -- if you add load balancers between tiers, this abstracts tiers meaning that you can now horizontally scale tiers by adding instances (the tiers are no longer connected to each other but to LBs. Still not decoupled though.
- QUEUE-BASED Decoupled Architecture: The queue would sit between components (tiers) to decouple, the components no longer require answers; now the components are using async communications
- MICROSERVICE Architecture: tiers/components become Microservices that can be duplicated. There are Producers, Consumers, and Both
    - Producers produce data, Consumers consume data/messages, 'Both' does ..both
        - Events are what are consumed. Queues can be used to communicate events
- EVENT DRIVEN Architecture: Event Producers / Event Consumers. You can have components that can do Both
    - Best-practice Event-driven arch's have Event Routers, which are highly available central exchange points for Events. Nothing is sitting around waiting, they are activated when an Event is sent to them; only consumes resources while handling events (serverless)

Lambda

- FaaS: Function as a Service - short running and focused
- Lambda function: piece of code that lambda runs
- Functions use a runtime like Python 3.8
- Functions loaded into a runtime environment; env has a direct memory (indirect CPU) allocation
- Billing: for the duration that the function runs
- Lambda is a key part of Serverless architectures
- Lambda Function: Think of it as the code + association wrappings/config; a deploment package. Stateless
- You define Lambda resources like Memory (128MB -> 10240MB in 1MB steps), which then scales how much CPU you'll get (you can't directly control how much CPU). Storage in Lambda stored in /tmp 512MB -> 10240MB

EXAM: Lambda function timeout 900s (15 minutes). Anything beyond 15 mins can't use Lambda directly

Common Uses of Lambda

- Serverless applications
- File processing

- Database triggers
- Serverless CRON
- Real time stream data processing (kinesis + lambda)

Lambda Networking Modes

1. Public (Default)
2. Private (in VPC)
3. Public: Could access Public AWS svc's like SQS, DynamoDB, or external like IMDB

- Lambda functions have no access to VPC unless public IPs are provided and security controls allow external access

2. Private (in VPC): Lambda functions running in a VPC obey ALL VPC networking rules (can't access external stuff unless configured as such). Eg. Could use a gateway endpoint to access dynamodb

Lambda - Security

- Execution Role: For Lambda env to access AWS p&s it needs an Execution Role to gain the role's permissions based on the role policy
- Lambda Resource Policies: similar to bucket policy on S3 which controls WHO/WHAT can interact with a specific Lambda function. Eg. services like SNS or S3 being allowed to invoke the function

Lambda - Logging

Lambda uses CloudWatch, CloudWatch Logs, and X-Ray

- Logs from Lambda executions -> CloudWatch Logs
- Metrics like invocation success/failures, retries, latency -> CloudWatch
- Distributed Tracing Capability -> X-Ray. Eg. Trace the path of a user through an application NOTE: CloudWatch Logs requires permissions via the Execution Role

Lambda - Invocation

Types:

- Synchronous
- Asynchronous
- Event Source Mappings

Synchronous: CLI/API invokes lambda function and then CLI/API WAITS for a response. Errors/Retries/Reprocessing must happen on the client side. Synchronous is generally invoked by a human

Asynchronous: Typically used when AWS svc's invoke lambda functions. Reprocessing is handled by Lambda

- Lambda function needs to be idempotent; reprocessing a result should have the same end-state. Eg. Adding +$10 to an account or Setting account to $10 explicitly. If adding fails in a certain way, you

could end up with 20, 30, 40 etc. Idempotent failures still explicitly set the account to $10 value over and over
- Events can be sent to DLQ's (dead letter queues) after repeated failed processing
- Lambda support destinatoins where successful or failed events can be sent like SQS, SNS, Lambda & EventBridge

Event Source Mapping: Typically used on streams or queues which don't support event generation to invoke Lambda (Kinesis data stream, DynamoDB streams, SQS queues, Amazon Managed streaming for Apache Kafka). Event Source Mapping polls these streams/queues looking for data and getting back 'source batches', split based on batch size, sent to Lambda function

