A gentle introduction to AWS

Detailed documentation: <https://docs.aws.amazon.com/index.html>

Whitepapers: <https://aws.amazon.com/whitepapers/>

White papers to read and review before exam:

<https://docs.aws.amazon.com/whitepapers/latest/aws-overview/aws-overview.pdf>

Skills heatmap

|  |  |
| --- | --- |
| Service | Level (out of 5) |
| IAM | 3 |
| KMS |  |
| CloudFormation | 4 |
| EC2 | 2 |
| MSK | 4 |
| S3 | 2 |
|  |  |

**Foundation of Cloud Computing**

## Understand Cloud Computing

Cloud = Servers in data centers that connect to

Services:

* Compute: EC2, Lambda
* Networking: VPC, Direct Connect
* Storage: S3, EBS
* Analytics: Athena, Redshift
* Development: Cloud9, CodeCommit
* Security: AIM, Macie
* Databases: RDS, DynamoDB

<https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html>

Physical server -> Virtual machines for you to use (own memory, networking ...)

Usage:

* On demand, Pay as you go: pay by hour / second

**Advantages of cloud computing**

1. Go global in minutes, AWS already has world coverage
2. No need to run and maintain data centres
3. Benefit from economy of scale from AWS
4. Innovate quickly, as no need to set up computing resources
5. No need to predict scale, it can expand and shrink as you wish
6. Pay for what you use, so potentially saves cost

Benefits in technical terms:

1. High availability = always up and running
2. Elasticity = expand and shrink like a rubber band when it comes to scale
3. Agility = innovate faster for your business
4. Durability = safety of data
5. Security

AWS allows businesses to focus on business value and revenue-generating activities, as opposed to managing infrastructure

Total cost of ownership

CapEx = upfront cost for hardware

OpEx = day-to-day costs of operation

Software licensing costs may change when migrating to the cloud

**Cloud Computing and Deployment Models**

Computing models

IaaS: building blocks, like web hosting (pay monthly subscription to have a hosting company serve your website)

SaaS: complete application (e.g., SageMaker), like email provider

PaaS: developer service (for development of software without worrying about infrastructure), like storefront website, think Heroku

Deployment models:

Private cloud: internal db and local network within company, this gives security

Public cloud: AWS, provides all benefits of public cloud

Hybrid cloud: Barclays. Sensitive data in private cloud, computation in public cloud. This is supported by **Direct Connect**

**Leveraging the AWS Global Infrastructure**

Region -> AZs -> Data centres -> Servers

Region is a physical location.

Each region is fully independent, so if one region blows up other regions are not affected. Each region has >= 2 AZs

Availability Zones (AZs): >= 1 physically separated data centres, each with redundant power, networking, and connectivity (all within 100km with each other)

There can be many AZs in one region. E.g.

N. Virginia region

US-EAST-1A ~ US-EAST-1F

AZs in a region are physically separated, connected through low-latency links, fault tolerant (one AZ goes down others are still up), allows for high availability

^^ A multi-zone service (spanning across many AZs) can add resiliency, if AZ fails, an adjacent AZ in the same region may be spun up to take requests.

Edge locations

* "mini data centre"
* Purely used to cache your static content to reduce latency (latency = response time - request time)
  + ^ by physically located close to users
* Many more edge locations than regions or AZs
* Currently ~215 edge locations
* Used by content delivery services like CloudFront and Global Accelerator

When to use multiple AZs?

* For high availability
* For high resiliency

When to use multiple regions?

* Disaster recovery
* Low latency
* Data sovereignty

**The AWS Account**

AWS Management Console: console (GUI) to manage cloud services

Root user: email used to create the account, very powerful (like sudo), protect with MFA

AWS CLI: console features in the command line. Programmatic access = access AWS resources from code (SDKs)

Tasks that require the root user: link [here](https://docs.aws.amazon.com/general/latest/gr/root-vs-iam.html#aws_tasks-that-require-root)

* Change account settings (name, email, root pswd, payment currency preferences ...)
* Restore IAM permissions
* Configure MFA
* Close AWS account
* .. and a few more

Moral: use root user only for tasks that only root user can do

**Technology**

Ways to access AWS:

* Programmatic access / APIs (think boto3 for Python)
* SDKs
* AWS Management Console (web app)
* CLI
* IaC

**Compute Services 💻**

**EC2 - Elastic Compute Cloud**

Rent and manage virtual servers (instances) in the cloud, that is elastic

EC2 is not "serverless"

Workflow:

1. Provision an EC2 instance at the click of a bottom
2. Use a preconfigured template called an Amazon Machine Image (AMI) to launch instance
   1. AMI is a read-only filesystem image (filesystem = directories) that contains an OS and some other software
   2. Region, OS, Architecture (32-bit or 64-bit), launch permissions, storage for root
   3. AMIs are region specific
   4. If the AMI is deleted, the EC2 instances that are running on the AMI will still be running, as the AMI is already installed. The issue only comes up when we try to build new machines with that AMI.
   5. Backed by EBS or Instance Store
      1. EBS-backed EC2s can be stopped by Instance Store-backed EC2s can’t (as the data will be gone)
3. Deploy applications directly to EC2 instances
4. 750 compute hours per month on Free Tier

Example use: deploy db, deploy a web application

Ways to access EC2 instances:

* Through console
* SSH
  + Generate key pair (private and public key)
* ECI (EC2 Instance Connect), use IAM policies to remove need for SSH $$
* AWS systems manager

EC2 pricing models:

* On demand, good if application unpredictable, or no long-term commitment
* Spot: computation only run when there is unused EC2 capacity, can save up to 90% of on-demand prices
  + useful when compute time is unconcerned: big data, containerized workloads, CI/CD & testing, media rendering, high-performance computing
  + Not useful for persistent workloads, critical jobs (don’t want them to be terminated), databases
  + You specify a spot price, when machines below that price becomes available, you will get to use the machines
  + If the spot price goes above your maximum, you have 2 mins to decide if you want to terminate your instance
  + Spot block -> block EC2s from being terminated when hourly price goes above spot price, can be between 1 and 6 hours
  + Diagram

    Description automatically generated
  + One-time spot request -> wait till price is lower than spot price to execute
  + Persistent spot request (see image on RHS)
  + Spot fleets: try and match the target capacity with your price constraints
    - Strategies:
      * capacityOptimized: Spot instances come from the pool with optimal capacity for the number of instances running
      * lowestPrice: Spot instances come from pool with lowest price -> default strategy
      * diversified: Spot instances distributed across all pools
      * InstancePoolsToUseCount: Spot instances distributed across pools you specify (only valid when combined w/ lowestPrice)
* Reserved Instances: commit to particular instance, typically for 1-3 years. Save up to 75% off on-demand prices, can pay all upfront (for a further discount), partial upfront, or no upfront
  + useful when you have steady capacity
* Convertible RIs: can change machine type to higher grade machine type
* Scheduled RIs: capacity for defined time window
* Dedicated hosts: commit to particular physical server, save up to 70% off on-demand prices
  + useful when you have corporate compliance demand (e.g., no multi-tenant virtualization, need dedicated hardware)
  + On-demand or reserved
  + Also useful when the license you have don’t support multi-tenancy or cloud deployments
  + Basically, **licensing = dedicated host**
* Savings plan: commit to computation usage for 1-3 years, save up to 72% off on-demand prices
  + useful to lower your bills across multiple computer services

Features:

* Load balancing (across different servers = EC2 instances)
  + Classic, Application, Gateway, and Network
  + <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html>
  + <https://aws.amazon.com/elasticloadbalancing/features/#Product_comparisons>
  + Application: application layer (HTTP/HTTPS) load balancing through path-based routing
  + Network: transport layer (TCP/SSL), using some flow hash routing algorithm
  + Classic: transport layer, but relationship between load balancer port and container instance port is fixed
  + Gateway: network layer
  + An elastic load balancer will direct traffic to a set of targets (e.g., EC2 instances), making sure traffic is only directed to healthy targets
  + For ELB to offload traffic, register the AZs and targets within those AZs
  + Cross-zone load balancing:
    - Without cross-zone, each zone gets even traffic, which is then evenly distributed to targets inside the zones
    - With cross-zone, all targets in merged zones get even traffic
  + <https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html>
* Auto scaling: adds or replaces EC2 *instances* automatically across AZs
  + This is horizontal scaling (more computers of the same power), not vertical scaling (= upgrading existing computers)
  + Increase *availability* of your applications

#### Auto Scaling

<https://docs.aws.amazon.com/autoscaling/ec2/userguide/get-started-with-ec2-auto-scaling.html>

<https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html>

You can first create EC2 instances using launch templates or launch configuration (templates are recommended as it has the newest set of features)

Auto Scaling Groups can be created to manage instances:

* Making sure the # of instances running fit into the desired range. Performs periodic health checks on instances, terminate ones that fail and launch new replacements
* Spread instances across specified AZs

Launch failures troubleshoot: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ts-as-instancelaunchfailure.html>

Note, when launching EC2s with EBS included in the template, you need to add auto scaling service to the key policy so it can encrypt the EBS volumes when it launch the templates. Details see the link above.

**EC2 misc**

* Name of machines are just tags (Name -> EC2 name)
* Bootstrap Script (script to run during start up to automate set up process)
  + Add it as “User Data” when choosing AMIs
* Metadata
  + curl http://<EC2 IP>/latest/meta-data
* User data = your bootstrap scripts
* Networking cards
  + ENI (elastic network interface): 1 private IPv4 address, 1 public IPv4 address, many IPv6 addresses, MAC address, 1 or more security groups
    - Low cost, high availability
    - Good for network separation -> just have ENIs for each network you have (management, logging …)
  + EN (enhanced networking)
    - Higher I/O performance: 10 ~ 100 Gbps (higher bandwidth and lower latency)
    - Depending on your instance type, EN can be enabled using Elastic Network Adaptor (ENA) or Intel 82599 Virtual Function Interface (VF)
  + EFA (elastic fabric adapter)
    - Lower and consistent latency & higher throughput
    - Good for high performance computing and ML applications
      * EFA can use OS-BYPASS
* Placement groups
  + Cluster PG – grouping of instances within a single AZ -> low network latency and high throughput
  + Spread PG – each in distinct hardware infra -> good for physical separation
  + Partition PG – each with their own power & network -> good in case one instance goes down others will not be impacted
  + Only certain types of instances can be launched in a placement group
  + AWS recommends homogeneous instances within cluster placement groups
  + You can’t merge placement groups
* You can deploy vCenter on the AWS Cloud using VMWare
* EC2 instances assume roles via instance profiles. The credentials are constantly being rotated.

