NETWORKING
IN
AWS

Networking: Background

In this section we will discuss the fundamentals of networking, how AWS global network is organized and the components it is comprised of. A concise list of the topics is given below:

- Brief History of The Internet
- OSI Model
- IP Addresses and DNS, what they are and how they work
- AWS Global infrastructure, Availability Zones and Local Zones
- Caching and CDN, what they are and how they work

If the reader believes themselves to already be familiar with the listed technologies, then they can feel free to skip this section and move on to the rest of the chapter.

Brief History of The Internet

We would not be talking about cloud computing if it were not for the internet, this monumental invention of humans that allows a diverse range of computers quite geographically dispersed to communicate with each other. It is after all, the chief way in which most businesses relying on AWS access their rented resources and the chief way in which your customers will access AWS resources if you're running a public-facing operation. As such, I find it necessary to provide a brief history (albeit a very diluted one) on how the internet was created.

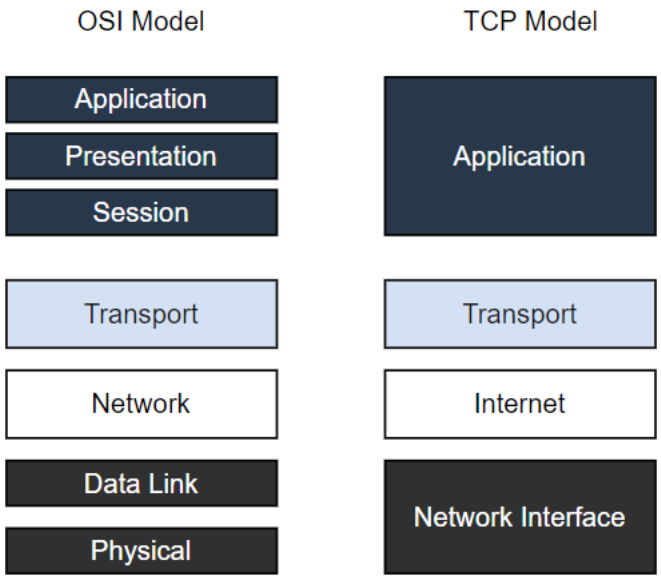
Before the advent of the internet, the largest networks of connected computers were all private networks, they were networks owned either by government agencies, university campuses and large corporate houses which were all using separate pieces of technology to organize and operate their networks. And since these networks did not utilize standardized ways of operating, communicating across networks was very difficult. A computer in a network owned by a university

campus therefore, could not talk and share information with a computer owned by a large corporate house, or even another university.

The United States government has an agency called the Defense Advanced Research Projects Agency (DARPA), which is responsible for performing R&D operations for the American military. In 1969, DARPA started a project that would later become known as ARPANET, a project to unify various private networks and allow their constituent computers to talk with one another. The success of ARPANET led to others trying the same and in 1973, a proposal to connect these disparately connected networks into a single, large network through an **internetworking protocol** was put forward: The internet was born.

OSI and TCP models of the internet

The internet today has evolved works through not just one but through many internetworking protocols layered on top of one another. There are two major models of these interlayered protocols taught in universities, **The OSI Model** and **The TCP Model**.



Now, every layer in that model has its own purpose in the sending and receiving of data across the internet, and every layer has its own set of protocols and rules for doing so. Now, of the two models, The OSI model is mostly an academic model, used to teach students about networking, while the TCP model is the one actually used in the real world, the one that reflects what businesses are using as I write this very sentence.

As such, the TCP model, short for the Transmission Control Protocol model, is the one that this section will elaborate on and explain, but the reader may feel free to look into the OSI model if they find the topic to be interesting. The four layers of the TCP model are as follows:

Layer 1, Network Interface: The network interface layer, also known as the data link layer, handles the physical infrastructure that lets computers communicate with one another over the internet. This covers ethernet cables, wireless networks, network interface cards, and so on. The network interface layer also includes the technical infrastructure, such as the code that converts digital data into transmittable signals, that makes network connection possible. Some major layer 1 protocols are: **Ethernet**, **Wifi** and **ARP**.

Layer 2, Internet: The internet layer, also known as the network layer, controls the flow and routing of traffic to ensure data is sent speedily and accurately. This layer is also responsible for reassembling the data packet at its destination. If there's lots of internet traffic, the internet layer may take a little longer to send a file, but there will be a smaller chance of an error corrupting that file. **Internet Protocol (IP)** is for example, a layer 2 protocol.

Layer 3, Transport Layer: The transport layer provides a reliable data connection between two communicating devices. It's like sending an insured package: the transport layer divides the data in packets, acknowledges the packets it has received from the sender, and ensures that the recipient acknowledges the packets it receives. **TCP** and **UDP** are two common layer 3 protocols.

Layer 4, Application Layer: The application layer is the group of applications that let the user access the network. For most of us, that means email, messaging apps, and cloud storage programs. This is what the end-user sees and interacts with when sending and receiving data. **HTTPS** and **FTP** are two very common layer 4 protocols.

IP Addresses and DNS

Now, IP addresses are like the ID assigned to every computer that is part of the internet, it has two types, IPv4 and IPv6 and looks something like this: `192.168.1.1` (IPv4 addresses at least). IP addresses also help us locate a device geographically when we are either sending or receiving packets to them, and every website, API or service on the internet that the reader has ever used is in part powered by them. These numerical addresses however, are quite hard to remember and asking people to remember them every time they wish to visit a website might prove difficult, enter DNS.

A **Domain Name** is a unique name that identifies a website on the internet, imagine nytimes.com or google.com. We map these domain names to IP Addresses like the ones above, allowing us to navigate to the devices possessing the address without having to remember their complex numerical designation.

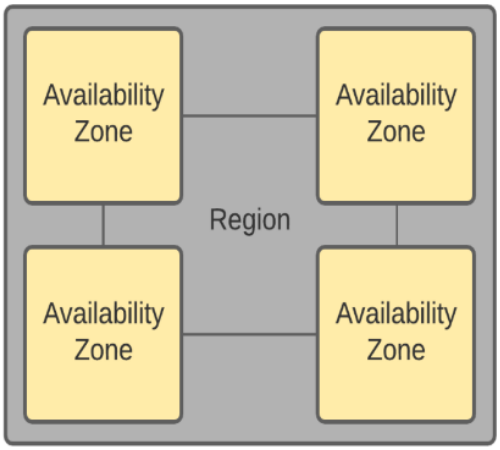
DNS (Domain Name System) then is like the phonebook of the internet, its a giant registry of domain names and the IP addresses that they point to. Whenever we go to a webpage, say Amazon.com, our browser consults the DNS to find out which IP Address it must navigate and send requests to.

AWS Global Infrastructure

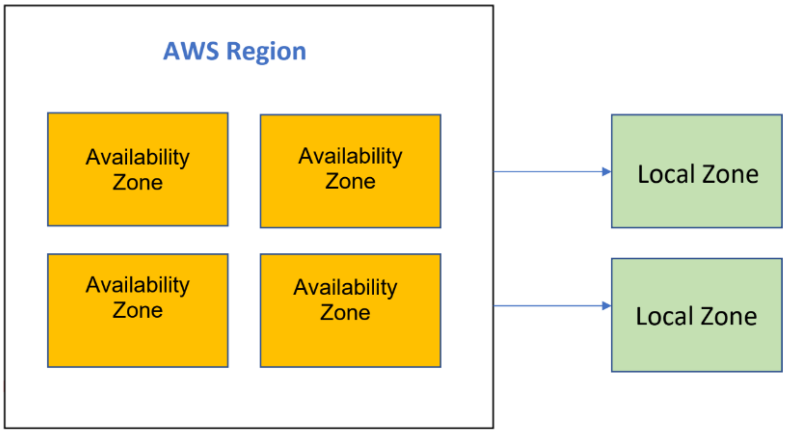
We've mentioned before how AWS operates infrastructure in over 245 countries, with presence in every one of the seven continents. This infrastructure is organized into **Regions**, **Availability Zones**, **Local Zones** and **Edge Locations**.

