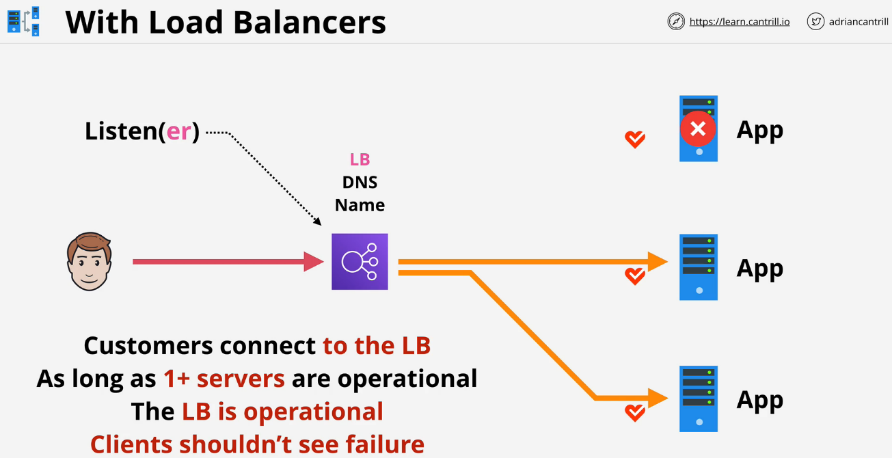
**HA- and-Scaling**

**Load Balancing Fundamentals**

Using one server is risky because that server can have performance issues or be completely unavailable, thus bringing down an application.

A better solution is to use multiple servers. **Without load balancing**, this could bring additional problems.

* The servers can end up with **uneven load**.
  + Some requests take more CPU than others.
* **Failed instances will still show up in DNS cache.**
  + Due to TTL values, a user can be directed toward a dead server.

**Load Balancers Architecture**

The user connects to a load balancer that is set to listens on **port 80(http) and 443(https).**

Within AWS, the configuration for which ports the load balancer listens on is called a **listener**.

The user is **connected to the load balancer and not the actual server**. Behind the load balancer, there is an application server. At a high level when the user connects to the load balancer, it **distributes that load to servers on the application server**. The users client thinks it is talking directly to the application server.

LB will **run health checks against all of the servers. If one of the servers does fail, the load balancer will realize this and stop sending connections to that server.** From the users client, the application always works.

As long as 1+ servers are operational, the LB is operational. Clients shouldn't see errors that occur with one server.

**LB Exam PowerUp**

* Clients connect to the **listener** of the load balancer.
* The load balancer connects to one or more **targets** or servers.
* Two connections in play.
  + **Listener connection:** one connection between the client and listener.
  + **Backend connection:** one connection between load balancer and target.
* The LB abstracts the client away from individual servers.
* Used for **high availability, fault tolerance, and scaling**

**Application Load Balancer (ALB)**

ALB is a **layer 7** or Application Layer Load Balancer. It is **capable of inspecting data that passes through.**

It can understand the application layer **http** and **https** and take actions based on things in those protocols like **paths, headers, and hosts.**

Capacity that you have as part of an ALB **increases automatically based on the load which passes through that ALB**.

This is made of multiple **ALB nodes each running in different AZs**.

This makes them scalable and highly available.

**ALB nodes are 1 per AZ.**

Load balancing can be **internet facing or internal**. The difference is whether the nodes of the LB, the things which run in the AZs have public IP addresses or not.

* **Internet facing LB** is designed to be connected to, from public internet based clients, and load balance them across targets.
* **Internal load balancer** is not accessible from the internet and is used to load balance inside a VPC only.

Load balancer **sits between a client and one or more servers**.