**Lambda**

Serverless code running = run code without managing servers

Allows developer to focus on writing code

Example: Data file -> (upload) -> S3 Storage -> Lambda Function -> DynamoDB Table

User don't need to worry about how that lambda code is executed

Features:

* Support popular programming languages: Java, Go, Python, C# ...
* Comes with dev environment
* Can be triggered
* Scales automatically

Pricing model

* Compute time (the "duration")
* Request count (# of executions)
* Always free (1M free requests each month after free tier access expries) $$

Share responsibility model:

* You: the application code
* AWS: service, coding environment, and language support

**Other compute services**

**Fargate**: serverless compute engine for containers (like Docker)

**Lightsail**: deployment of simple projects that is pre-configured (like WordPress)

**Outposts**: run cloud services in your internal data center

For regulatory needs. AWS will come install servers in your internal data center ​ This supports **hybrid** deployment model

* Outposts rack: Available starting with a single 42U (unit) rack and scale up to 96 racks -> large deployments
* Outposts Servers: Individual servers in 1U or 2U form factor -> small deployments
* Process:
  + Order from AWS console
  + AWS staff will come on-prem to install
  + Launch instances on your Outpost on-site
  + Start building your on-site AWS environment

**Batch**: process large workloads in small chunks (= batches)

**ECS**: Elastic Container Service - fully managed container orchestration service

**EKS**: Elastic Kubernetes Service - fully managed service to run Kubernetes in AWS cloud or on-prem (scheduling containers, managing application availability, storing cluster data, and other key tasks)

**Storage Service 📝**

**S3 (Simple Storage Service)**

#### Overview

S3 basics:

* Objects (or files) are stored in buckets (directories)
* Essentially unlimited storage that can hold millions of objects per bucket (scalable)
* S3 server also scales with # of requests to keep availability high
* Files can be up to 5TB each
* Objects can be private or public -> security can be set at bucket or individual object level
* Can upload objects via the console, the CLI, or programmatically from within code using SDKs

More stuff

* **Versioning**: Can enable versioning to create multiple versions (version ID)
  + So when objects are changed (including deletion), the new object is stored alongside older versions of that object with a new version ID
  + Note if your bucket policy allow public access to all objects in bucket, it doesn’t apply by default to previous versions of these objects. We need ACLs for this
  + We delete the delete marker to restore an object
  + Once versioning is
  + Integrates with lifecycle rules
  + Support MFA (to delete objects)
* Bucket names must be unique globally (even though buckets are regional)
  + [https://[bucket-name].s3.[region].amazonaws.cm/[object-name](https://[bucket-name].s3.[region].amazonaws.cm/%5bobject-name)]
  + S3 is a key-value db, key is the object name and value is the object itself
* **Lifecycle management**: rules to make transitions (delete objects after some time, transition to cheaper storage if not accessed for 30 days …)  
  Can be used in conjunction w/ versioning => perform actions for objects with certain version  
  Note, in lifecycle policy it’s only possible to move objects from more frequent to less frequent access tiers
* **S3 Object Lock**
  + Store objects using write once, read many (WORM) model => achieved using versioning
  + Object lock can be applied on object or entire bucket level
  + 1 Governance mode: most users can’t overwrite or delete an object version or alter its lock settings
  + 2 Compliance mode: a protected object version can’t be overwritten or deleted by any user (incl. root)
  + Retention period: attaches to an object version, protects an object version for a fixed amount of time => this creates a timestamp in the metadata of the object  
    After period expiration, object version can be overwritten or deleted
  + Legal hold: basically a non-expiring retention period on an object. It can be deleted
  + Glacier Vault Lock -> apply S3 object lock to glacier storage
* **Security**
  + Encryption at rest: server-side encryption (though user usually need to turn this on)
    - SSE-S3: S3-managed keys, using AES-256
    - SSE-KMS: AWS KMS-managed keys
    - SSE-C: customer-provided keys
    - Encryption scheme is given in the S3 PUT request
  + Encryption at rest: client-side encryption
    - You encrypt the files yourself before uploading to S3
  + Encryption in transit
    - SSL/TLS, HTTPS
  + ACSs –> permissions for individual objects
  + Bucket policies -> permissions for whole bucket
* **Performance**
  + S3 Prefixes
    - Mybucketname**/folder1/subfolder1**/myfile.jpg  
      prefix = the bolded part above
    - S3 has extremely low latency, you can get the first byte out of S3 within 100-200 milliseconds
    - 3500~5500 requests per second, per prefix
    - So spreading objects over 4 prefixes will improve performance by 4x compared to all under same prefix
    - Note there are KMS encrypt-decrypt quotas (which cannot be change currently) that can impact performance. So maybe use SSE-S3 for better performance
  + Multipart Uploads
    - Recommended for files over 100MB
    - Required for files over 5GB
    - Upload chunks of big file in parallel
  + S3 Byte-Range Fetches
    - Parallelize downloads with multiple byte ranges
    - Speed up download or download part of a file
* **S3 Replication**
  + Replicate from one bucket to another
  + Versioning must be enabled on both the source and destination buckets
  + Once replication is turned on, all subsequent objects will be replicated automatically, but no replication will be done w/ objects that were in the source bucket before. You can either reupload that object or `sync` it over.
  + Delete markers are not replicated by default
  + You can change the storage class for replicated objects
* **Strong Read-After-Write consistency**
  + After a successful write, any subsequent read request immediately receives the latest version of object
  + After a successful write, listing of objects give the last versions of objects
* **Public access (e.g., for hosting static website)**
  + By default, public access is blocked. First thing to do is to uncheck block public access on the bucket
  + Then, we could attach the following statement to bucket policy. Tweak the `Resource` part of the policy to grant access to specific objects you need



* + Another way is to enable ACLs for the objects you need public access for and click “make object public with ACLs” to make it public
* Note you can store OS/database etc., the files are not meant to change, they are statics

Data accessibility

* Durability = will my data be there tomorrow?
* Availability = how quickly / when can I access my data?

S3 has 11 9's of durability and 4 9's of availability by SLA

Service limits

* For each S3 bucket, the # of objects to hold is *infinite*. There is no service limit on this and the bucket will scale horizontally as needed
* The # of S3 buckets in an account does have a service limit. To increase the # of S3 buckets beyond that limit, the user needs to file a service limit increase request.

Storage classes

* Standard: general-purpose storage, data stored **across multiple AZs (>=3)**, low latency and high throughput
  + Good for: frequently accessed data (e.g., content hosting, static websites)
* Intelligent-tiering:
  + Good for: data with unknown or changing access patterns (per-GB storage & pre-GB retrieval)
* Standard-Infrequent Access (IA): access infrequent but requires rapid access, cheaper than standard
  + Cost 20% less than Standard
  + Good for: long-lived data, infrequent access
* One Zone Infrequent Access (IA): only in 1 AZ, so data can be lost!
  + Good for: data that can be re-created, infrequent access
* Glacier Instant Retrieval: retrieval is instant (milliseconds)
  + Good for: long-term very infrequent access data archive, cost very low if retrieval frequency low
* Glacier Flexible Retrieval: retrieval takes long time (from a few minutes to a few hours depending on type => expedited or normal versions)
  + Good for: long-term data archive
* Glacier Deep Archive: even longer retrieval time (12hrs - 48hrs)
  + Good for: long-term data archive
* Outposts: object storage on premise
  + Good for: data needs to be kept locally or low latency needs

S3 always stored in >=3 AZs apart from One Zone IA

Glaciers are charged per retrieval

^^ Basically, the less availability or durability you need, the cheaper

Uses:

* Static websites
* Data archive
* Analytics systems (put data on buckets)
* Mobile applications

Bucket policies vs. IAM policies vs. Bucket ACLs (access control lists)

* IAM policies: can allow certain actions on resources like S3 buckets; attached to identities
* Bucket policies: can only be attached to buckets; contains a list of "principles" which are people that are allowed to perform specified actions on that bucket
* Bucket ACLs: ACLs are old ways of managing controls and will be deprecated soon. The advantage of ACLs over policies is that it can specify controls for each individual object within the bucket, instead of uniform policy for the entire bucket

Other points on S3 bucket policies:

* Previously if you had principal as “Principal: {“AWS”: BUCKET\_OWNER\_ACCOUNT\_ID}”, the policy will only apply to the root user in the bucket owner account. From June 2022 it’ll apply to all Principals in that account.   
    
  A recommendation is to not make the deny statement too broad in accounts, otherwise you might end up with a full lockout.

**Other storage services**

EC2 storage:

* EBS (elastic block storage): a physical storage device (called "volume") that can be attached and removed from instance (~= like hard drive)
  + Data persists when instance not running
  + One instance per EBS, but could have many EBS for each instance
  + Tied to 1 AZ (same AZ as EC2 for low latency)
  + Can resize EBS on the fly
  + Good for: quickly access data, running a db on an instance, and long-term data storage
  + Create a file system, run a db, run an OS, store data, install apps …
  + Built for production workloads, highly available (replicated within single AZ), scalable (dynamically increase capacity with no performance impact)
  + Volume types (at least 1 volume per EC2 machine, called the root volume)
    - gp2/gp3 (general purpose SSD)
    - io1/io2 (Provisioned IOPS SSD -> high-performance)
    - st1 (low-cost HDD volume)
    - sc1 (lowest cost HDD, good for “cold” access)
    - => IOPS vs. Throughput
      * IOPS = I/O per second -> transactional speed, read and write quickly
      * Throughput => bits read or written per second -> bulk speed, read or write a lot
    - *Can change volume types on the fly*
  + Snapshot = a volume at a point in time, consecutive snapshots store the deltas (so first one will take significantly longer to create)
    - Only capture data that’s been written by EBS volume, so best to stop instance then take a snapshot
    - Can share snapshot but can only be shared to same region. You can also just copy over the snapshot to another region before sharing there.
    - They exist in S3
  + Encryption:
    - Low impact on latency
    - If volume encrypted, will also encrypt data in transit & snapshots
    - If volume isn’t encrypted, you can encrypt the EC2 by 1) creating a snapshot, 2) encrypt the snapshot, 3) rebuild the EC2 from AMI built from the now encrypted snapshot
  + EC2 can also hibernate (so RAM (< 150GB) is stored on EBS instead of deleted like in termination) => faster restore time
    - Can’t be hibernated for more than 60 days
* Instance store: local storage physically attached to the host computer (~= like RAM)
  + Data lost with termination of instance (ephemeral)
  + But I/O speed is very very fast
  + Good for: temporary storage needs
* EFS (elastic file system): serverless network file system for sharing files (~= Dropbox)
  + Only support Linux file system (uses NFSv4 protocol)
  + Accessible across different AZs in 1 region
  + Scales automatically, pay for what storage you use (no need to pre-provision)
  + Serves thousands of concurrent users
  + More expensive than EBS
  + Good for: shared directories, content management, web servers
* FSx:
  + FSx for Windows => fully managed native MS Windows file system => easily move Windows-based applications
  + So basically EFS for Windows
  + Also FSx for Lustre which has higher performance

Storage Gateways: hybrid storage service (some in cloud, some locally)

* Good for: moving backups to cloud, reducing costs for hybrid cloud storage ...

ECR: Elastic Container Registry

<https://aws.amazon.com/ecr/>

Basically a repo for compressing, encrypting, and controlling access to container images for your reployment.