Lambda - Versions

Lambda can have versions... v1, v2, v3. The VERSION of Lambda function is 'code + config'

- Versions are immutable when published and has its own ARN. $Latest points at the ..latest... version
- Lambda functions can have aliases like DEV, STAGE, PROD which can be changed

Lambda - Startup times

- An 'execution context' is the environment a lambda function runs in
- A 'cold start' is a full creation and configuration including function download
- A 'warm start' can occur if another lambda function is invoked shortly after a first, now it doesn't have to rebuild the execution context (no build process)
- Event source mapping uses permissions from the Execution Role to access the source svc that it pulls data from

CloudWatch Events and EventBridge

CloudWatch Events and EventBridge have visibility over events generated by supported AWS services within an account.

- They can monitor the default account event bus - and pattern match events flowing through and deliver these events to multiple targets.
- They are also the source of scheduled events which can perform certain actions at certain times of day, days of the week, or multiple combinations of both - using the Unix CRON time expression format.

EventBridge is replacing CW Events as EventBridge can do all the same things but also handle third-party and custom app events (AWS recommending using EventBridge)

Key Concepts, CloudWatch Events and EventBridge

- If X happens, or at Y times... do Z
- EventBridge is basically CW Events v2
- A default Event Bus exists on an AWS account for both CW Events and EB
- In CW Events, only 1 event bus. In EB you can have add't event busses
- Rules created match incoming events (or schedules) -- Two rule types: 1) Event Pattern Rule 2) Schedule Rule

DEMO - Automated EC2 Control Using Lambda and Events

Gain experience of using Lambda for some simple account management tasks.

1. 1 click deploy
2. Give Lambda Permissions: https://learn-cantrill-labs.s3.amazonaws.com/awscoursedemos/0024-aws-associate-lambda-eventdrivenlambda/lambdarole.json > IAM > Role > Create Role > Entity Type, AWS Service, select Lambda > Next
3. Create IAM Role Policy: Click 'Create Policy' > JSON tab > Paste JSON provided > Next > Policy name "Lambdastartandstop" > Create Policy
4. Create IAM Role with Policy: IAM Role tab > Select Policy for IAM Role > Role name "EC2StartStopLambdaRole" > Create Role
5. EC2 > Copy both instance ID's to notepad/clipboard
6. Create Lambda Function: Lambda > Create function > select Author from Scratch > name "EC2Stop" > runtime "Python 3.9" > dropdown "Change default exec role", select Use Existing, select the IAM Role you created > Create Function > paste in code to Lambda code block from file lambda_instance_stop.py > click Deploy
7. Creat env variable for EC2Stop: Lambda function, configuration tab > Environment Variables "Edit" > click "Add env variable" > key: EC2_INSTANCES, value: "ec2_instance_ID_1,ec2_instance_ID_2"
8. Test Event: EC2Stop Lambda Function, Test tab > Event name "test" > Test. Observe EC2 instances being stopped.
9. Create second function "EC2Start" > Lambda, Functions "Create Function" > name "EC2Start" > Existing IAM Role > Create function > add new code from lambda_instance_start.py > Paste code, Deploy > Environment Variables "Edit" > click "Add env variable" > key: EC2_INSTANCES, value: "ec2_instance_ID_1,ec2_instance_ID_2" > Save
10. Test Event: EC2Start Lambda Function, Test tab > Event name "test" > Test. Observe EC2 instances being started.
11. Set up Event-driven Lambda Function: Lambda > New Function name "EC2Protect" > Runtime python 3.9 > Select IAM role that we created > Create function > Add code from file "lambda_instance_protect.py", Deploy
12. Create EventBridge Rule for Lambda to receive: EventBridge > New Rule name "EC2Protect" > desc "Start protected instance" > select "rule with an event pattern" > Event Source "AWS events or EventBridge partner events" > Event pattern "AWS Services", "EC2", event type: EC2 Instance State-change Notification, Specific States: Stopped > Specific Instance IDs, paste instance 1 ID > Next > Target 1 AWS Svc, "Lambda function", Function: EC2Protect > Create Rule
13. Clean up: Lambda, delete functions > EventBridge, delete Rules > IAM Policies, delete Lambda policy > Delete Lambda IAM Role > CloudFormation, delete Stack

Serverless Architecture

The Serverless architecture is a evolution/combination of other popular architectures such as event-driven and microservices.

