Preparation for AWS Cloud Developer Associate Exam

Notes were created alongside of a Udemy course AWS Certified Developer Associate - AWS Certification from Ranga Karanam from in28MinutesOfficial. This course has taught me the practical skills needed to pass the exam and I can only recommend it.

Link to the course: https://www.udemy.com/course/aws-certified-developer-associate-step-by-step/

Author: VojtaKai

In case of any complaints/issues please open a PR with your commentary.

Multiple Regions, some regions have multiple availability zones (AZ) – called –a, -b, -c, …

EC2

- Virtual servers
- Work with load balancing
- Has one instance storage (500) – disappear after termination, has one root EBS (SSD drive) attached, plus can have multiple more EBS
- Instance types – Compute optimized – usual, memory, disk, network, they include - Instance family of the instance – e.g. t and instance generation – 2, higher number, more improved, instance size – nano, micro, small,…, xlarge – bigger is more powerful.
- **Change instance type: stop the instance, change it, restart it**
- General info about EC2 instance can be drown directly from itself through metadata – http://169.254.169.254/latest/meta-data/
- Dynamic data - http://169.254.169.254/latest/dynamic
- IP addresses – public – can be enabled, when stopped it changes (stays after reboot though), private – created always, elastic IP = static public address – **can be assigned to any one EC2 Instance within the same region, has to be manually detached after instance is terminated**
- Userdata (boostraping them)– script run at EC2 launch – install OS patches, software, slow, not ideal
- Launch Template – automized way to launch EC2 instances with preconfigured configuration, can use spot instances or spot fleets,
- AMI – Mirror image of an existing ec2 instance and its setting, can be integrated into launch template -> we can created hardened image including necessary settings, configuration and software, **most useful, fast setup, launch, you should typically stop the instance before making an AMI image.**
- AMI types – by AWS, customized (created by you), aws market place – you'll be billed for services/softwares included in the template
- AMI can be share across accounts, if you want to use it in different region/AZ (for AZ you have to specify it which subnet=AZ you want), you have to copy it there, you can set up permissions who can use your image, stored in S3 in the specific region, make back up in several regions
- Key pair – cryptographic login credentials to EC2 instance – good security habit, you don't need it but is adviced – private and public key, public in EC2 instance, private you store it, set chmod 400, you can use these keys for multiple EC2 instances or generate for each new (but then you have a lot of private keys to keep track of), **however, I don't think you can have more than two active at the same time. You can delete one, it takes a couple of days, not totally sure, maybe case with KMS keys**
- Connection via Manager console – EC2 Instance Connect, standalone SSH client (SSH, Putty), Session Manager, ssh –i private-key instance-user@public-ec2-IP-address.compute.amazonaws.com
- When having a windows server – you get private key and encrypted psw for login, you use private key to decrypt the password and then use the password to connect to the instance
- **Protect EC2 instance against accidental deletion – termination protection on – not effective against spot instance shut down, OS shutdown, ASG**
- **VM Import/Export service allows you to use VM image from on-premises server and use it to create the same image in EC2**
- **EC2 command timeout – check SG**

- **If you stop EC2 you won't be billed for it, but you can be billed for the storages and elastic IP if you have one**

Security Groups – allows connection to and from EC2 instance, typically stateful, has rules – deny by default, allow only, protocol + port + source (all IP addresses, only limited range), e.g. http – 80 – 0.0.0.0 all

- Up to 5 SG for a EC2 instance
- Separate rules for inbound and outbound traffic – but still stateful!
- SSH – 22, http – 80, https – 443, RDI – 3389
- **Assigned at the launch of the template, but can be changed in a runtime, takes effect immediately**

Elastic Load Balancer

- Distributes traffic across multiple EC2 instances in one or more AZs in a **single region**
- ELB scales itself automatically, requests are going through it!
- Used with ASG and EC2 inst
- Health checks of instances by default, Health check of ELB can be enabled
- 3 types: Classic (old), Application (most common), Network
- Private, public – public needs port 80 open

ELB + EC2 instances

- http port open on ELB
- EC2 instances with SG that only allows traffic from ELB -> **requests wont reach EC2 Insts, should be only sent to ELB**

Layers:

- Network – IP – Internet Protocol – unreliable, bytes and bits = 0s and 1s
- Transport – TCP – reliable > performance, TLS- secure, UDP – performance > reliable – streaming, gaming, high performing internet speed demanding applications (**layer 4**)
- Application – http, HTTPS – secure, Websockets, SMTP – email protocol, FTP – file transfer protocol – web application, REST APIs (**layer 7**)

Listeners

- ELB has one or more listeners
- Listeners listen to requests from clients
- **Listener = Protocol + Port**
- Listener(Protocol + Port) + Rule(s) -> http:80 – forward traffic to this target group

Target Groups

- Group of instances where traffic is distributed from ELB
- Target group of EC2 instances, lambdas, set of IP addresses (different web apps)
- CLB can only have one, ALB can have many
- Target (EC2 instance) can be part of multiple target groups

Distribution strategy

- Round Robin – 1. req to 1. target, 2.2 to 2., 3. to 3., 4. to 4, 5. to 1., 6. to 2.
- Least outstanding – least busy target gets the request

Target group configurations

- Slow start – newly created target, traffic is slowly shifted there, gradual increase
- Deregistration delay = connection draining – when we are going to terminate an instance, we stop traffic to this instance (unregister the instance) and give time to the instance to finish processing of the in-flight requests, 0 – 3600 seconds, default: 300s
- Sticky session – requests from one user send to one instance (or target group?), cookies used, CLB and ALB support

Microservices

- ALB and NLB can take multiple target groups, CLB cant
- Separate target group for each microservice

Listener Rules

- identify where the request should be sent, to which target group
- multiple listener rules possible for single listener
- **order is important, default rule executed last**
- **ALB** – **advanced routing** – based on path (/path), host/domain address (a.path, b.path), http header and methods (put, post, delete, get), query strings (/path?target=a), IP addresses

ASG

- Scaling in/out EC2 instance (single target of a target group)
- ELB + ASG + EC2 instances, ASG is the target
- **Can ASG be a target group?**
- Target group have typically static set of instances (constant count of instances)
- **Min, max, desired count** of instances, scaling based on traffic
- Health checks on instances by default
- **Created through launch configuration or launch template (both can have AMI)**– supports on-demand, spot or both instances
- Health check of instances by default
- Health check grace period – time for which no health check is performed on EC2 instance, applies to newly launched instances so they are not marked as unhealthy because they are not serving traffic when they are setting up. Default 300s if ASG created using mgmt console else 0s
- has Auto Scaling Policies, schedule policies
- any changes to EC2 instances – create new launch template, assign it to the ASG, terminate the old instanced in batches, ASG will create new EC2 instances with the changes

Auto Scaling Policies

- rules for scale in/out
- what to monitor (CW alarm), what action (ASG?) -> CW alarm triggers action in ASG and it launches/terminates EC2 instances
- **Cooldown period** – prevent frequent scale-in/out of the ASG, for the cooldown period the next action will be ignored, default 300s, align CW monitoring – no need for 1 min monitoring instead of default (and free) 5 min interval
- **Warm up period** – configure for each instance, time for which EC2 instance is not avaliable to CW metrics and therefore traffic, when time passes, the traffic will be sent to the instance

Dynamic auto scaling policies (static min=max=desired=const EC2 instance count)

- **Target tracking scaling** – metric + target value – e.g. CPU at 70 %, add +1 instance, simpliest but most useful, automatically sets the CW alarms for when the Instance should be terminated
- **Simple scaling** – metric + upper and lower limit required, CPU > 70 % -> +5 instances, CPU < 60% -> -3 instances, cooldown period can be set
- **Step scaling** = more complex simple scaling - metric + upper and lower limit required, +1 if CPU between 70 and 80 %, +3 if between 80 and 100 %, -1 if CPU between 40 – 60 %

ASG Lifecycle Hook

- Set up in ASG
- For setting up EC2 instance between it's available to the customer/traffic is sent there
- Lifecycle of an EC2 instance – sets it in wait state and you set action what you want to happen, then it move to next state
- Configuration that you want to set, e.g. run a script or use user data, before the EC2 has traffic assigned and before

ASG instance termination at scale-in

- Default termination policy – 1. Priority – EC2 instances should be in all AZs of the region, 2. Priority – newer instances will be kept, older go
- or oldest terminated first

ASG protect newly launched instances from scale-in

- enable instance scale-in protection
- tip: adjust CW monitoring

Classic Load Balancer

- Support layers 4 and 7
- Not recommended anymore
- CLB migrated to ALB

Application Load Balancer

- Layer 7 only
- Load balances between, possible targets: EC2 instances, containerized app (ECS), web app (IP addresses), lambdas
- Cross zone enabled by default, can be specified to only some of the AZs
- enables multi value headers (array of headers) and routes them accordingly using advance routing techniques

Network Load Balancer

- Layer 4
- For high performance use cases
- Can be assigned static IP/Elastic IP
- Load balances between, possible targets: EC2 instances, containerized app (ECS), web app (IP addresses), **no lambda functions**

Serverless = no worries about infrastructure, flexible scaling, high availability, pay for use

Lambda and API Gateway

Lambda

- Running code written in popular languages
- Serverless
- Price depends on: memory of lambda, number of requests, duration of requests
- Memory: 128 MB – 10 GB with 64 MB increments
- CPU is proportional to memory
- Memory influences duration of request -> play around for optimum pricing
- Execution time: 3s default, max 900s (15 mins)
- Integrates with CW – monitoring, logs
- Integrates with X-Ray (request tracing across aws services and resources)
- 500 MB storage – non-persistent - /tmp directory
- **Environmental variables** to pass operational parameters
- Place Java .jar dependency files in **/lib directory of the deployment package** - reduces time to unpack it – faster and cheaper lambda

Lambda Reserved Concurrency

- Ensure that lambda runs by reserving number of parallel LF instances
- Region quota for all lambdas is 1000

Lambda Provisioned Concurrency

- Makes lambda runs continually -> + Execution context is always ready
- Costly, however, lambda doesn't need to prepare execution context (lambda initialization) after longer time of inactivity, useful only when execution context is big
- Specified for lambda version or alias

Lambda Execution Context

- Temporary runtime environment created by invoking lambda function
- Cache static assets locally
- cached in 512MB /tmp directory
- Reusable by all running instances of lambda functions
- Only things outside of lambda handler function will stay initialized in execution environment and can be reused
- Things common for you lambda request should be initialized before function handler (order of code is important)
- Cold start – first initiation of execution context

Lambda throttling

- When hitting region quota – requests cannot be processed, lambda will fail with 429 status code
- Exponential back-off

Synchronous invocation

- Aws lambda invoke CLI command
- Lambda runs the function and wait for response
- Services: API Gateway, CloudFront, Lex (ML)

Asynchronous invocation

- Lambda does not wait for function response
- Lambda places events on an event queue
- Events from S3, SNS queue
- Exponential backoff – default 6 hours, retry attempts, dead letter queue

Lambda request context

- Handler(event, context)
- RequestID, user(cognito identity, userpool)
- Function name, version, arn, memory, logStreamName, logGroupName

Lambda@Edge

- Runs lambda function to customize CloudFront content
- Improves performance of the web app that runs on your servers, edge devices are closer to your clients and therefore the web app is faster
- Only Python and NodeJS
- At different edge locations (200+)
- 4 possible trigger points – can work either with request or response
- Client – CF(cache, Lambda@Edge) – Origin Server(EC2 instance, regular lambda)

Lambda function versioning

- You publish versions – immutable copies of you LF
- Function code + dependencies, lambda runtime, unique arn, environment variables + other settings
- $LATEST – points to the latest version

Alias

- Pointer to a specific lambda version
- Prevents that lambdas will automatically change with new version code
- Alias can be used as a policy for resource-based policies, or blue/green deployment

Layers

- Allows sharing libraries and dependencies among lambda functions
- ZIP file
- You should keep deployment packages small < 10 MB (allows you to use Mgmt Console) + add Layer(s)
- 1 LF can have **up to 5 layers**
- They get extracted to **/opt directory** and made available to your lambda functions
- Use SAM or CF to assign Layers to LF

Lambda logging

- Automatically implements CW, lambda role needs permission to be able to send the logs to CW

Lambda tracing

- Enable X-Ray, give lambda execution role permission to upload to X-Ray (same as above)

API Gateway

- Frontdoor to your APIs – API Gateway actions as a frontend for different backend services
- Service to publish, maintain, monitor and secure APIs at any scale
- Supports HTTP(S) and WebSockets
- Serverless – pay for use – API calls and connection (request) duration
- Resources (/machines/{clientId}/{machineId}) and actions – HTTP methods (get, post, put, delete)
- Authentication and authorization, set limits of request also for a specific customer– usage plans, API keys, multiple version of your API up at the same time, allows monitoring, caching, etc.
- 3 types: REST API, HTTP API, WebSockets API

REST API

- Restful API – uses http methods like GET, POST, PUT, DELETE, ..
- More complex than HTTP API
- Request/Response validations, custom transformations – mapping templates (VTL – Velocity Template Language), test invocations, API caching
- Integrates with CW, X-Ray, WAF – firewall around your API
- 2 types: Custom Integration (default), Proxy Integration

Custom Integration

- Method Request, Response – checks event and what is required in path/query strings/headers, event (model) validation, status code
- Integration Request/Response – request/response transformation – mapping templates

Proxy Integration

- Predefined structure for request and response transformations
- Provides a customizable wrapper around your request
- Proxy uses standard transformation templates – event and lambda response have standardized structures
- Enable it in Integration Request - checkbox
- Request must have headers, query strings, body
- Response – required: status code, body; not required but often – (response) headers

HTTP API

- Restful API
- Lightweight version of REST API
- Newer, simpler, cheaper, low-latency
- Automatic deployment possible – you can cherry pick which resources you exactly want
- Makes OAuth authentication simple
- Route – method + resource path + integration target (LF)
- Integration
- Regular proxy structure for request and response

HTTP 1.0

- For migrating REST API to HTTP API
- Typical proxy structure – request and response as REST API

HTTP 2.0

- For new APIs
- Typical proxy structure – request and response as REST API, request supports cookies, response can be just body (status code, headers automatically attached)

WebSockets API

- Persistent connection

Endpoint types

- Edge optimized – uses CF and edge location – usecase – for globally distributed clients
- Regional – for clients in a single region
- Private – API Gateway can only be access within your VPC through VPC endpoint (in API Gateway)

Integration types

- LF – connect via proxy or custom integration
- HTTP – connect to HTTP(S) endpoints in or outside of AWS
- Mock – mocked response
- AWS Service – connect to AWS service endpoints (DynamoDB, Kinesis)
- VPC link – for private endpoint, connect only to resources inside a VPC

Deployment stages

- For different environments (dev, test, prod) – create stages and deploy your endpoint there
- **Stage variables** – API – Stages – Stage variables -> connect to different LF or alias based on the stage variable in the given stage

Caching

- Saves money
- Enable cache for specific stage of the endpoint only
- TTL – how long a record is saved in cache, default – 300s, max 3600s, 0s for no caching
- CW metrics – CacheHitCount, CacheMissCount – check if caching works
- Cache keys – only cache responses for requests with a specific headers, path, query string parameters, set directly on the methods of the resources
- No caching on HTTP APIs

Identity Federation

- Authentication of users with credentials from external authentication system and allow them into yours, check if they are authorized and if yes, generate credentials (token) to access resources on the cloud
- Corporate (Enterprise) Identity Federation – enterprise DB of employees – SAML protocol (XML based)
- Web Identity Federation – Social IDs – OpenID protocol

Cognito

- **Authentication and authorization service + syncs data across devices**
- **Authorizer for API Gateway**
- Create your own DB of users – scales automatically

- Provides customizable web UI sign-up, sign-in pages, password reset
- Integrate with web identity provides
- User pool, Identity pool
- **Stores user data – can sync across multiple devices**
- Allows MFA – phone, email

User pool

- Database of users
- Can integrate with **ALB** and **API Gateway**
- Authenticates users (requests)
- **Users authenticated by user pool get JWT user pool token**
- **Third party authorizers** – Amazon, Google, FB, Apple, SAML, OIDC – configure with user pool – return **JWT identity provider tokens**
- **These tokens can be exchanges in Identity pool for temporary AWS credentials**

Identity pool

- Provides AWS credentials to grant users access to Cloud resources
- Integrates with these identity providers: userpool, social IDs, OpenID Connect (Web Identity Federation), SAML 2.0 – Corporate Directory (Corporate Identity Federation)
- **After uses are authenticated (get tokens) – they can exchange them for AWS credentials in a form of role. Even unauthenticated users can assumes a specific role (if configured)**
- **AssumeRoleWithWebIdentity, SAML**
- Can integrate with more than one authentication service (mentioned above) – there is a tab with all these identity providers and you manage authentication and authorization for any of them.

Customize Userpool workflow – trigger points

- User pool offers points in time - during and after user authentication – when LFs can be triggered
- Trigger a LF at specific times
- Sign up – accept or deny sign up request, post sign-up
- Sign in – accept or deny sign in request (authenticate or not), after user authentication, after sign-up
- User migration – when user is migrated from an existing user directory to user pools

API Gateway – possible Authorizations

- Open – no authentication or authorization
- **IAM** Permissions – IAM User + AWS credentials of the user, is this user allowed to invoke this endpoint? IAM user needs to have IAM Permission – effect: allow, action: invoke-api, resources: api-of-endpoint (API Gateway – Method Request- authorization set to AWS_IAM)
- **Cognito** authorizer – authenticate users with methods defined in Identity pool – userpool, socialID, corporate directory, check if the authenticate user is authorized to invoke this endpoint, API Gateway – Authorizers – Create New Authorizer – type: Cognito – connect to existing user pool. 3 steps: user signs up for userpool, user signs in and gets identity token back, passes the user's identity token to the request as a header
- **Lambda** – custom authorizer to validate bearer token (OAuth or SAML) or request parameters, output: user policy + principal (who can use the policy) -> Action: invoke-api,

effect: allow/deny, resource: endpoint arn, API Gateway then evaluates the lambda output – denies/gives access

API Gateway + Lambda

- **Max request process time for lambda is 30 seconds because API Gateway times out after 30 seconds**
- When client doesn't want lambda response from back-end only (not cache), user needs execute-api:InvalidateCache IAM Permission, send request with Cache-Controle:max-age=0 in header

API Gateway

- Customized plans for you API customers – basic, premium, full – usage plans (who can access certain APIs/methods resources and setting of quotas for requests) + API keys + other forms of authorization are possible
- Authorization possibilities: Congito, API-key, custom LF to check permissions
- API authorization only for existing AWS users in your account (root account) – IAM Permissions Authorization
- Create users when they use API – implement Cognito Userpool

S3

- Storage service – object storage, inexpensive, stores all types of files – for large objects
- Stored as **Key-Value pairs**, Key – "object name", Value – the object itself (**Key unique in a bucket**)
- Path to any object: s3://bucket-name/object-key
- REST API, V something Endpoint
- 4x 9s availability, 11x 9s durability – **objects replicated across multiple AZ, at least 3**
- different storage classes – the less you access your data and the slower you can retrieve it, the cheaper the storage you can use, also implement data lifecycle
- usecases: media files, archives, application packages and logs, backups of data (DB or storage devices), staging during migration from on-premises to cloud
- **Name of the bucket globally unique for your account – S3 is GLOBAL SERVICE, however, buckets are created in a specific region**
- **Object lock** – if you don't want the object removed, **enable bucket versioning**
- **Bucket names are part of URLs – can contain only lower case letters, number, hyphens, periods**
- **Unlimited objects in a bucket**
- **Max** object size – **5 TB**
- **3500** requests/sec to add data, **5500** to retrieve data **per prefix**
- For standard buckets – Mgmt Console, APIs, CLI
- For Glacier – only APIs and CLI
- **Transfer acceleration** – paid, moves data faster

S3 Versioning

- Protects against accidental deletion
- Versioning can be enabled any time, older objects will be saved as version 0
- You cannot turn it off, only suspend it for new objects created

Server access logging

- turn on bucket level
- not free, paid, detailed logging who accessed an object
- logs will be saved in another S3 bucket – on this bucket create permission that the first bucket can write the logs in this bucket (ACL)
- takes hours before logs are delivered

Static Website Hosting S3

- place index.html there and all static content
- enable Static website hosting
- block public access
- Bucket policy – enable public read

Object level logging

- Recording all API calls
- Can be implemented with CloudTrail (All API calls and events – reads, writes, upload will be tracked by it)

Bucket Policies

- Resource-based policies
- Sid(Policy name), Effect (Allow/deny), Principal(who can access), Action([s3:GetObject]), Resource (what can be accessed – bucket arn)
- Controls access to your bucket and objects
- Cross account, public

Default data encryption

- In Rest
- Optional, for Glacier and Deep Archive required
- Enable data encryption in flight using SSE – S3, KMS
- In Flight – SSL/TLS or client side encryption
- If default data encryption not enabled but you still want to encrypt the data at rest, you have to send encryption with them and not allow anyone to post objects without encryption

S3 tags

- Key-value pairs applied to a bucket
- Useful for data lifecycle policies, cost tracking, automation
- Can be added and removed whenever

S3 Event Notification

- Configure yourself notifications (events) for events that happen in your bucket
- Created, deleted, replication – e.g. between regions, object lost
- Event destinations: SQS queue, SNS topic, LF
- Bucket-Properties-Notifications-select when LF should be triggered