Regions in AWS represent data centers clustered around a large geographic area i.e. a continent or a part of a continent. Example, us-east-1 represents a group of

data centers clustered around the eastern coast of the United States. An Availability Zone (AZ) on the other hand, is simply one of the data centers within it. This is succinctly illustrated in the diagram below:

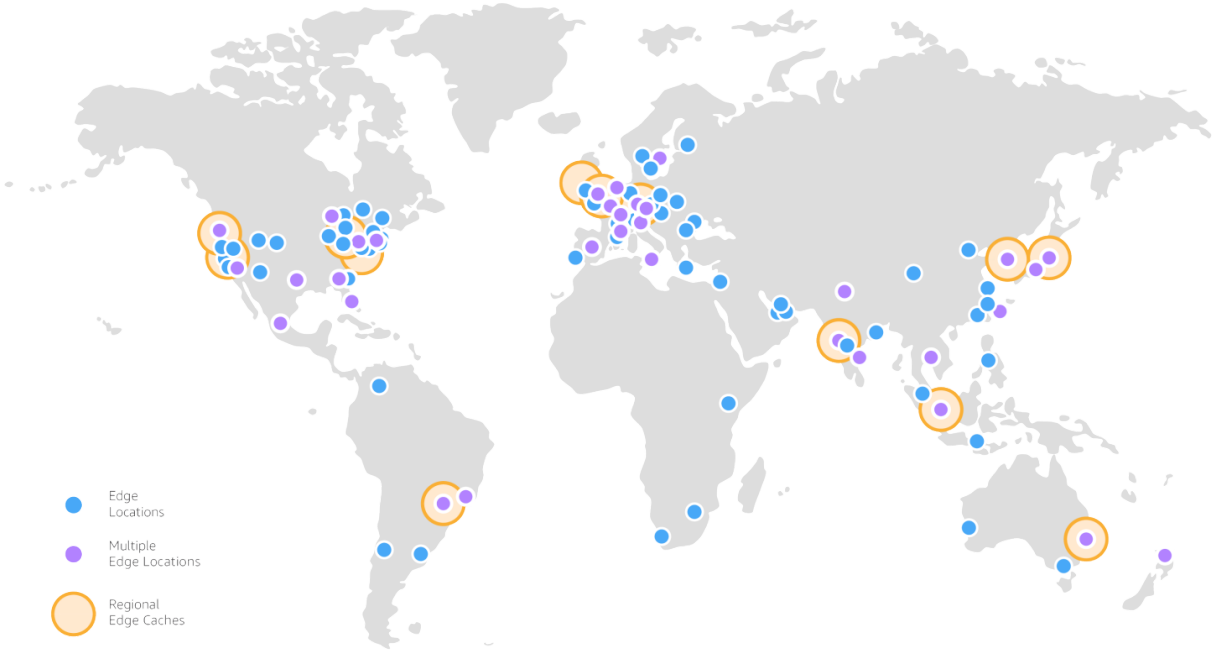


There may however, be times where a data center is needed outside the boundaries of these regions, due to a large cluster of users being concentrated in areas at a significant distance from the region, which is usually the case for large cities. In order to address such situations, AWS created the Local Zone. The local zone is effectively, a data center closer to the user. Organizations place systems that require low latency in these local zones. A diagram of a local zone is given below:



Finally, Edge Locations are the small pieces of infrastructure used to power CloudFront, Amazon's Content Delivery Network. They mostly serve the purpose of improving internet performance and website security. CDNs and CloudFront are both discussed liberally throughout the chapter.

At the time of writing, AWS infrastructure spans across 34 Geographical regions, 108 Availability Zones, 34 Local Zones and 600+ Edge locations. A map of the regions and edge locations is given below:



Caching and CDN

Caching is nothing but the process of storing frequently accessed data in a location that can be retrieved quickly, minimizing the time and resources it takes to serve that data to a user. Its like putting the car keys in a drawer right by the door.

We use the keys frequently, and it saves time to have them in a convenient, easy-to-reach spot rather than searching the entire house for them each time we need to go out. In web terms, this means data that users often request (like images,

website code, or video content) is temporarily stored closer to them, resulting in less loading times.

Now, if we were to create a full-blown distributed network of these caches, and connect them both to AZs and to each other, what we would get is a **Content Delivery Network (CDN)**, a system designed to speed up the delivery of critical and/or frequently requested data to the user.

Application Programming Interfaces (APIs)

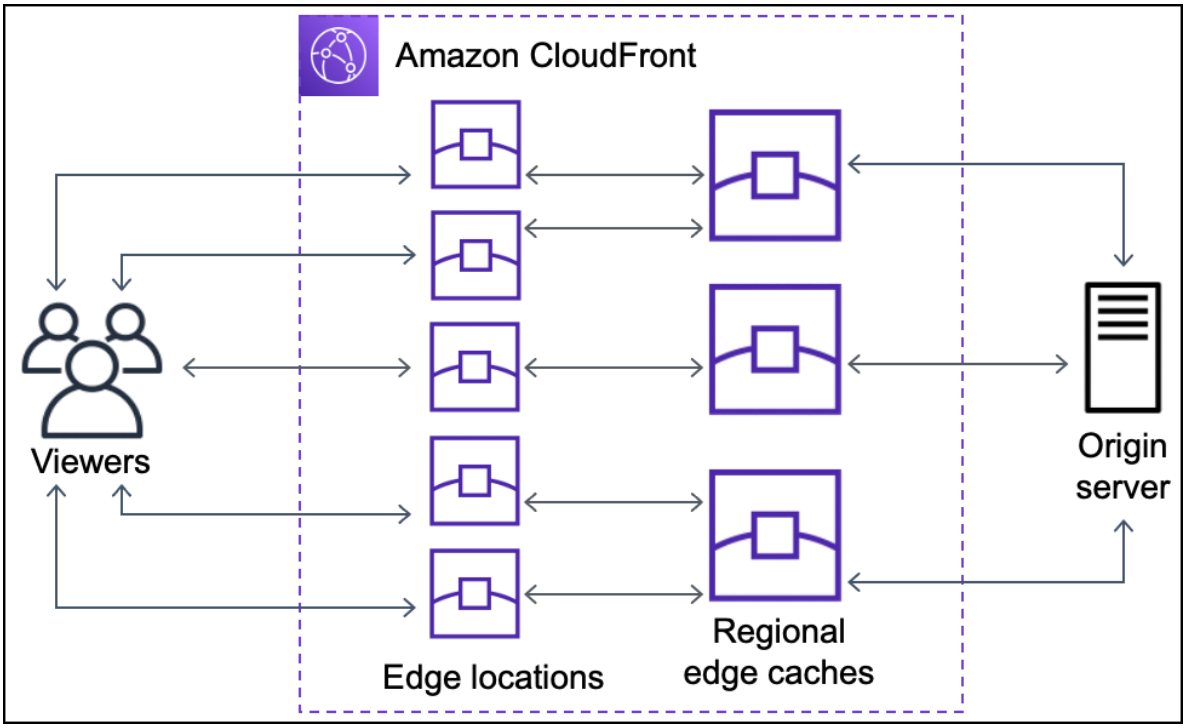
An unplanned addition to this section, APIs (Application Programming Interfaces) are sets of rules that allow different software systems to communicate. They are genuinely the cornerstone of many internet applications, and dedicating a paragraph to the concept in this section was thus believed to be necessary.

APIs, in simple words, enable one system to request or send data to another without needing to know how the other system works internally. They handle the creation of a standard channel for communication between two systems on our behalf, allowing two systems to talk with each other in extremely convenient ways.

AWS has its own set of APIs, with the python Boto3 API in particular being a popular tool in many cloud engineer’s arsenals. Also many of the individual services like S3 and EC2 also have their own APIs, allowing developers to more easily automate tasks and build integrations.

CloudFront

AWS CloudFront is a fast and highly secure content delivery network (CDN) service that accelerates the delivery of your websites, APIs, video content, and other web assets. It improves the performance of your applications by caching content at edge locations around the world, reducing latency and providing a better user experience.



Architecture diagram for inner working of CloudFront

CloudFront improves performance and reduces latency for both static content stored in a S3 bucket as well as dynamic content stored behind say, multiple Application Load balancers and EC2 instances. (i.e. Cloudfront distributions can have multiple services as their origins.) Also worth noting is that all Cloudfront distributions offer DDoS protection by default due to them having AWS Shield Advanced installed on them out-of-the-box.

Global Accelerator

Applications hosted using AWS often have to serve information to users and clients quite far from their geographical data center locations, resulting often in considerable latency and less-than-desirable data transfer speeds.

AWS however, as mentioned before, has an entire global network of infrastructure at its disposal. AWS Global Accelerator is a service designed to take advantage of said infrastructure in order to reduce latency and improve data transfer speeds. It also provides the applications using it with features such as automatic failover, helping it ensuring higher availability and reliability alongside faster loading and operating times.

This makes it an excellent fit for non-HTTP use cases such as gaming (UDP), IoT, and Voice over IP, where low latency and fast data transfer are critical. In fact, there are many use cases where the utilization of Global Accelerator is basically a pre-requisite such as the updating of financial boards and various IoT applications.

That does not however mean that the service is only meant for non-HTTP applications, and can enhance the connections for HTTP applications by providing a more consistent and reliable experience as well.

Cross Origin Resource Sharing

A major source of headache for engineers around the world, Cross-Origin Resource Sharing (or CORS for short) is a security feature in web browsers that controls how web pages can request resources from a different domain. It ensures safe interactions between web pages and resources across different origins, protecting against **cross-site scripting (XSS) attacks**.

As a general rule of thumb, enabling CORS is basically essential for public-facing services like Amazon S3 and API Gateway. For S3 in particular, it allows web applications to directly request and interact with S3 buckets from different domains, useful for functionalities like file uploads. While for API Gateway, CORS settings enable APIs to be accessed from web applications on different domains. This integration allows client-side applications to connect to serverless backend services securely and efficiently.

Also worth noting is that Amazon CloudFront also leverages CORS through its `Access-Control-Allow-Origin` header. This header specifies which origins are permitted to access resources on a CloudFront distribution. By configuring the `Access-Control-Allow-Origin` header, we can control which domains can make cross-origin requests to your CloudFront-distributed content, ensuring secure and efficient delivery of static and dynamic content across different domains.

