Dexter's "Barely passed" AWS Cram Notes

Access Management Basics – A principal is any entity that can make either an authenticated or unauthenticated request to a resource. Authentication is the process of validating a principal against an identity they present, which proves they are who they say they are. Once authenticated, a principle is associated and represented as an identity on the system they've authenticated to. Authorization then determines whether that authenticated identity can access various resources within that system.

Shared Responsibility Model – You share responsibility when deploying solutions on AWS, security in the cloud is your responsibility, security of the cloud is AWS' Thus AWS manages all the low level physical infra, DCs, and regions to provide a service. You still own whatever data and/or config you perform within AWS itself.

Cloud Service Models – *IaaS* – EC2, Customer manages everything above the VM (OS-Runtime-App-Data) but AWS manages anything below the virtual guest OS. *PaaS* – You bring your apps and runtime, AWS executes it for you, they manage anything OS and below. *SaaS* – You bring your data and business, everything else managed for you. *FaaS* – You give a function to execute for a specific task, AWS handles everything else for you.

HA vs. FT – HA – designing a system to quickly recover in case of a failure but doesn't prevent a failure from happening in the first place. FT – designing a system to run through a failure. Priority of HA is to recover from a failure quickly, while FT aims for zero-interruption in the event of an issue. FT is more expensive b/c of more resources required to deploy. Choosing btw 1 or the other depends on business needs.

RPO vs. RTO – RPO – how much data can you use in an adverse event, this would determine the frequency of backups and snapshots taken. RTO – how much time do you have to recover systems back to operational state before the business starts to realize financial impacts. Higher RTO or shorter RPO = higher cost. Each AWS product has its own RTP/RPO SLAs.

Scaling – allowing a system to size up or down with changing demands, statically. Horizontal scaling is when new servers are added to an existing workload cluster for load-balancing. Vertical scaling is when you up the spec on an individual server. Horizontal scaling has no size limitations but the app must support multi-server processing = more complexity. Vertical scaling is limited to the physical hardware capacity on the box (you can only add so many CPUs on a motherboard).

Elasticity – First came vertical scaling, then came horizontal scaling, then came Elasticity. Elasticity is an upgrade from both vertical and horizontal scaling where you provision just the right amount of resources based on demand for that particular point in time. It eliminates the stepwise effect of scaling in a static manner. The way you achieve elasticity is to use automation tools like launch templates, launch configs, auto-scaling, or use AWS serverless products that scale dynamically to demand.

Tiered Application Design –A monolithic app is when you have all the presentation, logic, and data tiers coded into a single piece of code – not scalable, hard to debug and maintain. Vertical scaling is only possible w/ monolithic apps. Tiered apps split out their presentation, logic, and data tiers as separate pieces of code that can run on diff. infra components and allows each to be able to scale independently, allowing for higher flexibility and easier maintenance.

Encryption – You either encrypt at-rest or in-transit. You can perform either symmetrical or asymmetrical encryption. Symmetrical encryption is when the same key is used to encrypt-decrypt data, is fast but is harder to exchange keys btw the 2 parties. Asymmetrical encryption is when you use a private and public key. Sender uses your public key to encrypt, you use your private key to decrypt. You can use asymmetric encryption as means to exchange keys to use in symmetric encryption of the actual data payload.

Design Pillars of the AWS Well-Architected Framework – Operational Excellence (deliver value while continuously improving), Security (protect assets from risks), Reliability (ability to recover from disruptions), Performance Efficiency (using resources efficiently to meet business objectives), Cost Optimization (running systems to deliver value at lowest price point). The *Design pillars* lead on to a set of design principles – (1) stop guessing capacity and provision just what you need at that time, (2) test systems at production scale, (3) use automation to simply the architecture, (4) allow for evolutionary architectures, (5) use data to drive decisions, (6) improve through game days.

Undifferentiated Heavy Lifting – Basically wholesaling infrastructure. Even though you built a custom solution specific to your needs, you're still running on commodity platforms and infrastructure that other people run on. This allows AWS to consolidate infrastructure management to provide economies of scale = lower cost for all its customers.

AWS Accounts – An AWS account is the basic way in which resources under a particular administrative control are isolated from each other. Think like an AD forest. It serves 3 functions – 1) as an IDAM domain, 2) providing an isolated blast radius, 3) provides separate billing from other accounts.

AWS Global Infrastructure – AWS runs at a global scale. The GI is made of regions connected via a global backbone. Within each region are multiple AZs. Within an AZ are multiple datacenters in the same local area. *AWS regions* serve to provide high-level isolation and compliance functions. *AWS AZs* are essentially datacenters that you host your infrastructure on. Each AZ is separated geographically from other AZs so a failure in one won't affect others. Thus you can host multiple instances of the same infrastructure on 2 different AZs within the same region to provide for high-availability and fault-tolerance. *AWS Edge locations* are smaller datacenters with limited service offerings that you can use for specific use cases in specific physical locations to reduce latency (usually it's CloudFront).

----------S3----------

S3 – Object storage system that is a global service. Buckets you create are per region only though. You create a bucket to store objects that can be accessed using web-protocols. It's perfect for anything that needs to be accessed over the web or if you need to share a common set of data/objects across diff. AWS products. A ***bucket*** is the basic entity inside S3 that is nothing more than a flat container that stores objects. It's not a file system and its name needs to be globally unique in S3. Objects in S3 can be internet-accessible via a unique URL if correct permissions are set. Buckets are a flat container that has no concept of directories or hierarchies but you can add / in file names to emulate the effect of a directory. S3 couldn't care less because it only identifies objects by their key (object name) and their value (object data). **Objects can be from 0 bytes – 5 TB. You can have up to 100 buckets per account and if you ask AWS they can give you up to 1,000 buckets per account. It is NOT a filesystem so you can't mount buckets.**

S3 bucket permissions can be set via Identity policies, resource policies, bucket ACLs, and public access settings. **The AWS account that created the bucket owns and has full control over it.** No one else has access by default. You can configure permissions from 2 perspectives – identity policies and resource policies. *Identity Policies* are created from the perspective of IAM entities and can be applied to our IAM users, roles or groups but NOT anon or external identities. **You can only use identities on principals you control.** *Resource Policies* are created from the perspective of the bucket and are used to grant access to identities OUTSIDE of our AWS account (incl. anon access). (Deny-Allow-Deny). **Bucket ACLs** are an old way of granting access but is not recommended for prod anymore.

There are **2 ways to move data to S3 –** Single PUT operation and Multi-Part uploads. **SPO for anything < 5 GB. Beyond that, use MPU.** A Single MPU can have up to 10k parts each from 5MB – 5GB in size. The *recommendation* is to use MPU for anything over 100 MB, over 5GB it becomes mandatory.

Encryption in S3 is PER OBJECT not per bucket. 4 choices available – Client-side, SSE-C, SSE-S3, and SSE-KMS. **C-S =** you encrypt your own data, **SSE-C =** S3 encrypts/decrypts but you manage your own keys, **SSE-S3 =** S3 encrypts/decrypts data and uses its own internal keys, **SSE-KMS =** S3 encrypts/decrypts but using your AWS KMS CMKs (used for role separation). By default AWS will pick the encryption algo for you but you can change this using bucket policies.

Dexter's "Barely passed" AWS Cram Notes

CORS is a feature that allows web apps running in one bucket to be able to reference resources in another bucket. Cross-Origin Resource Sharing (CORS) is a protocol that enables scripts running on a browser client to interact with resources from a different origin. This is useful because, thanks to the same-origin policy followed by XMLHttpRequest and fetch, JavaScript can only make calls to URLs that live on the same origin as the location where the script is running.

You can only enable versioning on an S3 bucket once and you can't disable it after (although you can suspend it). When enabled every object is given a version ID. Every-time you upload the same object you create a new version. You are charged for every version stored in the bucket. When you delete an "object" you're only deleting one version of it and the way it's done is S3 adding a DELETE marker on the version. You can remove this marker to UNDELETE the version. If you delete all versions of an object, you permanently delete that object.

You can configure MFA delete where it prompts for MFA tokencode before deleting an object.

Pre-signed URLs are an easy way for outside users access to specific files w/o having to open the entire bucket permissions up for public access. You generate a pre-signed URL that embeds the necessary creds to impersonate your identity and its access into the bucket. Give the URL to some outside party, they can use it for a limited period of time. If you delete the user that created the PSU, that PSU's access to the bucket is also removed.

CloudFormation – IaC product that allows you to essentially "code" your AWS infrastructure, either in advance or to make changes like any other piece of code. It can be used for QA/DEV environments to rapidly bring up and tear down those environments, or it can be used as part of a DR plan to drastically reduce the RTO required to bring up a "cold site". Basically everything starts with a **CF template** which is either a JSON or YAML doc. You define local resources in the template. You give the template to CF to create a **stack**. In the stack, CF then provisions *physical resources* according to and that maps back to the logical resources you've defined in your CF template. When you update a stack, you make changes to existing *logical resources* in the CF template, give it to CF. CF then provisions/deprovisions/modifies the *physical resources* according to the logical resources defined in the new version of the CF template. **When you delete a stack, all the resources that were created as part of that stack is deleted along with it.** Up to 200 resources per template, but you can nest template together.

Any new logical resources will create new physical resources, any removed logical resources will remove the physical resources. Any changes to existing resources will also modify the associated physical resource. Changes to physical resources may cause some disruption depending on the nature of the change.

S3 has 4 storage classes – standard, standard-IA, OneZone-IA, and Glacier (Deep Archive). **Standard** is the default for almost everything. You get 3 AZ replication and 11x 9 availability. **Standard-IA** is for infrequently accessed files, is cheaper than standard but you have min. 30 day usage charge for the object as soon as you upload it and comes with a retrieval fee. **Onezone-IA** is the same as standard IA but is cheaper as the data is only stored in 1 AZ and used for noncritical data. **Glacier (and Deep Archive)** has 3 AZ replication, a min 90 day, and a min 40KB charge.

You can automatically move data from one storage tier to another using **intelligent-tiering** (where S3 automatically moves infrequent access from one storage class to another for you), or configure **lifecycle policies** to tell S3 when to transition an object from one class to another. I-T only supports 2 storage classes whereas you can configure many storage classes in lifecycle policies.

CRR is cross-region replication. You create 2 buckets each in a separate region and **enable versioning on both.** Then configure a role with access to both source and dest. Buckets. Then you configure CRR. CRR only works one way from source bucket to dest. Bucket. You can't do CRR with SSE-C encrypted data as data needs to be decrypted before it's replicated. It only takes effect on objects added AFTER the feature is enabled. You can set CRR to automatically change ownership of objects copied into the second bucket to be the owner of the that bucket. CRR is basically backing up objects in S3.