AWS backup: backups across AWS services

* Integrates with all AWS storage services (EBS, EC2, EFS ...) $$
* Create a backup plan on retention and access frequency

**Content Delivery Services 🚚**

Content delivery network (CDN): think YouTube

**CloudFront**: CDN that deliver data and applications **globally** with low latency $$

* Speeds up delivery of static and dynamic web content
* Use edge locations to cache content (that's how it's fast)

Users <-(view request/response)-> CloudFront (edge location and distribution cache) <-(origin request/response)-> AWS Cloud (S3 bucket, E2 instance, Elastic Load Balancer)

Note, CloudFront alone does not guarantee high availability! You need to separately deploy to multiple regions or AZs $$

Uses:

* Used with S3 to deploy content globally

Security features of CloudFront: $$

* IP address blocking (= geo restriction)
* Prevention of certain attacks like DDoS (by reducing attack surface, hide your db behind CloudFront)

**Global Accelerator**: sends users through the AWS global network

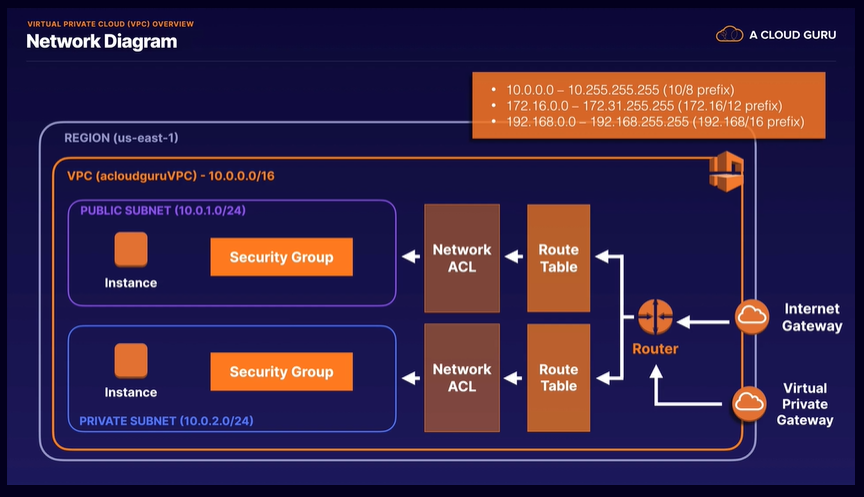
* Improves latency and availability of single-region applications -> up to 60%
* Does this by directing traffic through more available parts of the internet

**S3 Transfer Acceleration**: improves content uploads and downloads to and from S3 buckets

* Fast transfer of files over long distances

**Networking Services: VPC and Subcomponents 🕸**

**Virtual Private Cloud (VPC)**



a protected, own slice of the AWS cloud

* Private virtual network
* Launch resources like EC2 instances inside VPC
* Isolate and protect resources
* Spans AZs in a region

Why do we need VPCs?

* Launch instances
* Assign custom IP addresses and subnets
* Configure route tables between subnets
* Use internet gateway to connect to public internet
* Better security control over AWS resources
* NACL (network access control lists)

Default vs. Custom VPC:

* Default simpler to set up than Custom

Creating a new VPC creates:

* VPC
* Main route table
* Network ACL

Creating a subnet

* A subnet can only span 1 AZ
* 5 addresses are reserved per subnet for AWS routing use
* Eg VPC has CIDR 10.0.0.0/16 and now creating a subnet with CIDR 10.0.0.0/24
  + 10.0.0.0: Network address
  + 10.0.0.1: Reserved for VPC router
  + 10.0.0.2: Reserved for DNS server
  + 10.0.0.3: Future use
  + 10.0.0.255: Network broadcast address

Creating an internet gateway

* Attach it to VPC
* One internet gateway max per VPC
* Create a route table (if we modify the main route table, all subnets without route table associations will be associated with that route table, which isn’t very secure)
* Associate subnets to that route table (make sure these subnets have automatic public IPv4 address assignment enabled)

Let instances in private subnet connect to public internet -> use a NAT gateway

* Redundant inside the AZ (max 1 NAT gateway per AZ)
* Put the NAT gateway in a public subnet
* Starts at 5 Gbps and scales currently to 45 Gbps
* No need to patch
* Not associated with security groups

Different VPCs can connect to each other through *peering*

Many VPCs can be hosted on the same AWS cloud, but are semantically separate from each other through use of 1) private IP addressing and 2) VPN. Hence, it's as if the VPCs are physically separated.

VPN is essentially an extension of a private network to the wider internet, where computers in a global location can be within the same private subnet. The VPN protocol used by AWS for VPCs is IPsec VPN (<https://en.wikipedia.org/wiki/IPsec>)

Example:

* Set up a VPC (10.0.0.0/16)
* Set up a subnet within that (10.0.0.0/24)
* Put a EC2 instance within that subnet
* Open up that subnet to the public by 1) creating an internet gateway, 2) attaching that gateway to the subnet, 3) let subnet allocate for public addresses, and 4) create routing table so public traffic can access the instance through the subnet

Note, by default a private subnet in a VPC can only communicate with other subnets of the same VPC (public or private). To connect to the internet, an internet gateway is required.

Note: S3 buckets are managed AWS services and don’t reside in VPCs. This is unlike EC2s where you can put them in VPCs. It’s good practice to restrict access via bucket policies where only authorized VPC endpoints can access it.

<https://stackoverflow.com/questions/52093540/s3-buckets-are-not-residing-in-vpcs>

Common port numbers:

* Linux SSH (Port 22)
* Windows RDP (Port 3389)
* HTTP Web Browsing (Port 80)
* HTTPS Encrypted Web Browsing (Port 443)

**VPC endpoints**

VPCEs: [https://aws.amazon.com/blogs/architecture/reduce-cost-and-increase-security-with-amazon-vpc-endpoints/#:~:text=A%20VPC%20endpoint%20allows%20you,or%20AWS%20Direct%20Connect%20connection](%20https://aws.amazon.com/blogs/architecture/reduce-cost-and-increase-security-with-amazon-vpc-endpoints/#:~:text=A%20VPC%20endpoint%20allows%20you,or%20AWS%20Direct%20Connect%20connection).

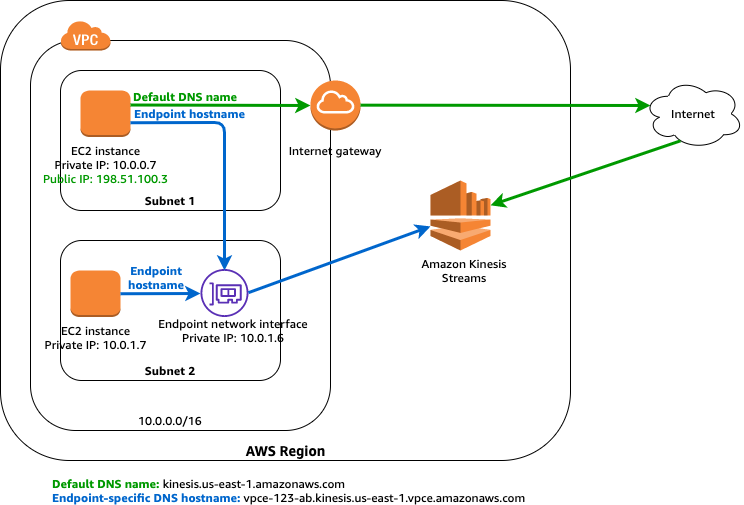
PrivateLink: <https://docs.aws.amazon.com/vpc/latest/privatelink/concepts.html>

A VPC endpoint allows resources within that VPC to access services outside of it. Each endpoint has a designated service resources are allowed to access. VPC endpoint policies control what actions resources in the VPC can take via that endpoint.

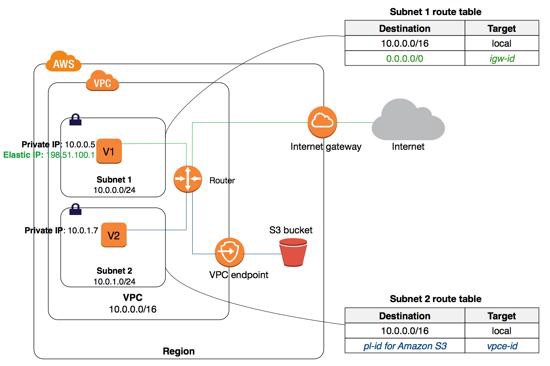
There are 2 types of endpoints, interface and gateway.

Interface endpoints uses AWS PrivateLink to establish secure connection between endpoint and the service, where data will not travel via the public internet. Interface supports most services.

Gateway endpoints needs routing tables to help route traffic outside the VPC to the endpoint. It currently only supports S3 and DynamoDB



An EC2 instance using a VPCe interface to access Kinesis



VPCe gateway set up, with routing table specifying that to connect to S3 we need to go to vpce.

#### Security groups

Security groups control what traffic is allowed inbound and outbound a resource. We specify the restrictions using rules.

Security group basics:

* Naming: must be unique within the VPC, cannot start with “sg-“
* They are stateful. If you send a request from an instance, the response traffic for that request is always allowed. Similarly, if you receive connection from another resource, response traffic out to that resource is always allowed.
* You can specify allow rules only. All traffic is denied by default
* By default, a new sg has one inbound rule that permits all traffic from same security group and one outbound rule that allows all traffic
* When you update a security group, the rules get updated across all resources associated with that group

Security group best practices:

* Authorize IAM principals to create and modify security groups
* Restrict as much as possible (ports, IP ranges …)

Anatomy of a security group rule:

* Protocol(s)
* Port range – specific port or a port range
* ICMP type and code
* Source or destination
  + Single/Range of IPv4/IPv6 addresses
  + ID of a security group
    - This means traffic to and from resources associated with that security group is allowed
    - Note, no rules from that security group is merged into the current rules
* Description (optional)

Example rule:

|  |  |  |  |
| --- | --- | --- | --- |
| **Inbound** | | | |
| **Source** | **Protocol** | **Port range** | **Description** |
| 0.0.0.0/0 | TCP | 80 | Allows inbound HTTP access from all IPv4 addresses |
| ::/0 | TCP | 80 | Allows inbound HTTP access from all IPv6 addresses |
| 0.0.0.0/0 | TCP | 443 | Allows inbound HTTPS access from all IPv4 addresses |
| ::/0 | TCP | 443 | Allows inbound HTTPS access from all IPv6 addresses |
| Your network's public IPv4 address range | TCP | 22 | Allows inbound SSH access from IPv4 IP addresses in your network |
| Your network's public IPv4 address range | TCP | 3389 | Allows inbound RDP access from IPv4 IP addresses in your network |
| **Outbound** | | | |
| **Destination** | **Protocol** | **Port range** | **Description** |
| The ID of the security group for your Microsoft SQL Server database servers | TCP | 1433 | Allow outbound Microsoft SQL Server access |
| The ID of the security group for your MySQL database servers | TCP | 3306 | Allow outbound MySQL access |

Troubleshooting:

Use the AWS Network Reachability Analyser: <https://docs.aws.amazon.com/vpc/latest/reachability/what-is-reachability-analyzer.html>

^^ This is like policy simulator but for security groups

Security group rule aggregation

It’s done permissively. So if *one* security group allows the traffic, the traffic will be allowed.

#### Network ACLs

Network ACL basics:

* Like a firewall to VPC
* Router -> Route table -> Network ACl -> Security -> …
* Default network ACL allows all inbound and outbound traffic
* Custom network ACL by default denies all inbound and outbound traffic
* Only 1 network ACL per subnet, but same ACL can be associated with multiple subnets. If you assign a new network ACL to a subnet, the previous ACL is removed
* ACL = a numbered list of rules that are evaluated in order, starting with the lowest numbered rule
* ACL have separate inbound and outbound rules, each rule can either allow or deny traffic
* ACLs are stateless
* Network ACL further controls traffic on the subnet level  
  
        A VPC with two subnets and a network ACL for each subnet.
      