S3 Prefix

- Whole or portion of the object key (path)
- Pass as query string ?prefix=2030/10
- Prefix can be used in bucket policies

ACLs

- Access Control Lists – something like bucket policies
- **Primarily used to grant permissions to public or other AWS accounts**
- Limit access to buckets and objects
- Depends who own the bucket and the object
- Typically bucket owner is the object owner – user Bucket/Object ACL
- Sometimes object owner differs from bucket owner – user Object ACL to access the object
- Allowing CORS must be also allowed

ACLs x bucket policies

- ACLs can't have conditions, cannot explicitly DENY access, grant permissions to other individual users except public/bucket owner/object owner

S3 storage classes

- Standard, Standard-IA(infrequently accessed), Onezone-IA, Intelligent Tiering, Glacier, Deep Archive
- **Trade-off between access time and cost**

S3 Lifecycle

- Generally data usage reduces with time
- Set up lifecycle to move objects between storage classes to save costs
- Actions: transition – move object to another storage class, expiration – delete object after X days
- Use **prefixes and tags**
- Usually not for free

S3 replication

- Objects or list of objects will be copied to other buckets within the same region or cross region or environments (dev to test)
- Cross account possible
- Versioning should be enabled on source and destination bucket
- Configured at bucket lvl, prefix, object(using tags)
- Only new objects are replicated automatically, for older has to be explicitly said
- Why? Reduced latency, regulations
- Replication takes time, min 15 mins
- For replicated object can change – storage class, object ownership (if you were owner of the original object)

Object level configuration

- Can override bucket configuration
- Storage class, encryption, Object ACLs

S3 Consistency

- Same data is stored in different AZs/regions, the change takes a while to be effective in each copy
- **Read after write for PUTs** – new object is created, it is guaranteed that the object will be available

- **Eventual Consistency for overwrites PUTS and DELETE** – you might get a previous version of data immediately after object update
- **Guarantees it will never return partial or inconsistent data**

Presigned URL = Query String Authentication

- Temporary URL that allows access to cloud resources
- Lasts up to 7 hours
- **URL contains: security credentials, bucket name, object key, expiration date and time (as query strings typically)**
- Created by AWS SDK API

S3 Access Points

- **Network endpoints** that you attach to a bucket to **perform object level operations** (you cannot modify or delete a bucket)
- Useful when you have multiple applications
- **Application specific access point on a bucket and attach it application specific policies**
- **Limit resources in a bucket each application can reach/access**
- More granular than to attach policies to a single bucket with conditions that it comes from a specific application
- Can't perform cross region replication through AP
- Up to 1000 APs/region – soft limit
- Access to AP from Internet or VPC
- No additional cost

Prevent object from deleting – S3 Object lock

Protect against accidental deletion - bucket versioning

Unauthorized bucket deletion – MFA, owner of the bucket

Enable cross domain requests to S3 hosted website over another website – enable CORS headers

S3 costs

- Storage costs, retrieval charges – Standard-IA, monthly fees – Intelligent Tiering, **data transfer fee**

Free data transfers

- Into S3 (buckets)
- From S3 to CloudFront
- S3 to services in the same region

Analyze storage access patterns and decide the right storage – Intelligent Tiering, Storage Class Analysis report – in Bucket – Mgmt

Reduce costs – Lifecycle managements and rules, expiration policies, proper storage class

Large files upload – **Multipart Upload API**, recommended > 100 MB, mandatory > 4GB, 1. Split into smaller portions, upload simultaneously, if fails you can restart just the portion. Pause and resume portion upload. Begin uploading before you know the final file size. -> use **Byte-Range Fetches: Get request + Range HTTP header: specify the portions used in Multipart Upload**

AWS resources using S3 bucket should be in the same region as the S3 bucket.

Requester pays – for data transfer and requests, storage is still paid by bucket owner

S3 inventory report – what is saved in the bucket, created daily

Change metadata, tags, ACLs or whatever in many bucket objects – generate inventory (S3 inventory report) + S3 Batch Operations – create what should be done based on the list of objects from inventory report (reference the report directly)

S3 Server Access Logs – default: off, enable it -> bucket and prefix where to send the logs

S3 Glacier

- Files – objects -> archives
- Containers – buckets -> vaults
- Archive keys are generated by service itself
- Only API and SDK and CLI access
- Once created can never be updated
- Archive limit 40gb
- Mandatory KMS S3 encryption – rest and transfer
- Bucket lock policy -> Vault lock policy
- Extremely cheap
- Longer retrieval times, minutes, hours, days
- 2 step retrieval – request, actual downloading, event notifications to SNS, SQS or
- Use HTTP RANGE HEADERS to retrieve only the necessary portion
- Glacier – expedited – 1 – 5 mins, standard: 3 – 5 hours, bulk – 5 – 12 hours
- Expedited – not always possible unless you activate provision capacity (paid)
- Deep Archive – standard – up to 12 hours, bulk – up to 48 hours
- Similar access settings as regular S3

IAM – Identity and Access Management

- Identities access (users, resources) cloud resources
- Authentication, Authorization
- AWS users, Federated users
- IAM users – credentials (user name/password, access keys)
- IAM groups
- Roles – temporary identities users or resources can assume, does not have credentials attached
- Policies – permissions – AWS managed, Customer managed (created by me, available to other resources), Inline policies (added directly to one specific role, user, group, are global available to anyone else), **permission versioning – up to 5, you select which one should be active**
- Policies – principal, effect –allow /deny, action, resource
- Don’t use ROOT user/account, instead create other IAM users/accounts
- Cross account access – mutually permitted - establish trust relationship. Account that want to enable access to its resources creates a role. This role has trust relationship that allows the other account, maybe even a specific user/group to access the resources. The other account has to allow users to access STS Service to assume a role. The user from the other account

uses account number of the first account and the role name it wants to assume. Tada, it works.

- Cloud resources (EC2 instance, DB) use Roles, assume role, Assume Role API call to AWS Security Token Service
- Instance profile – container for IAM Role for EC2 instance, in mgmt. console created automatically while creating a Role (EC2), using CLI and API you have to create it manually and map instance profile to a role
- Corporate Directory Federation + IAM – establish trust relationship, SAML protocol, for Microsoft AD – use AWS Directory Service to establish trust, otherwise set up custom proxy that translates user identities to IAM roles. I think you can actually just user AWS Directory Service. -> external user gives external directory it's credentials and gets a token back, uses this token to validate with AWS IAM using SAML Corporate Role API, if okay, gets IAM credentials back, uses those credentials to access the cloud resources.
- Web Identity Federation – use socialIDs to access cloud resources, OpenID protocol, Amazon Cognito supports connection with Identity Providers, or Amazon, FB, Google independently, **Create a Web Identity Role** as trusted entity.
- **Identity-based policies –** attached to IAM users, groups, roles | managed and inline | what resource? What action? | cross-account access – IAM roles | supported by all AWS services
- **Resource-based policies and ACLs –** Attached to a resource – S3 buckets, SQS queue, SNS, AWS KMS Keys | inline only | Who? What action? (resource given by where we create it) | cross-account access – no need to switch accounts
- **Access key rotation of an IAM user account** – max 2 active at a time, create one, use new one everywhere, disable original one, test and verify, delete original (can't be deleted immediately)
- IAM policies – deny by default
- IAM users are **global entities** (not specific to region)
- IAM users – created and managed in AWS, Federated users – managed outside AWS, Web identity federation users – Amazon Cognito, Amazon, FB, Google – OpenID Connect
- IAM root user should not be used on regular basis (lock credentials securely away), create an IAM admin user and operation everything over it
- Password specification and expiry date of it can be set for an IAM user
- Use MFA for all important IAM operations and extra security of accounts and actions

AWS KMS & Cloud HSM

- Generate and manage keys, perform encryption and decryption
- Data encryption protects data when someone unauthorized gets to it
- KMS – creates and stores KMS keys for encryption, some services can have a key generated in KMS but managed in the service itself
- Data at rest – data sitting somewhere – cloud, harddisk, USB, DB, file
- Data in transit – data transferred over a network – copy from on-premises to cloud – 2 types – over internet – usually a good idea to encrypt on client's side (Client Side Encryption) or within AWS – secure – no encryption needed
- Data in use – active data processed in a non-persistent state – data in RAM
- **Symmetric Key Encryption** – same one key for encryption and decryption, issue: if moving data from one place to another, key has to go with the data – dangerous if intercepted. Issue how to securely move the key to the other party. Okay for data in rest.
- **Asymmetric Key Encryption (Public key encryption)** – two keys – **private** (stash securely, chmod 400) and **public** (share with anyone), **Encrypt – public key, decrypt – private key**,

create asymmetric keys in KMS or Cloud HSM. 2 types, 1. Encryption/decryption, 2. Sign (private key) and verify (public key)

KMS

- Create and manage the keys
- **KMS perform encryption of very small pieces of data, < 4 kB!!!! -> Typically just to encrypt the plain data key!!!**
- **< 4 kB!!!!**
- **KMS doesn't usually perform encryption of larger files by itself, typically it's done in another AWS service – e.g. S3**
- Control use in your App or AWS Services (e.g. S3)
- Define key usage permissions = **key policies - cross-account possible**
- **Track key usage** – CloudTrail
- Integrates with any AWS service needing encryption
- **Automatically rotates CMK** (Customer Master Key) once a year
- **Schedule key deletion** – to verify it's not user anywhere, minimum wait 7 days, max 30, you can cancel it, if you notice the key is used somewhere, key can be disabled though
- **SSE with KMS – Encrypt an object** – 1. S3 gets a new object, 2. S3 gets plain data key from KMS (create CMS that only stays in KMS, CMS created plain data key, KMS encrypts the created plain data key -> encrypted data key), 3. S3 encrypts the object (object + plain data key + encryption algorithm), 4. Stores encrypted data and encrypted plain data key that corresponds to this one.
- **SSE with KMS – decrypt an object** - 1. S3 sends encrypted plain data key to KMS, 2. KMS decrypts it using CMS and returns plain data key to S3, 3. S3 uses the plain data key to decrypt the object data
- Tip: remove plain text data key from RAM asap, AWS services need permissions to access CMK in KMS
- Encryption context – additional layer, key-value pair, value is used for encryption and need to be supplied for decryption as well
- **Envelope Encryption** – **encryption used in KMS** – plain text master key encrypts plain text data key that encrypts the data | 1. Data is encrypted by plain text data key, 2. Data key is encrypted by Master Key, 3. Master Key never leaves KMS
- **!!! Envelope Encryption for larger objects > 4kB!!!** , typically encrypted in an AWS service – GenerateDataKey from KMS, use the key in SSE or CSE
- **Customer Master Key (CMK)** – used for encryption, decryption and signing | **created in KMS** | 3 types: **Customer managed** (owned and managed by customer = stored at a customer or in the app – you or another IAM user, available for your AWS account only), **AWS managed** (managed by AWS on your behalf, available for your AWS account only), **AWS owned** (owned and managed by AWS, available for multiple AWS accounts, **LIMITED usecases**)
- S3, DynamoDB, EBS, SQS SNS – Customer managed or AWS managed
- DAX, CodeCommit – only AWS managed
- ReEncrypt API – called when creating a new CMK and removing the old one. Object gets decrypted using the old CMK and encrypted using the new CMK.
- KMS API quota – 5500 – 30000 reqs/sec., varies per region
- S3 calls KMS APIs to encrypt a file, GenerateDataKey/Decrypt, set up CMK key policy (each key must have exactly one key policy) allowing S3 bucket to access the CMK. IAM policy on the S3 bucket should allow API call

- **IAM policies are global, key policies are regional** -> key policy only effective in region where the CMK is created
- **CloudTrail** – track usage of CMKs
- **Encryption SDK** – use in your app to encrypt data, supports **Data Key Caching – reuse the same plain text data key and encrypted data key – doesn't generate a new one for each object you want to encrypt, reduces No. of API calls**

KMS and S3

- **SSE-C**: If you want to manage keys outside of AWS. Customer stores the keys somewhere and supply them to S3, encryption in S3. Customer sends the key with every request (S3 doesn't store the key), HTTPS mandatory – because user sends the key with the data
- **SSE-KMS**: keys managed in AWS KMS, CMK stored in KMS. You (the customer) manage the keys in KMS. CMK generates plain text data key and encrypted data key and sends it to S3. Encryption in S3. Request headers: x-amz-server-side-encryption(aws:kms), x-amz-server-side-encryption-aws-kms-"key-id-you-want-to-use"
- **SSE-S3**: Keys are managed in AWS S3. In the background it still somehow works with AWS KMS but you don't take care about it. No management on your part. Keys are rotated every month automatically. Request header x-amz-server-side-encryption(AWS256)
- **CSE** (Amazon S3 encryption client) – CMK stored within your app (client side). **Encryption client** talks to local storage or where the key is stored locally or in the app and uses it to encrypt the data that you want to send to S3.
- **CSE** (Amazon S3 encryption client) – CMK stored in KMS. **Encryption client** talks to KMS to get the CMK and uses the key to encrypt the data that you want to send to S3.

Server Side Encryption

- Client sends unencrypted data to AWS server, server service (S3), S3 talks to KMS and perform encryption
- Use HTTP**S** endpoints to ensure sent data are securely encrypted, all AWS services have HTTPS endpoint

Client Side Encryption

- Customer handles all the encryption by himself
- Sends encrypted data to AWS, e.g. S3 and the data are stored as they are received
- To encrypt data for S3 you can make use of a client library **Amazon S3 Encryption Client**

KMS and CW

- Use KMS to encrypt CW logs
- Permissions on your account/user to use KMS keys – the API to get a key and encrypt – kms:CreateKey, kms:GetKeyPolicy, kms:PutKeyPolicy
- Key policy to allow use of CMK from CW – set on a key in KMS

AWS CloudHSM

- It enables you to easily generate and use your own encryption keys on the AWS Cloud.
- Like KMS but only YOU (your account/user) can access the CMK in HMS. Once you lose the key, neither YOU nor AWS can access it.
- **TIP: Store copies of the same CMK in HSMs in multiple AZs**
- Scales

- SDK integrates with industry standard APIs - PKCS#11 or Libraries
- HSM = Hardware Security Module
- Security compliance – FIPS 140-2 lvl 3
- **Single-tenant** vs. KMS multitenant
- **Integrates with KMS – encryption, CloudHSM cluster– keys**
- CW – monitoring
- CT – track key usage
- **Offload SSL processing - Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are used to confirm the identity of web servers and establish secure HTTPS connections over the Internet. You can use AWS CloudHSM to offload SSL/TLS processing for your web servers.**
- **Certificate Authority (CA) - In a public key infrastructure (PKI), a certificate authority (CA) is a trusted entity that issues digital certificates. These digital certificates are used to identify a person or organization. You can use AWS CloudHSM to store your private keys and sign certificate requests so that you can securely act as an issuing CA to issue certificates for your organization.**
- **Digital Rights Management – signing, verification**
- **TDE (Transparent Data Encryption) for Oracle DBs**

Networking

- VPC – private network, you can't connect to some resources directly over the internet, e.g. a database. You should create an app in the same private network that can be access through the internet and can talk to other resources in the private network.
- **VPC** – isolated network in AWS cloud, from internet and all other VP networks
- **VPC** – created in a **Region**
- You control all the traffic
- **Best practice: Create all your AWS resources within a VPC so they can securely communicate and are protected from unauthorized access**
- **VPC Subnets** – spaces within a single VPC – typically for public (ELB) and private resource
- **VPC Subnets** – created in an AZ
- Resources within **public subnet** can be access from the internet | communication allowed from subnet to internet and internet to subnet
- Resources within **private subnet** cannot | communication allowed only from subnet to internet (download patches, updates, …)
- **Resources in public subnet can talk to resources in private subnet**
- Routers in Internet – forward your request to reach desired IP in as little steps as possible, set of rules
- Routing inside AWS – **Route Tables** (associated with VPCs an subnets) – they have sets of rules (**destination and target**) | What range of addresses (destination) should be routed to which resource (target) | **Rule at top has higher priority**
- **Internet Gateway** – enables internet communication to subnets **-> public subnets – have it, private don't** | uses Elastic Network Interface (ENI) – cloud network card, and private IP
- NAT (Network Address Translation) Gateway and Instance – allows private subnet resources download something from internet and connect to other AWS services outside of their VPC -> **NAT Gateway** – easier setup, **NAT Instance** – EC2 with NAT AMI configured as a Gateway – complete control, **Egress-Only Internet Gateways** – for IPv6 subnets -> Still need IGW to talk to internet

- Network Access Control List (NACL) – security groups control traffic to a specific resource of the subnet BUT **NACL stops traffic from even entering the subnet** | allow traffic from these IP addresses to this specific EC2 instance on this port | by default all allowed, usually we set everything to deny and they allow rules, priority number – lower -> more important. Stateless compared to security groups | **each subnet has minimally one**
- **VPC Flow Logs – monitor network traffic in and out of VPC and/or Subnet** | Accept or Reject traffic | logs in CW logs or S3 | testing NACL
- **VPC Peering** – allow communication between resources in different VPCs that are typically private and you don't want to make them public just for the sake of talking to each other | cross account possible | peering **request – accept / reject () week**
- Connect AWS and On-Premises: **AWS Managed VPN** – VPN Tunnels – encrypted IPsec protocol from VPC to customer (**VPN Gateway in VPC** and Customer Gateway On-Premises), AWS Direct Connect (DX) – private (=not over internet) dedicated network connection from client to AWS, expensive, takes long > month
- **VPC Endpoint** – Enable connection between your VPC and other AWS services outside your VPC | e.g. EC2 and S3. EC2 goes through IGW over Internet to S3 and retrieves it the same way. In order to have a **secure connection within AWS**, use service VPC Endpoint | IGW, NAT, Direct Connect or VPN are **NOT** required | 2 types | **Gateway Endpoint** – **S3, DynamoDB**, **uses Routing Table**, create a VPC endpoint within your VPC and set VPC endpoint as a target in your VPC routing table | **Interface Endpoint** – **all other AWS services**, **uses ENI+ENI private IP address**, create an endpoint in your VPC using ENI – **Elastic Network Interface and ENI private IP address**, we call this ENI private IP address that gets in touch with the service, powered by **PowerLink**

Databases

- RTO – Recovery Time Objective – Max acceptable downtime of DB
- RPO – Recovery Point Objective – Max acceptable period of data loss
- Snapshots – copy of the DBs, e.g. every hour, at the time of snapshot the database is slower
- Transaction Logs – storage that copies the changes of the DB since the last snapshot, prevents data loss and lowers RPO
- StandBy DB – **Synchronous** replication of data from the master DB to standby DB, you make snapshot from it so it won't slow down the master DB | When master DB deleted, standBys are deleted as well | **you can make StandBy a master – if master fails immediate failover to standby** | **Increases availability not scalability (for scalability use Read replicas) | uses DNS not IP Addresses of the servers where DBs run. When switching Master to StandBy, it just switches the IP but DNS stays the same.**
- Read Replicas – read-heavy databases, use **asynchronous** replication of data from the master DB | **can be upgraded to master DB** | When master DB deleted, read replicas are not, you have to remove them manually | **take snapshots** from them | read replicas of read replicas
- Synchronous – strong consistency | master slow if you have many standBys and read replicas
- Asynchronous – eventual consistency – record is distributed with a delay, most up to date data in master DB | Shorter time of eventual consistency -> Vertical scaling | **used when scalability is more important than data integrity**
- Read-after-Write consistency – inserts immediately available, updates and deletes are eventually consistent
- Availability – 4 9's – downtime a month – 4.5 min
- Durability – 11 9's – protects against data loss, once data is lost, it can never be recovered

- Hot StandBy vs Warm StandBy – Hot takes less time to boot up and be ready to take over the load (**lower RTO – 5mins vs 15 mins**), is more expensive, **offers automatic switchover** whereas Warm will scale up when master fails to , both synchronize data (**low RPO < 1 min**) between master DB and themselves. If I can **tolerate even longer RTO** (hours), go for **snapshots + transaction logs**. You will use the last snapshot to create a DB + add the logs. **If data can be lost**, e.g. cached data or not processed data, start a completely new server that can take over.
- Reporting and analytics Apps typically just read data from DB, don't use master, go for **read replicas**

Relational Databases

- Predefined schema, structured tables, tables and relations between them
- OLTP = Online Transaction Processing
- OLAP = Online Analytics Processing
- **BETTER DATA CONSISTENCY THAN ANY OTHER DB TYPE**

OLTP

- Small records = small size data but large volume
- E.g. Banking, most applications, CRM
- Popular DBs: MySQL, Oracle, SQL Server
- **AWS RDS** – implements with **Amazon Aurora** – based on PostgreSQL & MySQL, PostgreSQL, MySQL, MariaDB, OracleDB, MSSQL
- **Row storage**
- Efficient for small transactions = records

OLAP

- Big data – analyze petabytes of data
- **Column storage**
- E.g. Reporting app, data warehouses, BI app, Analytics systems
- **AWS Redshift**
- High compression of data of the same sort – e.g. billions of people countries – compressed till the same 179 countries, columns of one table can be stored in different cluster nodes, more efficient complex queries across multiple nodes

Document DB

- **DynamoDB**
- No solid table structure = schemaless
- **Very fast and scalable – ms responses for millions of transaction per second -> high volume read-write DB**
- **Lower consistency than RDS - > eventual consistency**
- **One table instead of many**
- **Data can keep changing structure**

NOTE: Data consistency and scalability don't go together well. Data consistency – RDS, scalability - DynamoDB