----------IAM----------

IAM – Very important AWS service. Everything you do touches IAM in one way or another. IAM is what decides whether you can or can't access resources or perform certain actions in AWS. It is the ***only way to implement best-practices IDAM infrastructure inside AWS.*** IAM is made of several components (1) identities (users and roles), (2) resources (ARNs), (3) groups, (4) policies, and (5) credentials.

By default, an IAM identity has no permissions when created, IAM has an implicit deny policy that takes effect. (E. Deny > E. Allow > I. Deny)

An IAM user is *anything* that can authenticate to AWS to get services or resources. If it "logs in" it's a user. It can be a service account, or an actual human account. An IAM user must have a 1-to-1 relationship with a particular principal. This is the only way a principal can "log in" to AWS.

ARNs are AWS' way of uniquely identifying any resources in any region within any AWS account. ARNs are globally unique across AWS. An account with an S3 bucket will have a different ARN that another account with an S3 bucket in the same region with the same name.

Groups are like AD groups. You add users to a group. You *cannot* perform authentication on a group but you can attach inline policies or assign IAM policies to a group and those policies will flow down *and be applied to the users within that group.* The group itself doesn't do anything, it's an administrative container. An IAM user can be part of many Groups, and a group can contain many IAM users. **Groups have no credentials, you can't assign an access key to a group. They are not true identities, just container.**

IAM Policies are JSON documents that define under which circumstances an *identity* is authorized (or unauthorized) to access resources or features within AWS. A policy defined on its own doesn't do anything, only when they're attached to users, groups, roles, or resources would their statements take effect.

IAM policy statements have 5 components – SID, Effect (Allow/Deny), Actions, Resources, Conditions.

You can create and attach policies in 2 places – if you attach an IAM policy to identities, it becomes an *identity-based policy* that defines what those identities have access to. If you attach the policy to a resource, it becomes a *resource-based policy* where you define what identities have access to that resource. The way you create policies is you first create a baseline policy for the entire AWS account as a managed policy, then you create inline polices for any exceptions to the baseline.

IAM Roles is a mechanism allowing a non-human or an a generic external entity outside of your AWS account *impersonate* an identity within your AWS account to get at resources in your AWS account. You can't login to roles, but you can *assume* one. It's used to temporarily grant access to anything that needs to perform actions inside your AWS account on your behalf. Anything = human, AWS services, or external apps. Roles have 2 policies associated with them – a *trust policy* which defines "what" can assume that role, and a *permissions policy* which defines what that role "has access to". When an **identity assumes a role,** AWS IAM STS issues a temporary security credential aka "a token" to that entity which grants them permissions to whatever that role has access to. Roles are used for (1) break-glass situation for emergency vendor root access, (2) allowing AWS services access to AWS resources in your account, (3) allowing cross AWS account access w/o having to manually create/re-create users on both ends *think IAM 5k user limit* (4) you have apps that need to call AWS resources to function. Roles are not used for anything that needs to interactively login to the AWS console or command line, anything that is a permanent/long-term identity that needs persistent access, humans.

IAM credentials is how IAM *authenticates an identity presented by a principal.* It can take 3 forms – static user/pass, access keys, roles.

Dexter's "Barely passed" AWS Cram Notes

IAM Access Evaluation considers *all the policies that can affect an IAM identity at that point in time in a single shot.* Thus, if a user has an identity based policy attached, and is part of a group that has a managed policy assigned, trying to access an S3 bucket with a resource policy applied, all 3 policies are evaluated together. Any deny statements are evaluated first, followed by any explicit allows, if none exists, the identity is denied access to the bucket.

IAM Authentication can be in 1 of 3 ways, using *username/password/token* if the IAM user represents a human as a principal, using *access keys* if the principal represents an application trying to authenticate, or *using roles* if the entity is external to AWS or only requires temporary access to resources inside the AWS account.

Authorization takes place after authentication where identity policies and resource policies (inline or managed) are evaluated to grant/deny access to resources for a particular identity.

IAM Limits – 5000 users per account, 10 group memberships per user, 10 managed policies per user by default, 2048 characters max. for inline policies, 1 MFA and 2 access keys per user. You can use roles to overcome the 5000 user limit, or use Cognito to allow users to sign in using an external identity and assume a role inside AWS.

IAM Access Keys – are used for command line tools and non-interactive authentication capable clients to authenticate with IAM to get at AWS resources. Access keys are made of the Access Key ID (AKID) and the secret access key (SAK). The AKID is a public identifier that can be referenced. The SAK is the secret that needs to be protected. **Max. 2 access keys per IAM user.** You can't modify a previously generated AK. You must delete/re-create. If you do this, any apps that use that AK will lose access and must use the new AK. AKs have no expiration dates. AKs along with the AWS console username and password **are the only 2 long-term (permanent) credential sets within AWS. Tokens (and the roles that uses them) are temporary.**

A large multi-national organization or a federal government with requirements for multiple AWS accounts can make use of **AWS organizations** as means to manage and organize all those AWS accounts together. Organizations is provide a few features like (1) consolidated billing across AWS accounts, (2) allowing for Service Control Policies to restrict account access to services at a high-level, (3) allowing more AWS accounts to be created from AWS organizations. To use, you basically configure a single AWS account to use organizations. That account is then converted to become an **organization master account.** The master account is **the root account for the organization and cannot be restricted in any way.** Other AWS accounts can then join the organization. The best practice is to leave the master account as a dedicated account for managing the organization, then use AWS accounts joined to the organization to host the actual services. Each account inside an AWS organization can choose to receive their own bill, or you can enable **consolidated billing** on the master account for that level. This basically have the bills from all the AWS accounts in the organization passed onto the master account as a single bill. As an individual AWS account has limitations on resources and reservations, using consolidated billing combines all the limits together so the entire organization has access to higher resource limits across its members.

AWS Organization SCPs – SCPs are used to limit what each AWS account in an organization can do. You start with a root node which is the organization root. From there you create OUs, you assign SCPs to the OUs, and add AWS accounts those OUs. You CANNOT use SCPs to restrict the master account in any way. So basically, the way to use organizations is to just use the AWS master account for consolidated billing, centralized logging , and to host IAM users/groups/policies for the entire organization. Then host actual services inside the sub-AWS accounts and use IAM roles to switch between AWS accounts for users.

You can have up to 2 AWS accounts in an organization by default, but you can ask AWS for more. Remember SCPs are used to restrict what an entire AWS account can perform within an organization.
https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scp.html

By default, SCP comes with "FullAccessPolicy" which grants AWS member account access to use all services. You can create a new SCP, link it to the relevant OU within AWS organizations, and then it will affect member AWS accounts in that OU and below. **You CANNOT use SCPs to restrict the master account in any way!!** (Even if you tie the SCP to the organization root.

AWS Organization Account Role Switching allows you to switch between AWS account in the same console session without having to log out and back in. When you first create an AWS organization a new IAM role called **OrganizationAccountAccess** role is created to allow you to role switch from the master account into another account within the organization. Basically, when you perform role switching, your IAM user identity assumes the **OrganizationAccountAccess** within the organization member account, and this is what allows you access into the console for that AWS Account. When you role switch, AWS makes **sts:AssumeRole** call for the **OrganizationAccountAccess** in the member account, you get assigned a token in exchange. Basically this allows you to have a single IAM user in the organization root AWS account that can be used to switch into all the other AWS accounts in that organization. The **OrganizationAccountAccess is automatically created if creating an AWS account from with AWS organizations.** If adding an existing AWS account into an existing organization, it is not automatically created. You need to create a role with the same name and configure it to trust the AWS master account using the trust policy.

----------ID/F AND SSO----------

IDF is an architecture where identities of an external IDP are recognized for SSO authentication/authorization to local systems in AWS. It works using an **Identity Provider (IdP)** which is an identity source like AD/FB/Google/Twitter/or your own custom IdP). **3 types of IdP** usable in IDF are **cross-account ROLES, SAML 2.0 federation (AD), and WebID Federation (Google/FB).**

IDF is deployed in AWS using Cognito or STS products. You use Cognito/STS to broker authentication from a 3rd-party IDP to grant short term temporary credentials to gain access to resources in AWS account. ID pools can be discrete pools of identities, or support an architecture where you merge multiple identities and treat them as one. **An important concept is that AN AWS ACCOUNT MUST BE USED TO AUTHORIZE ACCESS TO AWS RESOURCES, YOU CAN'T SIMPLY "REFERENCE" AN EXTERNAL IDENTITY. With IDF, you are EXCHANGING AN EXTERNAL IDENTITY FOR AN AWS IDENTITY THROUGH THE USE OF STS TOKENS.** You impersonate this AWS identity by presenting your STS token to gain AWS access.

SAML 2.0 Identity Flow

1. Authenticate to ADFS portal
2. ADFS returns SAML assertion token to prove successful authentication
3. SAML assertion delivered to AWS SAML endpoint
4. AWS SAML endpoint checks assertion with SAML server (using a pre-configured trust btw the ADFS server and AWS)
5. SSO endpoint talks to STS on your behalf and assumes a predefined role using the SAML assertion token.
6. The SAML assertion is swapped for AWS temporary security credentials.
7. The user gains access.

Web IDF Flow

1. For mobile and web apps
2. You login to the web app
3. Web app redirects you to an IDP (Google, FB)
4. You login to the IDP and get a token back
5. The webapp passes the IDP token to Cognito and requests to assume a role
6. You get a temporary security credential to access AWS resources

YOU ARE ALWAYS USING AN AWS ISSUED SECURITY TOKEN TO ACCESS AWS RESOURCES WHEN USING AN EXTERNAL IDENTITY TO AUTHENTICATE.

Dexter's "Barely passed" AWS Cram Notes

When to use IDF – 1) enterprise access to AWS resources using SSO/AD, 2) mobile web apps that need to grant customer access to S3 to "their own account" by logging in via social media, 3) TO HELP OVERCOME THE 5,000 USER LIMIT IN IAM for front-end facing apps using IAM. 4) using multiple AWS accounts with role switch.

----------COMPUTE----------

EC2 is a core AWS product that gives access to virtual machines known as *instances* that run on AWS hypervisors and hardware. It's a regional service so you have to create instances in the correct availability zone and subnet. You first select an AMI which determines which OS image to use, you can have either AWS provided AMI, bring your own AMI, or use 3rd-party marketplace AMIs. Then choose your CPU architecture (x86 vs. ARM). From there you select an instance type and size, then pick your networking, storage, security group settings, and finally tag your instance.

You have **2 storage options –** instance store and EBS. Instance stores are temporary storage located on the local Ec2 host and doesn't guarantee durability, data can be lost at any time. If you want reliable, long-term storage, create and attach an EBS volume to your instance.