* A Network ACL contains inbound and outbound rules, numbered from 1 to 32766. Each rule can be an allow or deny. The traffic is matched with the rules from lowest number to highest number. When a rule matches, that rule is applied and the traffic is either allowed or denied based on that rule

Network ACL rules:

* Rule number
* Type (SSH etc.)
* Protocol (<http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml>)
* Port range
* Source/Destination
* Allow/Deny

Default Network ACL:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Inbound** | | | | | |
| **Rule #** | **Type** | **Protocol** | **Port range** | **Source** | **Allow/Deny** |
| 100 | All IPv4 traffic | All | All | 0.0.0.0/0 | ALLOW |
| 101 | All IPv6 traffic | All | All | ::/0 | ALLOW |
| \* | All traffic | All | All | 0.0.0.0/0 | DENY |
| \* | All IPv6 traffic | All | All | ::/0 | DENY |
| **Outbound** | | | | | |
| **Rule #** | **Type** | **Protocol** | **Port range** | **Destination** | **Allow/Deny** |
| 100 | All traffic | All | All | 0.0.0.0/0 | ALLOW |
| 101 | All IPv6 traffic | All | All | ::/0 | ALLOW |
| \* | All traffic | All | All | 0.0.0.0/0 | DENY |
| \* | All IPv6 traffic | All | All | ::/0 | DENY |

The asterisk (\*) matches all traffic and will deny. This is a catch-all rule that denies all traffic that doesn’t match any of the rules. This rule cannot be deleted.

Tips for designing Network ACLs:

* You might want to add a deny rule in a situation where you legitimately need to open a wide range of ports, but there are certain ports within the range that you want to deny. Just make sure to place the deny rule earlier in the table than the rule that allows the wide range of port traffic.
* You add allow rules depending on your use case. For example, you can add a rule that allows outbound TCP and UDP access on port 53 for DNS resolution. For every rule that you add, ensure that there is a corresponding inbound or outbound rule that allows response traffic.

Example Networking ACL:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Inbound** | | | | | | |
| **Rule #** | **Type** | **Protocol** | **Port range** | **Source** | **Allow/Deny** | **Comments** |
| 100 | HTTP | TCP | 80 | 0.0.0.0/0 | ALLOW | Allows inbound HTTP traffic from any IPv4 address. |
| 110 | HTTPS | TCP | 443 | 0.0.0.0/0 | ALLOW | Allows inbound HTTPS traffic from any IPv4 address. |
| 120 | SSH | TCP | 22 | 192.0.2.0/24 | ALLOW | Allows inbound SSH traffic from your home network's public IPv4 address range (over the internet gateway). |
| 130 | RDP | TCP | 3389 | 192.0.2.0/24 | ALLOW | Allows inbound RDP traffic to the web servers from your home network's public IPv4 address range (over the internet gateway). |
| 140 | Custom TCP | TCP | 32768-65535 | 0.0.0.0/0 | ALLOW | Allows inbound return IPv4 traffic from the internet (that is, for requests that originate in the subnet).  This range is an example only. For more information about how to select the appropriate ephemeral port range, see [Ephemeral ports](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html#nacl-ephemeral-ports). |
| \* | All traffic | All | All | 0.0.0.0/0 | DENY | Denies all inbound IPv4 traffic not already handled by a preceding rule (not modifiable). |
| **Outbound** | | | | | | |
| **Rule #** | **Type** | **Protocol** | **Port range** | **Destination** | **Allow/Deny** | **Comments** |
| 100 | HTTP | TCP | 80 | 0.0.0.0/0 | ALLOW | Allows outbound IPv4 HTTP traffic from the subnet to the internet. |
| 110 | HTTPS | TCP | 443 | 0.0.0.0/0 | ALLOW | Allows outbound IPv4 HTTPS traffic from the subnet to the internet. |
| 120 | SSH | TCP | 1024-65535 | 192.0.2.0/24 | ALLOW | Allows outbound SSH traffic from your home network's public IPv4 address range (over the internet gateway). |
| 140 | Custom TCP | TCP | 32768-65535 | 0.0.0.0/0 | ALLOW | Allows outbound IPv4 responses to clients on the internet (for example, serving webpages to people visiting the web servers in the subnet).  This range is an example only. For more information about how to select the appropriate ephemeral port range, see [Ephemeral ports](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html#nacl-ephemeral-ports). |
| \* | All traffic | All | All | 0.0.0.0/0 | DENY | Denies all outbound IPv4 traffic not already handled by a preceding rule (not modifiable). |

Ephemeral ports

* Many Linux kernels (including the Amazon Linux kernel) use ports 32768-61000.
* Requests originating from Elastic Load Balancing use ports 1024-65535.
* Windows operating systems through Windows Server 2003 use ports 1025-5000.
* Windows Server 2008 and later versions use ports 49152-65535.
* A NAT gateway uses ports 1024-65535.
* AWS Lambda functions use ports 1024-65535.

#### Connections between VPCs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | VPC Peering | AWS Transit Gateway | AWS PrivateLink with a single NLB | WS PrivateLink multiple NLB |
| Bandwidth | Limited by instance network performance and flow limits. | Up to 50 Gbps | 10 Gbps per AZ | 10 Gbps per AZ |
| Pricing | Data transfer charge (free if data transfer is within AZs) | Data transfer charge + hourly charge per attachment | Data transfer charge + interface endpoint charge + Network load balancer charge | Data transfer charge + interface endpoint charge + Network load balancer charge |
| Scalability | Recommended for smaller number of VPCs | No limit on number of VPCs | No limit on number of VPCs | No limit on number of VPCs |

VPC peering

Connect 1 VPC with another via a direct network route using private IP addresses.

Instances behave as if they were on the same private network.

Can be cross-account.

Can be cross-region.

No transitive peering!! All peering must be direct.

VPC peering cannot be between overlapping CIDR addresses.

<https://docs.aws.amazon.com/vpc/latest/peering/working-with-vpc-peering.html>

Peering route table config: <https://docs.aws.amazon.com/vpc/latest/peering/peering-configurations.html>

Transit gateway

Diagram

Description automatically generated

Simplify VPC peering connection. Uses hub-and-spoke model (O(N) connections needed to connect all VPCs compared to a mesh, which is O(N^2)).

Transit gateway work across accounts. If multiple accounts are in the same region, their VPCs can connect to same transit gateway and talk to each other.

Only route traffic in single region, but you can connect transit gateways between regions to get cross-region connection.

Can use route tables to limit how VPCs talk to one another.

Transit gateway scales horizontally based on traffic.

Drawbacks:

* 2 hops so higher latency

PrivateLink

A service created when you need to publish a service inside a VPC to hundreds or thousands of client VPCs.

What you can do is to create a NLB on the service and ENI on the clients, then use PrivateLink to have a secure connection between the service and clients.

Note, PrivateLink only work with specific services, not entire VPC.

**Other networking services**

**Route 53**: AWS' DNS service

* Performs health checks on AWS resources
* Supports a hybrid model

**Direct Connect**: Dedicated network connection = super fast

* connects on-premises data center to AWS
* data travels over private network
* Useful for
  + connecting between private cloud and public cloud (in hybrid cloud)
  + transfer data in large datasets
  + transfer data directly to AWS, bypassing ISP
* Gateways for connecting VPCs
  + Attach virtual private gateway from each VPC to your direct connect gateway for cross-region connection
  + For VPCs in the same region, we just need a single instance of a transit gateway, connect all VPCs to it, and connect that to the direct connect gateway

**VPN**: AWS VPN, support same service as direct connect and it's cheaper

**API Gateway**: allows access to resources

* Example: client <-> API gateway <-> lambda <-> RBS db

**Databases 🛄**

Relational: RDS, Aurora

NoSQL: DynamoDB (key-value)

Graph: Neptune (for social interactions, say)

Document: DocumentDB

In-memory: ElastiCache

RDS:

* Support popular db engines (Amazon Aurora, Postgresql, mySQL, MariaDB, Oracle Database, SQL Server ...)
* Manages db with automatic backups, software patching ...
* Multi-AZ to improve availability (but not performance)
  + When primary db is down, its DNS resolves to backup db
* Launch read replicas across AZs and/or regions to increase performance and durability
  + Each read replica has its own endpoint address
  + They can be promoted to be their own databases, but that breads synchronisation
  + Requires automatic backup
  + You create replicas in different regions, then in each replica you can enable multi-AZ
  + Remember, but enable
* Good for online transactional processing (OLTP)
  + OLTP: Data processing and completing large # of small transactions in real time => RDS
  + OLAP: Data analysis over large amounts of data, as well as complex queries that take a long time => Redshift

Aurora (Amazon’s proprietary DB system that is compatible with opensource DBs)

* Compatible with Postgresql (3x faster) and MySQL (5x faster) <- these are the only SQL vendors Aurora supports
* Scales automatically
* Managed by RDS
* 2 copies of data contained in each AZ, minimum of 3 AZs, so 6 copies of your data
* You can share Aurora snapshots with other AWS accounts
* 3 types of replicas available: Aurora replicas, MySQL replicas, and PostgreSQL replicas. Automatic failover only available for Aurora replicas
* Can also be serverless

DynamoDB (like Aurora for NoSQL database)

* All stored on SSD
* Spread across 3 geographically distinct data centers
* Eventual consistent reads (default)
  + Consistency across all copies of data reached within 1 second -> a repeat read after a short time should return the updated data
* Strongly consistent reads
  + Returns a result that reflects all writes received a successful response prior to the read -> no need to wait a second
* DynamoDB Accelerator (DAX)
  + 10x performance improvement
  + Traditional caching: cache <-> app <-> db
    - Application needs to communicate w/ both cache and db
  + DAX: app <-> DAX <-> db
    - Application only communicatees with DAX, where cache misses are managed by DAX itself and communicated to db
* Fully managed and serverless
* Scales automatically => on-demand capacity
* Fulfils ACID (atomic, consistency, isolation, durability) conditions
  + In exam, ACID => DynamoDB
* Backups
  + On-demand backup: no impact on performance or availability, consistent within seconds, retained until deleted
  + Point-in-Time recovery (PITR): restore to any point in the last 35 days, not enabled by default, latest restorable is 5 minutes in the past
* DynamoDB streams
  + Time-ordered sequence of item-level changes in a table
  + Changes are grouped by shard
  + Used to implement DynamoDB global tables
* DynamoDB global tables
  + Managed multi-master, multi-region replication

DocumentDB

* Supports MongoDB
* Fully managed and serverless
  + No need to worry about cluster management, software, configuration backups, or monitoring production workloads
* Usually used to migrate MongoDB to AWS, freeing yourself w/ ops work

Amazon Keyspaces

* Runs Apache Cassandra
  + A distributed database that runs NoSQL
* Fully managed and serverless

ElasticCache

* Compatible with Redis and Memcached
* In-memory datastore

Neptune

* Graph db
* Fully managed and serverless

Amazon Quantum Ledger Databases (QLDB)

* Ledger DB: A NoSQL db that is immutable, transparent, and has a cryptographically verifiable transaction log owned by one authority
  + E.g., for cryptocurrencies, track shipping …
* Not using quantum by the way

Amazon Timestream

* Timeseries data: data by timestamps
  + Examples: IoT (e.g., for agriculture), analytics, DevOps (scaling of services)

**Migration and Transfer Services 🚀**

Inexpensive, secure, and fast ways to migrate data to the cloud

**Database Migration Service (DMS)**: migrate on-premises databases to AWS

* Support homogeneous or heterogeneous migrations (between same db vendor or different)
* ^^ so can migrate on-cloud Oracle db to on-cloud Aurora MySQL

**Server Migration Service (SMS)**: migrate on-premises servers to AWS

* Servers as new AMI
* Launch with AMI to EC2 instances

**AWS Snow Family**: physical product to transfer data

* **Snowcone**: 8TB of usable storage, can transfer offline or online (with DataSync)
* **Snowball and Snowball Edge**: Petabyte-scale data transfer, cheaper than through internet. Edge also comes with some computing resource like EC2 and Lambda (natively supported)
* **Snowmobile**: Multi-petabyte or exabyte scale. Literally physically transfer to S3. Comes with a truck and security vehicles.