Key-Value pairs

- **DynamoDB**

- **Scalable and fast – ms responses**
- **E.g.** shopping cart: user – cart content, gaming: user: score, sessions stores

Graph DBs

- Graphical representation of data and its relationships among themselves
- Data with complex relationships
- E.g. social networking – FB, twitter, fraud detection
- **Amazon Neptune**

In-Memory DBs

- RAM – Cache
- Super fast – us responses
- Can be used as a cache in front of a DB
- **Redis** – persistent data
- **Memcached** – simple caches
- **Amazon ElastiCache** – as a cache for RDS

Amazon RDS

- Managed service for your DB
- Supports all commercial DBs + their own **Amazon Aurora**
- Multi-AZ deployment – master in one AZ, standby in another | synchronous replication except for Aurora where asynchronous | snapshops from Standbys | **maintenance easy** – maintenance on StandBy, promote StandyBy to master, maintenance on old master, switch master back | app can access only master, for Aurora – master and all StandBys
- Add Read replicas – up to 5 for commercial, 15 for Aurora | Same AZ, Multi AZs, Cross Region | app can access all read replicas
- Storage autoscaling
- **Manual snapshots**
- **Automatic backups**
- **Automatically installs latest OS and DB patches**
- **You take care of the data put in it, table structures, users permissions**
- Vertical Scaling – change DB instance type, scale storage | changes are done during maintenance window or “applied immediately” | manually scale up to 64 TB, 128 TB for Aurora, MS SQL Server – 16 TB | **autoscaling possible**
- Horizontal Scaling – Read Replicas (StandBys won’t do it), for Aurora – multi master, single master + read replicas, global option
- CW – shows historic data that were saved
- CW alarms – reaching near max capacity – storage, CPU | Enhance Monitoring – to discover what causes performance issues for SLOW queries
- Automatic DB backups - go to S3, stay for 7 to 35 days then removed | backup windows – 30 mins selected by you or RDS itself
- RPO 5 min – data loss not more than 5 minutes
- Security – VPC Endpoint, security groups, IAM Authentication (user -> check permissions -> token or use IAM Role and no credentials are required) for Aurora, MySQL, PostgreSQL, encryption with KMS, when encryption enabled – master, SBs, RRs are all encrypted, Data in-flight protected by SSL certificates – select at DB creation

- Predefined schema, strong transactional capability, complex queries | not scalable as DynamoDB, upload using Get Rest API – S3 or DynamoDB, no heavy customization of your DB or need access to EC2 instance

Amazon Aurora

- MySQL, PostgreSQL compatible
- **Uses Cluster Volume = Multi AZ storage => 2 copies of data in a minimum of 3 AZs**
- Data replication works differently here (Typically: Master DB to StandBys and Read replicas): MasterDB to Cluster Volume, StandBys and read replicas read the data from CV
- Up to 15 read replicas
- **Global Database** – allows globally read data with low latency, minimal lag time, up to 5 secondary regions (those regions allow only reads)
- **Deployment Options**: **Single master** (One writer – master DB, multiple readers – StandBys, read replicas | if writer fails, readers can be promoted to writer), **Multimaster** (multiple writers), **Serverless** (storage available based on demand)

Billing

- DB operating time, storage – GB, IOPS, storage, backup and snapshot storage, data transfer – cross region has a fee attached

Full Database access -> custom database installation – EC2 + EBS, install it there

AWS Database Migration Service – migrate data from on-premises to cloud (same DB type – e.g. PostgreSQL to PostgreSQL)

AWS Schema Conversion Tool - migrate data from on-premises to cloud + **conversion** (different DB type – e.g. MSSQL to Aurora)

Deleting RDS – auto backups are deleted, **snapshots stay**, before deleting – prompted -last final snapshot

Reduce global latency and improve disaster recovery – **Multi Region RR, Aurora Global Option**

Subnet groups – select subnets to which RDS DB should be launch into

Adding encryption to an unencrypted DB - !!! No, not immediately enable | 1. Create DB snapshot, 2. Encrypt the snapshot using KMS keys, 3. Create DB from the encrypted snapshot | or allow encryption at the DB creation

RDS for at least a year -> **reserved instances, 1 or 3 yrs**

Stop DB – billing -> you pay for storage – data, snapshots, … but not operating time

Amazon RDS Proxy – manage DB connection efficiently – better user pool management for multiple apps connecting to RDS, more secure, faster RPOs – recovery times

Amazon DynamoDB

- NoSQL Key-Value & document DB (schemaless)
- Scalable – millions of TPS (transactions), ms responses
- Automatic partition of data
- **3 replicas within the same region(s)**
- **1 table**

- **RCU, WCU (max 10,000 RCU or WCU per table, soft limit) or serverless**
- Table –> item(s) -> attributes
- **Max 400 kB per item**
- **tables are region specific**, if users from across the globe – mark table as **Global table**
- Data types – **scalar** – string, number, Binary, **document** – list, map, **set** – string or number or binary set
- **Primary key - simple** (partition key = hash function) or **composite** (partition key + sort key = hash + range)| **Set at table creation, can't be changed** | **high cardinality** – tip: add random uuid | hash function used to divide data into partition | **for Time Series Data create a new table for each period – 1 week, 1 month, 1 day, … -> recent data are frequently access, as table get older – remove WCUs and reduce RCU to working minimum**
- Same partition keys are stored in the same partition ("tables") and sorted using sort key
- max 3000 RCU and 1000 WCU per partition -> make sure you pick your partition key well
- **LSI = Local Secondary Indexes – same partition key! + different sort key**, **up to 5 per table (hard limit), defined at table creation and can't be changed**
- **GSI = Global Secondary Indexes – partition key and sort key can be different! Can be added, modified or removed later, up to 20 per table (soft limit), RCU and WCU stored separately from the table (minimum the same RCU and WCU as the table has to avoid throttling), project fewer attributes (return less attributes)**
- **Eventual consistency - default** and strong consistency – set consistentRead (API) to true possible – strong consistency requires at least double the RCU compared to eventual, not available for GSIs
- Read/Write Capacity Units
- Provisioned – bought RCUs/WCUs, manual dynamic adjustments
- On Demand = serverless – good for unknown loads, serverless is expensive
- 1 RCU = 2 eventually consistent reads of 4kB, 1 strong consistent read of 4kb
- 2 RCU = 1 transactionals read of 4kB
- 1WCU = 1 Standard Write of 1kB
- 2WCU = 1 Transactional Write of 1kB
- Scan – reads every item in the table, by default returns all attributes – use ProjectExpression, Parallel Scan – divides table into segments, for tables > 20 GB recommended, **faster, requires more RCUs**, **useful when not all RCUs are used**
- Query – uses partition key and its value, optional: sort key and filters
- When name of **attribute (=column)** is a reserved word use **a hash**, e.g. #subCol **-> expression attribute name**, e.g. {"subCol": "desc"}, **for value us a semicolon**, e.g. :subVal -> **expression attribute value** {":subVal": {"S": "real value here"}}
- **Pagination** – on a single request you can retrieve certain amount of records, when not all retrieved it sends back the last processed item and a token. You will use these information in the next request – either number of items, e.g. 1000 items or size 1kB I think
- Limit number of returned items, --max-items
- --starting-token, where to continue pagination
- **--page-size – prevents AWS service call from timing out, smaller page size prevents sudden spikes**
- BatchGetItem API call, BatchWriteItem API call – reduces or eliminate network round trips, returns successful when at least one items is received/written. Get – returns unprocessedKey(s) if any, Write – returns unprocessedItems if any
- **Conditional expression** – update/delete when condition true, good for checking consistency, e.g. if deleted object is actually deleted

- API Status codes – 200 – OK, 400 – authentication failure, throttling – check RCU/WCU, 500 – AWS problem
- **TTL** – expire/delete data after some time, saved data have an attribute with a **timestamp of expiration**, batch kind of thing run in DynamoDB, When timestamp older than current -> data expired, time between expiration and actual deletion can be specified – e.g. 20 days, **doesn't consume WCU/RCU**
- **Optimistic locking** – to avoid accidental over writing of an item by another update, **use version attribute**, only update the item if the version matches to intended version selected by you and increase version by one, if no, throw an error
- **Best practices**: understand access patterns and select the best primary key and LSI and GSI, avoid scans, rather use queries, **if scanning -> for relevant or frequently accessed items use a new table where you duplicate them and scan them there or use caching** | use conditions in IAM Policies to restrict access to DynamoDB, e.g. partition key matches userID retrieved by Cognito | **Global Tables need DynamoDB streams | filter is applied after Query/Scan doesn't lower RCU**
- Access DynamoDB: Mgmt Console, CLI, SDK, **NoSQL Workbench** for DynamoDB – install on your local machine
- Monitoring – CW
- RCU/WCU alarms – CW alarms, set them up in the table itself
- RDS or MongoDB to DynamoDB – **AWS Database Migration Service**
- Enable point in time recovery – at table creation, max 35 days
- Encryption – **AWS owned keys -** AWS KMS key **owned and managed by DynamoDB** – **no fee**, **AWS managed** – key stored in your account and managed by KMS – **fee**, **Customer managed key** - owned and managed by you, stored in KMS in your acc – **fee**
- SSE – **always enabled – tables, streams, backups**
- CSE – you manage keys in KMS or CloudHSM, use DynamoDB Encryption Client – write your own code and use your keys (Customer managed keys)
- **DynamoDB does NOT support resource based policies (unlike S3) -> other services – EC2 Instance or LF should have roles that have permissions to access DynamoDB**
- **Main table can be throttled even when writes using GSIs**
- **Not making use of all queried attributes – projectedExpressions, for frequent – GSI + projected attributes only**
- **Deleting all data from table – 1. Backup data, 2. Drop the table, 3. Create a new one, 4. Upload backed up data**
- **Storing large images – store in S3, save reference**

AWS owned keys

- AWS owned keys are a collection of KMS keys that an AWS service owns and manages for use in multiple AWS accounts. Although AWS owned keys are not in your AWS account, an AWS service can use the associated AWS owned keys to protect the resources in your account.

You do not need to create or manage the AWS owned keys. However, you cannot view, use, track, or audit them.

AWS managed keys

- AWS managed keys are KMS keys in your account created, managed, and used on your behalf by an AWS service integrated with AWS KMS

Customer managed keys

- The KMS keys that you create are customer managed keys. Customer managed keys are KMS keys in your AWS account that you create, own, and managed

DynamoDB Streams

- **Streams capture time ordered sequence of data modifications in near real time**
- **Create, update, delete**
- **What is captured: keys only – partition and sort key, new image – new item, old image – old item, new and old images – both**
- NOTE: DynamoDB and DynamoDB stream have different endpoints
- **1 modification = 1 record**
- **Stream records are organized into SHARDS – ephemeral – created and deleted automatically**
- If you close stream, it shuts down all the shards
- Create lambda that will be triggered when dynamoDB table stream has a new record/records. Give lambda a role and permission to DynamoDBExecutionRole

DynamoDB	RDS
DynamoDB Accelerator (DAX)	Typically ElastiCache, MemCached
No upper limit scaling	64 TB
Difficult to run complex queries over more than one table	SQL, can run complex queries over multiple tables/entities

DynamoDB Accelerator (DAX)

- In-memory cache for DynamoDB
- µs responses
- write through – cache and database update simultaneously
- for hot items
- saves RCU
- **NOT for** strongly consistent reads, write intensive apps with few reads

Decoupling Applications with SQS, SNS and MQ

- Decoupling – allowing asynchronous communication
- Synchronous – step by step, if one link of the chain goes down, it affects functionality of the whole app | if one services experiences high load, it may go down and requests won't be processed and will be lost -> **in case of a problem in a single place, the whole app suffers** -> to prevent this, **decouple services** from each other -> **make it asynchronous** - uses **queues** to store task -> if there's a failure in request processing, you can still "store" the requests in a queue
- Queues or topics
- Allows scaling out processing services and take requests from the queue

SQS

- Pull Model – Producers and Consumers
- producers put messages on the queue, consumers poll on the queue – long polling useful, polls for 20 seconds at a time and not every minute
- **Queue and consumers have to be in the same region?**
- Unlimited scaling

- Pay for use – low cost
- **Standard Queue** – **unlimited throughput** (messages/second), Best-effort ordering – but order of events is not guaranteed, message processed at least once guaranteed (can be more than once)
- **FIFO** – **Order of events guaranteed** but lower throughput (300 message operations per second, if you use batch 10 (10 is max), then grows up to 3000), processed exactly once
- Message placed has assigned a globally unique messageID
- Consumer polls a message – **gets a message** (message body, messageId, message attributes, MD5 digest) **and a receipt handle** – information of a message reception
- Message stays in queue but cannot be polled by any other consumer
- Consumer processes the message – message removed from the queue using the receipt handle
- **delay or no delay** - Message placed on queue – **visible** – customer polls, **visibility timeout** – time to process the message, if it fails it is again available on the queue – either processed **or** put back on queue – **message retention period** – max time a message can be in a queue before it gets deleted or **retry policy** – attempts allowed for message processing, if it fails, move message to DLQ ideally or delete it
- Autoscaling – CW alarm – target tracking scaling policy – select reasonable SQS metric like ApproximeNumberOfMessages -> scales consumers accordingly to message demands
- **visibility timeout** – def. 30s, 0s -12 hours**,** consumers can **call ChangeMessageVisibility API** to increase visibility timeout of the processed message
- **DelaySeconds –** def. 0s, max 15 mins, set at time of queue creation or by SetQueueAttributes API, message specific delay can be set by **message timers – DelaySeconds**
- **Message retention period** – def. 4 days, 60s – 14 days
- **MaxReceiveCount** – max amount of attempts to process a message before it's sent to DLQ
- Security: producers and consumers should have IAM permissions or IAM roles with necessary permissions to place/process the message from the queue | **by default only queue owner can use the queue** | Cross-account possible – additional: configure SQS Queue Access Policy (resource policy) on the queue
- **IAM Role – Trust Policy** -> who or which service can assume the role, to assume a role -> 1. IAM User needs permission to AssumeRole, 2. IAM Role (being assumed) should have a trust policy allowing the IAM User, trust policy is required for services as well.
- **IAM Roles** have **trust policies** allowing users and services who can assume them **and policies** what you can do with the role
- Message deduplication – **message has a body, content and attributes** -> similar messages treated as unique – use **message deduplication ID,** 2 approaches – create message dedup. ID from message **content (typically the message itself contains a uuid)** or call **MessageDeduplicationID** Api every time you send a new message on a queue
- **MaxNumberOfMessages** - Receive 1 – 10 msgs from the queue
- **WaitTimeSeconds** – Enables **long polling** – always go for **20s** (max) – reduces number of api calls to SQS, reduces number of empty responses, receive a msg immediately as it arrives to the queue
- Consumer takes longer than visibilityTimeout to process the message -> it becomes visible on the queue again and can be poll by another consumer | use ChangeMessageVisibility API call to extend the visibility period
- Problem processing messages -> DLQ
- Send a large message (1GB) -> **Amazon SQS Extended Client Library – up to 2 GB, messages are stored in S3**

- Give **higher priority to premium consumers** -> separate queues for free and premium customers and to premium queue set more consumer instances -> faster processing

SNS

- Push Model – **Publishers – Subscribers** – works like an email
- Publishers send notification request to a topic, subscribers are subscribed to the topic and receive a notification
- 1. Create SNS Topic, 2. Publisher publishes event/notification request to the topic, 3. Event notification is broadcast to all subscribers (to the given topic)
- Use cases: workflow systems like Jira, mobile apps, monitoring apps
- **Mobile and enterprise messaging web services – 1. Push notification** to Apple, Android, FireOS, Windows devices**, 2. Send SMS to mobile users, 3. Send Emails – notification emails only (for larger amounts of emails, use Amazon SES simple email service)**
- Basic needs – SNS standalone, for complex – use SQS queue as a topic subscriber
- **Cross account access to the SNS – configure AWS SNS generated policy on the specific SNS topic**

MQ

- Managed message broker service for Apache ActiveMQ which is a queuing Tool used across a lot of different enterpreises
- Amazon MQ = SQS (queue) + SNS (topic)
- Restricted scalability compared to SQS and SNS
- **use Amazon MQ to migrate on-premises message brokers to cloud broker without making code change**
- **Start with MQ and then switch to SQS and SNS**

Data Streams

- **Continuously generated small pieces of data mostly associated with time**
- E.g. page views, link clicks
- **Services record changes done to their resources and generate information about it in a real-time. This information may be processed by other services such as AWS Kinesis or use them to trigger a LF.**
- **Data stream in S3**: **S3 Notifications** – create, delete, and update an S3 object. S3 Notification (-> SNS or SQS) -> LF
- **DynamoDB Streams** – DynamoDB Table records all changes done to the documents (**has to be enabled on the Table**) -> generates streams about it. DynamoDB Streams can process these streams immediately or keep them up to 24 hours and process them in batches (iteration through these processed/unprocessed streams for up to 24 hours again). DynamoDB Table Event -> DynamoDB Streams -> LF

Amazon Kinesis

- Primary Data Stream handling tool
- NOT recommended for ETL – Extract, transform, load – data extracted from source, transformed according to need in further processing and loaded into a single save destination
- **Kinesis Data Steams – process data streams**

- **Kinesis Firehouse – used for data ingestion of streaming data** (ingestion = data is transported from source – data stream to a storage medium) - S3, ElasticSearch, Redshift. **Basic processing before storing possible – transform, compress, encrypt**
- **Kinesis Analytics – performing real time analytics of streaming data – using queries**
- **Kinesis Video Streams – Video stream monitoring (image-data stream processing), typically drawing conclusion/intelligence in real time teaming up with ML tools.**
- **Min shard count** – based on throughput in and out, **min shards = max(bandwidth in / 1024, bandwidth out / 2048)**
- **Producers – put items in the stream, Consumers – process the items from the stream**
- **Producers set a primary key and set it along with the put item – high cardinality so no partition become a hot partition – this causes increase in latency**

Kinesis Data Streams

- Real time data stream processing
- **Producers – clients, devices, services generating streams -> Kinesis Data Streams – stream name and shards -> consumers – (underlying Kinesis Stream "consumer" apps – EC2 instances that process the data streams) process the events**
- Multiple streams possible to handle the load
- Alternative to Kafka
- Support multiple clients – each client can track their stream position -> **each client consumes the data stream and run its analysis on it (unlike SNS/SQS), consumers consume the event in a given order and each client know it.**
- **Retain and replay data** – def. 1 d, max 1 y, if data not deleted, you can make changes to a LF and reran the same data stream event, useful when there was something wrong in the LF
- **Application integration: make your app generate data streams and send them to an actual data stream in Kinesis Data Streams - toolkits** such as AWS SDK, Mobile SDK, Kinesis Agent, Kinesis Producer Library **or service integrations and** logs – AWS IOT, CW Events and CW Logs
- **Develop producer Apps for Kinesis DS – Kinesis Producer Library (KPL), Kinesis Agent** – prebuild java app
- **Develop consumer Apps for Kinesis DS – Kinesis Client Library (KCL) – Python, java, node.js, .NET**
- On-demand (auto scaling) or Provisioned (Shards)
- Shards – give us write and read capacity, 1 – 200, max per acc. 200
- Data Stream > Shards > Data Records
- **Partition key** (given by produces), Kinesis uses it to distribute the stream among the shards
- **Record ordering within a shard is guaranteed** – unique sequence number given at the moment of putting the record in the shard
- **Stream Data includes partition key, unique sequence number and data blob (actually data sent from producers)**
- Each shard **max 1000 records/sec or 1MB/s** whatever comes first
- **Increase stream throughput -> increase number of shards**
- **Resharding –** adapt shards any time using API, low level operations – splitting or merging, upper level operation – update-shard-count – still uses low level ops
- **BEST PRACTICE: Max No. of consumer instances = No. of shards (at one time only one consumer can read from a shard)**
- APIs: put-record, put-records(better throughput), register-stream-consumer, split-shard, merge-shard, update-shard-count, list-shard

Provisioned throughput exception – frequently -> reshard, infrequently – exponential backoff
Records not evenly distributed across the shards -> optimize the partition key

Kinesis Data Firehose

- Data ingestion from streaming data
- Receive DS, Process (transform – LF, compress, encrypt), store DS (S3, RedShift, ElasticSearch)
- Kinesis Data Streams -> Kinesis Data Firehose -> Consumers (places to store data)
- Further analytics (understand SQL queries) can be done from S3 using **Athena**, Redshift – normal, ElasticSearch – normal

Kinesis Analytics

- Kinesis Firehose/Data Streams -> Kinesis Analytics
- Real time analytics using direct DS or run queries against stored DS
- Create your own apps

Kinesis Video Streams

- Real time video -> Kinesis Video Streams -> AWS AI Services -> Real time alerting
- Get intelligence from pictures -> car crossed the street, the traffic light was red -> driver should be fined
- Integrate ML

Routing and Content Delivery

- **Content Delivery Network (CDN)** – content delivered to global audience with low latency using more than 200 edge locations
- You can publish content to these edge locations to lower the latency even more
- Edge locations offer **caching** – set **TTL**

AWS CloudFront - serve content directly from edge locations | **Content Distribution System**