Access **to** an Ec2 instance is via **security groups.** It like a virtual host firewall attached to the ENI of the instance (but appears as if it surrounds the instance itself). Security groups only have **allow** rules and no deny rules. The only deny is the implicit deny.

Ec2 integrates with CloudWatch and CW is the de facto monitoring solution for instances. Ec2 sends data at 5-min intervals.

An instance can be in 1 of 7 states – pending, running, rebooting, shutting-down, stopping, stopped, terminated. Instances follow a lifecycle through the states - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html

When you shutdown a guest OS, the instance keeps running and does not get stopped, you still get billed. The only way to stop an instance is to use the AWS instance controls. With EBS volumes you get a separate bill for EBS volume usage and you get charged for storage use as long as the instance hasn't been terminated.

When to use EC2 – when your app needs a traditional OS to run, when you need constant CPU time (i.e.: background processing jobs), for monolithic apps that still need an OS to run, a bursty application that needs to be active even if not being used (i.e.: Active Directory)

You have 5 families of Ec2 instances – General purpose A/M/T, Compute optimized C, Memory Optimized R/X/Hi Mem/Z), Storage optimized (I/D/H), and Accelerated Computing (F1/Inf1/P/G). Choose the correct instance type for your needs. https://aws.amazon.com/ec2/instance-types/

Within each family you have diff. instance sizes – large/small/medium/xl/2xl/nano. This is how much CPU/RAM/Network/Storage you get for that instance.

You also have special use instances like (a – AMD CPU), A (ARM-based), n (high-speed networking), d (NVMe storage)

Ec2 storage architecture – 2 storage types to choose from – instance store and block storage (EBS). **Instance stores** are attached directly to the Ec2 host and is used by the instances running on that host. You get the best performance but you sacrifice durability of your data. If the host fails or if the instance is deallocated to another host, data stored here is lost. Instance stores will survive a reboot as long as it comes up on the same host, but no guarantees. **EBS** is used for long term storage. You provision a separate volume on a SAN that your instance uses. You can use either HDD or SSD volumes. A volume only exists in ONE AZ so you still have a single-AZ risk, but you can configure snapshot and replication of an EBS volume to multiple AZs. You can use AWS KMS to apply encryption to an EBS volume. EBS performance measured in 2 ways – IOPS and Throughput. IOPS is how many I/O per sec the volume can handle, while throughput is the data rate in MB/s. **IOPS * Block size = Data Rate.** Each EBS volume type has a dominant performance attribute (SSD – IOPS, HDD – throughput) which will affect the storage type selected for your workload.

ST1 is for low-cost high throughput storage good for streaming video, big data workloads, log data, or anything where large datasets need to be read sequentially. It has a max IOPS of 500 MB/s.

SC1 is cold HDD. Worst performance, basically for archival and backups. You CANNOT use any mechanical drive as a boot drive.

GP2 is an SSD and is the default storage that provides good throughput and fast IOPS. Its performance is linked to its volume size. You get 3 IOPS of performance for every GB in size, with **min. 100 and max 16k IOPS.** You adjust performance by adjusting the size of the volume. Smaller volumes come with a **burst pool** feature that gives you I/O credits (5.4 million). If you use anything above the baseline, you use up I/O credits, if you consume less, AWS refills your I/O credits. The burst pool has a max of 3,000 IOPS. Max volume size = 16TB. Any volume larger than 1TB in size doesn't use the "burst pool" model, you get IOPS of 3x the volume size all the time.

IO1 is used when you pre-pay for performance. You set the max. IOPS maximum and volume size independently. You pay for both the volume size + its performance.

EBS Maximum – 64K IOPS for nitro instances, other instances is **32k IOPS. Anything above 80K IOPS –** *use instance stores instead.*

You create the EBS volume in the same AZ as your instance. Attach it to the instance then map it to a volume in the OS, then mount and format like any other drive.

Auto-Enable IO – if an EBS volume gets data consistency issues, IO is suspended automatically to that volume. This setting allows AWS to automatically re-enable IO to that volume after a timeout.

Ebs FOR durability.

You take **EBS snapshots** to (1) backup the EBS volume, (2) as means to migrate an EBS volume from one AZ to another. Best practice is to shutdown the instance before taking snapshots to avoid consistency issues. Snapshots are stored inside S3 that can be shared with other AWS accounts. You can use Data Lifecycle Manager (DLN) to automate creating/deleting snapshots. The first snapshot you create is a full volume snap, subsequent snaps are incremental. Incremental snaps only need the full snap to restore, it's not dependent on prior incremental snaps. If you snap an encrypted EBS volume, that snap is also encrypted using the same KMS key.

Security Groups – think host STATEFUL firewalls. A security group is attached to the ENI that's attached to an instance so in effect, they appear to be attached to an instance. Made of inbound and outbound **PERMIT rules only (no deny rules)**. Both inbound and outbound rulesets have an implicit deny that takes effect if no other permit rules match. An instance is assigned to the default sec group unless you specify a custom group to assign it to. Rules can either be inbound or outbound, and matches by SRC, Protocol, Port. Think about when adding entire subnets/networks to security group rules, you can't deny specific subset of IPs from that range. **Using security groups -** You associate a security group with a VPC, every security group belongs to a particular VPC .Security groups have an ID and they consist of inbound and outbound rules. Every sec group on a particular entity are evaluated at the same time. If none of the rules apply, implicit deny takes effect. You can reference other logical entities inside AWS in SG rules - other instances, other sec groups, including itself. You can also have a sec group reference itself as a source - you can allow traffic from the security group to the security group - this allows all EC2 instances belonging to the security group to communicate with each other.

Instance Metadata – you can use a page from within Ec2 to get details about the instance running – **http://169.254.169.254/latest/meta-data**

Dexter's "Barely passed" AWS Cram Notes

You need either a NATGW, an IGW or EIGW to assign a public IP to an instance.

EC2 instances are created from **AMIs.** There are 2 types of AMIs – instance-store backed (root volume for non-EBS instances), and EBS-backed (root volume uses EBS. Basically a VM template. An AMI contains – **Block Device mappings** (has snapshots of EBS volumes so the mapping contains details on how these snaps should be mapped to the new instance), **AMI instance permissions (**who can launch the AMI – public or private)

AMI-BAKING is the process of creating a new AMI from an existing instance. Basically taking a new image of the instance + a snapshot of the EBS volume. The AMI is a container that references the snapshot of the source instance and the block device mapping. You then can launch an instance from the freshly baked AMI. **All config is built into the AMI which if you over-configure it becomes very inflexible.**

Bootstrapping is when you give a series of OS commands to execute at boot time. So you **Bake an AMI** with pre-installed base software and **bootstrap** the instance with config commands or to install custom packages. Bootstrapping is done by defining a set of **user-data** within the instance settings – and user data can be a BASH or powershell script. **You can use CloudInit** syntax within bootstrap scripts. https://cloudinit.readthedocs.io/en/latest/

You use AMI baking to gain a consistent OS/application baseline. Then you use bootstrapping to customize the config, download patches or set up custom applications at launch time.

The only difference btw. a public and private Ec2 instance is whether it's launch into a subnet that has an IGW/NAT gw attached and whether the subnet has the "auto-assign public IP address" enabled. **Private instances** only have a private IP in the range of the VPC subnet. It can only communicate within a VPC (and to other VPCs if inter-VPC peer routing is configured). Private IP addresses are dynamically assigned at launch time and remains even if the instance is stopped. You can assign multiple private IPs to an ENI, the max is determined by the instance size. A private DNS name is associated with a private IP Address. If to terminate the instance the private IP is disassociated automatically.

Public Instances have both a private and public IP address. The translation is handled by either IGW or NATGW. The real IP address is always an RFC1918 address, the public IP is mapped on the IGW or NATGW and is shown on the instance settings. **You can never apply a public IP address directly to an instance's guest OS – you need the NATGW or IGW to handle this for you.**

EC2 instances also get a DNS name. It always gets a private DNS name but an optionally have an external DNS name assigned. If you ping the external DNS name from within AWS, you get the internal IP, if pinging from internet, you get the external IP.

2 types of public IP addresses can be assigned to public instances – (1) dynamic IP addresses – these are auto assigned at launch time along with the DNS hostname associated to that IP address, they change as the instance is stopped/restarted. (2) elastic IP addresses – you allocate and assign an elastic IP manually. These come with a DNS name but they don't change through reboots. This has extra cost. You can choose btw dynamic or elastic public IP for an instance, but you can't have both.

Tying Ec2 back to IAM – Instance Roles. Instead hardcoding AWS creds into any scripts running on an instance, you can assign an instance an IAM role. This allows the Ec2 instance and any apps running on that instance the ability to impersonate that role and gain access to resources. To use this feature, first create an **instance profile,** which acts as a container for the IAM role that you'll use to pass the permissions onto the Ec2 instance. The instance *and* the apps running on that instance can then assume the role as specified within the profile. **You can create an IAM role and associate it thousands of instances.** The temp credentials are rotated each time Ec2 tries to assume the role. This helps automate secure usage/delivery of AWS creds to many Ec2 instances. **Whenever you have a choice, use IAM roles. The only time you CAN'T use roles is when you have to login to the console interactively. USE INSTANCE ROLES.**

AWS Encryption uses AWS KMS to handle encryption keys. Can either use AWS Managed Keys or CMKs to encrypt a volume. KMS generates CMKs, EBS requests KMS to generate DEKs. KMS delivers both the encrypted and plaintext version of DEKs back to EBS. Encrypted DEK stored alongside the encrypted data, plaintext DEKs stored on the EC2 instance host which is used to encrypt/decrypt that data in transit.

EBS Volume Encryption for every EBS volume uses its own KMS-generated encryption key as long as it's encrypted while in a blank/fresh state. Encryption/decryption process is transparent to guest OS so no performance impact as process occurs on the Ec2 host. **If you create a snapshot from an encrypted volume, the snapshot is encrypted with the same DEK as the original volume. You CANNOT create an unencrypted volume from an encrypted EBS volume.** If You create a volume from encrypted snapshot, those volumes are also encrypted. If you create an AMI from encrypted volumes, the snapshots of those AMIs are also encrypted using the same DEK as the original volume. **NOT appropriate for when you need to manage the encryption at OS-level.**

EBS Optimization – EBS optimized mode is an architecture featured in new generations of Ec2 instances. Basically has 2 dedicated network paths – one for app data, and the other as a dedicated storage link. Gives faster access to storage = better I/O rates and higher transfer consistency. You can't disable EBS optimized mode for newer generation instances.