**DataSync**:

* Copies data over direct connect or the internet
* Or between AWS storage services
* ^^ cross region or account

**Analytics service 💹**

Data warehouse = a large data storage that aggregates historical data

Useful for querying, reporting, analytics, business intelligence ...

Not used for handling transactions (RDS usually used for that)

**Amazon Redshift**: AWS data warehouse solution, handles exabyte-scale data. Fast and efficient.

Useful for consolidating multiple data sources or use as relational database that doesn't need fast real-time access.

**Athena**: Query service for S3

Pay per query, considered serverless $$

**Glue**: ETL service, prepare and load data

ETL, which stands for **extract, transform and load**, is a data integration process that combines data from multiple data sources into a single, consistent data store that is loaded into a data warehouse or other target system.

**Kinesis**: Analyze real-time streaming data. Support video, audio, click streams, IoT ...

**Elastic MapReduce (EMR)**: Process *large* amount of data. Uses Hadoop and other big data frameworks like Apache Spark

EMR studio: Prototype with SageMaker-like interface, the underlying EMR EKS cluster is independent to studio where you can freely connect, environments are stored in notebooks and so can be shared (“environment as code”), collaborate in real-time, debug in console. To move to prod you need to build your own pipeline.

**Data Pipeline**: Move data between compute and storage services (e.g., between EC2 and Redshift)

Can move on intervals or conditions, sends a notif when successful.

**QuickSight**: Datavis service, like dashboards

Examples:

* Use Athena to query data in your db
* Use kinesis to do fraud detection with real-time data

#### MSK (Amazon Managed Streaming for Apache Kafka)

Manages underlying Apache Kafka infrastructure, so integrates seamlessly with data streaming applications that use Kafka.

It will create, update, and delete clusters as required.

Broker nodes: # of broker nodes per AZ you want at time of cluster creation

ZooKeeper nodes: nodes created for you

Producers, Consumers, and Topics: Kafka data-plane operations to create topics & produce/consume data

Flexible cluster operations: Perform cluster operations with the console

Resilience:

* Automatic detection and recovery from common failure scenarios
* Detection and replacement of unhealthy nodes
* Reduce data: tries to reuse storage from older brokers
* Reduce time required: impact time = detection to recovery, largely shortened

MSK serverless: serverless cluster management, fully compatible with Apache Kafka

MSK connect: stream data between clusters

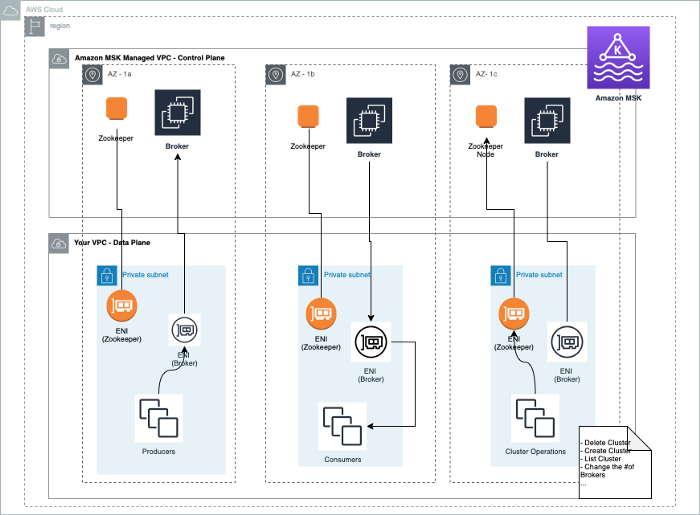
Security & Logging:

* Integrates with KMS for server side encryption
* Encryption at rest by default
* Encryption in transit by default by TLS 1.2
* Metrics gathered and sent to CloudWatch by default
* Can deliver broker logs to CloudWatch, S3
* All Amazon MSK API calls are logged to CloudTrail

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Details on Kafka setup

<https://aws.plainenglish.io/it-cant-get-simpler-than-this-setup-kafka-cluster-in-aws-431cd6cf914d>



1. Create a VPC
2. Deploy MKS cluster in VPC. In the example above, we have 3 subnets and 3 brokers. Each subnet contains one broker node and one zookeeper node, hence using 2 private IP addresses per subnet. You can view them at the AWS ENI (Elastic Network Interface)
3. Launch Kafka client as a EC2 instance. Follow the steps on the official AWS guide to see the CLI commands for this

**Updating your broker configurations**

Scaling up broker storage: <https://docs.aws.amazon.com/msk/latest/developerguide/manually-expand-storage.html>

Manual and autoscaling options available. Note, you cannot scale down. To scale down, you must do manual migration

Change broker type: <https://docs.aws.amazon.com/msk/latest/developerguide/msk-update-broker-type.html>

MSK will update the broker one by one, so your service will still be available while the brokers are being updated

Update MSK config: <https://docs.aws.amazon.com/msk/latest/developerguide/msk-update-cluster-config.html>

Expanding a cluster by adding more brokers: <https://docs.aws.amazon.com/msk/latest/developerguide/msk-update-broker-count.html>

**Machine Learning Services 🤖**

**Rekognition**: Image and video analysis, face and text detection

**Comprehend**: NLP service, e.g. for social media posts

**Polly**: TTS, voice across many languages, can create custom voice

**SageMaker**: Build, train, and deploy ML models quickly (DeepLearning AMIs)

**Translate**: Language translation, e.g. for adding localization

**Lex**: Conversational chatbots (like GCP DialogFlow), used for Alexa

**Developer Tools 🖊**

Tools to accelerate the software development cycle.

**Cloud9**: IDE in web browser (with needed SDKs and libraries)

**CodeCommit**: version control for private repos, works with Git repositories

**CodeBuild**: build/compile and test application source code (source -> **build** -> **test** -> deploy)

**CodeDeploy**: manages deployment of code to compute services in the cloud and on-premises

**CodePipeline**: automates software release process -> quickly deliver new features and updates (integrates with CodeBuild, CodeCommit, and CodeDeploy)

<https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html>

Good to read to understand how DevOps actually works

**X-Ray**: traces user requests throughout your application

**CodeStar**: help developers work collaboratively, connect development environments, contains an issue tracking dashboard

**Step Functions**: Integrate with AWS Lambda to set up state machines for step-by-step execution of functions -> branching, error handling, human in the loop, parallel tasks …

<https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html>

**Deployment and Infrastructure Management Services 🏘**

Infrastructure services allow you to:

* Quickly stand up new applications (in a consistent manner)
* Automate the management of infrastructure
* Leverage on version control
* Visibility of system health

Infrastructure as Code (IaC)

MyBucket:  
Type: AWS::S3::Bucket

"MyBucket": {  
   "Type": "AWS::S3::Bucket"  
}

IaC allows you to reproduce similar infrastructures, without using the console

**CloudFormation**: provision AWS resources using IaC (CloudFormation template)

**Elastic Beanstalk**: deploy web apps and web services to AWS -> provision resources, automatically deploy, and monitors application health via health dashboard

^^ Can only deploy on cloud, not on prem $$

**OpsWorks**: Deploy code and manage applications

* Manage either on-premises servers or EC2 instances in the cloud
* Works with Chef and Puppet for configuration automation

### CloudFormation

**Write code**: Supports either JSON or YAML formatting.

**Deploy template**: CloudFormation makes the necessary API calls on your behalf to deploy the product.

**Benefits of CloudFormation**

* Useful for creating immutable architecture -> high consistency and simple replication
* Version control possible
* Automate deployment process

**Concepts**

* Templates: Written in JSON or YAML, these specify the resources you want to create
* Stacks: You combine related resources into a stack, defined by a template
* Change sets: To make changes to the running resources in the stack, you update the stack. You can generate a change set before making changes to see how your changes might impact your running resources.
* Parameters: “Soft code” in your template that can be set at each deployment

Note: CloudFormation scripts use a very small subset of YAML so there is no need to learn the entire YAML language.

Hard-coded values and resource IDs usually can be the reason templates fail to create.

If it finds an error, CloudFormation rolls back to the last known good state.

#### Templates - How CloudFormation Works

To create templates, you can do so via CLI so you don’t need to upload the template to S3s every time:

aws cloudformation create-stack --stack-name sampleapp --template-body file://compute.template.yaml --role-arn <role-arn>

The role provided in <role-arn> is assumed by CF to create the resources (but you role must have iam:PassRole permission). If we omit this, CF will create a temporary session using the credentials in your current session in the terminal

Create template -> save in S3 bucket -> Use CloudFormation to create a stack based on your template.

If failure occurs during deployment, CloudFormation will unroll and delete previously deployed resources.

CloudFormation quotas: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cloudformation-limits.html>

Example templates:



* Resources is a mandatory field in each CloudFormation template
* Type is a mandatory field in each resource, taking the form `AWS::ProductIdentifier::ResourceType`
* Resources have property fields which may contain subfields. Some resources have more mandatory properties than others.



* References to resources in the current group use the `Ref` function
  + <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-ref.html>
* References to resources in the account can be specified by the literal itself, here `MyExistingSecurityGroup`



* We can specify parameters that are entered by the user during stack creation and reference them in the same template



* To find an attribute of a resource, we can use the `GetAtt` function. The function takes a list of 2 arguments, the resource and its attribute name.  
  More information on the `GetAtt` function: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-getatt.html>

^^ Note: You cannot use any functions for the logical name in GetAtt, but you can use functions for its attribute

Also, if resource B has a GetAtt in its parameters for an output of resource A, then resource B has an implicit `DependsOn` for resource A



* For parameters, only the `Type` field is mandatory. It can be a String, Number, or AWS specific type. Other fields are optional and pretty self-explanatory
  + NoEcho will conceal the input, though it does not include encryption  
    Rather than including sensitive information inside template, it’s better practice to refer to sensitive information outside of the template, such as in AWS Systems Manager Parameter Store or AWS Secrets Manager



* Use the `FindInMap` function to resolve mappings defined in the template  
  The function takes a list of 3 inputs: map name, top level key, second level key
* The pseudo-parameter `AWS::Region` is resolved during stack creation to the region the stack is created in.



* Use the `Join` function to concatenate strings. It takes 2 parameters, the delimiter and the list of strings to concatenate

**The `!Sub` function**

The function acts like the f-string in Python. Variables are interpreted using the `!GetAtt` or `!Ref` function depending on whether the variable is an attribute, or a stack or pseudo parameter.

The standard way to use it is to provide the string (with variables in ${}) then a list of mapping of variable name to value



You can also do a one-liner and not use variable names all together



Note on updating stack with Auto Scaling group

Each instance will run its update independently, which will cause the following issues:

* If the cfn-hup changes run on all Amazon EC2 instances in the Auto Scaling group at the same time, your service might be unavailable during the update.
* If the cfn-hup changes run at different times, old and new versions of the software may be running at the same.

To avoid these issues, consider forcing a rolling update on your instances in the Auto Scaling group. For more information, see [UpdatePolicy attribute](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-updatepolicy.html).

DependsOn attribute enables you to specify that one resource must be created after another.

DeletionPolicy attribute enables you to specify how CloudFormation should handle the deletion of a resource.

Metadata attribute enables you to specify structured data with a resource.

#### Template reference

Format version

Identifies the capabilities of the template. The latest template format version is `2010-09-09` and is currently the only valid value.