- Can distribute source content from S3, EC2, ELB, External Websites
- If content not available in edge location, it is returned from origin and cached
- Allows authentication and authorization
- **Use cases:** static web apps, audio, video, software downloads, dynamic web apps, supports media streaming
- Integrates with AWS Shield (DDoS attacks), AWS WAF (SQL injection, cross-site scripting)
- **Cost**: S3 <-> CF FREE
- **CloudFront Distribution = DNS domain + origins=content source (S3, EC2, ELB, External Websites) + Cache-Control (TTL, def. 24h, file level customization – cache control max age and expires headers)**
- **Caching** – **static content** – for a long time, **dynamic** – short time because it keeps changing
- Supports HTTP and HTTPS by default, HTTP can be redirected to HTTPS in CF
- **Web app has static and dynamic content – NOT** recommended to store all of it in EC2 Inst**, store static content in S3 and distribute it using CF, dynamic content will be served from an EC2 Instance**
- **Caching distribution** – can be different for **path pattern** (/photos) – wild cards (*) + path**,** query strings added to path pattern, use only https to access some resources on the path, TTL for the specified path

- **Private content in CF** – user – CF – content in S3 | content shouldn't be directly accessible | implement authentication in CF (**reject requests without any form of authentication**) – **signed URL (**usually for video stream or single downloads, client talks to app, app talks to CF and creates a signed URL using the client's credentials to the app, the app returns the signed URL back to the client, the client retrieves the S3 content through CF**), cookie key pair (**similar to signed URL, typically one cookie key pair for multiple files, instead of signed URL returns a cookie, the client uses it to retrieve the files, **only for HTTP/HTTPS apps available), Origin Access Identities (**prevents direct access to content – **S3 bucket policy,** account to retrieve the content -> **create a special CF user,** associate OAI with CF distribution**, create S3 bucket policy to allow only OAI)**
- Signed URL and cookie pair typically **for premium content**
- OAI typically used when you're offering a **lot of free content** but you want to make sure **all the requests go through CF ->** Benefit: users will **use CF caching** to retrieve all this free content, fast, at the same time very little load on the backend
- Tips: **Cache invalidation** – remove a specific object from all edge locations, **use versioning in object path name** – e.g. /profile125?version=1 -> prevents the need to invalidate cache, **Don't use** when request coming **from single location or corporate VPN** (sort of single location), **CF Geo – you can limit certain areas or countries to access CDN**

Route 53

- **Domain registration and DNS routing** (step 1 and 3) can be done through Route 53 (AWS service)
- **Routing done in DNS servers (Route 53** or external**) using DNS records (mappings) – mapping your website to servers where your website content is stored**
- Set up web page: 1. Buy a **domain** (URL) (domain registrar – vojta.com, **Buy domain in AWS or anywhere else**), 2. Setup your **website content – content that you want to delivery when someone connects to my domain** (website hosting), 3. **Route domain (URL) to my website content**, this is **done by DNS**- requests to my domain to my DNS (hosting web server that has all the content)
- **!!!** 1. Register a domain name, 2. Create a hosted zone in Route 53 and create DNS records – mappings/routings Name Servers to AWS resources (servers with web content – EC2, S3, CF), 3. Delegation – update registrar (where you sign your domain) with your correct Name Servers in Route 53 **!!!**
- **DNS = Domain Name System/Server** – maps domain to IP address of website content host, configure records – map domain to a server, api to api server, static content to http server, mail to mail server -> for this we need a **Hosted Zone = container of routing records (mappings),** private – routing in VPC, public – routing over Internet
- **Standard DNS records (mappings)** – **A** – maps domain name to IPv4 address (route vojta.com to web server with web content at 192.198.80.60), **AAAA** – maps domain name to IPv6, **most important NS (=Name Server)** – **4 servers automatically assigned to your hosted zone, root domain routes the traffic by default to those server (so called delegation set)**, **MX (=Mail Exchange)** – mail address to mail server mapping, **CNAME** – domain name to domain name mapping (CNAME **ONLY FOR NON ROOT DOMAINS, but it can be routed on CNAME record to it**) (route api.vojta.com to vojta.com (root) possible and it routes to traffic to where vojta.com routes the traffic)
- **Root domain** – vojta.com, **Non root domain** – api.vojta.com
- **Extra record** – only for Route 53, routing to AWS resources – **Alias Records** – they are used to **route traffic to selected AWS resources** – ElasticBeanStalk, ELB, S3 bucket, CF distr. – **IP**

addresses of those resources can change, we don't want to hard type them in, we just reference the AWS resource. (ALIASES **ALSO FOR ROOT AND NON ROOT DOMAINS**) ONLY WAY TO ROUTE ROOT DOMAIN TO AWS RESOURCES. **CF requires alternate domain name that need to exactly match to the DNS name (e.g. vojta.com to CF, vojta.com is the DNS name), same for S3 bucket name need to match the DNS domain name (vojta.com to S3 bucket name called vojta.com)**

- **!!!Main ROUTE 53 jobs is routing + routes between multiple servers!!!**
- Routing based on user's geolocation to the most suitable servers, e.g. ELB
- **Routing policies – Simple** – domain name to one or more IP Addresses (Round Robin if more), **Weighted –** single DNS name to multiple weighted resource – A, B, C – Canary Deployments, **Latency** – picks server with content with minimum latency, **Failover** – multiple servers – main, standby – disaster recovery, primary has a health check, if fails automatically switches to DR server, **Geoproximity** – nearest resource to the user, biasing – favoring of some regions to some resources, **Multivalue answer**, **Geolocation** – based on user location, specify what location uses what server/resource, more specific the location is defined the higher priority (Kentucky > USA > North America), **default policy necessary** – fall back in case no defined region applies to the user, otherwise returns "no answer"

DevOps

- Automated processes to deploy and test your code – prevents bug creation, tests are run automatically -> faster bug discovery, faster and reliable deployments, monitoring, faster feedback
- Bring together Business unit (task) + Software Development unit (create the product) + Operations Team (make the product run)

DevOps - CI, CD

- Continuous Integration – run tests and packaging (code + all dependencies required for the code to run properly) of committed code
- Continuous Deployment – deployment to another stage (test environment), e.g. master to test
- Continuous Delivery – deployment to prod
- **Guarantees code quality**
- Committed code should have **Unit tests (test single functionality)** and **Integration tests (harder, tests in global scope with other depending services that oftentimes need to be mocked)**

DevOps - CI, CD Tools in AWS

AWS CodePipeline

- orchestrates CI/CD pipeline
- from code commitment over testing and packaging to deployment and delivery (**release** manager tool)
- **Typical release workflow: source (your code in VCS - CodeCommit, Git) -> build (app package or container - Jenkins, CodeBuild) -> test (run unit and integration tests) -> Approval (also manual) -> Deploy (code to resources), stages: master -> test -> prod**.
- Pipeline consists of **multiple stages** depending on release workflow.
- Alternatives to: Jenkins, Bamboo, GitLab, TeamCity, Travis CI, AzureDevOps(=for cloud only).
- Integrates with various AWS & Third-party tools to automate end to end release process.

- Integrates with CW – monitor progress, CT - trace API calls, Eventbridge - use streams to invoke an event driven action
- Triggered on new commits, can be configure for S3 bucket or ECR Repository
- You can add manual approvals to corresponding stage in CodePipeline

AWS CodeCommit

- one part of CodePipeline, Code repository, AWS Control Version System (like Git)
- CodeCommit repositories – encrypted at rest and transit by KMS is managed by AWS
- Authentication: IAM User (recommended) – log in using SSH or Git credentials, IAM Role – when wanting to use different authentication system (**federated access, cross-account or other AWS services**) and allow whoever assumes the role to do something in CodeCommit (use policies to give/limit access)
- Monitoring – CW, on action (pull request, repo deletion) trigger LF, SNS, …
- Copy Github repo to CodeCommit repo -> Clone Github Repo to your machine, set up CodeCommit. You need IAM user who has policy AWSCodeCommitPowerUser. Configure git credentials for CodeCommit – recommended or use SSH key pair. Push Github repo to CodeCommit.

AWS CodeBuild

- **build** (application packages, e.g. java code turn into .jar or build a container where your app will be run and tested) **and test code**
- TEST RUN ONLY HERE
- Run CodeBuild locally or any On-Premises device – CodeBuild Agent
- compile source code, run tests, create artifacts
- **2 Steps: 1. BuildSpec file** – commands to build your project – create docker image, copy there something, run it, …, define build output files = Artifacts if any (e.g. log file recording which container image was build – its tag and number and time and write it into a file, this log file is an artifact) – will be stored in S3 so they can be reused later, **2. CodeBuild project** – define where is your source code – Git, CodeCommit, S3, …, define build environment – Operating System, Runtime, Tools to install
- To be able to create a stable build every time, an environment has to be set up first **-> Environment before Build**
- **BuildSpec** – **Always YAML**, **BuildSpec.yml at root of source directory**, contains **build commands**, **phases** – build can be divided into logical individual blocks or stages – **install** – e.g. install dependencies, runtime environment – python, java: openjdk8, **pre-build** – e.g. set up connection to docker repository, **build** – create docker image with your running app, or app package, **post-build** – e.g. push docker to repository (see pre-build), run tests, **environment variables, Build output files = Artifacts** (stored in S3)
- As part of your build, you run integrations test and need AWS CodeBuild to connect to your test DB or any AWS resource**, by default AWS CodeBuild can't access resources in your VPC**, however, you can configure: **VPC ID, Subnet IDs and Security Group IDs** OR use **VPC Endpoint** to connect to your VPC
- Provides preconfigured build environment (Docker image) for popular programming languages
- Integrates with CW for logs

AWS CodeDeploy

- automate deployment within **cloud**, deployment to AWS services (EC2, ECS, LF) and on-premises services/instances (CodeDeploy Agent locally or on the on-prem device + registration to AWS CodeDeploy, coordinates execution according to AppSpec.yml file)
- NO TESTS ARE RUN IN THIS STAGE
- What needs to be set up for proper deployment: what app should be deployed (built of the app), where to deploy it, deployment configuration – deployment rules and success/failure scenarios -> deployment group (which instances should be deployed – what AMI), target revision – app version to deploy (Lambdas have versions, new app packages have revisions = versions), deployment type – in-place x blue/green, traffic shifting – canary, linear, all-at-once
- **Targets: EC2, ECS, LF, On-premises Inst.**
- **Integrates with other third-party CI/CD (Jenkins) and IAAC(Puppet, OpsWorks) tools**
- Manual or Automated rollbacks if deploy fails, automated rollback triggered on fail of a deployment or CW alarms attached to deployment groups/instances
- 2 deployment types: **In-place deployment** – uses existing resources -> application is stopped, latest version deployed, application started and validated if it does what it's supposed to (**ONLY supported by EC2 and On-premises**) -> **Rollback harder**, you need to redeploy older version, **Blue/green deployment** – creates new environment/resources -> Creates new environment (e.g. EC2 - new deployment group – EC2 Instances, ECS – replacement task set, LF -> new serverless environment), deploys there the new version, traffic is shifted to the new environment (these new instances – in case of EC2 deployment, ECS – to the new task set, LF – to the new serverless environment) (**supported 3 out of 4 – EC2, ECS, LF**) -> **Rollback easier**, traffic will be shifted back and new environment will be deleted
- Blue/green deployment -> **shift traffic rate:** canary (part of traffic shifted to new instances, if it succeeds, shift the traffic to all of them), linear (N by N), all-at-once
- EC2 and On-Premises deployment – deployment (**=copying of files**) executable files – docker image or app package, configuration file (**AppSpec**), images, and more to run the app to deploy it to the EC2 instances, **for EC2 instance or On-Prem device – CodeDeploy Client install required,** identify resources using tags or use ASG.
- **!!! EC2 supports – in-place and blue/green deployment type, On-prem only in-place**, **Lambdas and ECS only support blue/green. !!!**
- **AppSpec manages the deployment, YAML and JSON formats allowed, at root of source directory, ONLY FOR CODEDEPLOY!**
- **EC2/On-Prem AppSpec** – OS CodeDeploy uses to deploy everything, files – mapping – source and destination (S3 to EC2 Instance), permissions for destination files, hooks + scripts(lambdas – name or arn)
- **Lambdas** – Resources – AWS::Lambda::Function, name, alias, currentVersion, targetVersion (CF format of writing), hooks + lambdas
- **ECS** – Resource – AWS::ECS::SERVICE, task definition, ELBInfo, hooks + lambdas
- CodeDeploy hooks – significant points at time of deployment, allows you to run scripts to check/execute/validate something. **Hooks are used in AppSpec.**
- Lambda deployment hooks – BeforeAllowTraffic – e.g. validate something using another lambda, AfterAllowTraffic – e.g. validate that all instances serve the traffic
- EC2/On-Prem In-place**:** Start deploy -> **BeforeBlockTraffic** to the instances -> BlockTraffic -> **AfterBlockTraffic** to the instances -> **ApplicationStop** on the instance -> DownloadBundle of the new app -> **BeforeInstall** of the new app -> Install -> **AfterInstall** -> **ApplicationStart** of the new app -> **ValidateService** – that instance and application work properly -> **BeforeAllowTraffic** to the new app -> AllowTraffic -> **AfterAllowTraffic** -> End deployment

- **NOTE: Only the bolded hooks can be used in AppSpec**
- EC2/On-Prem Blue/green: We are using new instances, similar hook order, however, the traffic to the old instances will be stopped at the end: Start deploy **-> ApplicationStop** on the instance -> DownloadBundle of the new app -> **BeforeInstall** of the new app -> Install -> **AfterInstall** -> **ApplicationStart** of the new app -> **ValidateService** – that instance and application work properly -> **BeforeAllowTraffic** to the new app and instances -> AllowTraffic -> **AfterAllowTraffic** to the new app and instances -> **BeforeBlockTraffic** to the old instances -> BlockTraffic -> **AfterBlockTraffic** to the old instances -> End deployment
- ECS – **BeforeInstall** -> Install -> **AfterInstall** -> AllowTestTraffic -> **AfterAllowTestTrafic** -> **BeforeAllowTraffic** -> AllowTraffic -> **AfterAllowTraffic**

DevOps – IAC (Infrastructure as Code)

- **Automate resource provisioning with a code alone**
- **Has nothing to do with CI/CD -> IAC is used to provision resources and install necessary dependencies/software**
- You can track changes based on the changes done to the code that provisions (buys) the resources (EC2 Instance, Lambda, DB, SNS, …) using control version system (Git)
- Introduces repeatability to your code – easier code
- **Infrastructure provisioning** – provision AWS resources, tools: **Terraform** – open source, AWS **CloudFormation**, AWS SAM, we use **CDK**
- **Configuration management** – install software and tools on the provisioned resources, tools: open-source (**Chef**, **Puppet**, Ansible), AWS **OpsWorks**

DevOps – CodeStar

- AWS service for project management of development of your app - entire set up of your CI/CD, issue tracking capability – like Jira, user permissions – PO, devs, viewers to CI/CD tools/services, repos.
- **Dash board – issue tracking + CI/CD + IAC**
- Automatically and FAST set up coding, building, testing, and deploying your app code

AWS CloudFormation

- IAC tool
- Can be part of code deployment in CI/CD pipeline
- Template file for provisioning of resources -> tracking changes
- Can provision same resources for different environments
- **JSON** or **YAML** file
- Understands dependencies – first creates resources on which other depend, e.g. VPC before EC2 in that VPC
- **Automatic rollbacks on error** - e.g. if DB creation fails, it deletes all the other resources created with that template
- **Template** = CF JSON or YAML file defining resources (to provision)
- **Stack** = A group of resources created from that CF template
- **Change Sets** = Tool to update or replace current stack. Are created from a current stack. It will show you how your new template or your changes affect existing resources of the current template. If you agree with the changes, proceed to execute.
- Important template elements: AWSTemplateFormatVersion, Description, Metadata, **AND**

Resources

- what you want to create
- MANDATORY template element
- list of resource objects to create
- **Type of resource –** format: **AWS::ProductIdentifier::ResourceType**, e.g. AWS::S3::Bucket
- **Different properties for different resource** – mandatory, optional

Parameters

- Values to pass to your template at runtime – e.g. what region, which stage, reference to another resource, ec2 instance type
- Parameters are a method within CloudFormation stacks that provide choices when a stack is launched
- make template dynamic -> slide in new inputs as template is running
- **max. 200 per template**
- **Parameter Name is mandatory ->** Other resources can reference to it, !Ref Parameter-Name
- **Type is mandatory:** String, Number, AWS specific type: AWS::EC2::Image::Id -> type is then id of EC2 image format
- **Also restrictive ability** – AllowedValues, MaxLength & MinLength– chars, MaxValue & MinValue, AllowedPattern – does it comply with regular expression?, Default -> more in docu
- Specify parameters in CF api call:
  aws cloudformation create-stack --stack-name S1 --template-body example-template **--parameters ParameterKey=ParameterName,ParameterValue=ParameterValueYouWantToUse**
- **Or parameters are passed in using a single file -> --parameters ./file-with-param-definitions**
- **file-with-param-definitions: [{"ParameterKey": "FunctionName", "ParameterValue":"ValueYouWantToUse"}, {…}]**
- Pseudo-Parameters – parameters predefined by AWS – e.g. AWS::AccountId, AWS::Region – where the stack is created, AWS::StackId – of the created stack, AWS::StackName -> !Ref: "AWS::StackId" or { "Ref" : "AWS::StackId"}
- Attributes are under resource **Properties:**
- CreationPolicy – common attribute for ASG -> Specify when ASG resource is complete -> ASG resource is complete once it's dependencies are in running state, e.g. CreationPolicy ->ResourceSignal-> Count: 3 -> ASG is complete when 3 EC2 instances are running
- DeletionPolicy – preserve or back up resources when stack is deleted – retain, snapshot, delete possibilities
- DependsOn – On resource starts to get created once another resource that it depends on is successfully created
- UpdatePolicy – for EC2 instances – reuse existing or create new ones -> AutoScalingReplacingUpdate, AutoScalingRollingUpdate
- **Conditions:** to resource or stack output, based on the condition the resource is created (or not), e.g. Condition:\n CreateResource: !Equal [{ !Ref EnvTypeParameter, "prod"}]

Mappings

- Key-value pairs
- E.g. configure different values for different regions
- **Mappings:**\n **RegionMap**:\n region-1:\n "AMI_key_1": "AMI_value_1"\n"AMI_key_2": "AMI_value_2"\nregion-2:\n"AMI_key_1": "AMI_value_1"\n"AMI_key_2": "AMI_value_2" -> Create Resource EC2 Inst, use AMI based on region and key **-> ImageId: !FindInMap [RegionMap, !Ref: "AWS::Region", AMI_key_1]**

Outputs & Exports

- Return values from execution
- Outputs in console, use in automation
- **Max 60 outputs per template**
- **Can be used to create cross stack reference by exporting the value -> Output variable, use export -> Name: "Name of resource to export"**
- **For each AWS account, Export names must be unique within a region.**
- **Outputs -> OutputVariableName -> Value** + description + condition
- Outputs are NOT encrypted -> not recommended for passwords or credentials

Transform

- **Set of macros to process your template = set of functions (AWS lambdas) to do something to your template before launching it**
- **Not really useful -> typically nested stack, export and import across stacks**
- 1. Create a lambda parsing template JSON/YAML and what changes you want it to make it, 2. Create a template with Macro resource and reference the lambda, 3. Use the macro in different stack -> in different stack -> section **Transform -> MacroName, Macro Method** – makes changes within the **whole stack -> to limit** the macro for only some section of the template: **Fn: Transform**
  Macro Methods below
- **AWS::Include -> inserts another template parts into your template**
- AWS::Serverless -> converts SAM template to CF
- AWS::CodeDeployBlueGreen

Intrinsic Functions – Native, natural function within CF

- **Built-in functions in CF**

Ref function

- Refers to other resources defined within the template OR existing in AWS environment (use arn of the resource) OR input parameters in the template
- !Ref Resource-Name-to-refer-to-it or {“Ref”:” Resource-Name-To-Refer-To-It”} or Fn:Ref
- **Returns attribute that identifies the resource**

GetAtt function

- Like ref but you can specify which attribute of a resource you want (if possible)
- **Fn:GetAtt or !GetAtt ‘logical-name-of-the-resource.attribute’**

Sub function

- substitutes variables in an input string with values that you specify
- In your templates, you can use this function to construct commands or outputs that include values that aren't available until you create or update a stack
- { "Fn::Sub": ["www.${Domain}", { "Domain": {"Ref" : "RootDomainName" }}]}
- Name: !Sub
- - www.${Domain}
- - { Domain: !Ref RootDomainName }

FindInMap function

- Make template more generic to use it across different regions -> **create mapping**
- Select from the map
- !FindInMap [RegionMap, !Ref “AWS::Region”, HVM64]

Join function

- Join multiple values – **delimiter, array of values to be joined**
- “Fn:Join” : [“:”, [“arn”,”aws”,”resource”,”attribute”]] -> arn:aws:resource:attribute

Cidr function

- The intrinsic function Fn::Cidr returns an array of CIDR address blocks. The number of CIDR blocks returned is dependent on the count parameter.
- { "Fn::Cidr" : [ipBlock, count, cidrBits]}
- cidrBits = The number of subnet bits for the CIDR. For example, specifying a value "8" for this parameter will create a CIDR with a mask of "/24".