EBS Snapshot optimization – when you launch an EBS-EC2 or create a new EBS volume, you get the max. perf. Of the new volume right away. But if you restore from snapshot, the EBS volume space is allocated right away but the data is gradually transferred from S3 as it's being read for the first time. To ensure max. perf of restored EBS volumes read all data blocks in advance.

Enhanced Networking – The Ec2 hypervisor emulates the underlying hardware and presents them to the instances. This has performance impact. By enabling enhanced networking, the host NIC can use SR-IOV to present itself as multiple NICs at the hardware level, each VM can then be given direct access to each of the NICs and the NIC hardware handles this interaction. Benefits = less CPU used on Ec2 host, lower latency, better consistency in performance.

Placement Groups are used when you want to specify where you want a group of instances to run. 3 types – cluster, partition, spread. Use **cluster PG** when you want max perf by grouping all instances together on the same Ec2 host, but you sacrifice resiliency in hardware failure. Use **partition PG** when you want to ensure app availability within the same AZ so that a rack failure does not take down the entire application. Use **spread PG** to ensure that only up to 7 instances can run within the same AZ and each instance is deployed in a separate physical partition, max availability but lower perf.

On-Demand Billing – Minimum 60 seconds. Used for when the load is variable and can't be forecasted. No prior contract just show up and deploy instances, and pay for what you use.

Spot and Spot Fleets – used for when you really want to save cost and use spare AWS capacity to run your instances. You bid a max price per hour for an instance. The spot price is set depending on load on the AWS infra at the time. If the spot price < max price, your instance is launched and keeps running. If spot price > max price, you don't get your instance or existing ones may be terminated. For batch/interruptible workloads that can tolerate failure.

Reserved Instances is when you pre-pay to guarantee capacity over a period of 1-3 years. Performance is guaranteed. By pre-paying you also get a cheaper price per hour. You can either pay all upfront, partial upfront, or no upfront. But you commit to that instance for a period of time. You can have either **zonal or regional reservation.** Zonal reservation allows to reserve capacity in a particular AZ but is locked to a specific instance size and AZ. Zonal reservations also reserves capacity upfront. Regional reservation has to restrictions in AZ or instance size. You can carve out instance sizes such that it you reserve a

t3.medium you can run 2x t2.small at the same price. Regional reservations don't reserve capacity. **Use for when you have a base/consistent load, when growth can be forecasted, for critical systems/services or for apps that don't have bursty workloads. Use on-demand when future usage is variable or can't be forecasted and the workload can't be interrupted. Use spot for best-effort workloads.**

Dedicated Hosts is usually used for compliance/regulatory purposes, to comply with enterprise licensing models with CPU core restrictions or to really control instance placement.

AWS Lambda is an event-driven, serverless FaaS service in AWS that allows to define a function by uploading code to perform a certain task. The function only runs as long as it needs to complete the task, or up to a max. 15 mins after which it terminates. You only pay for the time and resources consumed while the function is running. Functions can be invoked manually or via an event (i.e.: file upload to S3 event). Permissions for lambda functions are known as **execution roles.** A function will assume an execution role to get at various resources. At a minimum a function needs access to write logs to Cloudwatch. Every function has a duration for which it's executed and billed for – min. 100ms, max 15 mins. To configure a Lambda function to upload a ZIP file containing your code and required libraries, then configure the IAM execution role, then configure an event trigger that will invoke the function.

Lambda permissions – new functions by default are in the public VPC, they can access internet and any public AWS endpoints. But you can change the VPC the function executes in. Lambda functions create a temporary environment to execute, it pulls in any resources needed to execute to the end. Once it's finished, the environment is torn down, any output needs to be moved to persistent storage to preserve otherwise it's lost. You can scale Lambda to infinite levels. You can use Cloudwatch to monitor Lambda function execution metrics. You DON'T use Lambda when the app requires consistent CPU use or needs persistent access to OS functions. And remember – **15 mins max.**

An API gateway can be used to publish, monitor and secure APIs. It can be used with Lambda and handles HA/scaling for you. You make calls to endpoints hosted on the API GW and it calls underlying services on your behalf. Supports 2 communication methods – REST and Websockets. You can enable CORS if resources from diff. domain names need to reference this API. Methods are defined as ways we interact with the API resource and is configured on a per-resource basis. It determines the integration method of the API. Permissions are needed for the Lambda function's execution policy to set who can invokve that function. The last step are to configure the resources to ensure the flow is correct btw the API caller and the lambda function, and finally creating an API endpoint which publishes the API into Prod. No cost to host an API but charged when calling the API to service a call.

Step functions allow to build a serverless workflow by coordinating multiple microservices to do a bigger task than what a function can accomplish. It overcomes Lambda's limitations by providing a state machine to orchestrate other AWS svcs. State machines can run for up to a ear and basically runs from start to completion. Step machines are made of states – workflow steps that can be branching, logic, decisions, a task that runs a Lambda function, etc. Step functions = serverless SWS (it can do what SWS can do w/o the long running compute resources). Use cases – (1) when you need to coordinate many Lambda functions, (2) when you need large coordination , scale, visual workflows, (3) anything running > 15 mins.

ECS is Amazon's containerization service. It's a managed Docker service that allows to run containers w/o the overhead. It can run in 2 modes – EC2 or Fargate. **Ec2 mode** uses Ec2 instances provisioned by ECS to manage containers, **Fargate mode** manages the container process end-to-end without using any Ec2 instances. https://www.dragonspears.com/blog/aws-container-orchestration-101-ecs-vs-fargate-vs-eks. The basic entity in ECS is a cluster. A cluster is a config container that defines how you want an group of ECS infra to work – it can either be (1) networking only using Fargate (pure serverless), using EC2 linux and Networking (Ec2 mode), or using EC2 Windows and Networking (again Ec2 mode). After you create a cluster, you create a **task** definition which defines the task role, size, containers, container URL and port mapping (the container's specs). **Task definitions** on their own do nothing, to do something you have to create a task from the task definition. A **task** is a running copy of the task definition. ECS automatically scales containers up and down (to the specified limits) to meet compute demands of the task.

----------NETWORKING----------

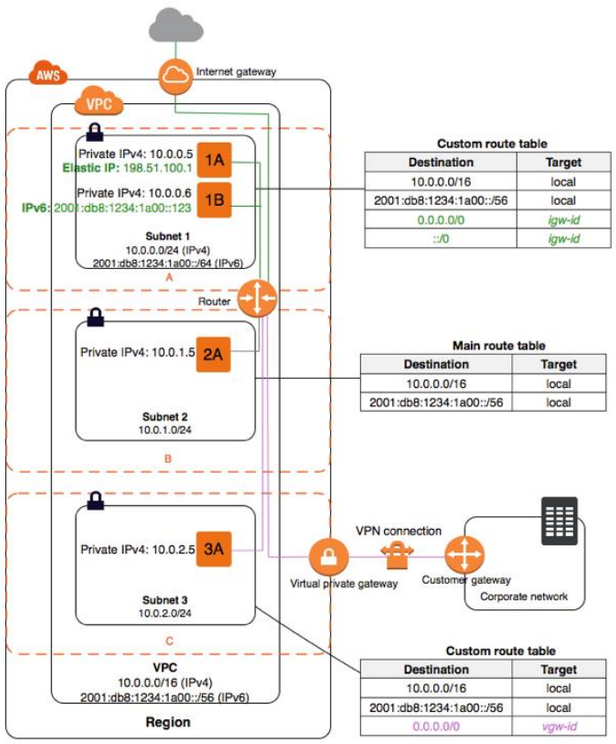
A **VPC** is a way to define isolated networks on AWS. A VPC is an isolated network infrastructure from all other VPCs in AWS and a resource inside a VPC can't talk to anything outside that VPC unless explicitly configured. VPCs are regional and be max /16 and min /28. Accounts come with a default VPC but you can create your custom one. VPCs can either use default tenancy where it's deployed to AWS shared hardware or it can be a dedicated tenant where it uses a dedicated hardware. If you launch a VPC inside a dedicated tenant, not all shared AWS svcs may be available. A VPC has a CIDR assigned to it. All subnets created afterwards inside this VPC must fall into the main CIDR block. You must create subnets within a VPC before being able to deploy services. While a VPC is regional, a subnet is AZ-specific. A subnet can only belong to one AZ but you can have many AZs in as many AZs within the region as you want. Subnets have their own CIDRs that can't be larger than the main CIDR block of the VPC. Every subnet always belong to a VPC and an AZ in that VPC. **Every subnet's IP block has 5 reserved IP Address**, the network address, the broadcast address, the network+1 used for the VPC router, network+2 used for DNS server, and network+3 for future use.

You can further customize a VPC using DHCP option sets. You can use subnet sharing to share a subnet with other AWS accounts so that you own the subnet and its config, but other AWS accounts can deploy their resources into your subnet.

A VPC has a router. The router has directly connected subnets and a route table. The VPC router directs traffic btw subnets within the VPC as well as to/from the IGW/EIGW for traffic destined to the internet. Route tables control what the VPC router does with traffic leaving a subnet.

Subnets in a VPC can either be public or private. A private subnet is one whose resources doesn't have a public IP address assigned. They come with a single route (local route) which is used by the VPC router to direct traffic btw subnets within the VPC. A private subnet *can be made public* by (1) enabling public IP address assignment to the subnet, (2) attaching an IGW to the VPC, (3) configuring a default route in the VPC to direct internet bound traffic to the IGW, (4) optionally enable DNS hostnames and resolution for the subnet.

VPC routes can be static or dynamic. Dynamic routes are ones that are auto-learned by the VPC via a routing protocol like BGP.



A security group is a stateful firewall. It also is intelligent, you can reference other resources like security groups in a security group rule.

An IGW allows bi-directional traffic to/from the internet to/from a VPC. A NAT gateway only allows the VPC to access the internet. A NAT instance is an EC2 instance that performs the same function as a NAT gateway. You can configure the VPC such that outbound internet traffic is default routed to the NAT gateway and inbound traffic is routed via the IGW.

NAT Gateways (1) **need to be deployed in a public subnet, (2) need to be deployed FOR EACH AZ THAT NEEDS INET ACCESS (3) can scale from 5-45 Gbps.**

Network access control in AWS works in 2 layers. Security groups are stateful firewalls that only allow permit rules (no explicit deny rules). NACLs allow you to create permit and deny rules but are stateless so you need a rule per direction of traffic for every application flow (but NACLs allow you to have deny rules). Use NACLs to control access in and out of a subnet at a high level, use security groups to control access to specific hosts.