- Aims to use 3rd party services where possible and FAAS products for any on-demand computing needs.
- Using a serverless architecture means little to no base costs for an environment - and any cost incurred during operations scale in a way with matches the incoming load

Serverless - Key Concepts

- Serverless isn't one single thing; aiming to manage few, if any, servers for lower overhead
- Stateless and Ephemeral env's; duration billing
- Function as a Service; FaaS used where possible for Compute functionality
- Managed Services are used where possible; web ld providers, S3 for storage, etc.
- Event-driven; consumption only when being used

Serverless: TL;DR and Architecture

Aim should be to consume as a service whatever you can, code as little as possible, use function as a service for any general compute needs

SNS - Simple Notification Service

The Simple Notification Service or SNS is a PUB / SUB style notification system which is used within AWS products and services but can also form an essential part of serverless, event-driven and traditional application architectures.

- Public AWS Service; network connectivity with a Public Endpoint
- Publishers send messages to TOPICS
- Subscribers receive messages SENT to TOPICS
- Messages are <= 256KB payloads
- SNS supports a wide variety of subscriber types including other AWS services such as LAMBDA and SQS
- SNS is REGIONALLY RESILIENT. Highly available and scalable
- Can provide delivery STATUS and delivery RETRIES
- SNS capable of server-side encryption
- SNS can be cross-account with a TOPIC POLICY

AWS Step Functions

AWS Step Functions addresses some problems within Lambda.

- Lambda is FaaS, each Lambda should be as simple as possible; you'd never put a full application in a Lambda Function (execution duration limit: 15 minutes). Theoretically, you can chain Lambda functions but this gets messy at scale
- Lambda runtime env's are stateless (can't hold state through different Lambda functions)

Step Functions let you create STATE MACHINES (a workflow with start point and end point). START -> STATES -> END

- States are THINGS which occur
- Max Duration of State Machines is 1 YEAR
- Two workflows in State Machines:

1. Standard Workflow (default); 1 YEAR execution limit
2. Express Workflow: designed for high-volume, event-processing workloads, etc for UP TO 5 MINUTES

- You can Export your State Machine as a JSON template called "Amazon States Language (ASL)"

- IAM Role used for permissions

Step Functions: State Machines: Types of States

- Succeed and Fail.
- Wait. Wait for certain period of time to elapse, or for a particular date/time that was set. Holds / Pauses workflow until duration passes
- Choice. State Machine can take different paths depending on input
- Parallel. Create parallel branches within a state machine
- Map. Takes list of items and performs action/set of actions on each item in list
- Task. Represents a Single unit of work performed by a State Machine

API Gateway 101

API Gateway is a managed service from AWS which allows the creation and management of API Endpoints, Resources & Methods. Endpoint/entry-point for applications

- The API gateway integrates with other AWS services - and can even access some without the need for dedicated compute.
- It serves as a core component of many serverless architectures using Lambda as event-driven and on-demand backing for methods.
- It can connect to legacy monolithic applications and act as a stable API endpoint during an evolution from a monolith to microservices and potentially through to serverless.
- API Gateway sites between applications and integrations (services)
- Can connect to svc's/endpoints in AWS or ON-PREMISES
- HTTP APIs / REST APIs / WEB SOCKET APIs

API Gateway - Authentication

- Can use COGNITO user pools for authentication
- Can use LAMBDA for authentication