Direct Connect

AWS Direct Connect, sometimes also referred to as simply DX, is a network service that establishes a dedicated, private, wired connection between on-premises infrastructure and AWS services, in fact AWS claims it to be the shortest path to the AWS resources we rent from them, providing us with a high-speed, low-latency connection that helps improve performance and allow us to receive a more consistent network experience with our cloud infrastructure compared to standard internet connections.

Direct Connect is ideal for applications requiring stable and high-throughput data transfer, such as large-scale data migrations, hybrid cloud architectures, and real-time data processing. Since the service allows us to connect to AWS resources through a wired connection, it essentially bypasses the public internet and therefore essentially eliminates the security threats and interruption risks associated with web-based traffic further enhancing the reliability and connectivity to AWS resources compared to standard connections.

Finally, it is also worth noting that because DX is based on dedicated private connections, it does not compete with the organization’s bandwidth allowing us to negate any internet-speed restrictions for example.

Subnets

As discussed before, VPCs (Virtual Private Clouds), i.e. the virtual boxes into which we place our AWS resources are an integral component of any cloud environment/architecture that is even remotely complex.

Well, to take the box analogy one step further, when placing a vast array of things into boxes, is it not so much easier and neater if we were to group all the similar items together? Such as putting all the toys in one compartment and all the books in another. Makes things a whole lot easier right? Subnets in AWS are similar to those compartments except for VPCs. They allow us to logically separate different types of resources within your VPC.

So one could for example, have a VPC containing both the web server EC2 instances as well as say, RDS database instances. Subnets allow us to segment portions of a VPC allowing us to logically isolate our EC2 instances from the RDS database instances. They allow us to allocate IP addresses to resources (like EC2 instances) and enforce network access controls through Network Access Control Lists (ACLs) and Security Groups (Will be discussed in later chapters) therefore playing a crucial role in organizing and securing your AWS resources within a VPC environment.

Let us illustrate the logical segmentation performed by the VPC using the aforementioned scenario as an example, utilizing CIDR blocks:

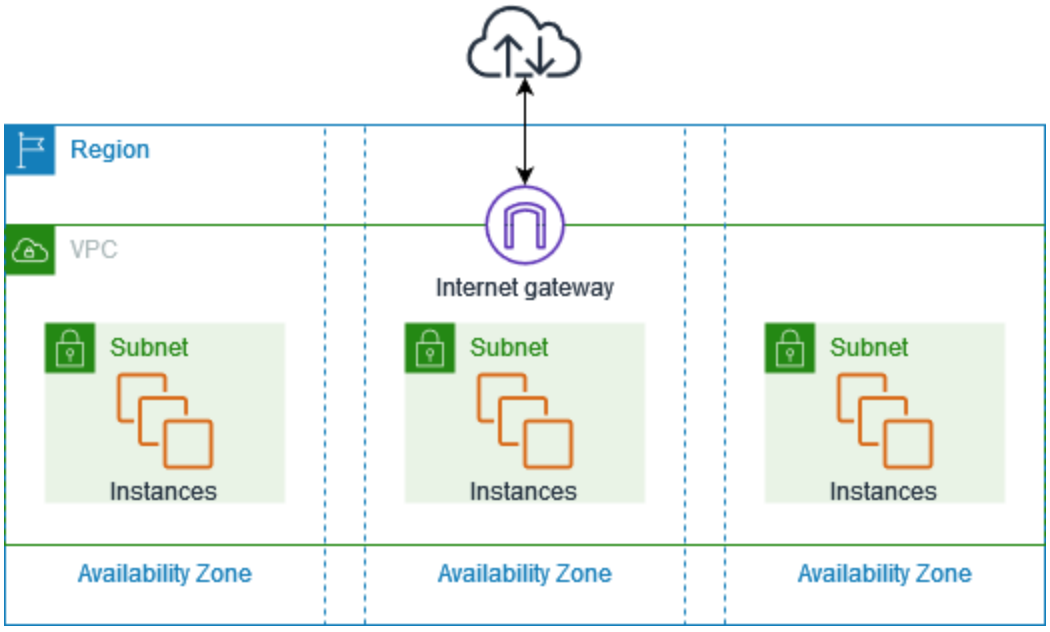
Imagine the hypothetical VPC CIDR block to be of the range: `0.0.0.0/16`

- This gives us a total of 65,536 IP addresses (from `10.0.0.0` to `10.0.255.255`)

Then, we can design the Subnet to be as follows:

- **Public Subnet for Web Servers**
  - **CIDR Block**: `10.0.1.0/24`
  - **IP Address Range**: `10.0.1.1` to `10.0.1.254`
  - **Purpose**: This subnet will host the EC2 instances that run the web servers, allowing them to be accessible from the internet.
- **Private Subnet for RDS Database Instances**
  - **CIDR Block**: `10.0.2.0/24`
  - **IP Address Range**: `10.0.2.1` to `10.0.2.254`
  - **Purpose**: This subnet will host the RDS instances. It is private, meaning the RDS instances are not directly accessible from the internet.

Note however that subnets are afflicted by a restriction which are not applicable to VPCs, as each subnet created by us is mapped to a single Availability Zone while VPCs have the capability to span across multiple availability zones.



Architecture of a VPC with 3 Subnets (Source: AWS)

Block Size of a VPC

In the context of VPCs, block size refers to the range of IP addresses defined by CIDR notation that can be allocated for subnets i.e. the range of addresses that can be used to define subnets. AWS VPC allows different block sizes, typically ranging from `/16` (65,536 IP addresses) to `/28` (16 IP addresses). The 10.0.0.0/16 CIDR block notation for example, theoretically gives us access to 65,536 addresses (from 10.0.0.0 to 10.0.255.255). However, not all of these are usable as AWS has something called reserved addresses, which is used for the operation of the subnet and are not made available to us. These IP Addresses, and their uses are as follows:

- **.0**: Network address, used to identify the network device
- **.1**: Reserved for the VPC router
- **.2**: Reserved for DNS
- **.3**: Reserved for future use
- **.255**: Broadcast address, the last address within the IP address range, it is used to communicate with other broadcasting devices

Thus, while a `/16` subnet may offer 65,536 addresses, only 65,531 of them are actually usable. Here are some examples of usable IP addresses within the 10.0.0.0/16 CIDR block:

- **First Usable Address**: **10.0.0.1**
- **10.0.1.1**
- **10.0.128.5**
- **Last Usable Address**: **10.0.255.254**

Though we have been using the `/16` subnet so far, it is obviously not the only CIDR block size available, a list of the most commonly used CIDR Block sizes and their number of IP Addresses:

CIDR Notation	Total IP Addresses	Usable IP Addresses	Suitable For
/8	16,777,216	16,777,214	Very Large Applications
/16	65,536	65,534	Large Applications
/20	4,096	4,094	Medium to Large Applications
/24	256	254	Small to Medium Applications
/25	128	126	Small to Medium Applications
/26	64	62	Small to Medium Applications
/27	32	30	Small Applications
/28	16	14	Very Small Applications
/29	8	6	Very Small Applications
/30	4	2	Smallest Applications

Note that block sizes with a larger amount of available IP addresses have a premium associated with them and as such choosing the right block size is as much an exercise in cost minimization as it is in IP Address or device management.