Description

Optional description of the template

Metadata

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/metadata-section-structure.html>

Optional, add any additional JSON or YAML text.

Some services retrieve data from the metadata field, such as `AWS::CloudFormation::Init`

Parameters

Parameters that can be specified on stack creation.

Info on parameter properties, types (including AWS specific types) are given here: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html>

Rules

To validate the parameters fit some requirements.



RuleCondition: determines when rule takes effect

Assertion: describes the rule for the parameters

Mappings

Self-explanatory

Conditions

Evaluates to a Boolean, which can then be consumed by `Resources` or `Outputs` to change what is produced based on that condition.

Conditions usually depend on a parameter, as otherwise the Boolean will be static.

Conditions can be nested (evaluate other conditions defined previously)



Transform

Macros that transform the templates

Resources

Self-explanatory

Output

Output as text to view on the stack or as input to another stack

Template reference:

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-reference.html>

Rule-specific intrinsic functions: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/rules-section-structure.html>

#### Stacks

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stacks.html>

A stack is a group of AWS resources that fit into a logical group, say a web applications.

CloudFormation provision resources where each template creates a stack.

We can create nested stacks by simply declaring `AWS::CloudFormation::Stack` resources in the `Resources` section. (<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-stack.html>)

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-nested-stacks.html>

The stacks will for a dependency that is a DAG (hopefully)

#### Stack sets

Allows you to deploy resources across accounts

<https://docs.amazonaws.cn/en_us/AWSCloudFormation/latest/UserGuide/stacksets-concepts.html>

#### Security

* Use multi-factor authentication (MFA) with each account.
* Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
* Set up API and user activity logging with AWS CloudTrail.
* Use AWS encryption solutions, along with all default security controls within AWS services.
* Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
* If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard (FIPS) 140-2](http://aws.amazon.com/compliance/fips/).
* Encrypt CloudFormation templates at rest when it’s stored in S3 buckets or SNS topics etc.
* Encrypt CloudFormation templates in transit by AWS (under Shared Responsibility Model)
* Use IAM policies for interacting with CloudFormation: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-iam-template.html>
* Do not embed credentials in your templates
  + Use dynamic references -> CloudFormation retrieves the value outside the stack when needed and passes it into the stack

#### CloudFormation – Best practices

Planning and organizing

* [Shorten the feedback loop to improve delivery velocity](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#shortenfeedbackloop)
* [Organize your stacks by lifecycle and ownership](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#organizingstacks)
* [Use cross-stack references to export shared resources](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cross-stack)
* [Use IAM to control access](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/security-best-practices.html#use-iam-to-control-access)
* [Reuse templates to replicate stacks in multiple environments](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#reuse)
* [Verify quotas for all resource types](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#limits)
* [Use modules to reuse resource configurations](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#modules-reuse)

Creating templates

* [Do not embed credentials in your templates](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/security-best-practices.html#creds)
* [Use AWS-specific parameter types](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#parmtypes)
* [Use parameter constraints](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#parmconstraints)
* [Use pseudo parameters to promote portability](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#pseudoparameters)
* [Use AWS::CloudFormation::Init to deploy software applications on Amazon EC2 instances](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cfninit)
* [Use the latest helper scripts](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#helper-scripts)
* [Validate templates before using them](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#validate)

Managing stacks

* [Manage all stack resources through AWS CloudFormation](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#donttouch)
* [Create change sets before updating your stacks](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cfn-best-practices-changesets)
* [Use stack policies](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#stackpolicy)
* [Use AWS CloudTrail to log AWS CloudFormation calls](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/security-best-practices.html#cloudtrail)
* [Use code reviews and revision controls to manage your templates](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#code)
* [Update your Amazon EC2 instances regularly](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#update-ec2-linux)

#### CloudFormation Misc

Parameter IDs should be the readable names given in Systems Manager -> Parameter Store, not the parameter value itself

Regular expressions follow the Java Regex pattern: <https://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html>

When there is an error in the build process, the failed event in the `Events` tab may generate a name for the log stream that records the failure. Unfortunately, it does not show the name of the log group. You’ll have to find the correct log group bruteforce.

Custom resource:

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-custom-resources.html>

You can define create custom resources using the Custom::<> resource type. The custom resource contains a service token (e.g., AWS SNS) and any input data. The service token will trigger the creation of a third-party resource. The third-party resource creator is responsible for listening into the service token.

When the resource is created, AWS will generate a S3 bucket where response from the third-party service will be recorded. The response will include SUCCESS or FAILURE alongside arbitrary name-value pairs. If the response is FAILURE, the CF resource will return in state failure and rollback will begin.

### Service Catalog

Create, organize, and control your curated IaC templates.

Cloud engineers can create IaC templates and put them on service catalog for other teams (developers, data scientists) to self-provision. Teams can then create standard products from Service Catalog. Service Catalog also acts as the central platform to manage this infrastructure

Service Catalog is the top layer/abstraction, CloudFormation is the engine the drives it in the lower level

Finding template for service catalog products

For some reason it’s difficult to find the template for products. One way to do this is to provision the product you want, navigate to CloudFormation and find the product stack, then view template from there.

**Messaging and Integration Services ☎️**

Loose coupling is good -> reduction in dependency between components, so change or incident on one component will not bring entire system down.

**SQS**: Component-to-component communicate using messages

* Multiple components can add messages to queue
* Messages are processed asynchronously (sender will not block until receiver receive message)
* Request processed in FIFO order

**Simple Notification Service (SNS)**: Send **email** (as plain text!) and **text** messages, using publisher-subscriber pattern

* Eg.: Get an email when CPU utilisation is >80% (connect with CloudWatch)

SNS instances are created from SNS topics, which act as the logical access point where multiple endpoints can subscribe to the messages (lambda, SQS ...)

**Simple Email Service (SES)**: Send **emails** with HTML (unlike SES) -> useful for marketing

**Auditing, Monitoring, and Logging Services 📊**

Answer questions like:

* Who signed onto the service?
* What is the current load on this EC2 instance?
* What is the root cause of this application error?
* Which execution path results in this error?

### CloudWatch

* Collects metrics, logs, and events (trigger event based on condition)
* Detect anomalies in your env
* Set alarms (e.g., billing alarms)
* Visualize logs (CloudWatch Metrics)

E.g. EC2 above 50% utilization -> CloudWatch -> SNS -> Email alert received

Note, CloudWatch cannot detect how an EC2 instance is terminated or who did it. It can only detect that it is. We need CloudTrail for this as it tracks user actions on AWS Management Console etc.

A **log stream** is a sequence of log events that share the same source. Each separate source of logs in CloudWatch Logs makes up a separate log stream.

A **log group** is a group of log streams that share the same retention, monitoring, and access control settings. You can define log groups and specify which streams to put into each group. There is no limit on the number of log streams that can belong to one log group

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/Working-with-log-groups-and-streams.html>

**CloudTrail**: track user activity and API calls

* Log and retain account activity
* Track activity through the console, SDKs, and CLI
* Identify which user made changes
* Detect unusual activity in your account

history log on a per-Region basis. Capped at 90 days

Can create custom trail to extend past 90 days

Things to track with CloudTrail:

* User
* Event time and name
* IP address
* Access key used
* Region
* Error code

**Additional Services**

**Amazon Workspaces**: Virtual desktop in the cloud

^^ Note this is technically a DaaS (desktop as a service)

**Amazon Connect**: A cloud contact centre

**Security and Compliance**

**Shared Responsibility Model**

There is a shared security responsibility between AWS and you.

AWS = Security of the cloud

You = Security in the cloud

AWS is responsible for protecting and securing their infrastructure

* Regions, edge locations, AZs
* Data centers are secured
* Network components: generators, uninterrupted power supply, computer room air conditioning ...
* Software: RDS, S3, ECS, Lambda ...

You are responsible for how the services are implemented and managed in you app

* **Application data**: Encryption and management of application/customer data
* **Security config**: Securing your account and API calls, rotating credentials, restricting internet access from your VPCs, encryption etc.
* **Patching**: Guest operating system (OS) -> patches and security updates
* **Identity and Access Management (IAM)**
* **Network traffic**: firewall config
* **Installed software**: make sure application code is free of vulnerabilities

Rule of thumb: if you can it in the AWS console, you are probably responsible for it

Note, training responsibility is shared between you and AWS. You are responsible for training your company staff, AWS is responsible for training AWS staff.

Examples:

EC2 shared responsibility model

| **You** | **AWS** |
| --- | --- |
| Installed applications | EC2 service |
| Patching the guest OS | Patching the host OS |
| Security controls | Security of physical server |

Lambda shared responsibility model

| **You** | **AWS** |
| --- | --- |
| Security of code | Lambda service: updating languages, endpoints ... |
| Storage of sensitive data | Underlying infrastructure |
| IAM for permissions | Software dependencies |

Report of abuse of AWS services

* Rotate credentials
* Email [abuse@amazonaws.com](mailto:abuse@amazonaws.com)

**Well-Architected Framework**

Best practices when creating architectures in AWS. Please read the white papers for more information.

Know the 6 principles

1. **Operational Excellence**: plan for anticipated failure, deploy small and reversible changes, learn from failure and refine
2. **Security**: automate security task, encryption, last privilege principle, track and log, ensure security at all application layers (e.g., security of infrastructure is no use if your code is vulnerable)
3. **Reliability**: recover from failure automatically, scale horizontally for resilience, reduce idle resources, testing
4. **Performance efficiency**: use serverless architecture first (as they're already optimized for the cloud), use multi-region deployment (for better latency for global customers)
5. **Cost optimization**: use only resources that your app requires
6. **Sustainability**: understand environmental impacts, establish sustainability goals

Where to find security documentations (best practices, whitepapers, official documents):

* AWS Knowledge Centre
* Security Centre
* Security Forum
* Security blogs

There are also 3rd party security products from AWS marketplace

**IAM (Identity and Asset Management)**

### Overview

Identities, Groups, Roles, Policies


      IntroToIAM_Diagram
    

Important to keep data safe and secure through **access control**. IAM is free to use.

Note, IAM is always in `global` region

Remember, access is a triple of Resource x Identity x Access

AWS **Identities**:

* Users: root, individual users (can be an application!). By default new users have no permissions
* Groups (usually for job functions like developers, administrators …)
* Roles

You can attach policies to identities. It’s best practice to apply policies to groups, and add users to these groups, instead of applying policies to users directly. This is because users come and go and can change teams. So, tracking the policies can be difficult.

When you create a new user, you get a access key ID and secret access keys. These work like username/password but are different from them. They are used to access AWS via the APIs and Command Line.

AWS **Principle**:

* A person or application
* … that uses the AWS account root user, an IAM user, or an IAM role to sign in and make requests for an action or operation on an AWS resource

AWS **Access**:

* Policies
* AWS managed policies (created template from AWS)
* Customer managed policies
* Permissions boundaries

AWS **Request**:

* Actions or operations
* Resources involved
* Principal involved
* Environment data – IP address, time of day, SSL enabled status
* Resource data – metadata relating to the resources being requested

^^ These then are wrapped into a *request context* which is sent to policies for authorization.

AWS **Users**:

* When you create your account, you are given the root user under that account
* You can create multiple other users in the same account to avoid using root user credentials extensively
* You can create federated users if your organisation already has authentication for different users


        Users who are already authenticated elsewhere can be federated into AWS without
          requiring an IAM user.
      