Fn:GetAZs – return an array that lists AZs for a specified region -> Fn:GetAZs: !Ref: “AWS::Region”

Fn:ImportValue – use output from another stack, cross stack only within the same region

"SubnetId" : {"Fn::ImportValue" : {"Fn::Sub" : "${NetworkStackNameParameter}-SubnetID"}}

Fn:Select – returns a single object from a list of objects by index

- { "Fn::Select" : [index, listOfObjects] } OR !Select [index, list]

Fn:Transform – specifies a macro to perform custom processing

- **You can't delete a stack if another stack references one of its outputs.**

- **You can't modify or remove an output value that is referenced by another stack.**

Cross stack reference

- Break up big CF script into smaller -> into logical sections
- Outputs->OutputVarName->Value + (Export->NameOfExportedVariable)
- !ImportValue NameOfExportedVariable

Nested Stacks

- It’s advantageous to have standard templates for different types of resources -> **allows template reuse**
- You can reference these resources in one “root” stack
- **Resource Type: AWS::CloudFormation::Stack, templateURL: “https::/s3….” Or another reference, you can pass in Parameters, these parameters need to be specified under the resource as well as the stack itself. The names of parameters have to be the same.**
- **Nested Stacks can contain other nested stacks**

Stack Set

- **To create same resources in different accounts across different regions**
- Lets you create stacks in AWS accounts across regions by using a single CloudFormation template **-> Stack set + CF template**

- **When you update a stack set, all associated stack instances are updated throughout all accounts and Regions.**
- A stack set is a regional resource. If you create a stack set in one AWS Region, you can't see it or change it in other Regions -> There are **stack instances in each Region**
- Administrator account is required to create the StackSets-> **cross account actions** -> trust relationship required between Administrator account and target account
- Operations: **create, update, delete**
- **When you update a stack set, all associated stack instances are updated throughout all accounts and Regions.**

Remember

- Deleting deletes all associates resource defined in the template -> **DeletionPolicy** or **Termination Protection** active for the whole stack, only **empty bucket can be deleted**
- You can run CF template using **CLI**
- **WaitCondition – wait until another resource/s will be created and then get created**
- Python has some helper scripts that make it easier to deploy on EC2 Instance

CF vs AWS Elastic BeanStalk

- Elastic BeanStalk – pre-packaged CF template with a UI -> you click what you want, CF will be created automatically, sets up automatically a run environment

Serverless Application Model = SAM

- Open source framework for building serverless apps -> shorthand record of resources – **"easier CF template"** -> automatically generates CF
- **YAML format**
- Hides complexity of provisioning resources, single deployment, built-in best practices – CI/CD, security, tracing, **local debugging and testing**
- Not only for AWS -> Google, Microsoft
- SAM Template – **mandatory**: **Transforms** – AWS::Serverless-Version (something like -2016-10-31), **Globals** – global variables, **Resources** – SAM resources and CF resources, optional: description, metadata, parameters, mappings, conditions – e.g. create when env = dev, outputs
- SAM Policy Templates – typically given to LFs to access other resources**,**
  SAM provides a **list of preconfigured policy templates**
- **CF is created as part of deployment package (sam package), deployment package is saved in S3**
- SAM – no cost, only pay for resources provisioned
- **Deploy app** - sam deploy (includes sam package) or aws cloudformation package + aws cloudformation deploy

SAM supported resources

- **Container App** – AWS::Serverless:**Application** – container for all provisioned resources, specified at the beginning of the template
- **Lambda Functions and Layers** – AWS::Serverless::**Function/LayerVersion**
- **API Gateways** – AWS:Serverless::**Api/HttpApi**(Rest)
- **DynamoDB tables** – AWS::Serverless::**SimpleTable**
- **Step Functions** – AWS::Serverless::**StateMachine** – to coordinate invocation of your LFs

- **Other resources have to use CF resource provisioning format**

Key SAM CLI Commands

- sam init – initialize serverless app using sam template
- sam validate – validate template
- sam build – builds the app
- sam local invoke – invokes a local LF
- sam package – bundle app code + dependencies into -> deployable package
- sam deploy – deploy app to AWS (includes sam package command)
- sam logs – fetches logs generated by your LFs
- sam publish – pushes sam app to aws serverless application repository – people can access it and reuse it as necessary

EC2 – Advanced

- Vertical scaling – bigger and stronger instance -> more CPU, Memory, HDD, **has limits, can be expensive**
- Instance size: micro to xlarge
- Instance family – M5 – general purpose, high network throughput, ENA adapter – Enhanced Networking Support, R5 – memory optimized - RAM, C5 – compute optimized - processor
- Horizontal scaling – **no limits**, scaling in/out -> more instances (more computers), **requires ELB, better availability, uses ASG – min, max, desired,** deployment: single AZ, in one region, multiple region – depends where your customers are
- **Tenancy** – Shared vs. dedicated
- **Shared** tenancy – default – one host machine (an actual computer on which run virtual machines) in AWS can provide instances to different customers. **Instances of AWS accounts are placed on a same physical machine.**
- **Dedicated** tenancy – **dedicated instances** – AWS assures that an actual physical machine (host server) will only be dedicated to you – your instances (your AWS account), but **the actual physical machine can keep changing! E.g. at deployment**
- **Dedicated** tenancy – **dedicated host** – AWS allocates you one (or more) of their physical servers and it will only be yours and it only carries your instances -> you can see the hardware -> **used for regulatory needs or server-bound software licenses (Windows Server, SQL server)**
- In the dedicated host you can only launch instances that the host supports
- **EC2 pricing model**
- On-demand – can be created any time, you can have them as long as you wish to, most expensive, **use:** spiky traffic for a web app, batch program with unpredictable runtime and cannot be interrupted, first time data migration from on-premise to cloud **-> spiky traffic, processes that cannot be interrupted**
- Spot – AWS gives these unused instances for very low price (- 75 % to – 90 %) because they can request them anytime back (**2 minute notice**). You set the max price (**price request**) you are willing to pay, based on demand and offers of others the price can grow and once spot instance becomes more expensive than what you are willing to pay, it disappears and capacity is offered to someone else. Old – people bid on instances, highest bid wins the instance. Suitable for batch **processes** when app progress **can be interrupted** = stopped **and restarted another time** – analytics of data, deployments that can be restarted.
- Reserved – reserved instances ahead of time for 1 or 3 years. They are not a specific instances but rather discount on On-Demand instances that match certain criteria, such as

size, family type, region... **Offering class** – **convertible** – can switch to different ec2 instance family, **standard** – no switch possible. Up to 75% off. Payment models: No Upfront - $0 upfront, monthly installments, Partial upfront, All Upfront – no monthly installments. The sooner you pay, the better the discount. Difference up to 5 %. **Use: constant workloads that run all the time – web apps**

- Saving Plans – commit spending $X/hour on AWS resources – Lambda, Fargate for 1 or 3 years. You can create instances as necessary with up to 66-72% discount. You can create any On-Demand instance and is NOT bound to any instance configuration/criteria as is the case for reserved instances. **Compute Saving Plan** – up to 66% discount on compute resources – Fargate, lambda, ec2 instances – can changes whatever configuration as you wish to. **EC2 Instance Savings Plans** – up to 72 %, only for EC2 instances, commitment to usage of individual instance families in a region **Use: constant workloads that run all the time – web apps and more flexibility regarding EC2 instance (virtual server) configuration**
- **Placement Groups** – sometimes we want to have better control over our instances, how they can talk to each other and how protected are they against failure – **latency, availability. You can specify placement of your instances!**
- **Cluster placement group** – low latency network communication – **examples: Big Data, High performance computing**, instances need to be placed close to each other, ideally the same host machine or a rack (group of host machines powered together from shared power supply, each rack has its own network and powersource) in a single AZ. **High Network Throughput, Low availibility**
- **Spread placement group** – avoid simultaneous failure -> high availability required, in this case we want instances in different rack or on different host machines -> you don't want all of your instances fail at the same time. Instances spread across multiple AZ within a single region. **Max. 7 running instances per AZ per placement group**
- **Partition placement group** – multiple partitions with low network latency and at the same time higher availability,
- **Placement Group (100s of EC2 instances) > Partitions = sets of EC2 instances (placed on the same rack -> low latency network communication) > EC2 instance (You select to which partition is each instance assigned)**
- **No two partitions of the same placement group share the same rack.**
- **Partitions across different AZs within a region.**
- **Max. 7 partitions per AZ per placement group.**
- Typically used by large distributed and replicated workloads, such as Hadoop, Cassandra, and Kafka
- Maximum of 500 placement groups per account in each Region.
- Name of your placement group must be unique within your AWS account for the Region.
- **Insufficient capacity error** – you typically get it when rack doesn't have enough capacity for your newly launched instances, typically caused by simply not enough capacity of the rack (in this case, stop and restart all instances in a placement group OR relaunch placement group again -> instances will be migrated into a new rack with more capacity to satisfy all requested instances) or when you want to launch different type of instance that the hosted machine/rack doesn't support and placement group conditions might not be adhered to. -> **Recommended: within a placement group use only one type of EC2 instances, launch all instances in a single launch**
- **Assign EC2 to a placement group when you set up configuration at EC2 launch**

- **Elastic Network Interface (ENI)** – represent **virtual network card (síťovka)**, provides EC2 instance with private and public IP addresses **used for communication**. **Support IPv4 and IPv6.**
- 1 ENI – **1 primary private IP address (default, eth0)** and multiple secondary (eth1, eth2, …), 1 public IP, 1 Elastic IP add per private IPv4 address, one or more SGs
- Secondary IP address can be **dual homed** – part of 2 network subnets
- You can create network using EC2 instances and these ENIs.
- **Hot attach** – attach ENI to EC2 instance while running, **Warm** – while stopped, **Cold** – at launch

Block and File Storages

- Block storage = typically a harddrive, 1 harddrive can typically be only connected to 1 computer, however, 1 computer can have more harddrives, block storages in AWS: **AWS EBS** (persistent storage for your EC2 instance)**, Instance store** (pherimeral storage of your EC2 instance or your lambda?? NOT SURE, VALIDATE)
- File Storage – can share data with multiple computers, e.g. Google Drive, your company harddrive that any one employee can access, file storage in AWS: **Amazon EFS** (for Linux instances), **FSx Windows File Servers**, **FSx for Lustre** (high performance)
- NOTE: S3 is an object storage
- **Instance store** – block storage, storage for EC2 instance (**only some instance types support it**), ephemeral storage = temporary data (lives as long as the instance is not terminated or faulted, if EC2 becomes unavailable so does the storage), **storage is located on the same host computer as the EC2 instance (=physically attached), very fast** (I/O 2-100x of EBS), **no extra cost**, good for storing temporary data – e.g. cached data, slow boot up time – instance takes longer to be launched, **cannot take a snapshot** and restore data from it, **fixed size** – size depends on EC2 instance type
- **Elastic Block Storage** – block storage, storage for EC2 Instances, permanent storage = persistent data (when EC2 instance is terminated, EBS stays intact and can be reconnected to another instance), network storage – EC2 instance and storage do not share the same host machine, they communicate via AWS internal network. Paid storage, increase size as necessary (**even when attached to an instance**), **replicated within the same AZ, snapshots supported**
- AWS offers EBS Multi-Attach – Block storage that works like file storage, single harddrive can be connected to multiple computers (up to 16 instances from single AZ), works only for some specific types of EBS – IOPS SSD for example
- EBS – **HDD** (Hard Disk Drive) **vs. SSD** (Solid State Drive) – SSD has significantly better IOPS (I/O per Second), both have high data throughput, SSD is faster but more expensive, used as boot volume (where OS is installed), HDD – lower price, suitable for big data or large streaming workloads, HDD – smaller number of transactions, however, larger sequential data reads, SSD – large number of transactions. HDD – for big data that require a lot of storage because it is cheaper, SSD – faster but more expensive
- **Amazon EFS** – file storage + file management (inflatable Google Drive), autoscaling – petabytes of data, compatible with **Linux** based EC2 instances
- **Amazon FSx for Lustre** – file system optimized for **performance**, **automatic encryption at rest and in-transit**, high performance computing.
- **Amazon FSx Windows File System** - fully managed shared storage built on Windows Server, and delivers a wide range of data access, data management, and administrative capabilities.

File System is accessible from Windows, Linux and MacOS machines/instances, integrates with Microsoft AD (Active Directory), automatic encryption at rest and in-transit.

- AWS Storage Gateway – Hybrid storage (on-premises + cloud, all services before were cloud solutions) -> provides unlimited storage for on-prem apps and users with high performance, encrypts data by default like Glacier
- AWS Storage File Gateway – move shared files from on-premises to cloud, 2 options - **Amazon S3 File Gateway** enables you to store file data as objects in Amazon S3 (and Glacier) cloud storage for data lakes, backups, and ML workflows (Gateway on-premises allows caching, http/s communication between local gateway and S3). **Amazon FSx File Gateway** provides low-latency for team shared files, on-premises access to fully managed file shares in Amazon FSx for Windows File Server (SMB clients on-prem, FSx Windows File Server in cloud talks to S3 storage)
- AWS Storage Tape Gateway - enables you to replace using physical tapes on premises with virtual tapes in AWS, compresses data, protects physical tapes from wears and tears. On-premises low-latency caching. Stored in S3 and Glacier. Or you can use **AWS Snowball** – for humongous volume of data. They send you a machine that reads up on all the tapes and then they store in into S3 Glacier Flexible Retrieval or Deep Glacier
- AWS Storage Volume Gateway – on-premises block storage to cloud, as primary data storage or as a backup. **Cached** – storage is primarily moved to the cloud (S3), frequently accessed data is stored locally in your computer storage – low latency. Plus, no need for bigger and faster local storage. **Stored** – all data stored locally on your storage – low latency **for all data**, back-ups to the cloud (S3)

AWS Elastic BeanStalk

- AWS service for easy deployment and scaling of your web app (end-to-end application management). Supply code and set resources using UI.
- Supports: **programming languages** (Java, .NET, PHP, Node.js, Python, Ruby, Go), **application servers** (Apache, Nginx, Passenger, and IIS) and **Docker containers**
- **For free**
- Manages resources, health checks, autoscaling, **platform updates**
- **Application** = container for versions (= code), environments, configuration (platform – Python, Docker, Server, configuration of resources)
- **Application Version** = **specific state**/version **of** deployable **code**
- **Environment** = application version deployed to configured **AWS resources** (EC2 instances, DBs, S3 buckets, …) + environment properties, diff environments – set up of resources - for different stages
- Environment Tiers – worker tier – for batch (of scripts) app run in a terminal – not very common, **web server tier** – for web app
- **Worker Tier** - ASG + EC2 + SQS, EC2 instances process messages from SQS
- **Web Server Tier (Not ideal for microservice architecture!!!)**
- **Single instance environment** – EC2 + Elastic IP – code deployed to this one instance
- **Load-balanced environment** – ELB (+ Elastic IP) + ASG + EC2s
- Create **DB** as part of your environment has its caveats – DB disappear after termination of an environment. **Workarounds:** Enable Delete Protection on RDS OR take a snapshot and restore.
- **Source bundle** = deployment package – built code + dependencies, **one and only one bundle file - .zip or .war (do not place in any folder)**, **max size 512 MB**

- **Configuration files –** customize EB environment, settings of AWS services/resources in a subfolder **ebextensions in a root folder (next to source bundle)** > xray.config, healthcheckurl.config, **YAML or JSON**, **configuration file can refer to a CF template** to create another resources, set up additional settings
- Deploying new version – create new application version = deployment bundle and save it in S3, use UI – upload and deploy or CLI – eb appversion – create new version, eb deploy – deploy it. Deployment packages require storage -> limit number of last application versions to keep in EB and should also delete corresponding versions from s3? Configure **application version lifecycle policy**
- Logs can be saved in S3 or CW Logs
- Metrics in CW
- SNS to check on resource health

EB deployment methods: V1 to V2

EB – Environments – selected-env – actions – clone env/swap env URLs/upload and deploy

All-at-once – V2 will be deployed to **all EC2 instances**, **existing instances,** downtime during deployment, manual redeploy. Reduced capacity possible, can get to 0.

Rolling – deploy batch after batch, only start another batch after first one was successful, if one batch fails, another batch will not start. Deployment on existing instances. Zero downtime. Manual deployment. Reduced capacity possible.

Rolling with an additional batch – Additional batch solves reduction of original capacity -> No reduced capacity -> first batch is deployed on newly created instances, rest is deployed on existing. If first batch fails, deployment will start and new instances will be removed. If successful, last existing instances removed. Manual redeploy. Zero downtime.

Immutable – second ASG with newly launched instances to which we deploy V2. Existing and new instances are running simultaneously until new instances pass a health check, then old ASG is replaced by new ASG, old instances are terminated. Zero downtime. Rollback terminates new instances if they don't pass health checks.

Traffic Splitting = Canary testing approach – create a couple of new instances, deploy V2, redirect portion of the traffic, if bad -> fail -> redirect the traffic back and new instances terminated, if successful -> launch new instances and deploy V2, redirect all traffic, terminate all old instances. Zero downtime. Percentages of redirected traffic can be temporarily impacted.

Blue/green deployment – create completely new environment, launch new EC2 instances, deploy V2 there. Test if V2 does what is should. If yes, swap URL to the new env. If no, swap URL back to original env.

Containers and Container Orchestration

- **Micro service architecture** – instead of monolithic app, separate code into logical units (=micro services), when you modify code, you do not need to deploy a huge app but just portion. Modern way to go. Micro services can be written in different languages -> only problem is that each deployment would be different -> **containers – standardizes deployment for any language!**
- **1 MS = 1 Docker Image**

- **Docker** – docker image for each microservice, **docker builds a container from a docker image. This container can be run on any operation system/platform exactly the same** , be it your local machine, corporate data center or cloud **–** but the platform requires installed **DockerEngine**
- **Docker Image** = contains everything a MS needs to run – **Application runtime** (JDK, Python, NodeJS), **Application code**, **Dependencies**.
- **Docker container** = build by Docker using a docker image, lightweight version of virtual machine
- **Docker – isolation of containers –** one container doesn't know about the other containers
- **Container Orchestration** – manage large number of containers, supports **auto scaling** – scale containers (= MS instance) based on demand, **load balancer** – distribute load to multiple containers of the same MS, **Self Healing** – container health check, replacement, **zero downtime deployments**, **service discovery** – help MS to find one another
- **(Docker images + configuration to have them run on container orchestration service) -> Container Orchestration -> creates cluster of instances (containers)**
- Container orchestrators: Kubernetes – neutral, AWS offers Elastic Kubernetes Service (EKS) – manage cluster manually – manual increase and decrease of MS instances (containers), Elastic Container Service (ECS) – same, cluster managed manually, Fargate = serverless version of ECS – cluster scaled on demand

Amazon Elastic Container Service

- Managed AWS service for container orchestration
- **Cluster = Group of one or more EC2 instances (where containers are deployed)**
- **Fargate = Serverless version**, you don't care about EC2 instances, no visibility in EC2 instances – you cannot enter them, only the cluster
- **Container Instance = EC2 instance + container agent** -> in ECS use **AMIs with preinstalled** container **agent**, EC2 instance **contains a config file - /etc/ecs/ecs.config** – that defines to which cluster the ec2 instance belongs
- **ECS capacity providers –** cluster auto scaling
- **ECS – Task Definition** – what container for a specific task – container cpu, memory, docker image, name of container, mapping for container – hostPort (on which port is container exposed to the world, if not assigned, dynamic host port automatically assigned), protocol (tcp, …), containerPort (internal port of container), launch type – ecs or fargate, volumes attached to the container, task permissions
- **Typically: 1 Task = 1 container**, Exception: If two containers have exactly the **same lifecycle** – e.g. both use the same volume, 1. writes in, 2. reads from it, **sidecar pattern** – usually container that every task has – taking care of some side jobs like **logging**, **metrics**, proxy requests
- **1 EC2 Instance can run multiple Tasks. A Task can run multiple containers.**
- **Task permissions** – Task Role (Role for Task Definition) x Task Execution Role (used by container agent/Fargate agent)
- Task Role define IAM Role, specify permissions, **more secure than using EC2 instance roles, 1 Task Role for each Task Definition that should correspond to each container/MS**, activate container agent while launching EC2 Instance via ECS using ECS_ENABLE_TASK_EXECUTION_ROLE=true
- Task Execution Role ("container role") – provides ECS container agent and Fargate agent access to: **pull container images from ECR** (Elastic Container Repository) or any other container image repository, **publish container logs to CW**

- **Cluster > Service > Task Definition > Container Definition**
- **Container instance = EC2 instance**, interchangeable naming
- **Service** – is a tab in a cluster – scaling of task definitions – desired count of tasks, deployment type – rolling update, blue/green, **task placement** – which task should be placed on which instance, **task autoscaling** – No. of EC2 instances for a specific task will increase/decrease
- **Task placement** – which task should be placed on which container instance (EC2 instance), based on Task Requirements on CPU, memory and ports – specified in Task Definition, Placement Strategies and Placement Constraints. **Task placement by default is done automatically by ECS which places task (containers) across EC2 instances within the cluster and spread them across AZs.** Task placement works like a filter, it takes all EC2 instances within the cluster, selects those that comply with container requirements, constrains (applied first), and strategy and pops up a single instance where is then task placed.
- **Constraints –** distinct instance – every task placed on unique EC2 instance, member of – only places task on a specific set of attributes around your container instance (EC2 instance, interchangeable naming) (defined attr. or custom), e.g. run task only on EC2 instance types t2.micro. To specify attributes and constraints, use **Cluster Query Lnaguage**
- **Strategy types**: field (ECS AZs, memory), type (3 – **binpack** – place tasks on as few instances as possible, optimize for memory OR CPU, **random** – places tasks ad-hoc across your instances, take any available EC2 instance that complies with Task requirements and place Task there, **spread** – spread tasks evenly across, e.g. different host instances – InstanceIds OR AZs). **Combine strategies and prioritize => chaining.** 1. Spread tasks across AZs, 2. Binpack tasks in those AZs.
- ECS uses **ALB** to balance load between container instances. ALB offers **dynamic host port mapping** – multiple tasks from the same service run on the same EC2 instance. In **Task Definition** we do not set ports -> automatic assignment of ports for each running task. **ALB** knows about these assigned ports and can route traffic correctly. **Path-based routing** – used in microservice architecture, multiple micro-services (**instances**) can use the same port for communication with ALB. Specify path for each instance app.com:5500/microservice-a, /microservice-b and ALB routes traffic based on it to the corresponding instance.
- Elastic BeanStalk – you can deploy Docker container, use container images from container repositories or build your own container images during deployment – include Dockerfile, **Docker compose** supported (docker-compose.yml – file with defined containers) **– easy way to run multiple containers at the same time**, OR you can run containers using Dockerrun.aws.json version 2= **standard definition of your Tasks** – like docker-compose. Options: Single Container – directly to EC2 inst., Multiple (uses ECS in the background)
- **Elastic Container Registry (ECR)** – place to store container (docker) images, repository fully managed by AWS, alternative to Docker Hub, like GitHub but for docker images
- **Dockerfile -> docker build -> docker image**
- Docker push – push docker image to the repository, docker pull – pull an image from the repo to your local machine
- To pull from and push to ECR you have to login first: aws ecr get-login-password -> returns authentication token. Using token, it is not very secured, stored in the instance in the container that called the token creation in root/.docker/config.json. It is better to install amazon-ecr-credential-helper- to root/.docker/config.json write {\n”credsStore”:”ecr-login”\n}