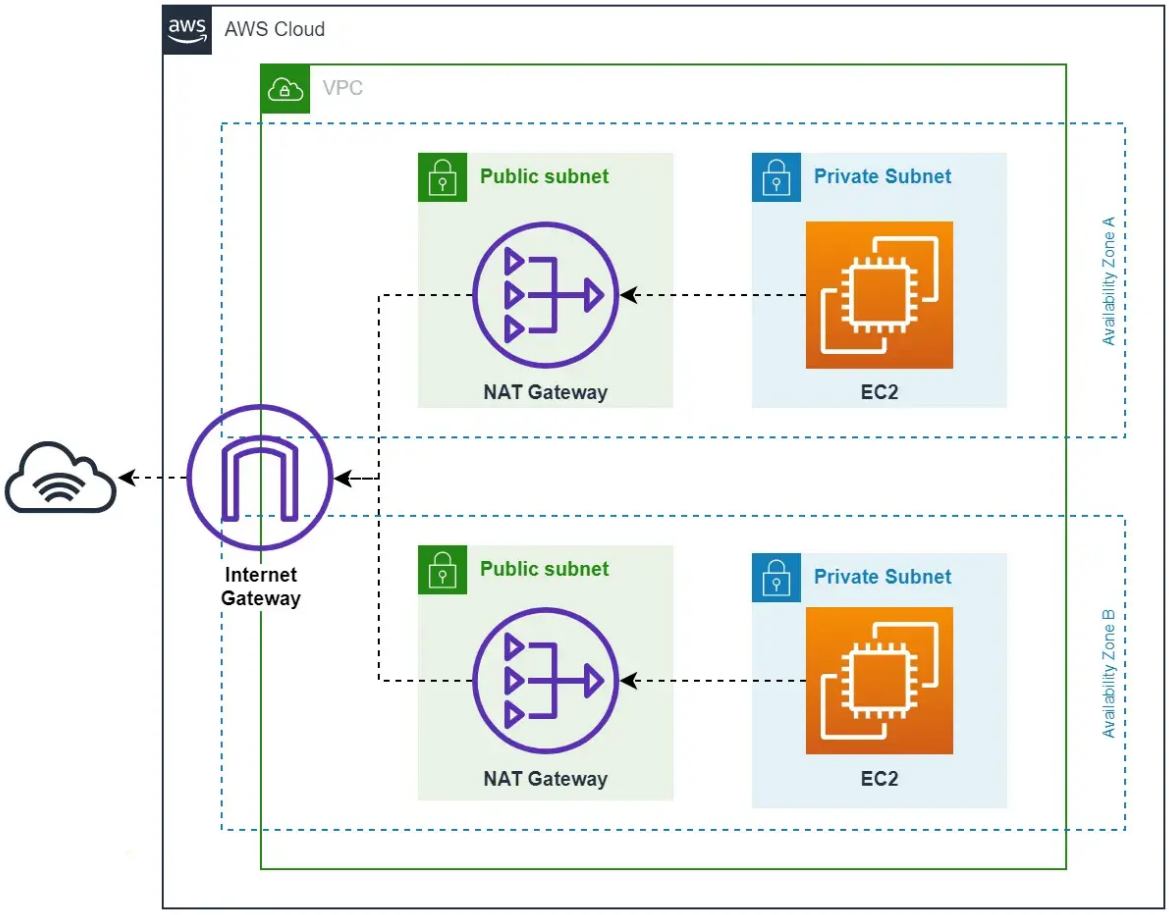
NAT and Internet Gateways

Network Address Translation or NAT for short, allows instances in a private subnet to access the internet while preventing the internet from initiating connections to those instances. This is crucial for maintaining security, as it helps shield private resources from external threats. In modern AWS environments Network Address Translation (NAT) services are provided by something called NAT Gateways. NAT gateways are used so that our AWS resources such as EC2 or RDS instances that are occupying a private subnet can connect to external services i.e. services outside the VPC, but said external services cannot initiate a connection with the instances inside the private subnet, allowing our web servers and databases to scour the internet or fetch software updates for example, without the risk of malicious actors accessing them.

When we create a NAT gateway, we must specify one of the following connectivity types for it:

- **Public** – The default connectivity type, instances in private subnets can connect to the internet through a public NAT gateway, but cannot receive unsolicited inbound connections from the internet. You create a public NAT gateway in a public subnet and must associate an elastic IP address with the NAT gateway at creation. You route traffic from the NAT gateway to the internet gateway for the VPC. Alternatively, you can use a public NAT gateway to connect to other VPCs or your on-premises network. In this case, you route traffic from the NAT gateway through a transit gateway or a virtual private gateway.
- **Private** – Instances in private subnets can connect to other VPCs or your on-premises network through a private NAT gateway. You can route traffic from the NAT gateway through a transit gateway or a virtual private gateway. You cannot associate an elastic IP address with a private NAT gateway.

Internet gateways on the other hand, are a more general purpose VPC component, allowing the resources within a VPC to have full access to the internet, and scour it in a manner which is more similar to how we browse it ourselves, allowing both inbound and outbound connections to be established. Internet Gateway therefore serves as both a bridge for outbound traffic from your VPC to the internet whilst providing a path for inbound traffic as well.



Architecture showing NAT Gateways connected to an Internet Gateway

Note in the above diagram how the NAT Gateways are visualized as an additional layer between the EC2 instances and Internet gateway, this is because the presence of an internet gateway is necessary for the functioning of a NAT Gateway though the opposite is never true.

NAT Gateway vs NAT instance

NAT Gateways and NAT instances both serve the same purpose of providing Network Address Translation capabilities within an AWS ecosystem. However, they possess characteristics that make them quite different from each other. These characteristics and how they make both offerings suitable for slightly different use cases is explored below:

A NAT Instance is an EC2 instance configured to perform NAT functionality. The burden of installing and managing said NAT software, its scaling, and perform maintenance on it however falls entirely upon the cloud administrator.

A NAT Gateway on the other hand, is a fully managed service provided by AWS, the cloud administrator therefore does not need to manage the underlying infrastructure with AWS handling the scaling and maintenance of the NAT infrastructure.

NAT Gateways are a newer offering compared to NAT instances, and treated by AWS as successor to NAT instances, a much older services. NAT gateways are, as such, kind of configured for modern environments out-of-the-box, something not true for NAT instances which make them a sort-of legacy offering by default.

Therefore, it is worth noting that when dealing with NAT, AWS encourages the use of NAT Gateways in almost all possible cases, especially for newer projects. In fact, AWS suggests that if feasible, cloud administrators managing legacy projects replace all older NAT instances with NAT gateways and might be planning on making NAT instances defunct soon (though it has not done it so far, at least at the time of writing).

A more detailed comparison of the two is provided below:

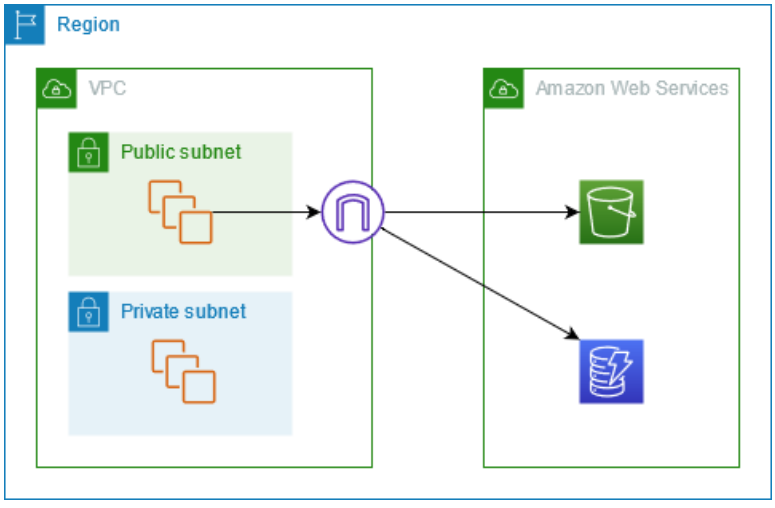
Feature	NAT Instance	NAT Gateway
Management	User-managed (you control the EC2 instance)	AWS-managed (fully managed service)
Cost	Generally cheaper (pay for EC2 instance)	More expensive (pay per hour + data processed)
Scalability	Manual (need to manage scaling)	Automatic (scales based on usage)
Availability	You need to set up redundancy and failover	Highly available by default
Performance	Limited by instance type and size	Generally better, with higher bandwidth
Maintenance	User responsible for updates and patches	No maintenance required
Use Cases	Custom configurations or special requirements	Standard use cases with no special requirements

In summary, if you're looking for ease of management, scalability, and reliability, a NAT Gateway is the way to go. If you have specific customization needs or are operating on a tight budget, a NAT Instance might be more suitable.

Gateway VPC Endpoint

Now, so far we have discussed the use of internet gateways and NAT instances, however these require us to access AWS services being hosted in the same data centers where the NAT infrastructure is through a connection external to AWS itself. Setting up and maintaining the networks using these connections therefore may be victims of needlessly redundant network hopping and operating costs.

Gateway VPC Endpoints aim to alleviate this problem somewhat by allowing us to access the AWS services of **S3** and **DynamoDB** directly from our VPC without connectivity to the internet, no NAT services required. They have become quite popular among cloud architects in recent years, probably also because of the fact that Gateway VPC Endpoints are **free** and have no additional charge associated with them, no matter the amount of inbound and outbound traffic passing through them.



Architecture diagram showcasing Gateway VPC Endpoints

VPC endpoints and how they work will be discussed in future sections.

For now, remember that VPC Gateway Endpoints differ from normal VPC Endpoints because they do not use AWS PrivateLink, unlike the other type of VPC endpoints.

Elastic Load Balancers

Now, when making complex architectures we may often find ourselves operating multiple versions of a resource to meet various scaling and operational needs. When one EC2 instance is not sufficient to satisfy all the requests for example, we often spin up and add more EC2 instances to the VPC using Auto-Scaling Groups as discussed before.

Now, in the above scenario, how do we ensure that all the EC2 instances are having the network traffic distributed properly among them, and that certain instances are being overburdened with work while others lay relatively dormant? The answer is **Load Balancers**.

Load balancers are systems that distribute network or application traffic across multiple servers, ensuring that no single server is overwhelmed with too much traffic, which can lead to performance degradation or outages.