This saves time in creating new users

* You can also create multiple accounts if you want more strict separation
* When to use users, accounts or groups: <https://aws.amazon.com/blogs/security/guidelines-for-when-to-use-accounts-users-and-groups/>

Ways to define access:

* Attach policies to accounts
* Attach policies to groups
* Attach policies to roles
* Resource-based policies
* Organizational SCPs
* Access control lists (ACLs)

**Policy types**

Identity-based:

* AWS managed policies – AWS creates this
* Customer managed policies – You create and edit this
* Inline policies – you create & is stuck with a single user, group, or role. This is not recommended

Resource-based:

* Customer managed polices – You create and edit this, always inline so stuck with the resource

Permission boundaries: granted to IAM identity. Defines the maximum permissions identity-based policies can grant to an identity (but does not grant any permissions itself). It does not define maximum permission a resource-based policy can grant to an entity

Organizations Service Control Policies (SCPs): Defines the maximum permissions for account members of an organization or OU, limits both identity-based and resource-based policies granted to that account (but does not grant any permissions itself).

Access control lists (ACLs): Like resource-based policies but does not use JSON structure.

Session policies: Limit the permission that the role or user grants to a session

**Groups**:

* Administrator: perform admin tasks such as creating new users
* Developers: use compute and db services to build products
* Analysts: run budget and usage reports

^^ We can add users to groups to grant them these permissions

**Note**: IAM Group is not an EC2 security group

-> EC2 instances need inbound and outbound rules to create a firewall for traffic. It's tedious to set one up for each instance so we can create security groups each with a specific set of rules, then just simply add EC2 instances to a relevant security group.

-> Security group is *per instance* and act like default firewalls. On the other hand, network access control lists is *per subnet* and covering all instances in the subnet with the same rule set.

**Roles**: temporarily assumed to gain some access (defined using policies)

e.g., User + DevOps Engineer Role = temporary access to CodeCommit and CodePipeline

Lambda + Lambda-Execution Role = List contents of S3 bucket + Query DynamoDB

Roles are useful to grant temporary permission without giving away access keys

**Policies**

{  
   "Version": "...",  
   "Statement": [  
      {  
           "Action": "rds:\*",  
           "Effect": "Allow",  
           "Resource": [...]  
      }  
  ]  
}

^^ Full access to RDS

Policies give *authorization* to requests.

* By default, all requests are denied
* An explicit allow in any policy overrides the deny
* If many policies (Organizations SCP, IAM permissions boundary …) exist, they must all allow the request for it to be allowed
* An explicit deny in any policy overrides any allows (think of policies as AND)

Full evaluation logic:

<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html>

IAM best practices:

* Enable MFA for privileged users
* Implement strong password policies (e.g., preventing users using previous passwords, change password over time ...)
* Create individual users instead of using root
* Use roles for EC2 instances to hide long-term credentials
* Use temporary credentials for workloads (bits of code)
* Rotate access keys regularly
* Regularly review to remove unused users, roles, permissions, policies, and credentials
* Use conditions in IAM policies (conditions are additional attributes to the access control relation, such as all requests must be sent using SSL)

IAM Credential Report: Gives overview of credentials in your account $$

users, passwords, access keys, MFA

-> Used for auditing and compliance

**ABAC** (Attribute-based access control)

The above access control paradigm is called **RBAC** (role-based access control), where you have roles and you create the identity x resource x access relation by 1) adding roles, 2) adding policies and attaching to roles, and 3) adding users to roles.

The idea of ABAC is to remove the need for policies all together. In ABAC, we add tags to both identities and resources, and an identity gets access to a resource if their tags match.

Advantages of ABAC over RBAC:

1. Scales with new products: If people under tag heart decides to add a new EC2 resource, we just need to tag new EC2 with heart and the people in that team will have access. No need to fiddle with policies.

etc.

How companies use IAM?

We can define permissions in different levels, much like what we do with CSS.

1.SCPs

2.Permission boundary (we can write this) – AppRole, SupportRole …

3.Policies to specific roles, resources …

At each abstraction, we focus on an extra level of detail (e.g., SCPs we block all actions to EC2s, at permission boundary we block all IAM account creation unless it follows some naming convention, at role level we define the individual conditions etc.)

### How policies work


          JSON policy document structure
        

*JSON structure of policies*

Statement structure:

* Version – Specifies the version of the policy language that you want to use. As best practice, use the latest `2012-10-17`
* Statement
  + Sid (Optional) – Statement ID
  + Effect - `Allow` or `Deny`
  + Principal – required for resource-based policy
  + Action – list of actions the policy allows or denies
  + Resource – required for identity-based policy
  + Condition (Optional) – Specify further circumstances which the policy grants permission

Example:



More examples see documentation.

**Granting least privilege**

It’s always best practice to include the least amount of permissions necessary for a user to perform a task. Here are some tips on doing that:

* Understand access level groupings (`List`, `Read` …) and only grant permissions within a group to restrict a specific type of action. E.g., choose actions only from `List` and `Read` to give read-only access
* Validate policies using IAM Access Analyzer
* Generate a policy based on CloudTrail access activity and constantly review it

**Permission boundaries**

A permission boundary looks just like a normal JSON policy, but the permission it specifies it the maximum permission that can be granted to the identity. Example:



This boundary allows the user to only manage S3, CloudWatch, and EC2

**Controlling access to AWS resources using tags**

Sometimes we want to grant permissions conditional on some tags. We can pass them as a conditional to the policy. Example:



There are other complicated ways of doing this, details see: <https://docs.aws.amazon.com/IAM/latest/UserGuide/access_tags.html>

**Policy evaluation logic**

Definitional interpreter for the logic as a flow chart


        Evaluation flow chart
      

1.Deny evaluation: Any explicit deny will deny the evaluation

2.Organization SCPs: Implicit deny form SCPs (if exist) will deny the evaluation. If SCPs exist, it must have an explicit Allow to let the evaluation continue

3.Resource-based policies: In most cases, an explicit allow would allow the evaluation  
Details see: <https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html#policy-eval-denyallow>

4.Identity-based policies: straight forward

… (straight forward)

In practice when we write policies, we follow certain patterns. For example:

* By default, all requests are denied
* An explicit allow in any policy overrides the deny
* If many policies (Organizations SCP, IAM permissions boundary …) exist, they must all allow the request for it to be allowed
* An explicit deny in any policy overrides any allows (think of policies as AND)

The above evaluation logic is for a single account. But we can also allow a principal in one account to access resources in a second account. This requires cross-account evaluation, which is a little more complicated:

<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic-cross-account.html>

### Using policies

**Creating policies**

It’s useful to consult the specific resource you need to policy for to get a feel of the relevant policies.

For specific details (e.g., the conditions available), consult the API reference

AWS policy CFG: <https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_grammar.html>

**Testing IAM policies**

You can use the policy simulator to simulate fake requests using the same evaluation engine to test our policies. Remember to add `iam` permissions to test your policies:

<https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_testing-policies.html>

**You can also version IAM policies**

<https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-versioning.html>

**Using summaries to understand your policies**

The IAM console includes *policy summary* tables that describe the access level, resources, and conditions that are allowed or denied for each service in a policy. Policies are summarized in three tables: the [policy summary](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_understand-policy-summary.html), the [service summary](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_understand-service-summary.html), and the [action summary](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_understand-action-summary.html). The *policy summary* table includes a list of services. Choose a service there to see the *service summary*. This summary table includes a list of the actions and associated permissions for the chosen service. You can choose an action from that table to view the *action summary*. This table includes a list of resources and conditions for the chosen action.


      Policy summaries diagram image that illustrates the 3 tables and their
        relationship
    

Policy: S3, RDS, …

Service: S3

Action: S3:ListObjects

**Policy reference**

<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_principal.html>

Principals



You specify principals as a dictionary, mapping principal types (AWS account, roles …) to values in that type. If there are many values in the same principal type, use a list.

A principal in evaluated against all principals specified in the resource policy as a logical OR. Once it matches one of the principals specified, access is granted.

A few principal types below (more see documentation)

*AWS account*

Use the account-ID or full arn (`arn:aws:iam::account-ID:root`)



*IAM role*



*IAM user*



When the policy is saved, IAM transforms the ARN to the user's unique principal ID. This is a safety feature that prevents entities from deleting the user and recreating it for privilege escalation. The newly created user will have a different principal ID and hence will not be granted access by the original policy.

User ARN <-> User’s unique principal ID

User ARN is what is displayed, user’s unique principal ID is what is stored

You can’t use \* for users to grant to all users.

*All principals*



The 2 statements are equivalent.

This policy grants access to all principals, including anonymous users, hence giving public access.

For principals in other accounts, they must also have identity-based permissions in their account that allow them to access your resource. This is called [cross-account access](https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic-cross-account.html).

Conditions

Conditions match key with different values (strings, dates, numbers …). If the key does not exist, the condition is automatically evaluated to false.

*String conditional operators*

StringEquals, StringNotEquals, StringEqualsIgnoreCase, StringNotEqualsIgnoreCase, StringLike (match using \* and ? (single-place wildcard)), StringNotLike

Example:





*Numeric condition operators*

NumericEquals, NumericNotEquals, NumericLessThan, NumericLessThanEquals, NumericGreaterThan, NumericGreaterThanEquals



*Date condition operators*

DateEquals, DateNotEquals, DateLessThan, DateLessThanEquals, DateGreaterThan, DateGreaterThanEquals



Date format must be [W3C implementations of the ISO 8601 date formats](http://www.w3.org/TR/NOTE-datetime) or UNIX epoch time

*Boolean condition operators*



*ARN condition operators*

ArnEquals, ArnLike, ArnNotEquals, ArnNotLike



### Misc

A service with a role to assume another role

<https://nelsonfigueroa.dev/aws-iam-allowing-a-role-to-assume-another-role/>

* If the service is assuming a role in the same account, we need to modify the policy of the target role
* If the service is assuming a role in a different account, we need to modify the policy of the target and source role

How to troubleshoot access denied error:

Use policy simulator: <https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_testing-policies.html>

**Application Security Services**

Software-based security tools

**WAF**: Web Application Firewall. Protects against $$

* Common attack patterns
* SQL injection $$
* Cross-site scripting (injecting malicious code to benign/trusted website, end-user of that website will then execute this code)

**Shield**: Managed DDoS protection service $$

* Shield standard: *free* - protection against common attacks
* Shield advanced: *paid* - enhanced protection + 24/7 access to AWS experts

How Shield detects and mitigate common DDoS attacks: <https://docs.aws.amazon.com/waf/latest/developerguide/ddos-event-mitigation-features.html>

Works with:

* CloudFront
* Route 53
* Elastic Load Balancing
* AWS Global Accelerator

**Macie**: Use ML to discover and protect sensitive data

**Additional Security Services**

**Config**: access, audit, and evaluate the config of your resources. Use SNS for config change alerts.

**GuardDuty**: uses ML to detect unauthorized behavior

**Inspector**: install agents to EC2 instances and report vulnerabilities <- only works with EC2 instances

**Artifact**: on-demand access to AWS security and compliance reports (SOC, PCI ...) -> For AWS compliance, not your own accounts!

**Cognito**: control access to mobile and web applications built on AWS. Basically an authentication and authorization tool.

AWS config in detail

<https://docs.aws.amazon.com/config/latest/developerguide/WhatIsConfig.html>

With AWS Config, you can do the following:

* Evaluate your AWS resource (EC2, EBS ...) configurations for desired settings.
* Get a snapshot of the current configurations of the supported resources that are associated with your AWS account.
* Retrieve configurations of one or more resources that exist in your account.
* Retrieve historical configurations of one or more resources.
* Receive a notification whenever a resource is created, modified, or deleted.
* View relationships between resources. For example, you might want to find all resources that use a particular security group.