You can peer multiple VPCs in the same or diff. AWS accounts and in diff. regions together. You configure a peering connection and what happens is a transparent IGW object is created in AWS in the background that is used to connect the 2 VPCs together, each VPC has routes to networks in the other VPCs. **A Single peering connection can only contain 2 VPCs, if you need more peers, you need more peering connections.** You CANNOT peer 2 VPCs with overlapping CIDRs. Don't forget to configure routes to networks in the "other" VPC (next hop = peer VPC). Once a peering connection is configured you can reference security groups in the "other VPC". Security groups are not automatically configured/inherited to your resources so don't forget to configure the access appropriately.

By default, any public instance gets a public name. if you ping from outside, you get a public IP. Ping from inside VPC = private IP. The peer VPC is considered to still be "outside" the local VPC so if you need the resolution to return an internal IP on the peer, **enable requester DNS resolution and accepter DNS resolution** settings on the VPC peering connection.

Transitive routing is NOT possible over a peering connection. You can't reach a 3rd VPC "via" a peer VPC. You need to peer to both. Peering connections are NOT transitive.

For efficient routing from a VPC to AWS services, you configure a VPC endpoint. When you do that the traffic towards those AWS services defined by the endpoint will traverse the AWS internal backbone instead of over the internet. You can configure 2 types of endpoints – gateway endpoint and interface endpoints. A **gateway endpoint** is attached to the VPC. It becomes like another IGW. When attached, the endpoint auto-adds routes used by AWS services to the VPC routing table such that traffic to those services automatically get routed to the GW endpoint. As with a VPC, GW endpoints are regional, so you only need 1 per VPC. **Interface endpoints** are virtual ENIs that are attached to specific subnets in the VPC. You need to pick which subnets it gets attached to. One thing that an interface endpoint allows to do is to **attach security groups to control access to the endpoint ENI.** When you create an interface endpoint, you also get an endpoint-specific DNS name per AZ that the endpoint exists in plus a DNS name for that service that is not AZ specific (uses best available AZ endpoint). The endpoint maps to a private IP address for the AWS service.

You can **Enable Private DNS name** such that if you try to resolve the public AWS service DNS name it returns a private IP Address. Interface endpoints can be affected by NACLs.

S3 and DynamoDB uses gateway endpoints which uses routes. All other services use interface endpoints which uses DNS.

IPv6 is supported in AWS, but not across all products/services. You enable IPv6 for the entire VPC and AWS will give you a /56 range. You can't change or modify this range, but you can further subnet it into smaller ranges. DNS names are not allocated for IPv6 addresses. All IPv6 addresses are publicly routable (no RFC1918 or NAT). EC2 instances are also given public REAL IPv6 addresses. You use a EO-IGW with IPv6 for outbound IPv6 traffic only, or you can opt to use IGW for bi-directional traffic.

VPCs support VPN and DirectConnect. VPN is IPSEC while DirectConnect is MPLS.

----------DNS----------

You should know how DNS works. **Route 53 is AWS' managed DNS hosting service.** It supports hosting both private and public DNS zones. You can use Route 53 to register your own domain names. You should also already know how DNS zones work.

In Route 53 there are 2 types of zones – public and private. Public zones are the default ones that get created when you register/migrate a domain to Route 53. It has a public resolvable namespace (i.e.: .com). Private zones are for internal use within a VPC only. It can have the same name as a public hosted zone for split-view DNS resolution. Only hosts inside a VPC can resolve private DNS names. In order for Route 53 private zones to work - ensure both EnableDnsHostnames and EnableDnsSupport options are enabled within the VPC settings

DNS SplitView is when you create a private zone with the same name as your public zone. What split views allow for is to set up the same records but with diff. targets depending on whether you're resolving from internally or externally. It allows to have one resource available on the internet using one name and have a different resource on the intranet using the same name. If you have 2 zones, a public and a private - if in any of those zones you have the same records (i.e.: www), inside a VPC, the private zone takes precedence over the public zone. If you have 2 zones, and the private one doesn't have the record, the public zone's records take effect. A private zone has no effect unless it has the same DNS records as the public zone.

You should already know the various DNS record types commonly used.

Route 53 can perform health checks on your public resources for you. The check can either be an endpoint check, it can be a check on other health checks at an aggregate level, or it can monitor a cloudwatch alarm state. The most common type is the simple endpoint check which can be used with either IP addresses or domain names. The simple check can either be a HTTP/HTTPS check, or a TCP SYN connect check. By default checks are done every 30 secs but can be changed. The failure threshold determines the number of consecutive failed checks before a resource is deemed unavailable. A HTTP(s) check looks for a HTTP/200 response. With string matching checks, the

checker looks in the body of the response for a particular text. By default can come from any AWS region but you can lock it down to a particular region (not recommended though). 0.50 per check on AWS resources. 0.75 per checks on non-AWS resources.

With Route 53 you can configure routing policies for DNS requests. There's simple, failover, weighted, latency-based, geo-location based, and multi-value answer.

Simple routing defines a single record for each resource but with multiple addresses. Route 53 returns all the configured addresses in any order. You get some load distributed but you can't guarantee that it will be evenly spread across all the configured addresses.

Failover routing uses health checks. You first define a health check on the primary resource, then change the routing policy of the A record from simple to failover. This creates a primary record set where you can define more A records with the same name. You can have (1) primary record and (1) Secondary record. Each record must be configured with a Set ID that's unique amongst all record sets of the same name. After you create the primary record and specify failover routing, you create as many secondary records to point to other resources/IP addreses as you want. You then specify the failover routing policy to be **failover. Remember – only 1 primary record type and only 1 secondary failover record.** You can chain multiple routing policies together to create complex routing trees inside Route 53.

Weighted routing policy assigns weights to DNS records in relation to the aggregate weight of all the DNS records for that resource. You need to specify 2 addt. Attributes in the DNS records – the weight of the record, and the Set ID to group records together. Route 53 adds the weight of all the records in the set together, then it uses the weight of each individual record as a percentage of the total weight to determine the % of traffic to send to each record. Is NOT load balancing, but the ideal use case is when you want to have a limited roll out of new versions of a website and direct only a portion of traffic to the new site to limit potential issues. you can create multiple DNS records of the same name, the only thing that needs to be unique is the set ID. You need to set the weight for each record, keeping in mind its relation to the total weight, which determines how often individual resources are returned.

Latency-based routing allows you to specify a region for every record you create. Route 53 maintains a database of average latencies btw internet endpoints and each geographic location. When a client requests for DNS resolution Route 53 returns the record with the least latency for the client's location. Latency does NOT equal geographical proximity. It's based on speed. A site with the least latency may not be the physically closest one.

Geo-location routing is the one that routes based on geographical location. Again is based on creating a set ID for a group of DNS records, Route 53 only returns records created for a particular region. If no geo-specific record exists, a default record can be created as fail-safe. If that doesn't exist, the resolution fails.

Multi-value answer – Create a unique set ID for each record, specify multiple records w/ the same name, set multi-value answer and Route 53 returns all the records for the one query.

----------CLOUDFRONT----------

Cloudfront is a CDN. It takes content from one location and distributes it to caches around the world to improve website performance in various geo regions. In CF, you start with an ORIGIN which is the server hosting the original content. You configure a CF distribution which specifies that Origin's URL. You can specify to use all global caches or use specific regional caches only. Your origin needs to be accessible by Cloudfront (either a public resource or using S3 Origin Access Identifier [OAI]). You configure your DNS as a CNAME to point to the CloudFront provided DNS for your cache.

When you browse to a CF URL you first check with the closest edge location, if the object exists it is returned. Else it checks with the regional cache, if the object exists then the process also ends. The final step is to perform an **origin fetch** which pulls down content from the origin server down to the regional cache. Objects have their own TTL per-object. The default CF distro name is in the format **randomstring.cloudfront.net.**

By default all CF distros are public but you can configure **trusted signers** using signed URLs or signed cookies like S3. You can use an OAI to restrict an S3 bucket so that it can only be accessed by CF so the public can't browse directly to the bucket's URL and bypass CF. When an OAI is created a virtual identity is created for your CF distro and you can apply S3 bucket policies to that identity.

----------EFS----------

Basically an NFS file share that can be mounted to an Ec2 instance. It's different from EBS in that multiple instances can mount that share just like a regular network drive. EFS is regional and data is stored in multiple AZs but you can create ONE separate mount points per AZ for as many AZ that your VPC has. You can choose from 1 of 2 throughput modes – **Bursting mode** when you get a certain level of performance based on the FS size (100 MB/s base and 100MB/s burst per 1TB of storage), **Throughput mode** when you set size and throughput independently. You can choose from 1 of 2 performance modes – **GP** is default but if you have lots of instances use **Max I/O. You don't specify the FS size it grows dynamically. Encryption at rest uses AWS master key.** You access EFS using a DNS name from local VPC, across VPC peers, across DirectConnect BUT NOT DIRECTLY FROM THE INTERNET. **You can mount EFS volumes onto Ec2 instances. VPC security groups can be used to restrict EFS network access.**

----------DATABASES----------

There are 2 main categories of databases – relational and nonrelational. Nonrelational databases have 4 types – Key-value stores, Document stores, Column databases, and graph databases. **If you need to store data from a traditional DBMS, use RDS or Aurora. If you have key-values or document-style data, use DynamoDB, if you want to store analytical data – Redshift, and if you want to create a graph database – Neptune.**

RDS is AWS' relational DB solution. It is a DBaaS supporting MySQL, MariaDB, PostgreSQL, Oracle, MS SQL and Aurora engines. It is a server-based solution where you configure DB subnet groups and DB instances to provision. AWS manages these instances and just presents you a database connection CNAME you can use to connect. It performs automatic backups during configured maintenance windows. It is created **per availability zone but high-availability across multi-AZ can be congfigured.**

There are 4 types of backups in RDS – snapshots, automatic backups, and point-in-time.

snapshots (you manually create these and remain forever even when DB is deleted), good for long-term backup.

automatic maintenance and backups (these are retained from 0-35 days and happens once a day during the configurable maintenance window. AWS provides backup storage equal to the DB storage *provisioned* at no extra cost). If you delete the original DB the backups are retained for a short period of time for you to restore then it gets deleted too. The first backup is a full backup and all subsequent are incremental. This is **NOT A LONG-TERM BACKUP SOLUTION.**

Point-in-time is for transactional reverts. RDS has a journal that is uploaded every 5 mins. You can restore to any point in time for the **last 35 days. ALL AUTOMATIC BACKUPS IN RDS HAS MAX RETENTION TIME OF 35 DAYS.**

When you restore an RDS snapshot you're always creating a new DB instance, you can't restore over an existing DB instance. You get a new DB CNAME that you have to reconfigure all the security groups, and clients to point to that new endpoint DNS. Outages will occur during the cutover.