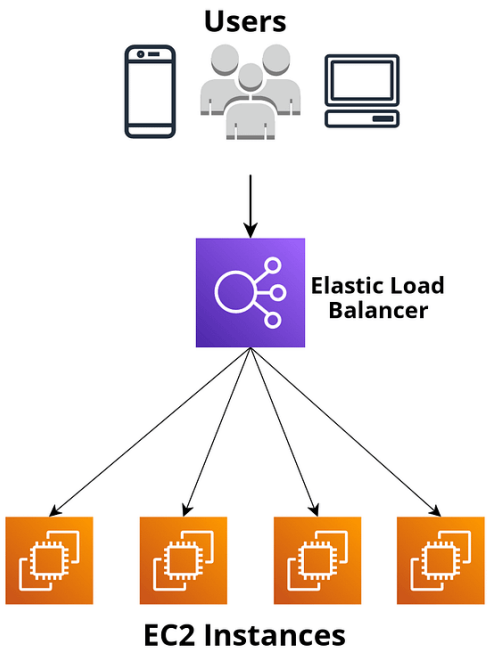
Load Balancers serve several critical purposes:

- **Traffic Distribution**: They balance incoming requests to prevent any single server from becoming a bottleneck.
- **High Availability**: Load balancers enhance the availability of applications by routing traffic to healthy instances. If one server goes down, the load balancer reroutes requests to other operational servers.
- **Scalability**: As demand increases, load balancers make it easier to add more servers, ensuring that applications can handle varying traffic loads.
- **Performance Optimization**: They can help optimize response times by directing requests to the most capable or least loaded servers.

In summary, load balancers allow us to combat **Single points of failure**, **Traffic spikes** and **Resource management**.

Elastic Load balancers (ELB) is the package provided to us by AWS that allows us to implement load balancers, distributing incoming application traffic across multiple AWS resources, such as EC2 instances, containers and IP addresses, enhancing the availability and fault tolerance of your applications by balancing the load and ensuring traffic is directed to healthy instances as mentioned above.

Though there are different types of Elastic Load Balancers (will be discussed in the next section), they all occupy similar positions in AWS network architecture, usually serving as a middleman between user agents and a swathe of compute or database instances as shown below:



Simple architecture diagram showcasing an Elastic Load Balancer

Finally, it is worth noting that Elastic Load Balancers, just like subnets within a VPC, can only be connected to Availability Zones in a single region.

Types of Elastic Load Balancers

As mentioned in the last section, there are different types of Elastic Load Balancers, each of which have different characteristics and serve different use cases. These are as follows:

Application Load Balancer (ALB):

Application Load Balancers operate at the application layer, the seventh layer of the OSI model. This makes it particularly suited for dealing with HTTP and HTTPS traffic, two prominent application layer protocols. Its primary strength lies in advanced routing capabilities, which include host-based and path-based routing. This functionality allows ALBs to intelligently direct requests to different backend services based on the content of the request.

Additionally, they provide features like sticky sessions and request tracing, enhancing the user experience by maintaining session continuity which combined with their intelligent routing capabilities have led to the widespread adoption of ALBs in dynamic web applications, RESTful APIs, microservice architectures and containerized applications; all systems and services that require and/or benefit from having fine-grained routing logic.

Network Load Balancer (NLB):

In contrast, Network Load Balancers operate at the transport layer, the fourth layer of the OSI model and is optimized for handling traffic utilizing the TCP and UDP protocols. NLBs are designed to handle millions of requests per second while maintaining ultra-low latency, making them an excellent choice for applications that demand high performance, such as gaming servers, VoIP applications, and real-time data processing.

NLBs can also operate in scenarios where static IP addresses are required, providing a more stable endpoint for applications. Furthermore, NLBs also increase a systems capacity to handling sudden spikes in traffic, and are thus often utilized in high-throughput environments.

Gateway Load Balancer (GLB):

The Gateway Load Balancer is a newer addition to AWS's suite of load balancers, operating primarily at the third layer of the OSI model. Its primary purpose is to facilitate the deployment and scaling of third-party virtual appliances such as firewalls, intrusion detection systems, and other network security devices.

A more structured, tabular comparison of the three different types of load balancers is given below:

Feature	**Application Load Balancer (ALB)**	**Network Load Balancer (NLB)**	**Gateway Load Balancer (GLB)**
OSI Layer	Layer 7 (Application Layer)	Layer 4 (Transport Layer)	Layer 3 (Network Layer)
Protocol Support	HTTP, HTTPS, WebSocket, HTTP/2	TCP, UDP	Primarily IP traffic (integrates with other services)
Routing Capability	Advanced routing (host-based, path-based)	Simple routing based on IP and port	Distributes traffic to virtual appliances
Performance	Optimized for complex application traffic	High performance, low latency, millions of requests	Scalable for virtual appliances, but latency depends on appliances
Use Case	Microservices, dynamic web applications	Gaming, VoIP, real-time data processing	Security appliance integration, network monitoring
Health Checks	Application-level health checks	Transport-level health checks	Monitors appliance health, not application health
Sticky Sessions	Supports sticky sessions	Does not support sticky sessions	Not applicable

Health Checks

We mentioned Health Checks in the last section, but what are they exactly and what purpose do they serve? Well that is what we will discuss in this section.

Health Checks as the name suggests, are processes that monitor the status and performance of our AWS resources and components to determine whether or not they are functioning correctly, i.e. whether or not they are “Healthy”. As mentioned in the last section, they are an integral part of load balancers and are often used to assess whether backend assets (such as EC2 instances) are capable of handling traffic. They are quite widely used and help us better our systems by providing:

- **High Availability**: Health checks help maintain high availability by ensuring that traffic is only routed to instances that are operational. If an instance fails a health check, it is marked as unhealthy, and the load balancer stops sending traffic to it.
- **Fault Tolerance**: By identifying unhealthy instances quickly, health checks facilitate swift recovery and failover, allowing the system to redirect traffic to healthy instances and minimize downtime.
- **Performance Monitoring**: Health checks can provide insights into the performance of application components, allowing teams to detect and address issues proactively.

Of the three types of load balancers, Gateway Load Balancers do not support health checks leaving us with the other two, namely: Application Load Balancers and Network Load Balancers.

Application Load Balancers support HTTP health checks and are the best at it, allowing them to check specific API URLs and endpoints, directing the requests only to the paths that it has determined to be capable of performing work successfully.

Network Load Balancers on the other hand, primarily handle health checks related to TCP/UDP traffic, and are especially useful for operations requiring low latency and high throughput, such as real-time gaming or IoT applications.

Note however that Network Load Balancers are also capable of performing HTTP health checks, though NLB health checks are simpler in nature, typically only checking if a specific port is open for connection, and not performing anything beyond that making them extremely suitable for simple services that do not require complex health verification.

Finally, an often overlooked aspect of health checks in AWS are their integrations with other features such as Auto Scaling. Not many know this but we can actually use AWS Auto Scaling to automatically add or remove instances based on health check status.

For example, if an instance is marked as unhealthy, Auto Scaling can launch a new instance to replace it, automating maintenance and ensuring continuous availability. This is usually done by utilizing a sub-feature of Auto Scaling in AWS called Auto Scaling actions. I shall spare the reader with a more in-depth explanation of the sub-feature as I judge it to be beyond the scope of both this section and this book in general.

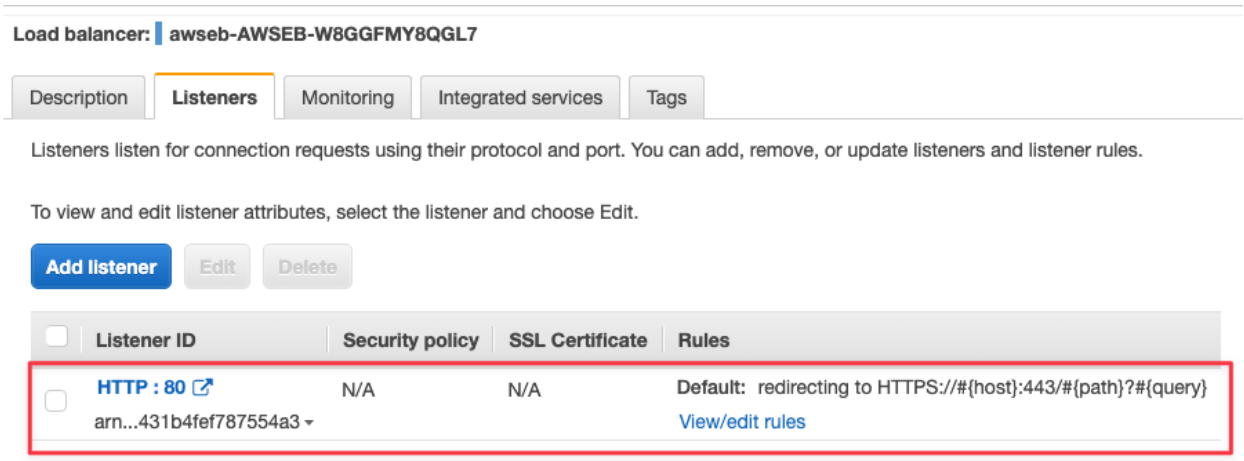
Listener Rules and Redirecting

As mentioned in the previous sections, Application Load Balancers (ALBs) have the ability to perform complex routing decisions, directing incoming traffic to appropriate backend services. Well, ALBs are able to make said decisions through the use of **Listeners** and **Listener Rules**.

Listeners are the processes that check for connection requests using the port and protocol we specify. Said listeners have a set of **rules**, called listener rules defined within them that determine how the load balancer should route requests to its registered **targets**. Both Listener Rules and their targets have to be defined by the cloud administrators when an Application Load Balancer (ALB) is being being created.