API Gateway - Endpoint Types

- Edge-optimized. Incoming requests routed to nearest CloudFront POP (point of presence)
- Regional. Clients in the same region
- Private. Endpoint accessible only within a VPC via interface endpoint

API Gateway - Stages

- When you deploy an API config in API gateway, you do so in a stage. APIs are deployed to stages, each stage has one deployment. Eg. prod and dev stages

API Gateway - ERRORS

EXAM - Memorize numbers

- Error codes generated by API gateway 2 categories:

1. 4xx - 400 series client-side error

2. 5xx - 500 series server-side error

- Error Codes: -- 400: Bad request; generic; many root causes possible -- 403: Access Denied; authorizer denies or WAF filtered -- 429: Throttling is occuring in API Gateway -- 502: Bad gateway excption; bad output returned by Lambda -- 503: Service unavailable. Backing endpoint offline or major service issues -- 504: Integration Failure/Timeout - 29s limit for any requests to API gateway Resource: https://docs.aws.amazon.com/apigateway/latest/api/CommonErrors.html

API Gateway - Caching

- Caching is configured per Stage
- Cache TTL default: 300 seconds; configurable min/max 0s/3600s
- Can be encrypted
- Cache size: 500MB -> 237GB

DEMO - Build a Serverless App - Skipping for now

Simple Queue Service (SQS)

SQS queues are a managed message queue service in AWS which help to decouple application components, allow Asynchronous messaging or the implementation of worker pools.

- SQS comes in 2 types:

1. Standard - best efforts; messages could be received out of order
2. FIFO - guarantees the order

- Like SNS, messages <= 256KB in size
- Polling: When a client checks for messages on a queue EXAM -- Received messages are HIDDEN (VisibilityTimeout - amount of time a msg is hidden). If the message isn't explicitly deleted, it will reappear (retry) in the queue
- Dead-Letter Queue: Can be used for problem messages. Eg if message is received 5 or more times
- SQS good for scaling: ASG's can scale based off SQS, Lambdas can invoke based on queue length -> This is good for ASG Worker Pool Architecture EXAM - Fanout Architecture: SNS fans messages out to multiple SQS queues

Delivery Methods - Standard / FIFO

- Standard = guaranteed to deliver 'at-least-once' with no promise on order of delivery
- FIFO = guaranteed to deliver 'exactly-once' and guarantee message order -- FIFO performance is lower: 3,000 messages / second with batching or 300 messages/second without batching. Not as good for scaling as Standard
- Billing: Based on "requests". A request can be 1-10 messages up to 64KB total
- Polling Types (2)

1. Short Polling (immediate)
2. Long Polling (waitTimeSeconds, up to 20s) - BEST PRACTICE; more cost effective as less requests per message. Will poll for a 20s duration collecting up to 10 messages and 64KB in data

- SQS supports encryption at rest (KMS) and in-transit

- Can add Queue policies

SQS - SQS Standard vs FIFO Queues

Standard

- Multi-lane highway. Scalable, nearly unlimited Transactions per second
- Best-efforts ordering; no rigid order
- Best for decoupling, worker pools, Batch for future processing

FIFO

- Single-lane highway (performance limited by width of lanes). 300 TPS w/o batching, 3,000 TPS with
- Delivered in a guaranteed order EXAM - FIFO queues Must have a .fifo suffix
- Exactly-once processing: removes chance of duplicate message delivery

SQS - Delay Queues

Delay queues provide an initial period of invisibility for messages; postpone delivery to consumers. Predefined periods can ensure that processing of messages doesn't begin until this period has expired.

- A message is automatically hidden once it hits a queue
- NOT the same as Visibility Timeout; VT is automatic re-processing of problem messages whereas this is an initial delay in processing of all messages
- DelaySeconds min is 0, it has to be non-zero to be a Delay Queue
- NOT SUPPOERTED ON FIFO QUEUES

SQS - Dead-Letter Queues (DLQ)

Dead letter queues allow for messages which are causing repeated processing errors to be moved into a dead letter queue.