You can make an RDS resilient by either **deploying a multi-AZ RDS database or using Read-replicas.** When you select to deploy a Multi-AZ RDS instance, RDS creates a standby instance and performs a synchronous replication from the primary to the standby automatically. If a failure occurs, RDS automatically fails

over from primary instance to the secondary by changing the CNAME of the entire database. If using anything but Aurora, this CNAME ALWAYS points to the primary instance unless there's a failure. You can't access the standby instance, it does nothing until a failure happens. You perform maintenance on the entire database but RDS will execute it on the standby first then promote it to become the new primary, after which maintenance will be performed on the original primary instance. There is no performance impact to having a standby node as it has its own storage separate from the primary.

Read replicas are used to improve DB read times. You can create R/O replicas of the DB in the same or diff. region from the primary instance and these can be addressed separately from the primary as well. A RR doesn't need any addt. Config but still needs a DB subnet group. The RR can be private or publicly accessible. You can have many RR in a multi-AZ deployment. The repl btw the primary and standby instance in the same region is always **synchronous** while any repl that happens across regions are always **asynchronous.**

You CAN promote an RR to become the new primary in the event that the primary RDS instance goes down. Much faster than doing a restore. **5 RR max per primary instance, but you can have RRs point to other RRs to create a hierarchy at the cost of performance. You can't address all RRs at once but you can address each independently.**

Aurora is a database engine provided as part of RDS which is compatible w/ MySQL and PostgreSQL. It's a managed database service. You start by creating a cluster with a choice of either using Aurora Global or Regional. Then you create a database and VPC. A cluster is made of one primary node that performs all writes, all other replicas are R/O replicas. You can have many replicas. It is a server-based product so you still need to configure DB subnet groups, VPC security groups, etc.

An Aurora cluster starts off with one writer instance – it's the only one that can write to the Aurora shared storage, and 2 endpoints – a write endpoint and a read endpoint. Initially they both point to the one writer instance provisioned. You can add more reader instances afterwards but you can only have 1 writer instance.

Up to 15 read replicas in either same or diff. AZ but they all use the same cluster shared storage. All read replicas replica synchronously.

You can organize replicas into 15 failover tiers – tier 0 being the highest. When a failover happens Aurora looks for replicas in each tier sequentially starting from tier 0.

You scale write workloads vertically (size up writer instance), you scale read workloads horizontally (add more RR).

Every Aurora DB has 2 endpoints –a writer endpoint sends traffic to the writer instance, a reader endpoint sends traffic to any of the replicas.

Cluster access via the endpoints are secured using VPC security groups.

Aurora Backtrack to roll back the DB to a previous state and is the only engine that allows to revert bad changes of up to 72 hours to the DB w/o having to restore from backup. You can still perform a regular backup with a retention window from 1-35 days.

You **can failover the writer role to any one of the available read replicas in the cluster and it takes over the writer role.**

3 "optional" features of Aurora – (1) Parallel queries, (2) Global DBs, (3) Serverless

Parallel queries allow queries to be executed across all nodes at the same time but needs a compatible DB engine. **Global Databases** needs to be enabled when first creating a cluster but allows for one primary region for the writer role and one secondary region for a read-only replica. You can always add a replica from another region into the DB. *Usually only for specific use-cases* normally you just let Aurora handle this for you.

Aurora Serverless is Aurora delivered as a service for autoscaling. Relational DB w/o any admin overhead. You pay per ACU used. 1 ACU = 2 GB RAM. A serverless Aurora cluster can scale down to 0 ACU if no activity is detected, you still pay for the shared storage. The number of ACU you use depends on the load the DB experiences. It's basically autoscaling for databases. When the cluster scales down to 0 ACU, it pauses itself. You use it for apps that have unknown/variable DB usage or for dev work. The trade off is that it takes time for the Aurora application instances in AWS's infrastructure to spin up after scaling down. The Aurora app servers are stateless and don't hold your data, when they spin up the read from the DB shared storage in your account. AWS manages the instances, so it's "serverless" on your end. **Can only be used in 1 AZ, you have to multi-AZ redundancy. Failover takes a long time. A scaled down cluster takes time to scale up to facilitate reads. You can only access an Aurora serverless DB from a PrivateLink and VPC endpoints. Does NOT support inter-region VPC peering or VPNs for access. You can use an API with Aurora serverless.**

DynamoDB is a NoSQL database. In a NoSQL DB, tables are still used but in that tables instead of rows/columns you have items that nest attributes inside them. An item is just a row. Few key concepts incl. **Tables, Items, Sort Key, Attributes,** and **Operations.**

A table is just a collection of roles with no schema but it needs at least a simple primary key which is made of one partition key. The name needs to be unique within an AWS region. This unique ARN can be referenced form anywhere in the same AWS account.

An item in a table needs to be uniquely identified using the table's primary key. The key is the only mandatory attribute for an item. All other attributes can be optional. Different items in a table can have/use diff. attributes.

A primary key is how items are uniquely identified. It must be unique for each item in the table. If the primary key can't be unique you can add an optional sort key. A use case is when you have multiple entries for the same date but each entry at a different time. When using a primary and a sort key together, they form a **composite key**. The composite key is your last shot at making the record's key unique in the table. The primary or composite key in a DyanmoDB table is its **primary index.**

An item can have their own attributes. There is no rigid structure. An item can have different attributes from other items in the same table, with different data types. There is no defined data type for a column, it's just all JSON text to the DB engine.

You don't have to worry about data resiliency with DynamoDB because it is resilient regionally. Data copies are stored in at least 3 AZs without any further configuration.

You can perform 4 operations on a DynamoDB table – GET, PUT, QUERY, SCAN.

You GET data from a specific *item* in a database table. You PUT an item into a table. You SCAN all items in a table to find an item matching your criteria, and you QUERY items with filters to find only items matching your filter's criteria.

SCAN is a crap operation to perform but is the most flexible. You're reading every item in the table. Even if you add filters, those are just view filters, you're still going top-down all items then only displaying the filtered results. You USE A LOT OF RCUs when doing a SCAN.

QUERY is better. You use either the partition or sort key to narrow down the scope of the lookup. This way you only consume the RCU for the number of items found in the table. BUT the limitation is you can only query on a single value for the partition key. You can combine a partition key QUERY with a filter to return all partition keys beginning with x…for instance.

SCAN = more flexible, QUERY = more efficient.

You can manually enable **point-in-time recovery** which allows the database to be recovered to any point in time within the last 35 days, or perform a manual backup for a full backup of the table. You must restore backups to a new table with a unique name.

Dexter's "Barely passed" AWS Cram Notes

You don't worry about encryption in DynamoDB, it's enabled by default and you can't disable it but you can choose to either use Default (AWS MK) or use KMS+your CMK.

Global Tables is a feature in DynamoDB that allows you to create replicas of a table in one region and store them in other regions. Global tables require the **streams** feature to be enabled. When enabled, copies of your table exists in multiple AWS regions. All tables can be read and written to.

Cloudwatch Integration – full CW integration for your tables.

You don't use Dynamo if presented with relational data, only for key-values, JSON docs, or complex data types. It's a DBaaS, you purchase capacity units, not servers. It's a lightweight DB for non-relational data without a schema.

DynamoDB Partitions - A DynamoDB database starts with a table. A table is made of 1/+ partitions. As the number of items in a table grows, more partitions are added. Data is then divided amongst all the partitions. Any requests for capacity is balanced equally amongst all the partitions at that time. Each partition is replicated to at least **3 AZ.**

Reading Data from DynamoDB table - Each partition is made on 1 leader node + 2 non-leader nodes. When you read data you can choose btw. a **strongly consistent read** which means you only get it from the leader node, or an **eventually consistent read** where you can read from any of the non-leader nodes in addt. To the leader node (but you take the chance that data hasn't been replicated to the non-leader nodes yet).

You have 2 capacity modes with DynamoDB – On-demand and Provisioned. On-demand means you don't worry about performance, you pay as you go and your cost depends on the amnt. Of reads and writes performed. Provisioned means you specify an RCU and a WCU for a table. 1 RCU = 4KB/s, 1 WCU = 1KB/s. You worry about RCP/WCU calculations if using provisioned capacity mode. On-demand means you don't care and "just let it scale"

RCU = (ITEM SIZE (rounded up to the next 4KB multiplier) / 4KB) * # of items (Round up to the nearest 4 KB multiplier)

WCU = (ITEM SIZE (rounded up to the next 1KB multiplier) / 1KB) * # of items (Round up to the nearest 1 KB multiplier)

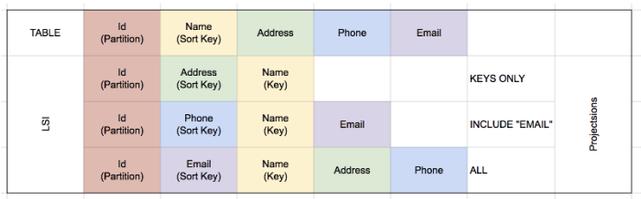
A DynamoDB stream is like a transaction log. It captures a time-ordered sequence of item-level changes to any DynamoDB table and stores this in a log for up to 24 hours. Applications can access the log and view data items as they appeared before and after they were changed in near real time. A DynamoDB stream is an ordered flow of information about changes to items in a DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table. Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A stream record contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams helps ensure the following:

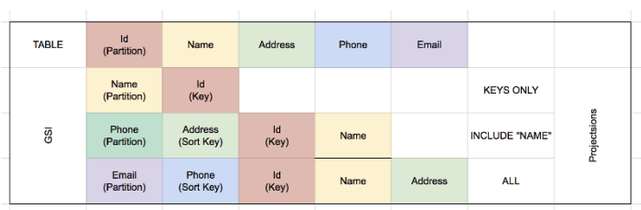
- Each stream record appears exactly once in the stream.
- For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

DynamoDB Indexes. Remember the table first uses a primary index for primary key attributes. You can specify a secondary index which allows you to manipulate data based on attributes that are *different* from that of the primary key. This avoids you having to perform a SCAN on the table, you can use the secondary index to find matching records instead. It's for access patterns that don't fall in line with the item structure. There are 2 types of secondary indexes – LSI and GSI.

LSI has the same partition key as the table, but with different sort key. If I want a subset of data that uses the same address, the second set will return the result much quicker.



GSI has different partition key from the table. This will make querying from different set of objects through an attribute much faster.



LSI - limited to 5, shares performance w/ main table, can only create them at time of table creation. GSI - limit of 20 but can be increased using support ticket and can be created after the fact. GSI allow specify alternative partitions and sort keys, LSI keep the partition key but allows an alternative sort key.

----------CACHING----------

There are 2 in-memory caching solutions in AWS – DAX and ElastiCache.