Listen Rules and ALBs allow us to perform a great many functions in conjunction beyond just simple traffic routing. One such function is the ability of ALBs to redirect HTTP requests to HTTPS, ensuring that all traffic is transmitted with the extra layer of security provided by HTTPS. Another such function is request manipulation, that is, the ability of Listener Rules to be able to modify the contents of a request before they reach backend services by, for example, adding headers to them. Request manipulation can prove especially handy when dealing with, say, CORS restrictions (a major source of headache for many people around the world).

An example of a Listener that checks for connections requests using HTTP on port 80, and redirects said request to HTTPS is shown below:



A listener rule for redirecting from HTTP to HTTPs (Source: Vanta)

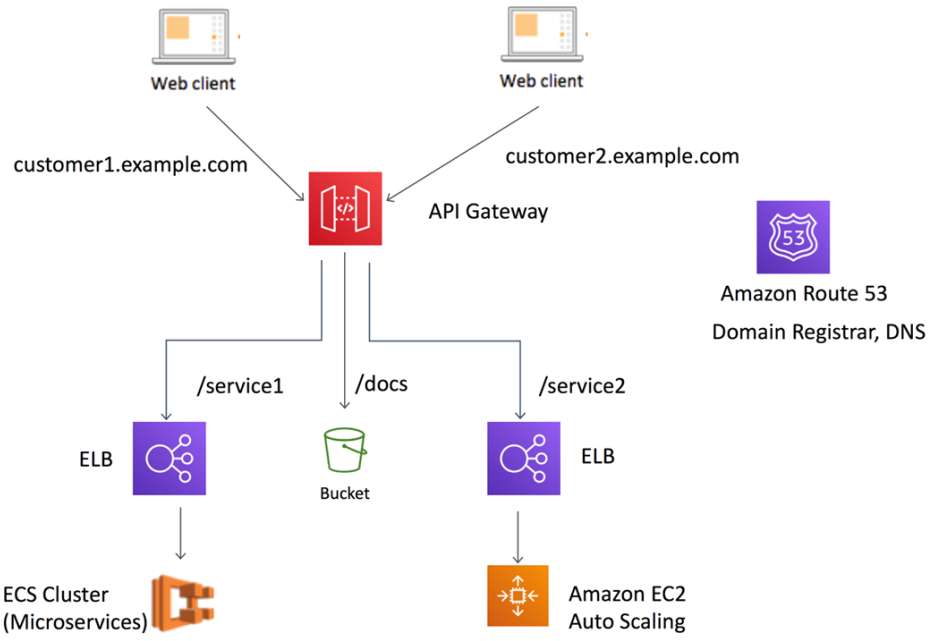
TLDR;

Listener Rules are the conditions set on an ALB that determines how requests should be handled. Used to redirect HTTP to HTTPS; Can also add headers to requests.

API Gateway

APIs are a widespread part of the modern technical world, serving as the middlemen powering communications between many software applications using the internet. In order to allow organizations to more easily create, publish, maintain and monitor APIs at scale, Amazon has a fully-managed service called API Gateway. It handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management thereby reducing a great deal of the hassle associated with APIs.

An architecture diagram of an organization that uses an API Gateway as an entry point to multiple software applications is shown below:



Source: AWS

Also worth mentioning is that the service has no minimum fees or startup costs and that we only have to pay for the amount of times the API was called and the amount of data transferred out. In fact, the mentioned advantages of API Gateway have led to a great deal of organizations moving their legacy API workloads to Amazon API Gateway, which often operate on the company’s domain name and corresponding certificate.

(Optional: Only necessary reading for AWS administrators)

To design the API Gateway URL with the company's domain name and corresponding certificate, the company needs to do the following:

1. Create a Regional API Gateway endpoint: This will allow the company to create an endpoint that is specific to a region.
2. Associate the API Gateway endpoint with the company's domain name: This will allow the company to use its own domain name for the API Gateway URL.
3. Import the public certificate associated with the company's domain name into AWS Certificate Manager (ACM) in the same Region: This will allow the company to use HTTPS for secure communication with its APIs.
4. Attach the certificate to the API Gateway endpoint: This will allow the company to use the certificate for securing the API Gateway URL.
5. Configure Route 53 to route traffic to the API Gateway endpoint: This will allow the company to use Route 53 to route traffic to the API Gateway URL using the company's

Route53 and Failover

Amazon Route 53 is a scalable Domain Name System (DNS) web service that translates domain names into IP addresses and routes users to endpoints, turning a user-friendly URL like `www.example.com` into a numeric IP address like `192.0.2.1`, usually representing an EC2 instance, a load balancer, or another type of endpoint or AWS resource.

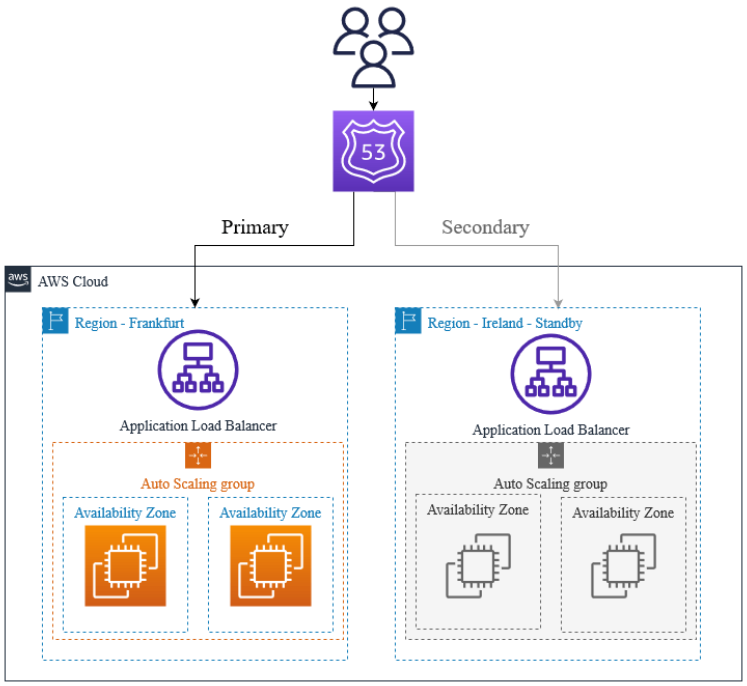
The service also offers a myriad of routing policies, capable of directing traffic to the necessary IP address based on specific criteria and conditions. A brief summary of the different routing policies is given below:

- **Simple Routing**: This is the most straightforward option, where Route 53 returns a single resource record set for a domain name. It's useful when you want to direct traffic to a single endpoint.
- **Failover Routing**: Failover routing allows us to configure primary and secondary endpoints and switch between them. If the primary endpoint becomes unhealthy (as determined by health checks), Route 53 automatically routes traffic to the secondary endpoint, ensuring minimal downtime.
- **Latency-Based Routing**: This policy routes traffic to the endpoint that provides the lowest latency for the user, enhancing performance by directing users to the nearest or fastest server.
- **Geolocation Routing**: This option lets you route traffic based on the geographic location of the user. You can specify different endpoints for users in different regions, which can be helpful for compliance or to provide localized content.
- **Geoproximity Routing**: Similar to geolocation routing, this policy allows you to route traffic based on the user's location and the location of your resources. You can also specify bias to favor certain endpoints.

- **Weighted Routing**: This policy allows you to distribute traffic across multiple endpoints based on assigned weights. For example, you might route 70% of traffic to one endpoint and 30% to another, useful for gradual rollouts or A/B testing.
- **Multi-Value Answer Routing**: This option allows Route 53 to return multiple IP addresses in response to a single DNS query. Clients can then choose which address to connect to, which can enhance redundancy and load balancing.

Now, while a more in-depth summary of all the different routing policies is beyond the scope of this book, we will be taking a closer look at Failover Routing in this section, with Geoproximity and Geolocation routing being the subjects of the next section.

Failover routing, simply put, allows us to direct traffic to one particular path when a resource (designated as the primary resource) is healthy and to direct it through another path when it is not. The resource that Amazon Route53 redirects to when the primary resource is unavailable is also called the secondary resource.



Failover Routing with Primary and Secondary Resource **(Source: StormIT)**

Failover routing furthermore, has two different configuration modes, each with its own modus operandi: **Active-Active Failover** and **Active-Passive Failover**

Active-Active failover in Route 53 allows traffic to be distributed across multiple endpoints, all of which are capable of handling requests simultaneously. This setup improves availability and responsiveness by distributing traffic among healthy endpoints.

Active-Passive failover, on the other hand, uses a standby endpoint that only becomes active when the primary endpoint fails. This configuration ensures minimal downtime by automatically redirecting traffic to the standby endpoint when the primary endpoint becomes unavailable.

Therefore, we use the Active-Active failover configuration when we want all of your resources to be available the majority of the time while the Active-Passive failover configuration is to be used when we wish that the secondary resource serve us mostly in a reserve capacity, being utilized only in cases where they system is facing significant disturbances.