More Serverless

AWS Lambda

- **Event Source Mapping** – some services cannot invoke lambda directly, e.g. **DynamoDB, Kinesis, SQS ->** but they **send events (streams)** when an action is performed on them, that's why they are called **Event Sources**
- **Event Source Mapping** - part of Lambda, Lambda resource, "lambda queue"
- Read from event source -> invoke LF
- Grouping of events = shards -> process them in batches
- FIFO – in-order processing guaranteed
- Automatic retries, if they fail many times, you can send them to an SQS queue or SNS topic
- **NOTE: S3 and SNS do not use Event Mapping – you configure lambda invocation in that service**
- Mention in mapping: **Source arn** (e.g. DynamoDB stream), **processing lambda arn**, **batch size** (from 1 up), other things like uuid, lastProcessingresult

AWS Lambda and ALB

- **ALB can route traffic** not only to EC2 instances but **Lambda functions as well**
- HTTP request comes to ALB, ALB converts it to an event and forwards it to the lambda function
- **Event is a JSON file that includes:** httpMethod, path, query-strings, headers, body, requestContext, isBase64Encoded – true x false
- Lambda response – statusCode, statusDescription, isBase64Encoded, headers – "Content-Type": "text/html", **body – contains LF response**
- **NOTE: ALB makes synchronous call**
- **ALB needs a permission to invoke the LF** -> configure it in LF itself – resource based policy (principal: elb service, action: lambda invocation, resource: LF arn)

AWS Lambda Permissions

- Give LF permissions to access other services or resources -> **Lambda Execution Role** – role that Lambda assumes when invoked
- **Role has permissions** – 3 types: **Managed policies** – predefined policies by AWS, visible in mgmt. console and can be used in other roles, **customer policies** – policies created by you (=customer), they are visible in mgmt. console and can be used in other roles, **in-line policies** – permissions given to the specific role, not visible in mgmt. console and CANNOT be used by other roles – typically in CloudFormation template or CDK, but can be also written in manually in mgmt. console.

AWS Lambda Resource Based Policies

- on the LF
- allows other accounts or specific AWS service to invoke the LF
- principal – who can do action (account or service), action – what can do on the resource, resource – which resource – usually the lambda, its version, alias or layer, effect – allow/deny

AWS Lambda logging

- lambda logs automatically stored in CW logs – by default in this folder - **/aws/lambda/function-name**
- logs help with auditing and troubleshooting your code (console.logs are visible there)
- **Lambda's execution role has to have permission to publish log to CW logs**

AWS Lambda inside a VPC

- by default lambda runs outside of VPC – can connect to internet but cannot access private resources in a VPC
- make lambda run inside of a VPC -> assign VPC to LF -> lambda needs necessary permissions to work properly (lambda execution role: create, describe, delete network interface, use AWSLambdaVPCAccessExecutionRole managed policy to allow LF assume lambda execution role)-> lambda needs ENI
- Allow LF access to Internet – private subnet of VPC need to have routing to a NAT Gateway

API Gateway – CORS Configuration

- **CORS = Cross Origin Resource Sharing** – typically happens when your FE App is hosted somewhere (FE domain if the **CLIENT**) and then accesses API Gateway on another domain (Origin domain, **ORIGIN, SERVER**). (NOTE: if the domains were the same CORS wouldn't happen). Browser is suspicious of this request and will block it, unless it comes from FE domain that is allowed in headers on Origin (API Gateway in case of AWS or server).
- **To enable CORS on API Gateway for any request method – Options methods is required –** should specify **methods**, **headers** and **origins**
- **(** Requests with non-standard header needs to be preflighted, e.g. PUT, Delete, POST but in AWS for all of them – uses Option resource method on API Gateway where we specify allowed headers for server response)
- Headers, methods and origins specified on API Gateway to allow sharing between other domains.
- **Access-Control-Allow-Methods** – http methods that are allowed to reply to if CLIENT asks
- **Access-Control-Allow-Headers** – specific headers request needs to get an OK response, if you pass in credentials – apply **Authorization** header.
- **Access-Control-Allow-Origin** – **specify domains (FE URLs = Client) from which are requests allowed!**
- E.g. Client is on vojtaapp.com, then API Gateway Option method needs to have specified that Access-Control-Allow-Origin: vojtaapp.com | if I request with GET http method, API Gateway Option method should have specified Access-Control-Allow-Methods: GET
- Configuring CORS on different APIs:
- **Rest API – Lambda custom integration (non-proxy):** on resource or the whole API – Actions – Enable CORS – there you configure the allowed http methods, headers and origins.
- **Rest API – Lambda proxy integration (proxy)**: typically, a proxy response from Lambda requires only status code and response body, for CORS, you have to also return headers specifying allow-headers, allow-origins and allow-methods.
- **HTTP API** – On the HTTP API, under Develop section, there should be CORS -> Set up allowed headers, http methods and origins. Or you can return them as a response of your LF

REST API vs. HTTP API

- Authorizers: HTTP has extra to LF authorizer, IAM and Cognito, **OAuth 2.0/OIDC**
- Integrations: REST has extra to HTTP, LF, other AWS services, private integration, **a configurable Mock**
- Usage plans/API keys – REST yes, HTTP no
- API Caching – REST yes, HTTP no
- API Type: HTTP can only be regional, REST – Region, Edge-optimized, Private
- Request transformation: REST yes, HHTP no
- Request / response validation: REST yes, HTTP no
- Test invocation: REST yes, HTTP no

- AWS X-Ray: REST yes, HTTP no
- Default stage – HTTP yes, REST no
- Autodeployment – HTTP yes, REST no

API Gateway – Canary Release

- Canary release is used to test new deployment of an app/software but routing a portion of the current traffic to the new app/instances
- Requests come through API Gateway and this is where we configure the canary settings
- **1. API Gateway – API – Stages – select stage – tab Canary – Create Canary**
- **Canary stage variable** – same stage name (as the stage we redirect the traffic from), different value OR completely new stage variables
- **2. API – Resources – Deploy API - select the stage, you'll see there "Canary Enabled" – deploy -> Canary activate**
- **3. Test if the new version is working – if errors, it rolls back immediately**
- **4. API Gateway – API – Stages – select stage – tab Canary – Promote Canary** – you can update the whole stage with Canary's deployment, stage variables with Canary's stage variables and set Canary percentage to 0%. **Select all three or select individually** -> click Update

API Gateway – Throttling

- Throttling limits prevent your customers to send too many request to your API Gateway
- **AWS Throttling limits = Default Throttling Limits – set by AWS, can't be changed, applied across all accounts and clients in a region -> prevent you API and your account from being overwhelmed by too many requests**
- **Per-account limit – cumulative number of requests to all APIs of a single account can't applies to all APIs in an account in a specific Region. Limits the steady-state requests per second (RPS) and bursts across all APIs within an AWS account, per Region. Steady-state: 10,000 reqs/sec., burst: 5,000 reqs/sec. Soft limits. Less than AWS throttling limits though. Regional APIs: 600, Edge-optimized APIs: 120**
- **Per-API, per-stage throttling limits (**Method level limit**) – applied to API method level for the whole stage. Limits can be either same for all methods on the stage in the API OR different for each method. Less than AWS Throttling limits.**
- **Per-client throttling limits – API keys and Usage Plans. Less than per-account limits. Set up usage plan + API key for a customer. API Key has to be part of the request. Set a limit on customer's usage plan. Usage plans – you can set throttling target for all methods of an API or all methods of a stage, or specific quotas for a specific method of a specific API resource**
- NOTE: Default Throttling limit ≠ Account limit
- NOTE: Burst = maximum number of concurrent requests, within a ms
- **If you exceed the limits – API Gateway returns 429 = "Too many Requests"**

Control Access to invoke your API

- Resource policy is set up in API Gateway on your API
- **Restrict by:**
- Principal: IAM user or IAM Role
- Source IP CIDR blocks
- VPCs, VPC endpoints
- **You can give access to other account to invoke your API**

API Gateway – Monitoring

- API needs to be deployed to a stage! If there's none, it prompts you to create one and deploy there
- **Stage – tab Logs/Tracing** – set up monitoring there
- **CW metrics**– collects metrics – how many times it was called, error 4XX, 5XX errors, CacheHitCount
- **CW logs** – records and saves logs – responses, callings, prints – **used for debugging of request execution (e.g. in LF)**
- **CloudTrail** – keeps track of actions in API Gateway taken by IAM User or role or any AWS service. Records are stored in CT S3 bucket.
- **X-Ray** – Records complete travel of your response – starting in API Gateway to LF to DynamoDB, useful for discovering the bottle-neck. Detailed analysis of visited services and times spent there

Lambda - CloudFormation – Inline LF

- Simple Lambda Function can be created using CF – Resource name, resource type – Lambda::Function, Properties (Handler, Role, Code – here you write the actual code, Runtime) - like this you can immediately upload the LF
- However, most LF are more complex and depend on packages -> Create deployable Package

Lambda – Deployment package

- **ZIP archive** – compiled function code + dependencies -> **1 Zip file for 1 function**
- Dependencies should be part of the ZIP file with the code OR they could a part of Layers
- 3 ways to update deployment package (zip file) to a LF
- 1. Create a deployable package (ZIP file) locally, use CLI to update the LF code – **aws lambda update-function-code –function-name *function-name* –zip-file *local-path-to-the-ZIP-file***
- 2. **ZIP file > 50 MB** – upload the code first to S3 (you can use a CLI), update the code: **aws lambda update-function-code –function-name *function-name* –s3-bucket *bucket-name* –s3-key *object-key***
- 3. Using CF – AWS::Lambda::Function – Properties – Code – **S3Bucket, S3Key, S3ObjectVersion**. Make sure to **change** at least one of the three – bucket, key or version – otherwise CF won't give it another unique identifier and will further use the older version. It might not even exist but it still won't update it

Lambda Quotas

- Function memory: 128 MB to 10 GB, step: 64 MB
- Function Timeout: default 3s, max 900s (15 min) – **if you have a batch program that runs LF for more than 15 minutes, then LF can't be used**
- Function environment variables: 4 kB
- Function Layers: **5 Layers**
- Function burst concurrency: 500 – 3000 – varies with region
- **Deployment Package size (ZIP files)** – 50 MB (zipped, for direct upload), 250 (unzipped) – This quota applies to all the files you upload, including layers and custom runtimes, 3 MB (console editor) – zip files > 3 MB can't be modified in console editor
- **Container image code package size** – 10 GB
- /tmp directory storage – 512 MB – as soon as the execution context is terminated, the directory would be deleted

- Execution processes/threads – 1024

SAM – Deployment with CodeDeploy

- 3 types of deployments
- Canary10PercentXMinutes – select X
- Linear10PerentEveryXMinutes – select X
- AllAtOnce
- Supports hooks that can call LF before and after traffic shift
- Supports CW alarms – when alarms triggered – rollback initiated

AWS AppSync

- Synchronizing app data across multiple devices (desktops, phones, laptops, IoT devices), e.g. one device changes something, this change should subject to all devices
- Syncing should work even after a device is worked with in off-line state and then is connected to the internet
- Great for real-time collaboration
- Based on GraphQL – creates APIs that only return the data the request/query asks for
- App uses this API to retrieve all kinds of data faster from all different AWS services
- App accesses and combines data from multiple microservices via a single API
- Replaces Cognito Sync that limits to storing only K-V pairs

AWS Step Functions

- AWS service to build **serverless** workflows as a **series of steps**
- Especially useful when the same order of steps/actions is done repetitively -> This saves time and you can change the code of the steps as necessary
- **Visual approach**
- Output of one step flows as input into next step
- You can orchestrate multiple services into the – invoke LF, run containers in ECS or Fargate task, get or put items from DynamoDB, publish SNS topics or produce SQS messages, start new SF workflow, and many more!
- **Max. duration – 1 year**
- **You can expose your step function through an API**
- **Include human approvals into workflows**
- Long and short running workflows – e.g. machine learning model training, report generation, IT automation, IoT data ingestion, streaming data processing
- NOTE: Data ingestion is the process of connecting a wide variety of data structures into where it needs to be in a given required format and quality. This may be a storage medium or application for further processing.
- **For more complex orchestration** - launching child processes, external signals – **Amazon SWF – Simple Workflow Service**

AWS Simple Workflow Service (SWF)

- Like Step Function but for more complex workflows - fully-managed state tracker and task coordinator in the Cloud
- Build and run background jobs with parallel and sequential steps, synchronously and asynchronously, with human inputs (can wait indefinitely for human inputs)
- **Use cases: order processing, video encoding workflows**

- Workflows up to 1 year
- Components (so called Actors):
- **Starter –** initializes the workflow, triggered by an event, e.g. an order
- **Decider** – take a task, check the history, recognize next step, return decision back to SWF
- SWF schedules an Activity
- **Activity workers** - Activity is picked up, processed, and result is returned to SWF
- SWF updates the history
- SWF asks Decider …
- Runs until decider return a decision to close workflow

X-Ray

- Tool to gather information and provide visualization, view, filter and gain insights
- Tracks requests across applications and AWS Services
- Especially useful when it passes through multiple apps/MS/AWS Services
- **Used for troubleshooting, solving performance issues, is something working as we expected?**
- Uses tools to visualize request path and times spent in services – **Service Map**
- Each client request is assigned **unique trace ID -> header: X-Amzn-Trace-Id:Root=12345**
- Each component/service in request chain sends traces to X-Ray using the trace ID
- **Reduce performance impact** – we do not send trace ID with every request -> **Sampling** – we set percentage of requests to carry the ID. **By default 1 req./sec + 5% of others.**
- **Sampling – reserved size - 60, fix rate – 20%**, 200 req./sec – 60 reserved + (200-60)*0.2 = 88 sampled reqs/sec.
- **For some AWS service is X-ray integration really easy – you just enable it – however, for others not so much, which is the case for you BE or FE application as well. Update Code to be able to trace requests to API Gateway or other HTTP web services OR calls to AWS Services from the app directly using X-Ray SDK. Then install X-Ray agent that gathers information from the code (that uses X-Ray SDK) and sends it in batches to X-Ray Service in the cloud.**

Step 1 – implementing tracing using X-Ray SDK

- **Interceptors** – catch requests before they reach the service, add them an ID in the header
- **Client handlers** – when your app calls AWS service directly, e.g. DynamoDB, instead use an SDK client, e.g. DynamoDB SDK. Enable tracing on the call that is sent to DynamoDB using the SDK client.
- **HTTP client** – As you are making a call to other HTTP web services, add tracing in the call.

Step 2 – Sending Traces to X-Ray Service using X-Ray Daemon

- Traces using X-Ray SDK cannot be sent directly to X-Ray Service
- **Traces are first sent to X-Ray Daemon**
- **X-Ray Daemon listens on UDP port 2000**
- Gathers raw data and sends them in batches to X-Ray -> causes that tracing doesn't have high performance impact on you apps
- **Using X-Ray Daemon on:**
- Lambda or Elastic BeanStalk – just Enable tracing, make sure execution role has permissions to send data to X-Ray
- EC2 – install Deamon on the instance, assign role with permissions to EC2 instance to send data to X-Ray, you can download Daemon matching your OS from S3

- On-Premises – like for EC2, just cannot assign a role, instead create an IAM User and give permissions to him -> you have to configure aws_access_key_id and aws_secret_key of the IAM User in ~/.aws/credentials inside the On-Premise machine to be able to make calls to X-Ray
- ECS – use Daemon container image - prebuilt container image – amazon/aws-xray-daemon, deploy this daemon container along your usual containers – **sidecar pattern**
- **Necessary permissions for the roles, user, … are:** PutTraceSegments – create segments in X-Ray, PutTelemetryRecords – upload trace data into those segments

X-Ray Trace Hierarchy: Trace > Segment > Sub Segment

- **X-ray enables you to trace request across multiple apps-components-services**
- Trace has number of segments and each segment has a number of sub segments
- **Trace**: A trace collects all the segments generated by a single request. Tracks the path of a request across Applications and AWS Services (using a unique trace ID)
- **(trace) Segment**: All data points related to a single component in the request chain – The compute resources running your application logic send data about their work. Segments records information about the original request, information about the work that your application does locally, and subsegments with information about downstream calls that your application makes to AWS resources, HTTP APIs, and SQL databases.
- Segment consists of **System-defined data** (data provided by AWS services) and **user-defined data** (annotations – user + system) + **sub-segments**
- **Subsegments** – granular details about remote calls made from a component – calls to AWS services, external HTTP API calls or calls to SQL DB
- **Annotations** – for storing some specific information to X-Ray, KV pairs with system or user-defined data. Annotations can be done on a segment or a sub-segment. Are searchable (=filter expressions) because they are indexed. Value can be only alphanumeric + underscore
- Metadata – associated with segment or subsegment – KV pairs with values of any type – also an object or an array. NOT searchable because they are not indexed. You can view metadata in the full segment document returned by the BatchGetTraces API.