DAX is for DynamoDB Accelerator. It runs on 1/+ nodes within a VPC. The first time you read from DynamoDB the items are stored inside DAX in addition to being returned to the app. The next time these items are read again they're served out by DAX. DAX is made of 2 caches, an item cache and a query cache. Item cache is filled with results from GetItem and BatchGetItem and has a TTL = 5 mins. Query Cache stores results from QUERY and SCAN ops. You use DAX for the fastest response times and for apps that read small # of items very rapidly. Read-intensive apps w/o allocating a lot of RCUs to the table. It's not for strongly consistent or eventually consistent reads, or for write intensive apps.

ElastiCache is an in-mem cache similar to DAX but can work with non-DynamoDB products as well. Uses either REDIS or Memcached. It works by offloading DB reads by caching responses to avoid having to read the DB table every time. It also works with EC2 to store session state data where ELB is used.

----------LOAD-BALANCING AND AUTOSCALING----------

3 families of AWS LB are the Classic LB, ALB and the NLB. The CLB is the first and oldest type of LB. You shouldn't use this one because they're basic L3-4 devices that don't understand applications. **A CLB only supports 1 SSL cert per CLB so you need multiple if you have many domains/SSL certs.** Can be deployed as internet-facing or internal. You configure the CLB with a listener to define what protocol/ports to listen to. The CLB can perform SSL-offloading. Multiple AZ or subnets can be selected and AWS will deploy a CLB instance in each subnet. Is controlled by security groups which defines what can connect to the CLB. Back-end instance security groups should be locked down to only permit traffic from the CLB. **A CLB can perform health checks** which uses either TCP or HTTP requests to determine if the back end instances are still responsive. If a host fails the CLB auto-routes traffic to the remaining instances in the cluster. Health check thresholds can be configured. **The Connection Draining feature** basically tells the LB to NOT break existing connections when taking an instance out of service and let those existing connections complete. The ELB will allow existing connections to the instances to complete (or for a timeout to expire) before taking that instance offline, it will not make any new connections to the instances being taken down.

The ALB is the newer type of LB designed to replace the CLB. It runs at layer 7 and understands HTTP/HTTPS. You

create an ALB the same way you would a CLB except it works differently. **The ALB introduces the concept of a target and a target group.** A target can be an Ec2 instance, ECS/EKS container, Lambda function. A target group is a collection of targets. You point the ALB to the target group. You create a health check for the target group. You can register more EC2 instances to the target group. Ec2 instance security groups **will need to be configured to allow the ALB to talk to them.** Load balancing policies in the ALB are configured using **rules.**

ALB rules tell how ALB listeners should route traffic. For a CLB, every unique DNS name would need its own CLB to serve it. An ALB can support multiple DNS names and route to diff. instances in one single deployment. ALB rules formatted in IF…THEN format. IF conditions are based on host headers, actions can be to direct traffic to certain target group. **Unlike the CLB, the ALB understands the HTTP protocol and can make decisions based on the header information within the protocol's packets, which allows for a rule-based load-balancing architecture.** It is also **cost-efficient as this allows one ALB to support many websites. ALB is the preferred product to use in an LB architecture.**

NLB is used for load-balancing any TCP or UDP based protocols that are NOT HTTP/HTTPS. It's VERY FAST because it doesn't worry about application level protocols, and can scale to very high workloads up to millions of requests per sec. You can assign a static IP to an NLB and it can LB targets *outside* a VPC. It supports anything that is IP based. Again, for IP protocols that are anything BUT HTTP or HTTPS.

Auto-scaling is an Ec2 feature that allows a cluster of Ec2 instances to be elastic.

Auto-scaling is made of 2 major components – Launch configurations and Auto-Scaling Groups. A launch config is a list of options you specify when creating an instance – same as you would manually creating one (AMI, instance type/class, storage, key/pair, IAM role, userdata, etc). **The limitation with a launch config is that YOU CANNOT MODIFY A LAUNCH CONFIG ONCE CREATED, YOU NEED TO CREATE A NEW ONE. This is because a launch config may already be associated to an AS group so if you modify it that can mess up the group. You can create a new launch config and associate it to the AS group.**

Launch templates do the same thing as launch configs but are better. It has more functionality like versioning, inheritance from a base template, tagging, and **must be used with newer types of EC2 instances (Elastic GPU, T2/T3 unlimited settings, placement groups, capacity reservations, and tenancy options).** Launch templates support versioning, while you can't modify an existing launch template you can create a new version. **You can launch an instance manually from a launch template but not a launch config. You use AMI baking and launch templates to spin up a baseline instance, then use user data to add custom apps and further customize the instance to your liking.**

Neither the launch template nor the launch config can be changed after creation. With launch templates you can create new versions, but the original copy can never be changed. Launch templates can be used to access new features of the new Ec2 instance types. You can use a launch template to launch an Ec2 instance directly, you CANNOT do this with a launch config. Wherever possible, prefer launch templates over launch configurations

AUTOSCALING. Where the launch template define "what" you want to provision, Autoscaling defines "how" you want them provisioned. The base entity is an AS group. An AS group uses a launch template to define new instances. You can define the min, the max, and the desired number of instances to have in an AS group. You configure networking and AZ that the AS group can use and it will try to maintain an equal number of instances btw. those AZs. An AS group monitors metrics within an instance and can perform health checks. A health check has a grace period that allows a newly provisioned instance in the group to full initialize before checking them. The AS group can auto-tag newly created instances.

"How" an AS group scales is defined in **scaling policies.** 3 types – simple, target, and stepped. **Simple** is to monitor a metric as a condition (like CPU use), then take actions. **Target** is to specify a target metric value (like CPU below 20%), if the actual goes above, as many instances are added to the group as required to bring down the metric to acceptable levels. **Stepped** allows for diff. actions to be taken depending on certain thresholds. Like if CPU is 20-50% then add 1 instance, else if CPU is 70-90% then add 5 instances to the group. Scaling policies are triggered by **alarms.** You first create an alarm. When triggered, autoscaling change the desired instances in the AS group config automatically to respond to changes to conditions. After taking an action, it then **pauses for the duration of the configured cooldown timer** before checking the next action to take.

You can target an ALB to load balance an AS group. Any new instances spun up as part of that AS group will auto-register with the ALB.

An AS group will auto-terminate any unhealthy instances, it uses EC2 health checks in the background and it also monitors the instance host health as well. If integrating with an ELB, an AS group can just use ELB's health checks. Any instances that fail the health check will be auto-terminated, this will cause the # of running instances to fall below the desired #. AS will auto-provision new instances to bring the number up to desired capacity.

Highly available, self-healing infrastructure = Launch templates + autoscaling group + ELB.

----------VPC VPN----------

In a VPC VPN Deployment, a Customer Gateway is the remote peer, it connects via an IPSec tunnel to an AWS VPG. You use an S2S VPN when you need (1) quick setup time, (2) to service low or sporadic usage connectivity, (3) a cheap way to securely connect 2 sites. A VPN can be deployed in 1 of 3 architectures – No HA (1 VPG to 1 CG), AWS HA (2 VPG to 1 CG), or full HA (2 VPG to 2 CG). Routing within a VPC VPN follows the standard route metric preference – local routes, then longest-prefix match, then static routes, then dynamic routes, then the default route.

AWS DirectConnect (DX) is a physical fiber connection between your network in a supported Colo and AWS's infrastructure. You can select a port speed of either 1Gbps or 10Gbps. Once a physical connection is provisioned you can configure 2 types of VIFs – **public VIFs** connect your network to public AWS service endpoints like S3/DynamoDB. **Private VIFs** connect on-prem networks to a VPC's VPG. Once the physical connection is created it is your dedicated link to AWS. **Used when you need speed *and* consistency of that low speed. But it takes at least a week to provision and you need to be hosted within a DX location. It is NOT ENCRYPTED BY DEFAULT but you can run a VPN OVER A DX connection.**

Use DX for performance, large data transfer volumes, dedicated bandwidth and for long-term AWS connections. **Use VPN** for urgent needs, when you have low-end hardware on-site, when you need encryption in transit, when you want HA deployment flexibility, and while you wait for a DX connection to be provisioned.

Using both IS an option.

----------SNOW*----------

For moving data into AWS. Basically a large HDD enclosure. There are 3 types – Snowball, Snowball Edge, and Snowmobile. **Snowball** is a hard drive enclosure for migrating 10TB-10PB of data to AWS. You start a snowball job and AWS ships a drive to you then you put your data and ship it back to AWS. Drive is encrypted. Turnaround time of a few weeks. You can use either the Snowball client or S3 adapter to move data onto the drive. **Snowball Edge** is basically Snowball but with local compute ASIC and has larger size of 100 TB raw. It can connect to the network via copper or fiber. It is mounted using NFS. **Snowmobile** is a truck. It's a datacenter on a truck with a large portable storage, you don't use this for anything less than 10 PB. It's for EXABYTES of DATA.

Dexter's "Barely passed" AWS Cram Notes

----------Migrating Data and DBs to AWS----------
Storage Gateway is used to extend AWS storage services to on-prem for either data migration or to use as a local network storage service. It's a virtual appliance and comes in 3 types – File, volume, tape gateway. **File Gateways** are VMs that you deploy and it creates an SMB share and anything put into this SMB share are stored into S3. **Volume gateways** are the same but the VMs expose iSCSI mount points that can be used to mount as volumes. They're block storage so you just can't access the files individually without mounting the entire FS. **GW-stored VGs** store all the data on the GW first then a snapshot is periodically uploaded to S3. **GW-cached VGs** stores the primary data in S3 but caches frequently accessed data on the local gw. You CANNOT access this data from S3 directly.

VTL presents a tape library over iSCSI to a compatible backup software. Very high admin overhead, very expensive, very high risk. Most of the data stored in S3 w/ some small amnt. Of cacing. You can take a tape from the VTL and place it on the shelf which moves the data from S3 into Glacier for long term archival. This is used to move the overhead of tape backup, as a way to migrate data over time to AWS storage gateway or extend a backup system to AWS.

DMS is a service that allows to migrate RELATIONAL DATABASES to AWS. You can do either an **offline migration** where you stop a DB, take a full backup, and restore it on RDS and then reconfigure everything to point to RDS, or you can do an **online migration** where data is slowly replicated while the primary DB stays online, then you have a short cutover window to change apps to point to RDS. **DMS is an online DB migration tool –** you create a replication compute instance that runs to migrate data from Oracle/MS SQL/MySQL/PostgreSQL/MongoDB/Aurora to Redshift/S3/DynamoDB. It comes with a **schema conversion tool** (**AWS SCT)** to help transform data structure btw. DB engines as part of the migration. No outage is incurred during migration. You only pay for when the DMS instance is running.