TLDR;

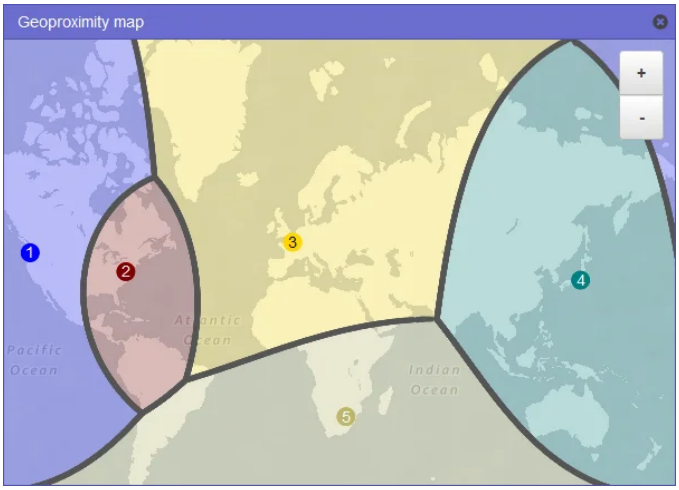
Route 53 is a DNS web service with support for failover policies that enable robust fault tolerance and high availability for applications, helping ensure a seamless user experience even during moments of endpoint failure.

Geoproximity vs Geolocation Routing

Both Geoproximity and Geolocation routing policies allow us to direct traffic based on geographical presence, but they differ in many important ways. However, a more deep dive into both routing policies is necessary to understand those differences.

Geoproximity Routing Policy

A Geoproximity routing policy is used to route traffic based on the location of our organization's resources, and shift the flow of traffic using a geographical paradigm more suitable to us.This policy allows us to direct users to different servers, even if those servers might be further away, using something called a *bias*.



Geoproximity map dividing the world based on four AWS Regions (numbered 1 through 4) and a location in Johannesburg, South Africa that is specified by latitude and longitude (5).

A bias allows us to impose and assign a certain set of resources as the de-facto traffic route for requests from a well-defined geographical boundary.

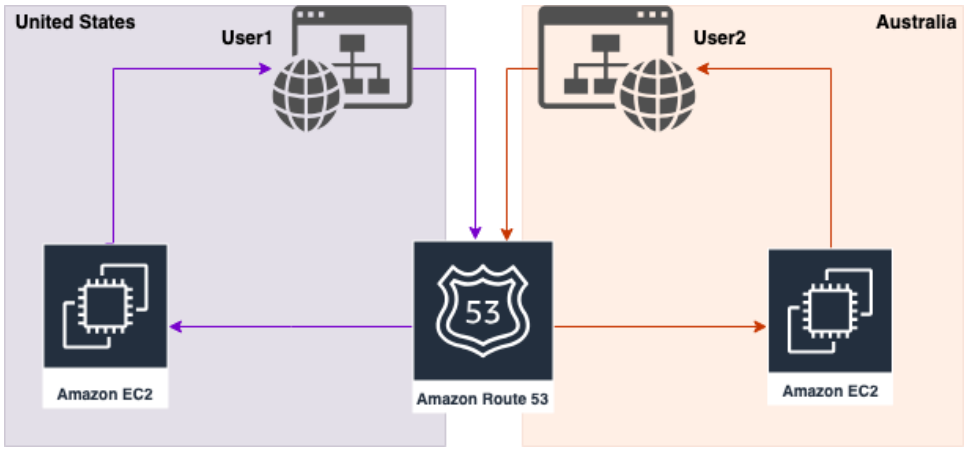
This is clearly illustrated in the diagram above where the world is divided into 5 areas, and any request originating within said area will be directed to its associated resource.

Geolocation Routing Policy

A Geolocation routing policy on the other hand, is used when when we wish to route traffic based on the location of our users. After all, many businesses these days have users all over the world, and it is natural for them to want to serve the appropriate content to those users as fast as possible. A geolocation routing policy allows you to allocate the resources that serve your traffic based on the location that users' DNS queries originate from.

Additionally, with geolocation routing, we can also localize the served content and restrict the distribution of specific content to only those locations where we are allowed to distribute, granting us the benefits of localization alongside the ability to adhere and comply with local rules and regulations.

For example, say, a streaming service like Netflix could have the international rights to an American movie and thus wants to prominently display it on its website in Australia, but not have the domestic rights to the same movie, and would have to serve something else in its stead for its users in America.



Example architecture for a Geolocation based application

Origin Access Identity and Origin Access Control

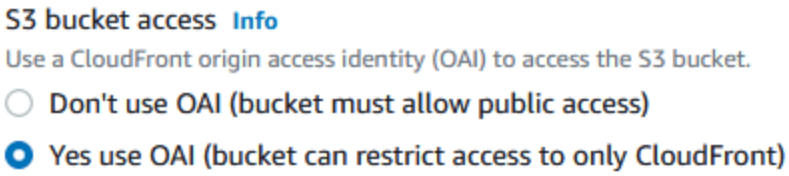
When using S3 with CloudFront, a setup often used when serving static webpages and application files, having an additional layer of security would not hurt. Well, AWS provides us with two methods of restricting access to an Amazon S3 distribution: Origin Access Identity (OAI) and Origin Access Control (OAC).

Origin Access Identity (OAI): A feature of the CloudFront service, it allows us to serve the content of a S3 Bucket without granting public access to the bucket. This restricts any unauthorized user from accessing the S3 files through its direct URL and forces them to use the CloudFront distribution URL instead.

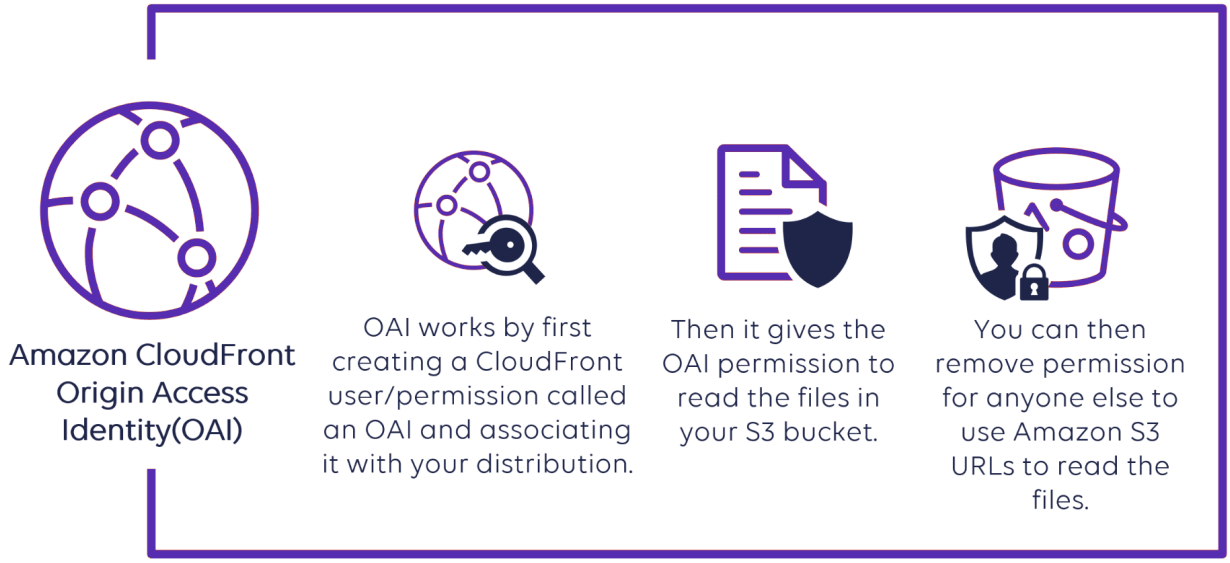
Origin Access Control (OAC): A relatively recent addition to the AWS feature lineup, it performs the same function as OAI, that of permitting access to a select group of S3 buckets, but it uses IAM principals to authenticate with the S3 bucket origin. As mentioned before, IAM is Amazon's proprietary access control and management service suite and AWS considers it a best practice to use it for the security of AWS resources.

However, OAC, unlike OAI is not exclusive to CloudFront, and can also be applied to other types of endpoints such as custom origins or origins hosted on EC2 instances. This makes it a much wider offering than OAI, and its reliance on IAM also makes it a much better integrated part of the AWS ecosystem (IAM and its features will be covered in more depth in future sections).

Refocusing back to OAI, it is worth remembering that OAI is not enabled by default and that it must be configured in the S3 bucket access settings, as shown in the image below:



Additionally, a brief description of how Origin Access Identity works is given in the infographic below:



Security Groups vs NACL

Once we have set up our Virtual Private Cloud (VPC) environments with EC2 instances or other resources within them, ensuring their security and putting guardrails in place such that undesirable users do not access our VPCs automatically becomes a crucial task. AWS provides us with two ways of achieving this: **Network Access Control Lists (NACLs)** and **Security Groups**. Short descriptions of them are given below:

NACLs (Network Access Control Lists): These operate at the subnet level and control traffic for all the instances within that subnet. Unlike Security Groups, NACLs are stateless, meaning that rules need to be explicitly stated for both inbound and outbound traffic separately. Each NACL has numbered rules, and traffic is evaluated against these rules in ascending order, starting from the lowest number. The first rule that matches the traffic is applied, and the rest are ignored. This allows for both "allow" and "deny" rules, providing more flexibility for granular control over traffic.