AWS CloudTrail

- **Tracks everything done to your resources**
- Tracks and logs events, API calls, changes/actions made to your AWS resources (Any changes done in AWS are done through API calls)
- Who made the request, what action was performed, what are the parameters used, what was the end result
- Logs are delivered to S3 by default and/or to CW logs log group
- Use cases: Regulatory compliance to records what is happening to the resources, **troubleshooting – locate a missing resource, what happened to the resource? Why did it mysteriously disappear? Use CT to clarify it**
- You can set up SNS notification for log file delivery -> use this SNS notification to trigger a LF or just receive an SMS msg or an e-mail
- 2 types – Multi region trail, single region
- Multi Region Trail – One trail of all AWS regions, **event from all regions can be sent to one CW logs log group**
- Single – Only events from one region, destination S3 bucket can be in any region, CW logs log group for each region
- **Logs are automatically encrypted with S3 SSE**

- **You can configure log lifecycle in S3**
- **Log file integrity – you can prove, using CT, no alterations were done to a log file**

AWS Config

- **Auditing =** complete inventory **(tracking of AWS resource configuration)** of all you AWS resources to a certain time, like GIT for all your resources (records the **resource configuration/setting**)
- **Resource history and change tracking** – find how a resource was modified at any point in time, configuration of deleted resource would still be tracked/maintained, delivers history files to S3 every 6 hours, take configuration snapshots of your account resources right now
- **Governance** – continuous evaluate compliance against desired configuration – you can use customized Config Rules for a specific resource or all resources of an account, get SNS notification for every configuration change
- **Consistent rules and compliance across AWS accounts: Group *Config Rules* and *Remediation Actions* into *Conformance Packs***. These packs can be used across AWS accounts. Config Rule (**=what you want to check**) can be: check if there are any unused Elastic IPs in a region, because we would be billed for it. Remediation action (**=what action/fix should be taken**) would be to delete this extra elastic IP.
- Config Rules – **predefined**, **custom**
- Each Config Rule is associated **with a LF that evaluates whether the resources comply with the rule.**
- You can **set up auto remediation** (=fix for the issue) for each rule – take immediate action on a non-compliant resources -> also **associate with a LF**, I would guess
- **More rules to check -> more LF invocations -> more expensive**

AWS CloudWatch

- CW is a monitoring tool
- **Collects metrics, logs, and events, sets alarms**
- Set alarm to notify you or trigger an action (usually a LF)
- Integrates with many many AWS services – probably all the ones you know

CW Metrics

- Gather information about services – invocation counts, successes/failures, CPU and memory usage
- Free metrics – update every 5 minutes (for EC2 inst)
- Detailed monitoring (paid) – update every 1 minute (for EC2 instance)
- Create custom metrics for your application
- CW visualizes these metrics, allows you to set alarms on those metrics, and you can search through them
- Examples
- ELB – how many requests receives 2XX status codes, how many 4XX, is the ELB healthy?
- DynamoDB – read/write capacity utilization, how much did you provisioned
- EC2 – CPU utilization (NOT memory! For this you need installed CW agent), network in and out, health check
- Lambda – throttles, errors, concurrent executions, durations

- API Gateway – Cache hits and misses, 4XX or 5XX errors, number of requests, latency – total request time, integration latency – how long the BE took to respond
- Terminology: **namespaces –** container in CW to group related metrics – for single or multiple services**, metric** – time-order data point (= timestamp + value)**, Dimensions** – name/value pair associate with a metric, e.g. InstanceID=ec2-1234, AZ=eu-central-1
- **Custom metrics –** publish your onw metrics – use **PutMetricData** API – add data (metric name, unit, value), add dimensions. Your IAM role has to have permission to call this API. **Tracking frequency** – standard (default) – 1 min, high resolution – 1 sec. (expensive – more api calls to send data to CW) -> **used for high resolution alarms** – action within 10 or 20 secs.
- **Metrics exists only in the region in which they are created**
- **Metrics can't be deleted and expire after 15 months**
- CW doesn't have access to operating system metrics of EC2 instances **-> install CW agent to gather metrics around memory**

CW Agent

- **Install on EC2 instances/On-Premise servers to collect system level metrics – Linux or Windows**

CW Logs

- place to store **system logs, app logs and custom log files**
- **monitor (= see what is happening) and troubleshoot (= debug issues)** your application
- **real time app and system monitoring**
- **CW Logs Insights** – query your logs and get insights for your next decision/action, monitor patterns in your logs, trigger alerts if there's something suspicious
- **CW Container Insights** – monitor, troubleshoot and set alarms for you containerized applications – EKS, ECS, Fargate
- **Log term log retention** - store logs in CW – **default: forever**, **expiry** can be set **on log group level**, or archive logs to S3 – 12 hour delay
- **You can stream real-time data to ElasticSearch (now OpenSearch) custer using CW log subscriptions**

Collecting logs from EC2/On-Premises

- Install **unified CW agent** – one agent for any OS
- Give CW agent permissions to publish logs to CW
- EC2 Instance – attach CWAgentServerRole IAM role to it
- On-Premise instance – IAM User + configure AmazonCWAgent profile (profile has credentials that are stored in ~/.aws/credentials, profile is in ~/.aws/config, you'll get your access key ID and secret access key in AWS Console)

CW Logs: Metric Filters

- **Metric filters** turn logs into metrics
- You want to create metrics from ErrorCount from your error logs
- Filter works only for new events, it cannot process the older ones

CW Alarms

- Alarms can be created on any metrics (general + custom) -> it monitors them and react to it – e.g. triggers a lambda, sends a notification, execute an auto scaling policy

- It takes a while for alarm to set off
- **Alarms states**: **OK** – metric within the defined threshold, **Alarm** – metric outside of threshold, **Insufficient data** – not enough data to determine alarm state (some alarms need certain number of DPs to make a decision)
- **Configure alarm**
- **Period: time-length to generate one DP**
- **Evaluation Period: Number of most recent DPs to evaluate**
- **Datapoints To Alarm: Number of DPs in evaluation period breaching defined thresholds to set the alarm off**

CW Events, Amazon EventBridge replaces and extends CW Events

- **Resources can emit information (=stream event) when an action took place, CW records these streamed events and can react to it, e.g. call a LF, send a notification, start an instance**
- Examples
- If EC2 instance stops, send an email
- If build status in BuildCode changes, publish to SNS Topic
- If ASG reacts to demand and adjusts number of running instances, produce message to SNS queue
- Original idea: event based monitoring, if something happens, let me know about it
- **Now: Build event-driven architectures -> based on action that took place on a resource -> create and event -> and trigger another (reactive) action**
- Event-driven architectures – react to events from your app, AWS services and partner services
- Event structure is defined in **Schema registry** – Code bindings are available
- **Event Buses** – **receive the events** – **default event bus** for AWS services, **custom event bus** for custom applications, **partner event bus** pro partner applications
- **Rules** – **map events to targets** (IAM Roles need **permission** to do something on the targets)
- **Event Targets** – service that processes the event – **LF, SNS Topic, SQS Queue**
- Event and Event pattern – **Event patterns are used to identify events based on their structure and route them to targets**
- You can also send events via http requests where event is inside the payload. You have to configure API destination endpoint for the 3rd party software – partner software. You can use input transformer to change the payload format to match the target. For the API destination set up a connection – which is authentication method – user name + password or OAuth credentials or API Key. For debugging of your payload and headers and connection you can use official API destination on webhook.site

AWS CLI

- run actions against AWS service from your command line
- structure: aws <service> (wait) <subcommand> --parameters parameterValue –profile specifyProfileWithCredentialsYouWantToUse (set up in .aws/credentials)
- wait waits until the subcommand happens, e.g. aws deploy wait deployment-successful –deployment-id 12345
- example: list all buckets in s3 –> aws s3 ls
- **HELP:** aws help – general aws help, aws <service> help
- Important options:
- **--output:** change format of output – json, yaml, text, table

- **--no-paginate:** display only first page, by default CLI retrieves all pages
- **--page-size:** avoid "timed out" error – number of items to return at a single response, default: 1000, specify less to avoid a time out.
- **--debug:** activate debug mode, it will print out a lot of info about what is happing in the background
- **--dry-run:** checks permissions, no actual request is sent
- **Logging-in**: create a credential file by "aws configure" – creates **credentials** file with the [profileName] and credentials = access and secret keys and **config** file with region and default output format
- **NOTE: Access keys for every IAM user – max two pairs, standard: only one active**
- Linux: ~/.aws/credentials + ~/.aws/config,
- Windows: C:\Users\USERNAME.aws\credentials
- You don't want to aws configure you IAM access keys on EC2 isntances because they will get stored there and anyone who accesses the instance can be in possession of them –> create a role with those access keys and assign it to the EC2 instance (and use instance profiles) – give the role permission to do something. Then your EC2 instance doesn't need your credentials, it has its own granted permissions
- **AWS CLI – Profiles: handle access into multiple accounts,** configure aws profile – access keys + region + output format -> **aws configure –profile profileName**, to use the profile in CLI call: **aws s3 ls –profile profileName**

AWS CLI call configuration settings and precedence

1) **Command line options:** --region, --output, --profile, override defaults
2) **Environment variables:** export ACCESS_KEY_ID=1234, setx ACCESS_KEY_ID=1234 (Windows)
3) **CLI credentials file:** credentials in ~/.aws/configuration
4) **CLI configuration file:** region + output format in ~/.aws/config
5) **Container credentials:** IAM role credentials from role assigned to ECS Tasks – Task definitions
6) **Instance profile credentials:** IAM roles credentials assigned to your EC2 instance

Using IAM role in CLI – On Premise

- Define profile for a role in ~/.aws/configuration defined locally
- Aws iam create-role –role-name roleName –assume-role-policy-document nameOfPolicy OR create a role in AWS Mgmt Console and add the role_arn and source_profile (OR credential_source) in the file manually
- **[your_user_profile]** has **aws_access_key=123** and **aws_secret_key=456**
- **[profile role]** has **role_arn = arn:aws:iam:XYZ:role/myrole** and **source_profile = your_user_profile** (an IAM user profile) OR **credential_source = Ec2InstanceMetadata or EcsContainer or Environment**
- Execute command using a role: **aws s3 ls –profile myrole**
- When you use role instead of an IAM user**, the CLI still uses the credentials of the source_profile user or credential_source** (there IAM role attached to instance profile of to EC2Instance or the container)
- CLI uses **sts:AssumeRole**: **IAM user/role** should have a permission to assume a role (sts:AssumeRole), **trust policy** – set on the IAM role to allow IAM user to be able to assume it

AWS Security Token Service (STS)

- Calling STS API allows **trusted users** to retrieve **temporary security credentials (roles)** with them can access AWS resources

- Works for any AWS resource, e.g. EC2 instance or any user, both can assume a role, if defined
- **Tokens are temporary credentials**, have expiration date (IAM users have permanent credentials)
- STS can be called even when authenticated by web or corporate identity federation
- STS global and regional, regional has better performance
- **APIs to get temporary credentials from STS:**
- **AssumeRole** – cross-account federation or AWS services (IAM user or IAM role with existing temporary security credentials) authenticated with a custom identity broker can assume role**, MFA possible, Duration: 1 hour**
- **AssumeRoleWithSAML** – for any users authenticated with enterprise identity broker, **MFA not supported, Duration: 1 hour**
- **AssumeRolWithWebIdentity** – for any users authenticated with Cognito or Amazon account or Social or any OpenID connect compatible identity provider, **MFA not supported, Duration: 1 hour**
- **GetFederationToken** – **Only for IAM users or AWS account root user can assume a role, MFA not supported, Duration: 12h. Allows you to use the role in Mgmt. Console for 12 hours**
- **GetSessionToken** – same account access with MFA. When MFA required for an API call, you have to call this API and pass your MFA code to the call as well, if code correct, it will return you credentials (a token) and then can make a call for another action that is MFA protected. **Temporary credentials for users working in untrusted environments. Use your own AWS IAM user credentials to generate temporary AWS credentials that allow you to communicate with AWS services. Only for IAM users or AWS account root user, MFA is supported, Duration: 12h**
- **DecodeAuthorizationMessage** – if you API call fails it typically returns encoded error message, use this API to decode the message and debug the error
- **GetCallerIdentity** – returns IAM user identity or role based on credentials provided in the API call

CORS Recap

- **By default browsers don't allow your frontend application running on one origin (domain), e.g. www.vojta.com to access resources (app BE or APIs) from a different origin (domain), e.g. api.vojta.com**
- Browsers send a preflight OPTIONS request to the other origin and get a response that need to contain allowed origin from which you send the request (here www.vojta.com), allowed HTTP methods and necessary headers, e.g. Authorization. If response doesn't contain the domain of the origin in allowed origins, normal request will fail with CORS
- How to allow CORS on AWS resources – S3 and API Gateway

CORS on S3

- FE web app is hosted on one S3 bucket (fe-bucket)
- Images it needs are hosted on a another bucket (images-bucket)
- Configure CORS configuration on images-bucket allowing access from fe-bucket URL
- **Go to images-bucket – permissions – CORS configuration -> you'll get a sample template**

CORS in API Gateway

- FE web app is hosted on S3 bucket and calls an API exposed by API Gateway

- On the API you have to enable CORS – either for the whole API or specific resources and methods
- REST API Lambda custom (non-proxy) integration – click Actions and enable CORS, configure CORS
- REST API Lambda proxy integrations – click ACTIONS and enable CORS, make changes to LF to return header with allowed origins, methods and headers.
- HTTP API – On the left enable CORS and configure CORS there

Configuration Management

- **Configuration management should help to set different values within your app based on environment where you want to deploy.**
- Passwords for all three environments should NOT be written in the code like if stage = dev, then psw = dev-password, else if stage = test, then psw = test-password. Use outside configuration – LF Environment Variables or Systems Manager Parameters Store
- Application configuration for each environment should be used when deploying to this environment. E.g. for dev use DynamoDB Table database-dev, all aws resources created should have a tag = dev designating current stage
- Part of the configuration can be, if you want to encrypt something, e.g. encrypt and version a bucket when on PROD

AWS Lambda – Environment variables

- **Key value pairs directly associated with a LF**
- Use them in your LF code, in JS: process.env.NameOfVaraibleName
- Reserved Environment Variables – set during initialization – AWS_REGION, AWS_LAMBDA_NAME, AWS_LAMBDA_FUNCTION_VERSION, AWS_LAMBDA_GROUP_NAME, many more
- **Environmental Variables are locked when a LF is published (=deployed),** if you want to change them, you change them in a code and publish the LF again
- You can **encrypt all environment variables using AWS KMS**

AWS Systems Manager Parameter Store (SSM Parameter Store)

- Manages Application Configuration and Secrets – externalizes configuration: app, build, and CI/CD pipeline configuration
- Externalized storing of environment variables
- Hierarchical – Create different environment variables for different environments - for dev stage set all dev env. variables, for test set all test env. variables – **Is determined by the path for the environment! E.g. /dev/service-to-which-it-belongs-to/VARIABLE_NAME**
- Types: string, list of string or secure string (KMS encrypted)
- **Maintain history of configuration over a period of time**
- **SDK - Retrieve a parameter: ssm_client.get_parameters(Names=[‘parameterName1’, ‘parameterName2’])**
- You can change configuration and LF will immediately apply it WITHOUT republishing
- Everything is monitored by CW, notifications can be sent on change using SNS, every change is recorded using CT
- Integrates with:
- KMS to encrypt parameters – key from AWS, if you want to restrict access, create your own

- Secret Manager – powerful password & secrets manager – automatic rotation
- CF, CodeBuild, CodePipeline, CodeDeploy – stored parameters can be made use of in build and deployment tools

AWS Secrets Manager

- Best place to save passwords for your app!
- Secrets: credentials, passwords, API-Keys, something what you want to keep save from others
- **Encryption of secrets by KMS**
- **You pay for it!**
- **Most useful operation:** automatic rotation of secrets without impacting the application – it doesn't impact the application run because the password change is automatically change at the place of configuration and configuration is where the app takes the parameters from. (Typically if you manually change a password, e.g. for database, you need to manually propagate this change to the configuration. Easy if you have a couple of passwords on a single account, it gets very difficult with many) – **SUPPORTED BY:** Amazon RDS, Redshift, DocumentDB (like DynamoDB, however, it is a managed service around MongoDB)
- Monitoring by CW, Notifications by SNS, auditing by CT

Caching

- **Reduce load on your DBs, improve application performance**
- Best usecases: caching **infrequently changing data**, **same data accessed often**, caching **user sessions** from apps (each EC2 instance would store session, you can **use a centralized cache**)
- People frequently request the same data, instead of performing the calculations and retrievals from DBs again and again, you temporarily save them results in a cache. Data can be then immediately sent to the client without further calculations.
- How long will the response live in the cache – set TTL (Time-To-Live) – the less data changes, the longer TTL can be and vice versa
- TTL prevents long term stale data

Caching Strategies

- 2 strategies: lazy loading (=cache aside) and write through

Lazy Loading

- If data is found in a cache, value from cache is used (CacheHit)
- If data is NOT found in a cache, calculations are performed, data are retrieved from DBs and send back and added to the cache (CacheMiss)
- All the frequently accessed data would be living in the cache

Write Through

- New incoming data that we want to store (**WRITES**) will be processed and response will be automatically added to a cache
- Cache is in sync with the BE -> all data in DB is present in the cache
- E.g. I want to save something to a DB, DB gets updated, it creates a record, and this record is our response. Response is added to the cache

Factor	Lazy Loading	Write Through
Stale data	Possible – configure TTL	Never stale

Node failure – distributed caches have number of nodes	If fails, we only lose data in the cache, cache is out of service and calculations performed	Cause failure because new data cannot be stored in the cache (prevention: replication of cached data/nodes)
Cache size	Low – only requested data is cached	High – all data is cached, as big as the DB
Reads	Can involve 3 steps 1) check the cache, if nothing, 2) retrieve it from BE and 3) update cache	1 step – reads directly from cache
Writes	Update DB only	2 Steps – 1) update DB, 2) update cache

Amazon ElastiCache

- Managed service
- **Highly scalable and low latency in-memory data store**
- Used for **distributed caching**
- **Cluster engines**: Redis, Memcached
- **Supported caching strategies**: Lazy Loading, Write Through
- Always **in front of DB** (DB layer), not API (Http – app. layer)

ElastiCache with Redis cluster engine

- Highly scalable and low latency in-memory data store
- **Complex storage ("memcached on steroids")**, provides more than caching
- Can be used as **a cache, DB or message broker (queue)**
- **Persistent Cache (contains as many data as a DB)**
- **Automatic failover to other nodes with Multi-AZ deployments (if enabled) –** there is a master node + read replica nodes, if master fails, other node can be promoted to master
- Supports **back-up and restoration of the service (cache/DB/Broker)**
- Supports **encryption at-rest (KMS) and in-transit**
- **Use cases:** cache, session store – all session from all apps in one place - centralized distributed cache, message broker – chat and messaging apps, or queues, gaming leader boards, Geospatial Apps – rides – Uber, Restaurant recommendation

ElastiCache with Memcached cluster engine

- **Pure caching**
- **Non-persistent -> pure cache**
- Key-Value storage
- **Fast session store -> Transient** session **store** – if node fails, you can lose session data
- **Ideal caching solution for data stores like RDS (DAX for DynamoDB)**
- **Distributed cache (=horizontal scaling with auto discovery – can connect and modify the actual nodes)** – up to 20 nodes
- **Not supported**: **back-up and restore**, **encryption** or replication, snapshots
- When node fails, all data in the node is lost -> to reduce impact of failure, have as many nodes as possible (max 20)

DAX (DynamoDB Accelarator)

- Cache for DynamoDB – couple of line of code to integrate
- ElastiCache can be cache for any DB – lots of code changes required to make it work

Caching Application Session

- If you want to cache application sessions -> create distributed session store
- Options
- ElastiCache-Redis: microseconds responses, node persistence – replication, backups, restore (durability)
- ElastiCache-MemCached: microseconds responses (data retrieved from in-memory), no persistence -> if node fails, sessions lost
- DynamoDB: slower than caching solutions -> milliseconds, persistent – replication, backups, restore (durability)

Amazon S3 Queries

- S3 and S3 Glacier – run your analytics directly from them
- **S3/S3 Glacier Select** - You can run SQL queries to retrieve subset of data – JSON, CSV, Apache Parquet - allowed formats -> **results of the queries are stored in S3**, **SQL select can be run as a part of your LF -> results of the queries are assigned to a variable in the code**
- **Recommended storing format is Parquet** – reduced storage by 85 %, improved querying up to 99 %
- You can also **compress the data** -> **GZIP recommended – supported by Athena, EMR, Redshift**
- Amazon **Athena** – direct SQL querying again data saved in S3, Athena uses a data-querying framework Presto – allowed formats – CSV, JSON, Apache Parquet, Avro
- Amazon **Redshift Spectrum** – run queries directly against S3 without loading complete data from S3 into a data warehouse, **RECOMMENDED if you are doing frequent queries against structured data**

AWS Service Quotas

- Every account has region-specific default quotas and limits for each service
- AWS Service Quotas – managed service for your quotas – there you can see all the quotas based on your current region

AWS Directory Service

- User data management service
- Provides AWS access to on-premise users without creating IAM users
- Can be deployed **across multiple AZs**
- **3 options**
- **AWS Directory Service for Microsoft AD** – if you have more than 5000 on-premise users, create a trust relationship between the local on-premise authenticating user directory and AWS AD -> **you can than add new/delete/modify users in the AWS service and it will be projected to the on-premise directory and vice versa**
- **Simple AD** – for less than 5000 users, powered by framework Sambda4, **also compatible with Microsoft AD,** does NOT support trust relationship with other AD domains -> this won't allow you to manage the users of the connected directory in the Simple AD service -> for this go with the 1. option

- **AD Connector** – use your existing on-premise directory with AWS cloud service – you don't want to create a new AD service within cloud but only extend the on-premise directory to cloud -> **users can use existing credentials for their on-premise AD to access AWS resources -> you manage users in the on-premise AD**

AWS Global Accelerator

- To prevent Cached DNS answers – when we access a domain, route 53 can map this domain to more than one target (servers or LBs), if the cached target fails, customer won't be able to access the other target because it has cached the first one. Or even though there's an update in Route 53, the customer won't get the update because it has cached old configuration
- To prevent high latency based on cached DNS answers – it communicates with only the one region that it has cached – if there is a new server somewhere closer, it will still ignore it
- For this use Global Accelerator -> once request hits AWS, AWS directs traffic to optimal endpoints to speed things up
- **You get 2 static IP addresses (not addresses of the final targets-here LBs) – those are edge location IP addresses** – low latency ensured, you can distribute traffic across multiple AWS endpoint resources in multiple AWS region
- **Internal routing with global accelerator work very well with NLBs, ALBs, EC2 instances,** Elastic IP addresses

S3 bucket policies

- Require HTTPS connection – "Condition": { "Bool" : { "**aws:SecureTransport**": "false" }}, "Effect": "**Deny**"
- Enforce use of encryption with KMS – based on the attributes of the incoming request you can recognize if it is encrypted using KMS or not: "Condition": { "StringNotLikeIfExists": {"s3:x-amz-server-side-encryption-aws-kms-key-id": "YOUR-KMS-KEY-ARN"}}

ADDITIONAL TEXT used for exam

AWS Batch Service

- Monitoring of progress CW Events

AWS DataPipeline

- Sfsdf

AWS Quicksight

- Quick visualization of data from a DB or another AWS service

AWS Inspector Service

- Sdfsdf

AWS Trusted Advisor

- Provides real-time guidance to help to provision AWS resources and follow AWS best practices. It can perform overall system utilization (performance report) but is not used to identify performance issues

AWS SSO

- Manage access to multiple AWS accounts and other account (GitHub, Jira, JFrog, DropBox) from a single place.