----------Application Integration----------
The 3 application integration products are SQS, SNS, and Elastic Transcoder. **SNS** is a pub-sub messaging service. You create a topic, other entities subscribe to that topic. Publishers send messages to a topic which is received by all subscribers. **A publisher is anything that can generate a notification like CW alarms, EC2, custom apps.** Is cross-AZ resilient by nature w/o any added config. Automatically scales to msg volume. It's a public service so you need an INET GW or NAT GW to access from inside a VPC. A topic can support both KMS encryption at rest and HTTPS/TLS in transit. **Resource policies can be used to restrict who can send msgs to a topic.** Msgs can be up to 256 KB in size. First you create a topic to define who can send and to aggregate msgs to that topic, then you create a subscription for that topic which defines an endpoint. An endpoint determines how subscribers get messages – HTTP/HTTPS, EMAIL, SQS, LAMBDA, APP ENDPOINTS, SMS. You add entities to a subscription which then receives the topic's msgs in a way determined by the endpoint. Remember – TOPIC + SUBSCRIBERS + PUBLISHERS + ENDPOINT.

You can use SNS in an SQS fan-out architecture.
https://aws.amazon.com/getting-started/tutorials/send-fanout-event-notifications/

SQS is a separate product that compliments any solution already using SNS. Basically a highly available message queue, a front-end worker node drops a msg into the queue for many other back-end nodes to pick up. Messages up to 256KB. Any msgs larger than this should have content stored in S3 and the message refer to that content in S3. You **poll** an SQS queue to take msg out of the queue – you can either do **short polling** or **long polling. Short polling** is when you send 1 API call, looks for any msgs and retrieves up to 10 if any, then disconnects. If there are no msgs you disconnect right away. **Long polling** is when you send an API call but you wait around for the duration of the WAIT_TIME, and anything that's delivered in this time is retrieved.

After a msg is retrieved it is hidden from the queue for the duration of the VISIBILITY_TIMEOUT. This prevents repeat retrievals while allowing for retries in event of a query failure. **Msgs are never automatically deleted from an SQS queue, the receiver must delete the msg using that msg's ReceiptHandle.** With SQS you get 2 queue types – FIFO and Standard. A **standard queue** has near unlimited throughput but msgs are send with a guarantee of at least 1 delivery attempt but could be in any order and could have duplicates. A **FIFO** queue ensures that each msg is delivered only once and in the order they're received in.

When you poll a message in a queue, it only gets hidden for the duration of the visibility timeout and doesn't get deleted. To permanently delete a msg use the message's ReceiptHandle to reference that msg. You can configure resource policies on a queue, encryption, and cloudWatch monitoring. You can invoke Lambda functions whenever a msg is added to as queue.

Elastic Transcoder is a media conversion service. You create a **pipeline** which is a queue of jobs processed in order and those jobs retrieve source media files from an S3 bucket to perform conversion. Once a pipeline is created, 1/+ **conversion jobs** can be created. Jobs are processed in the order they were created.

AWS Elemental MediaConvert is a file-based video conversion service that transforms media into formats required for traditional broadcast and for Internet streaming to multi-screen devices.

----------Analytics----------

Athena is a dataset querying service that allows you to perform queries against data w/o having to pre-create a schema. You store data on S3 (avoids having to pay twice for data) and use Athena to perform queries and it will format the output as a table per the schema. Athena performs **schema on read** meaning that you define the schema in advance that says how you want the OUTPUT TABLE formatted, Athena gathers the requested data and formats it on display as per the schema. You only pay for the queries run by Athena and the storage used by S3.

Athena uses the concept of databases but it's not a DB. It's just an unstructured container. You define a virtual table. You run SQL queries against a set of data, Athena "reads" the data through the lens of the schema and formats the output as a table. The original data is left unchanged. Good for ad-hoc queries where you don't want to be running and maintaining a database.

Other Analytical offerings by AWS include EMR, Kinesis and Firehose, and Redshift.

EMR is used to analyse big datasets of semi-structured and unstructured data. It's a managed Hadoop instance. You create an EMR cluster made of nodes run on EC2 instances managed by the service. Data is stored in S3 and can also be used as output location for the analysis. The cluster runs a shared EMRFS file system which is an enhancement of HDFS on S3 that allows data to persist even if the cluster goes down. You use it for ad-hoc short-term big-data analysis. You can use reserved instances for long-running analysis tasks.

Kinesis is a real-time data streaming platform designed to ingest huge quantities of data in real time and doesn't have a max. performance limit. **Producers** put data into a kinesis stream via the API. Kinesis stores a 24-hour rolling window of this data. A stream can scale indefinitely using shards, A shard gives 1MB of ingestion capacity and 2MB of consumption capacity per second. A data record can be up to 1 MB in size.

Kinesis is NOT a queuing system unlike SQS. SQS is designed to take msg from one side and send it off to one entity on the other side. Kinesis is designed to stream real-time.

You add Firehose to Kinesis to allow streaming data from Kinesis to be stored permanently into S3, Elastisearch Clusters, and Splunk.

SQS used for decouple things, Kinesis is used for massive scale data ingestion, Firehose is used to store streaming data persistently

Redshift is a petabyte scale data warehousing solution. It's a column-based database used for long-term batch reporting. It's used as a final data storage point for other analytical products and intermediate databases.

Dexter's "Barely passed" AWS Cram Notes

---------- LOGGING AND MONITORING ----------

AWS Logging and monitoring solutions include CloudWatch (and CloudWatch Logs), CloudTrail, and VPC Flow Logs.

Cloudwatch is the de facto AWS monitoring solution, it's used by all AWS products to send metrics for analysis and operations. It has an API and an agent so that non-AWS systems can send data to Cloudwatch for monitoring. **Cloudwatch monitors metrics** which is basically a collection of time series data points for a specific measurement(s). The frequency that data is injected into CW depends on the AWS service and may be configurable.

Cloudwatch data automatically changes granularity depending on the age of that data. 60 secs for the last 3 hours, 60 seconds for data older than 3 hours but within the last 15 days, every 5 mins for anything older than 15 days up to 63 days, and hourly for anything older than 2 months up to 400+ days. The older the data, the less granular it becomes

Data is added to metrics, metrics are grouped into namespacs.

Metrics allow you to create alarms. An alarm forms the basis for Cloudwatch actions. Alarms trigger based on metric threshold values. A threshold can be static or anomaly. Alarms trigger actions to be taken. Actions can be as simple as a notification, or as complex like adjusting an AS group.

Beyond CloudWatch which is a monitoring tool, the rest are logging tools. That would be **CW Logs, CloudTrail, and VPC Flow Logs. CloudWatch logs is used to store log data for any events inside AWS products deployed in the environment.** The base entity is a log event which is made of many timestamped data. A collection of log events is a log stream. Multiple log streams can be group into a log group. You configure params on the log group like filters, retention and monitoring. **Metric filters are configured for the log group, nothing else.** Metric filters can be used in CW alarms to take actions.

CloudTrail is not real-time, it records audit activity of actions taken by users, roles, or services in an AWS account. Enabled by default. Records up to 90 days of activity in an AWS account by default. An event is a JSON entry that is stored in the CloudTrail Event History. A **trail** can be configured to define advanced options for events like what is logged and where the log data should be logged to. A trail is configured **per region but if you create it within an AWS organization it can apply to all regions and member AWS accounts.** A trail can monitor either mgmt. events (like account changes), or data events (like S3 or Lambda events). You can encrypt log data using SSE-KMS, configure checksums to detect tampering, or send SNS notifications for new log batches. **Logs are delivered in batches to destinations.**

VPC flow logs are used as traffic debugs for packets going in and out of interfaces being monitored. It does NOT monitor the contents of the packet, only the packet's header information. Flow logs are not real time and does not capture all types of traffic. You can attach flow logs to a VPC which then monitors all attached ENIs on that VPC, a specific subnet which only monitors ENIs within that subnet, or directly on one ENI. A flow log relies on IAM roles to write to log destinations (CW logs or S3 buckets). Any logs created at one level auto-applies to entities at lower levels (i.e.: enable on subnet, applies to all attached ENIs…enable on VPC, applies to all subnets and ENIs and GWs). **It does NOT capture traffic to the AWS DNS server, to instance metadata URLs, windows licensing, DHCP, or VPC router internal traffic.**
https://www.sumologic.com/insight/aws-vpc/

----------OPERATIONS----------

Cloudwatch Events is a sub-component of Cloudwatch that adds real time event monitoring in an AWS account. It delivers a near real-time stream of system events that describe changes in Amazon Web Services (AWS) resources. Using simple rules that you can quickly set up, you can match events and route them to one or more target functions or streams. CloudWatch Events becomes aware of operational changes as they occur. CloudWatch Events responds to these operational changes and takes corrective action as necessary, by sending messages to respond to the environment, activating functions, making changes, and capturing state information.

CW events uses rules to match specific event patterns and delivers "actions" to targets. Rules can be time-invoked or event-invoked. You use CW events to take actions from sources and take actions on targets. **Events/patterns/auto-remediating/responding to certain events in an AWS account = CW events.**

----------AWS KMS----------

Whenever you use encryption in AWS, you use KMS in one way or another. KMS manages Customer-Managed Keys (CMKs). There are 3 types of CMKs – **AWS Owned** (used by AWS back-end and not available for use), **AWS managed** (when AWS products manage the encryption and keys for you, you can't manipulate these), and **Customer managed** (KMS keys that you create and manage).

If you're configuring encryption inside an AWS product and you get to select which key to use, it's likely to be a **Customer-Managed CMK**. If using default encryption, it's likely to be **AWS Managed CMK**.

For a customer-managed CMK, you can set to rotate every year. AWS managed CMK, rotate every 3 years is the only option and can't be changed.

AWS Re-Encrypt operation is when you provide it with the encrypted ciphertext, and provide it with the new key to use, KMS will take the data and re-encrypt it using the new key without ever having to see the plaintext version of that data

YOU HAVE TO USE A DATA-ENCRYPTION KEY (DEK) TO ENCRYPT ANYTHING LARGER THAN 4 KB. Envelope encryption is used to encrypt large files. You give AWS your CMK, AWS returns a plaintext version and encrypted version of the DEK. Use the plaintext DEK to encrypt the file, store the encrypted DEK with the encrypted file.

To decrypt, pass the encrypted DEK to KMS, KMS CMK to decrypt the DEK and sends back the plaintext DEK to decrypt the file.

If it's using encryption, it's using KMS.

FIPS 140-2 Level 3 = CludHSM.

----------DEPLOYMENT----------

Elastic Beanstalk is a full PaaS product that eliminates the need for you to manage any infrastructure.

OpsWorks is used to manage large infrastructure deployments in AWS.