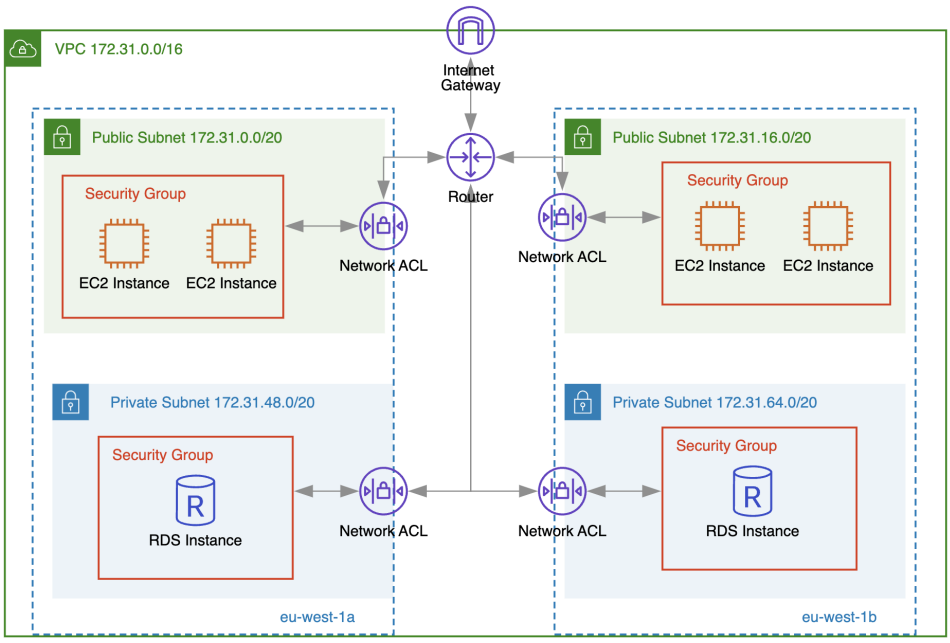
By default, NACLs have a rule that denies all inbound and outbound traffic unless otherwise specified. This makes NACLs useful for applying broad security controls to an entire subnet, as opposed to individual instances. However, their stateless nature means that we need to create separate rules for incoming and outgoing traffic, which can lead to more complex configurations compared to Security Groups. NACLs are often used in scenarios where subnet-level protection is needed, such as when managing public-facing resources that need additional security.

Security Groups: A more granular, EC2 instance focused offering, these act as firewalls at the instance level, controlling the traffic going in and out of individual

EC2 instances. Also, Security Groups are stateful, meaning that if we create a rule allowing an inbound request, the outbound response is automatically allowed, and vice versa. This simplifies configuration compared to NACLs, where each direction must be handled separately. Security Groups do not have an ordered rule set; instead, all rules are evaluated equally, and any traffic that matches an "allow" rule is permitted.

By default, Security Groups has a rule that denies all inbound traffic and allows all outbound traffic. However, they only support "allow" rules, meaning you cannot explicitly block traffic using a Security Group. This limitation is offset by their granular control over instance-level traffic. Security Groups are ideal for managing access to specific EC2 instances for example, allowing us to define precise security policies and protocols.

Furthermore, such that it becomes easier for the reader to visualize Network Access Control Lists (NACLs) and Security Groups, an architecture diagram showcasing the placement and bounds of both are given below:



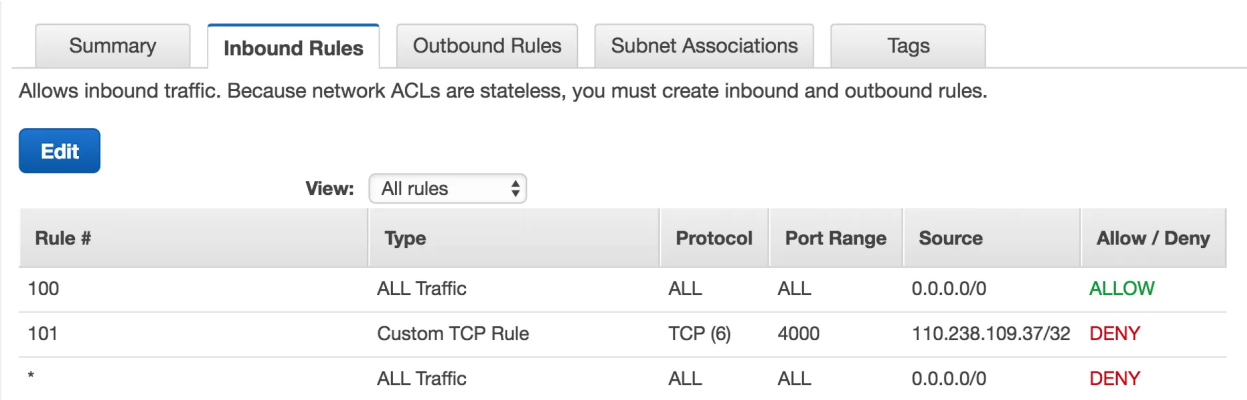
Source: CloudViz

As the descriptions might have already betrayed, though they serve the same purpose, Security Groups and NACLs operate at different levels and have quite a few contrasting characteristics. These characteristics have been tabulated for the ease of the reader below:

Security Group	Network Access Control List
Acts as a firewall for associated Amazon EC2 instances and other specific resources	Acts as a firewall for associated subnets; Ambivalent to resources within the subnet
Controls both inbound and outbound traffic at the instance level	Controls both inbound and outbound traffic at the subnet level
You can secure your VPC instances using only security groups	Network ACLs are an additional layer of defense
Supports allow rules only	Supports both allow and deny rules
Stateful i.e. Return traffic is automatically allowed, regardless of any rules)	Stateless i.e. Return traffic must be explicitly allowed by rules
Evaluates all rules before deciding whether to allow traffic	Evaluates rules in number order when deciding whether to allow traffic, starting with the lowest numbered rule
Applies only to the instance that is associated to it	Applies to all instances in the subnet it is associated with
Has separate rules for inbound and outbound traffic	Has separate rules for inbound and outbound traffic
A newly created security group denies all inbound traffic by default	A newly created NACL denies all inbound traffic by default
A newly created security group has an outbound rule that allows all outbound traffic by default	A newly created NACL denies all outbound traffic by default
Instances associated with a security group can't talk to each other unless you add rules allowing it	Each subnet in your VPC must be associated with a network ACL. If none is associated, the default NACL is selected.
Security groups are associated with network interfaces	You can associate a network ACL with multiple subnets; however, a subnet can be associated with only one network ACL at a time

NACL/SG Rule priority

Both NACLs and Security Groups have been discussed in the last section and how they rely on a sequence of rules. In order to better understand how these rules are organized and evaluated, an example set of NACL inbound rules are given below:



Summary | **Inbound Rules** | Outbound Rules | Subnet Associations | Tags

Allows inbound traffic. Because network ACLs are stateless, you must create inbound and outbound rules.

Edit

View: All rules

Rule #	Type	Protocol	Port Range	Source	Allow / Deny
100	ALL Traffic	ALL	ALL	0.0.0.0/0	ALLOW
101	Custom TCP Rule	TCP (6)	4000	110.238.109.37/32	DENY
*	ALL Traffic	ALL	ALL	0.0.0.0/0	DENY

Source: TutorialsDojo

Lower the rule number, the higher priority it is treated with. The “*” rule in particular being the lowest priority and highest number rule. For example, the “*” rule as shown in the image is the default and lowest rule of the NACL. Additionally, higher priority rules will overpower lower priority ones. In the list of rules present in the above example, rule “100“ will be called first, then “101“, then “*“. So, because all traffic goes through rule 100 first, any and all inbound requests will be passed through regardless of protocol or port range.

A similar format is followed for the organization and evaluation of NACL Outbound rules and Security Groups as well, and therefore I shall refrain from regurgitating the same set of information for the aforementioned categories.

AWS PrivateLink

It was mentioned in a previous section that there are two different types of VPC endpoints: **Gateway VPC Endpoints** and **Interface VPC Endpoints**. Since we have already spent a section discussing Gateway VPC Endpoints, I find it to be worth spending a section discussing the other type of VPC endpoint too.

Now, the main differentiator that we can use to distinguish between Interface VPC Endpoints and Gateway VPC Endpoints are its use of AWS PrivateLink and the range of services covered by it. Let us start with the second one because it is in my opinion, the more important one. Unlike Gateway VPC Endpoints which are limited to the S3 and DynamoDB services, Interface VPC Endpoints allow us to create a private and secure connection with any publicly available AWS service (well technically, any service that supports PrivateLink, which is almost all of them) like EC2, ECS, RDS and all the others that we have discussed in prior sections.

Secondly, Interface VPC Endpoints create these secure connections by utilizing private IP addresses via a service called AWS PrivateLink which allows the service to act as a bridge between VPCs, AWS services and on-premises networks without ever having to expose the infrastructure and traffic to the public internet.

Therefore, whenever we wish to connect one AWS service to another, say EC2 to SNS (Will be discussed in later sections), then utilizing interface VPC endpoints is usually the most convenient way of doing so, especially when privacy and not exposing the resources to the public internet is a major concern.