AWS Redshift

- Warehouse, OLAP DB
- Cluster should be encrypted at its creation but it is possible to encrypt the DB at any time – AWS handles migration of unencrypted data to new encrypted cluster
- Once encrypted DB setting cannot be reversed

Step Function

- **Activity workers should have a set timeout** otherwise they never stop running until a year passes
- When LF – activity worker – returns Service Exception – use lambda retry code when the correct error is returned

Cognito

- Identity Pool – the **role** for authenticated users have **a trust policy – specifying who can assume them and permission policies – what the users can do with it in AWS**

RDS

- **You create a DB**
- **At creation time you can create a standby**
- **After creation – click actions – create RR**

RDS creation

- **Restore RDS from S3 snapshot**
- **DB parameter group – container for engine configuration values that are applied to one or more DB instances**
- **DB cluster parameters groups – only applicable to Multi AZ DB clusters**
- **Regular backups – set how long to keep them, set back up window**
- **Specify VPC and subnet group – which IP addresses DB instances can use**
- **Create a standby in another AZ – synchronous replication**
- **IOPS – choose how many I/O actions per second your DB instance should support – 1000 – 256 000**
- **SSD – general purpose – less IOPS, with growing allocated storage slightly grows IOPS, provisioned IOPS – specify IOPS you want to have, fastest storages – 1000 – 256k, magnetic – not recommended – max 1000 IOPS and 3TB storage**
- **Enable storage autoscaling – serverless – just specify min and max limits**
- **DB classes – run on EC2 instances – standard (M), memory optimized (r, x), burstable (t)**

Redis

- **Parameter groups – common values across nodes for memory usage and item size**

Enabling X-Ray

- **LF – via CLI, LF needs permissions AWSXRayWriteOnlyAccess**
- **API Gateway – checkbox in stages**

X-ray

- **Encryptes data at rest – SSE AWS owned or you can supply your own CMK**

DynamoDB

- **Query – primary key always equal to a value, sort key can contains certain actions like greater than, between = key condition expression**
- **Queries can use project expressions**
- **Scan – project expressions**
- **Scan, Query – filter expressions – are applied after the query/scan – so no saving of RCUs – it just returns more restricted list of found items**
- **Partition and sort keys types: string, number, binary**
- **Enable Kinesis Data Stream on a table – Table captures item level changes and replicates them to the DS. You can consume the stream with a LF – charges apply. ENABLE -> choose a stream -> DONE**
- **DynamoDB Streams – Table captures item level changes and pushes them to a DynamoDB stream. You can access them through DynamoDB Streams API -> types: key attributes of the changes item, complete old item, new item or both -> stream created ARN -> set up TRIGGERS – choose a single LF and assign batch size, you can create more than 1 trigger**
- **GSI only support eventual reads**
- **LSI and GSI are also called just "index"**
- **NO RESOURCE BASED POLICIES**
- **Encrypted at rest – at creation or whenever after – SSE – DynamoDB – free, SSE-KMS (AWS managed) – paid, SSE-C (AWS KMS Customer key) – paid**
- **Export to S3 – Point In Time Recovery has to be on**
- **CW metrics, alarms, insights possible**
- **To get total RCU how much a query consumed using CLI – use flag ReturnConsumedCapacity and set it to TOTAL**
-

Aliases

- **Point to a specific published version**

LF, Alias

- **LF publish version – once the version is published it cannot be modified and it receives an unchanging ARN**
- **Qualified ARN – publish version or a specific alias, denoted by ":versionNumber/Name"**
- **Unqualified ARN – points to a LF – always $LATEST version**

CF helpers

- **cfn-init – additional setting for your EC2 instances, fetch and parse metadata from CF stack and its resources, can fetch them also from another stack if credentials available and allow it.**
- **cfn-signal – Signal back to CF that certain action was performed – EC2 Instance successfully created,software configuration finished, end of creation policy – Wait state in ASG Lifecycle**
- **cfn-get-metadata – fetch stack metadata to a standard output, also can specify subtree of the stack using a key**
- **cfn-hup – daemon that detects changes in resource metadata and runs user-specified actions**

SQS

- **for proper scaling of LF – standard queue should use long polling, FIFO queue – message group ID – used for ordered message delivery**
- **Use SDK, not a console, to create SQS with S3 bucket for items greater than 256kB. You can set if all items or just ones greater than 256kB should be moved there.**

Amazon S3 encryption:

a) **using KMS key – 1. Client – requests keys from KMS Generate API - request for each item, 2. Return two keys – symmetric plain data key and its encrypted version (delete the plain data key ASAP from memory), 3. Use Key algorithm, plain key to encrypt the object, encrypted keys is included in item metadata. 4. Send it to S3, 5. Upon retrieval, send the encrypted key to KMS – it returns corresponding plain data key, 6. Use the key to decrypt the item**
b) **Using Root key stored in your app (symmetric or asymmetric) – 1. S3 encryption client generates a symmetric key – data encryption key – used for encryption of the item, 2. S3 encryption client uses the root key to encrypt the data encryption key – again part of the item metadata = material description – sent as x-amz-metadata-x-amz-key header, 3. Encrypt the object and send it to S3, 4. Download the object, check the material description to know which root key to use to decrypt the encrypt key -> get decrypted key -> use to decrypt the object**

API Gateway

- **CORS – for all response except 200, we have to specify Access-Control-Allow-Origin = * or specific origin in OPTIONS method to fulfill the pre-flight handshake – 4XX – Allow-Origin = *, 5XX – Allow-Origin=***

API Gateway – Canary Release

- **You can test another version of the function – you will be using stageVariables to point to the correct one**
- **In the stage – tab totally on the right – activate canary release. Make changes to your function. Deploy the function but to the same stage but you will see there that canary is enabled. Override stage variables, set ratio of traffic between the two functions. Same invocation of the endpoint will forward the request to a function based on the split ratio**

ECS + X-Ray

- **In Task Definition – specify secondary container with X-Ray Daemon image, sidecar pattern. Container definitions should have necessary permissions to publish the collected X-Ray data to the cloud service.**
-

Application on On-Premise

- **On-Prem server has an IAM user which can assume a role – using SDK and securely communicate with AWS resources. Or just retrieve a session token – GetSessionToken.**

S3

- **is a global service but tied to a region to which you deploy it**
- **bucket name has to be unique within a region**

- **If S3 has enabled SSE, you can send and item and in header x-amz-server-side-encryption specify default SSE or a specific KMS key -> this will encrypt the uploaded item using the specified algorithm or key**
- **Eventbridge event notifications – generates event for any action on the bucket – enabled/disabled**
- **Event notifications – set up for what events and event should be produced, you can specify prefix or suffix, destinations: LF, SNS topics, SQS queue**
- **Server access logging – can be saved on the same bucket but typically on another one**
- **Set up data replication or object lifecycle**
- **DEFAULT ENCRYPTION: SSE-S3 (S3 managed keys) and SSE-KMS (AWS managed and customer managed)**
- **Another Encryption: CSE – encryption on clients side – e.g.Amazon S3 Encryption agent**
- **Web hosting – enable WebSite hosting, configure index document, permission for website access on the bucket**

CloudFront

- **Secure encrypted traffic between the viewer and CF – Viewer Protocol Policy – HTTPS only or redirect HTTP to HTTPS and**
- **CF and Origin – Origin Protocol Policy – HTTPS only or Match Viewer – when Viewer Policy set as above**

Route 53 weighted routing

- **Form of blue/green deployment**

EB

- **One of the deployment types: blue/green deployment – Behind ELB are two separate environments – each has an ASG**
- **Deployment package needs to be at the root level of the working directory – deployment package includes all source code, its dependencies, executable files, and others**
- **Requires AMI image for the underlying servers + region where to deploy should correspond with the AMI image save location**
- **When no fitting AMI image found, you can create you custom platform using Open source Packer – platform from scratch – OS, software, scripts, custom metadata**
- **Use dockerrun.aws.json version 2 to run Docker images in EB**
- **EB – create App – app can have multiple environments – environments have multiple application versions**
- **Single ZIP/WAR file – deployment package (source bundle) < 512 MB, cron.yaml – in the source bundle – period operations**

Lambda – LF

- **Every LF needs to have HANDLER – this is what is called to invoke/run the function**

DynamoDB – RCU/WCU – transactions – isolated and atomic operations

CodePipeline

- **when using console wizard or SDK, you set up S3 bucket for artifacts**
- **Encryption on the buckets – AWS**

- **AWS owned and managed keys (cannot be changed, rotated or deleted) or customer managed (can be changed, rotated and deleted) – both use symmetric keys for encryption**

CodeArtifact – repository for packages, like JFrog

CodeCommit

- **repository for you source code, images**
- **CVS – Control Version System**
- **Encrypted at rest by KMS**
- **Access CodeCommit – a) Access keys of the IAM + CLI – credential helper, b) GIT username + psw generated in AWS you can store and use for codeCommit, c) SSH into codeCommit – upload public access keys from you SSH asymmetric key**
- **Supports NO RESOURCE BASED POLICIES**
- **Tags to CodeaCommit resources – very useful**
- **Notifications – open PR, comments, creating branch, merging, commit – any action produces a notification – destinations: LF, SNS, EventBridge + SNS**

CodeBuild

- **Compilation of code -> build, runs tests, produces artifacts – libraries or built federated modules – published outputs of the build**
- **Scaling of build servers**
- **Like JENKINS**
- **Pay for length of builds only**
- **Builds are run in preconfigure environment – OS, programming language runtime, build tools – npm, maven, docker**
- **BuildSpec.yaml – build instructions – at the root of you deployment package or uploaded during creation of CodeBuild Project**
- **CW metrics – detect outcome of a build**
- **Optional encryption of artifacts**
- **CW build logs – S3, CW**
- **IAM role for build permissions – e.g. build project needs to access RDS in a private subnet – specify VPC, subnets and ID of SG of the RDS – additional configuration required**
- **Secrets – env.variables or mainly SSM parameter store**
- **Run CodeBuild locally – Docker + CodeBuild Agent -> better debugging**
- **CodeBuild Project – name**
- **Code Source – GIT, S3, CodeCommit**
- **Environment – image of build servers – a) image by AWS CodeBuild + OS (Amazon Linux 2, Ubuntu) + runtime, b) Docker image, environment type (ARM, Linux, Linux GPU) + SERVICE ROLE**
- **Additional – build timeout, certificates, compute setting – memory, CPU, VPC it wants to access, env. variables**
- **BuildSpec file – YAML only**
- **Artifacts – saved to a bucket or no artifacts produced, artifacts – ZIP or not, encryption.**
- **Logs – CW and/or S3**
-

CodeDeploy

- **EC2 Instances and On-Premise server have to have CodeDeploy agents**

- **Deploys apps to EC2, on-premise, ECS and LF**
- **Source code files, packages, executable files, multimedia files**
- **Similar tools – Ansible, Terraform, Chef, Puppet**
- **WHAT to deploy, WHERE to deploy, HOW to deploy**
- **Deployment Groups**
- **Deployment types**
- **Deployment configuration**
- **AppSpec – YAML or JSON, at the root of the**
- **HOOKS – at certain steps of deployment you can run scripts**
- **CodeDeploy Agents**
- **Revision – updating to a new version, new revision includes everything to entirely replace the old version – AppSpec, app fioles, executable files, configs - make sure the bundle is not missing anything => DEPLOYABLE package + AppSpec file => Archive file – ZIP**
- **App name**
- **Compute platform: EC2/On-Premise, LF, ECS**
- **Deployment group EC2/On-Premise – service role, deployment type – in-place On-Premise, in-place/blue-green fro EC2 instances, env. config – EC2 ASG or EC2 instances or On-Premise servers and their tags, deployment config – allAtOnce, canary, linear, ELB – ALB, NLB – port mapping – choose target group, advance: triggers, alarms, rollbacks – rollback when deployment fails, alarm threshold is met – automatic/manual**
- **Deployment Group LF – name, service role, blue-green always, deployment config – canary, linear, allAtOnce, advance: triggers, alarms, rollbacks**
- **Deployment Group ECS – name, blue-green always – new Task Set created, service role, env. config – choose ECS cluster, ELB + target group, deployment config – all at once, canary, linear. Advance: triggers, alarms, rollbacks**

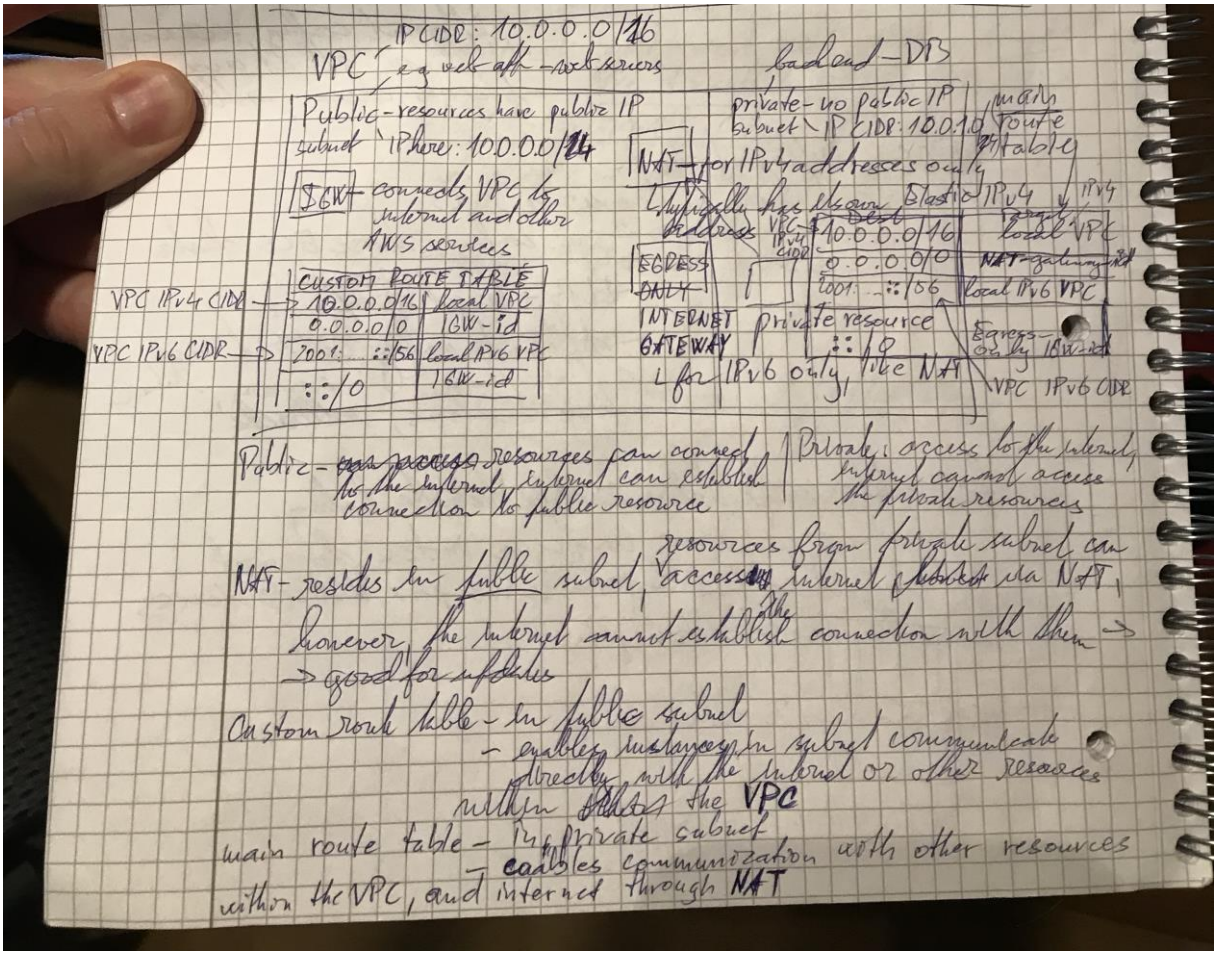
On-Premise servers can only have IAM user credentials – Access and Secret access key and don't have instance profiles as EC2 instances to which a role could be assigned.

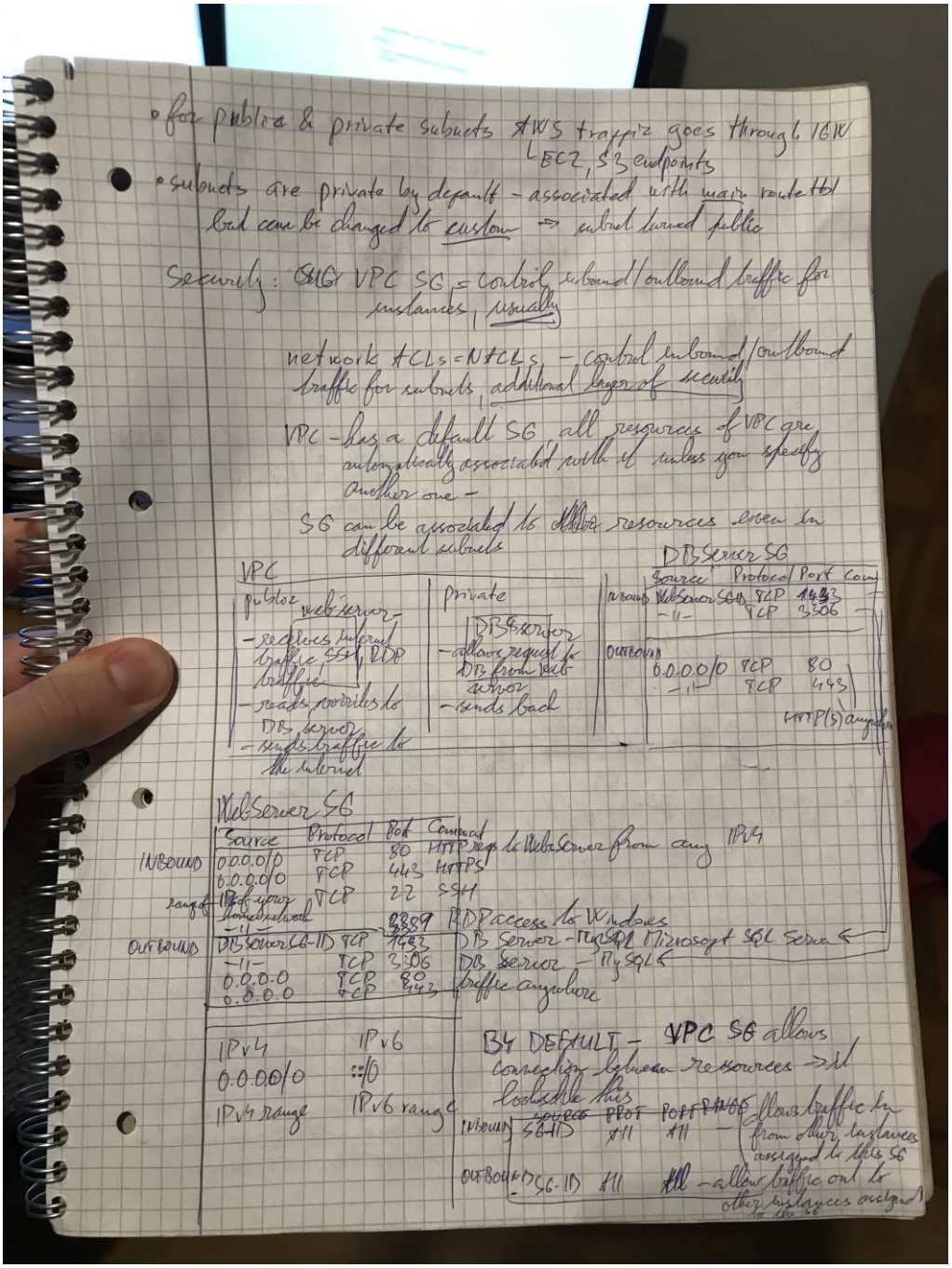
EC2 instance with ECS container and inside is a container agent – when instance is stopped, container status is still ACTIVE, container agent's status is false

IAM roles are distributed credentials rotated regularly – access token, refresh token, expiry, session token.

OpsWorks – configuration of provisioned resources – Chef, Puppet

NAT - https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Scenario2.html

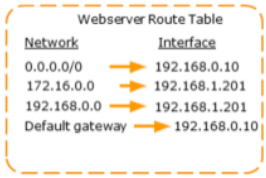

IP CIDR: 10.0.0.0/16
VPC
e.g. web app + db servers
backend - DB's
Public - resources have public IP
subnet IP here: 100.0.0/24
IGW - connects VPC to internet and other AWS services
CUSTOM ROUTE TABLE
VPC IPv4 CIDR
10.0.0.0/16 local VPC
0.0.0.0/0 IGW-id
VPC IPv6 CIDR
2001: ::/56 local IPv6 VPC
::/0 IGW-id
private - no public IP
subnet IP CIDR: 10.0.1.0
main route table
NAT - for IPv4 addresses only
typically has always blocked IPv4
addresses
EGRESS ONLY INTERNET GATEWAY
for IPv6 only
private resource
::/0
like NAT
Dest.
10.0.0.0/16 local VPC
0.0.0.0/0 NAT-gateway-id
::/56 local IPv6 VPC
Egress-only IGW-id
VPC IPv6 CIDR
Public - resources can connect to the internet, internet can establish connection to public resource
Private - access to the internet, internet cannot access the private resources
NAT - resides in public subnet, resources from private subnet can access internet via NAT, however the internet cannot establish connection with them
-> good for updates
Custom route table - in public subnet
- enables instances in subnet communicate directly with the internet or other resources within the VPC
main route table - in private subnet
- enables communication with other resources within the VPC, and internet through NAT



ENI - https://www.cloudsavvyit.com/3993/what-are-aws-elastic-network-interfaces-enis-and-how-do-you-use-them/#:~:text=Essentially%2C%20ENIs%20are%20virtual%20network,communicate%20on%20two%20different%20subnets

Virtual network card for EC2 instances -> allows the EC2 instance to send and receive traffic
One by default
Two – 2 network itnerfaces can connect to two networks at once -> enables possibility to route traffic to another subnet – can host LB, proxy servers, NAT servers
ENI have SGs just like EC2 instances – like firewall

Common use – creating of management network



Network – odkud to skutečně chodí
Interface – jak se to v rámci VPC prezentuje, přes co to tam přichází

ENIs are also often used as the primary network interfaces for Docker containers launched on ECS using Fargate.