AWS config concepts

AWS resource: EC2, EBS ..

Config history: configuration items for an AWS resource in a given time frame

Config items: a snapshot of various attributes of that AWS resource in a certain time (like a git commit)

Config recorder: store configurations of resources as config items (records every change by default)

Config snapshot: config items for an AWS resource

Config stream: an automatically updated list of config items of a resource under recording, each time the resource is changed (created, modified, deleted), AWS config adds the corresponding config item to the stream

Resource relationship: e.g., EBS is attached to EC2 instance which belongs to some security group

Config rules: rules that define a desired configuration of your resource. AWS has some managed ones and you can create custom ones as well

This is basically AWS take on CMDB. Currently they are expanding config rules to more services. New services will all have config rules attached to them

Multi-Account Multi-Region Data Aggregation: <https://docs.aws.amazon.com/config/latest/developerguide/config-concepts.html#multi-account-multi-region-data-aggregation>

Managing AWS config: config console, CLI, APIs, and SDKs

How AWS config works under the hood

AWS config is not the one actually configuring the items, it is simply a passive tracker for the state changes of its underlying AWS resource.

Changes in a resource act as a trigger for config items to be generated.

If there are config rules in place, each config rule is associated with an AWS Lambda function which contains the evaluation logic for the rule. The function is executed at each change, and if the result is negative, the rule and the resource is flagged as non-compliant.

How AWS config notifies you

Configuration history file is sent to the designated S3 bucket every 6 hours

SNS topic send notifications about various config stuff

AWS config support resource types: <https://docs.aws.amazon.com/config/latest/developerguide/resource-config-reference.html>

Anatomy of a config item: <https://docs.aws.amazon.com/config/latest/developerguide/config-item-table.html>

Service limits: <https://docs.aws.amazon.com/config/latest/developerguide/configlimits.html>

**Data Encryption and Secrets Management Services**

Data in flight/transit: data moving from one location to another

Data at rest: data inactive or stored for later use

**KMS**: Key Management Service

* Generate, store and manage keys digitally

Key deletion is default 30 days (to prevent accidently locking yourself out). The shortest deletion time you can set is 7 days.

**CloudHSM**: Hardware Security Module

* Generate, store and manage keys using physical device

**Secrets Manager**: $$

* Rotate, manage and retrieve keys
* Basically a managed service for a password database (encrypt and store secrets + decrypt and return secrets, upon request)
* Can be used for services like RDS, Redshift, and DocumentDB
* Can be integrated with KMS

**Pricing, Billing, and Governance**

**AWS Pricing**

3 fundamental drivers of cost

1. Compute: hourly from launch to termination
2. Storage: data your store in the cloud
3. Outbound data transfer: data in flight moving between systems

Free offer types

1. 12 months free: from initial sign-up 12 months
2. Always free: doesn't expire
3. Trial: short-term free trials

Within these trial timeframes, you can use the relevant services free of charge under certain constraints.

$$ Skim through the ones below:

EC2 pricing

1. On-demand: pay by hour or second
2. Savings plan: pay by hour for 1 or 3 year term
3. Reserved instances: pay 1 or 3 years, regardless of usage
4. Spot instances: launched only if there is spare capacity
5. Dedicated hosts: an entire physical server just for you

Lambda pricing

1. #. of requests
2. Execution time (start to end)
3. Always free (1M requests per month for free)

S3 pricing

1. Storage class (standard, glacier ...)
2. Storage size (# and size of objects)
3. Data transfer (transfer out of S3 region, within S3 region is free)

RDS Pricing

1. Running clock hour (time of termination of db - time of start of db)
2. Type of DB
3. Storage size of DB
4. Purchase type: on-demand or reserved?
5. Database count: # db instances
6. API requests
7. Deployment type: single AZ, or multiple?
8. Data transfer: inbound is free, outbound costs money

Total cost of ownership (TCO)

^^ Financial estimate of both direct and indirect cost of AWS

(There used to be a TCO calculator, but it's no longer available)

^^ New tool is called the **Pricing Calculator**

AWS Price List API

* Query prices using JSON or HTML
* Get discount alerts

**AWS Billing services**

Tracking ongoing spend

**Budgets**: set budgets (cost, usage, and reservation) and add alerts when you overspend (e.g., do this for a free tier limit)

Cost budget: how much to spend on service

Usage budget: how much to use a service

Reservation budget: how much to use reserved instances or saving plans

**Cost and Usage report**: download comprehensive report, lists usage for each service category, aggregate usage data on a daily, hourly, or monthly level

^^ when something goes wrong ...

**Cost explorer**: visualize cost over time (past 12 months), forecast for 3 months

**Cost allocation tags**: label resources with key-value pair (eg. "Marketing")

**AWS Governance Services**

Maintain control over cost, compliance, and security

### Organizations

manage many AWS accounts under one umbrella

* Group multiple accounts
* Single payment for all accounts -> saves money
* Automate account creation
* Centrally apply access policies (SCPs - Service Control Policies)

Organizational units (OUs): account grouped together as they are similar (e.g., IT OU, Marketing OU)

Benefits of using organizations:

* Consolidated billing = volume discounts
* Account governance = easy to create and manage accounts

Details

An organization is an entity you create to consolidate your AWS accounts under a single unit. You create **one** organization with **one** management account (usually called Org Master account).

The management account can create accounts, move them across OUs etc. All other accounts in the Org are member accounts. The management account is also responsible for paying for all accounts.

Every Org contains the root OU, which contains all accounts. The root OU can also contain other OUs, which can contain other OUs within them etc. Accounts are then leaves of this tree.

Policies applied to an OU affects all accounts under it (whether they’re directly under the OU or nested within an OU under it). So naturally, policy applied to root OU affects all accounts.

**Control tower**: helps to ensure accounts conform to company-wide policies (sits on top of AWS organizations) $$

* Helps set up new accounts (using multi-account strategy)
* Enforces best use of services
* Provide dashboard to manage accounts

**Systems Manager**: visibility and control over AWS resources

* Automate ops tasks
* Groups resources and take action on the group
* Patch and run commands on EC2, RDS instances

**Trusted Advisor**: real-time guidance to help provision resources following **AWS best practices**

Some checks free, some need Enterprise or Business support enabled

Can also help with keeping an eye on service limits

**License Manager**: manage software licenses (on-prem and AWS licenses, also for Oracle, MS, SAP ...)

**Certificate Manager**: provisions and manages SSL/TLS certificates

**AWS Management Services**

Services that helps you migrate to the cloud

**Managed services**:

* Augments your internal staff
* Provides ongoing management of your infrastructure (reporting, compliance ...)
* Reduces operational risks and overhead

**Professional services**: helps enterprise customers move to cloud-based model (propose, architect and implement solutions)

**AWS Partner Network (APN)**: global community of approved partners

* Technology Partners: help *build* resources on AWS - software, hardware provision ...
* Consulting Partners: system integrators, strategic consultancies, agencies, managed service providers (MSPs), and value-added resellers
* Approved vendors with deep AWS experience

^^ use if team lacks technical or operational expertise

**Marketplace**: solutions that you can buy/sell

**(Personal) Health Dashboard**:

* Visibility of system health
* VS. Systems Manager - cannot configure systems like Systems Manager, but focuses more on the visibility side of the system (though Systems Manager also has visibility side which makes it confusing)

**AWS Support Plans**

4 support plans

| **Basic** | **Developer ($29/month)** | **Business ($100/month)** | **Enterprise ($15000/month)** |
| --- | --- | --- | --- |
| - 24/7 support via email | - 1 primary contact, business hour via email | - 24/7 via email, phone, or chats - Full set of trusted advisor checks | - 24/7 via email, phone, or chats - Full set of trusted advisor checks - Technical Account Manager - Concierge Support Team - Infrastructure Event Management |
| N.A. | <24hrs general guidance $$ <12hrs system impaired | <24hrs general guidance <12hrs system impaired <4hrs production system impaired <1hr production system down | <24hrs general guidance <12hrs system impaired <4hrs production system impaired <1hr production system down <15min business critical system down |

Support case types:

1. Account-related and billing questions -> free
2. Service limit increase request (guardrails AWS places to ensure you don't spend a crazy amount of money) -> free
3. Technical support case -> for paid plans only

**Miscellaneous Concepts**

In cloud, there are many ways to achieve the same thing. Sometimes, a more intricate solution might achieve the same goal with lower cost, better security etc.

**AWS Landing Zone**

[https://aws.amazon.com/solutions/implementations/aws-landing-zone/#](https://aws.amazon.com/solutions/implementations/aws-landing-zone/)

A template for setting up AWS architecture that follows the AWS well-architected framework.

Multi-account structure:

AWS Organization account: oversees the other accounts, with central account baseline and provisioning, code pipelines etc.

Shared services account: VPCs etc.

Log archive account: logs flowing to that central account

Security account: be able to access other accounts, do audits etc.

**Elastic Load Balancing**

Distribute network / application traffic across multiple targets (EC2 instances etc.)

750 hours free per month

Comes in form like application / gateway / network load balancers (see EC2)

**AWS Auto Scaling**

Automatically scales your plans adapting to your usage, prioritizing either cost or performance

Unlike EC2 auto scaling, AWS auto scaling can be used across a wide range of services.

**IEM**

<https://aws.amazon.com/premiumsupport/programs/iem/>

IEM = Infrastructure Event Management

AWS experts lead a highly focused engagement to provide you with architectural and operational guidance for your **planned event** using a prescriptive, phased approach:

1. We work with you to understand your success criteria and desired business outcome.
2. We assess the readiness of your AWS environment, help identify and mitigate risks, and document your plan.
3. You confidently execute your event with AWS experts by your side.
4. We help analyze results post-event and scale services to normal operating levels, so you can focus on planning your next event.

^^ Note, this can also be mocked events to stress test your architecture.

Timeline:

Week 0 - Initiation, AWS Support to understand your needs

Week 1-8 - Assessment of environment, mitigation of risks

Week 9 - Your event

Week 10-11 - Post-event review

**Boto3 stuff**

If you use

boto3.set\_stream\_logger(name='botocore')

boto3 will print out logs for function calls, which can be useful for debugging

# Exam guide (SAA-C03)

Response types:

* Majority on real-world scenarios
* Multiple choice: 1 correct out of 4
  + Wrong choices have at least 1 thing wrong
* Multiple response: 2 correct out of 5

Exam structure:

* Design resilience architectures – 26%
  + Multi-tier, highly available, fault tolerant, decoupling mechanisms, appropriate resilient storage
* Design High-Performing Architectures – 24%
  + Elastic & scalable compute, high-performing and scalable storage, high-performing networking, high-performing database
* Design Secure Architectures – 30%
  + Secure access to AWS resources (IAM, SSO), secure application tiers, data security
* Design Cost-Optimized Architectures – 20%
  + Cost-effect storage, compute, database, network architectures

1000 marks total, 720 to pass

65 questions

130 minutes

Tips:

* Don’t spend too much time on hard questions -> flag & review later
* Watch for keyword indicators
* Remember well-architected framework pillars
* Read **well-architected white paper** before exame

Graphical user interface, website

Description automatically generated

Services you need for SAA-C03 exam ^^

Graphical user interface, application

Description automatically generated