- Eg. Message received but keeps re-appearing after visibility timeout and not explicitly deleted (this could technically happen forever unless you set up a DLQ) -- Each time a message re-appears in queue after timeout, the ReceiveCount tally is incremented -- when ReceiveCount > maxReceiveCount && message is NOT deleted, it's moved to DLQ

Kinesis Data Streams

Kinesis data streams are a real-time, scalable streaming service within AWS designed to ingest large quantities of data and allow access to that data for consumers

- Ideal for dashboards and large scale real time analytics needs
- Kinesis data firehose allows the long term persistent storage of kinesis data onto services like S3
- Producers send data into a kinesis STREAM, which can scale from low to near infinite data rates
    - Streams store a 24-hour moving window of data
        - This storage is included and can be increased to MAX 365 days of moving data window (add'l costs)
        - Multiple consumers access data from this moving window

Kinesis VS SQS

- SQS = decoupling, asynchronous communications. No message persistence, no window of data stored
- Kinesis = huge-scale data ingestion, real-time data streaming. Made for multiple consumers and has a rolling window of stored data for persistence (default 24 hrs, up to 365 days for add'l cost)

Kinesis Data Firehose

Fully managed service to load data for data lakes, data stores (like S3), and analytics svc's -> This lets data be persisted beyond the rolling window of Kinesis Data Streams

- Auto scaling, fully serverless, resilient EXAM - `NEAR real-time` delivery (~60s) which is unlike the main Kinesis Data Stream which offers real-time delivery
- Firehose supports transformation of data on the fly using Lambda
- Valid destination endpoints for Firehose: HTTP, Splunk, Redshift, ElasticSearch, S3

Kinesis Data Analytics

Amazon Kinesis Data Analytics is the easiest way to analyze streaming data, gain actionable insights, and respond to your business and customer needs in real time.

- Real-time processing of data using SQL
- Ingests from Kinesis Data Streams or Firehose
- Destinations of analyzed data: Firehose (and subsequently S3, Redshift, ElasticSearch, Splunk), AWS Lambda, Kinesis Data Streams -- If OUTPUT to firehose, data becomes NEAR REAL-TIME. If output to Kinesis Data Streams or Lambda data stays REAL-TIME

When to use Kinesis Data Analytics

- streaming data needing real-time SQL processing
- time-series analytics like elections, esports
- real-time dashboards like leaderboard for games
- real-time metrics like for security/response teams

Kinesis Video Streams

Amazon Kinesis Video Streams makes it easy to securely stream video from connected devices to AWS for analytics, machine learning (ML), playback, and other processing. Kinesis Video Streams automatically provisions and elastically scales all the infrastructure needed to ingest streaming video data from millions of devices

- ingest LIVE video data from things like security cameras, phones, cars, drones, time-serialised audio, thermal, depth, RADAR
- consumers can access data frame-by-frame or as-needed
- Can PERSIST and ENCRYPT (in-transit and at-rest) EXAM: CANNOT directly access the data via storage, only via APIs
- Integrates with other AWS svc's like REKOGNITION and CONNECT

Amazon Cognito - User and Identity Pools

A user pool is a user directory in Amazon Cognito. With a user pool, your users can sign in to your web or mobile app through Amazon Cognito

- Users can also sign in through social identity providers like Google, Facebook, Amazon, or Apple, and through SAML identity providers
- Whether your users sign in directly or through a third party, all members of the user pool have a directory profile that you can access through a Software Development Kit (SDK).
- Amazon Cognito identity pools (federated identities) enable you to create unique identities for your users and federate them with identity providers. With an identity pool, you can obtain temporary, limited-privilege AWS credentials to access other AWS services.

Cognito provides the following for web/mobile apps:

- authentication
- authorization
- user management

Cognito has 2 components:

1. User pools: for sign-in. You get a JSON Web Token (JWT) if successful (but most AWS svc's don't use this JWT)

- user directory mgmt/profiles, sign up and sign in (with UI), MFA and other security features
- most AWS services CANNOT be accessed using JWT via User Pools
- can use internal or social logins for sign in

2. Identity pool: Swap external ID for temporary AWS credentials.

- Unauthenticated ID's; guest users

USER POOLS = SIGN IN / SIGN UP IDENTITY POOLS = ABOUT SWAPPING ID TOKENS (with Google, Apple, etc) for temporary tokens to access AWS services

Cognito - Web ID Federation (using User Pools and Identity Pools in conjunction)

- First, User Pool JWT replaces all the different external token configs you'd need for handling different external IDs (User Pool removes admin overhead of handling all external social logins)
- Next, app can pass this User Pool token through to an Identity Pool
- Federated Identities can swap (Eg. give Google creds to swap for AWS creds)

AWS Glue

- Fully managed SERVERLESS extract, transform, and load (ETL) service which makes it easy for customers to prepare and load their data for analytics -- Can create and run an ETL job with a few clicks in the AWS Management Console.
- Simply point AWS Glue to your data stored on AWS, and AWS Glue discovers your data and stores the associated metadata (e.g. table definition and schema) in the AWS Glue Data Catalog. Once cataloged, your data is immediately searchable, queryable, and available for ETL.
- Moves/transforms data between source/destination

- Data Sources: Stores: S3, RDS, JDBC Compatible, DynamoDB. Data Sources: Streams: Kinesis Data Stream, Apache Kafka
- Data Targets: S3, RDS, JDBC Databases
- Crawls data sources and generates the AWS Glue Data Catalog

AWS Glue Data Catalog

- Persistent metadata about data sources within a region
- One catalog per region per account
- Configure crawlers for data sources

Glue Jobs

Extract, transform (using script), and load Jobs

- Can be initiated manually or via EVENTS using EventBridge

Glue VS Data Pipeline (both do ETL)

- Glue is serverless, ad-hoc, more cost-effective
- Data Pipeline requires a server

Amazon MQ

AmazonMQ is an open-source message broker; an AWS implementation of Apache ActiveMQ. Like a merge of SNS/SQS

- Supports open standards such as JMS, AMQP, MQTT, OpenWire and STOMP
    - If you need to support any of these, and use QUEUES and TOPICS - AmazonMQ is the tool to use.
- Provides Queues and Topics (like SQS/SNS); one-to-one or one-to-many EXAM: Amazon MQ is NOT a public service; runs on a VPC and requires private networking to access it EXAM: AWS Integration required? Use SNS/SQS and not MQ. EXAM: Migrate from an existing system with little to no app change? Amazon MQ EXAM: Need APIs like JMS or protocols like AMQP, MQTT, OpenWire, STOMP needed? Amazon MQ

Amazon AppFlow

Amazon AppFlow is a fully managed integration service that enables you to securely transfer data between Software-as-a-Service (SaaS) applications like Salesforce, SAP, Zendesk, Slack, and ServiceNow, and AWS services like Amazon S3 and Amazon Redshift, in just a few clicks.

- With AppFlow, you can run data flows at enterprise scale at the frequency you choose: on a schedule, in response to a business event, or on demand.
- You can configure data transformation capabilities like filtering and validation to generate rich, ready-to-use data as part of the flow itself, without additional steps.
- AppFlow automatically encrypts data in motion, and allows users to restrict data from flowing over the public Internet for SaaS applications that are integrated with AWS PrivateLink, reducing exposure to security threats. -- Can use AppFlow Custom Connector SDK to build your own integration if the integration doesn't already exist

- Exchange data between applications (connectors) using FLOWS; SYNC or AGGREGATE data
- PUBLIC endpoints, but works with PrivateLink
- CONNECTIONS store config/creds to access apps. Connections can be reused across many flows