**Front end** or listening side, **accepts connections from a client.**

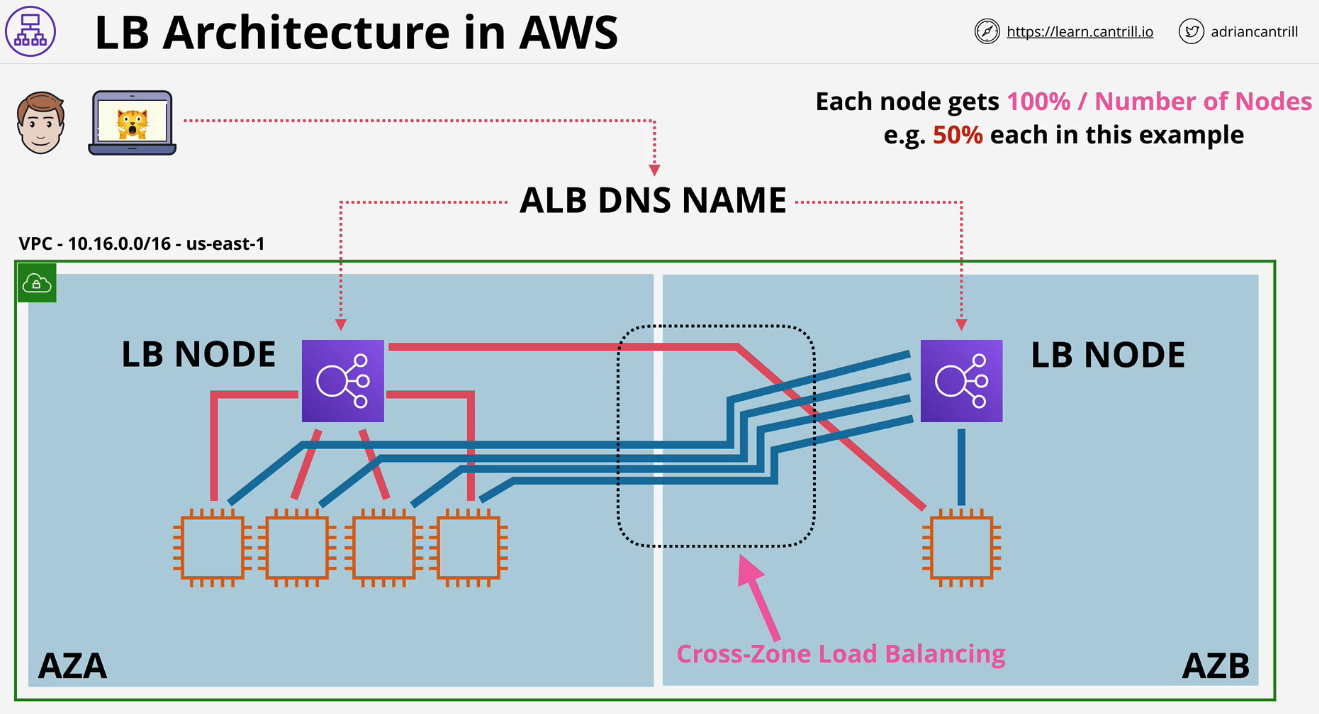
**Back end** is used for **distribution to the targets.**

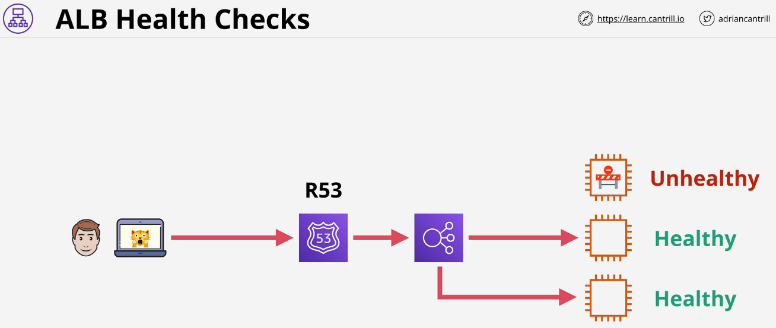
**Billing:**

LB billed on hourly rate and **Load Balancer Capacity Unit** LCU.

LCU that you consume is based on the highest value for all of the individual measurements. You pay a certain number of LCUs based on your load over that hour.

**Cross zone load balancing**

Each node that is part of the load balancer is able to distribute load across all instances across all AZ that are registered with that LB, even if it’s not in the same AZ. It is the reason we can achieve a balanced distribution of connections behind a load balancer.

****It can also **provide health checks on the target servers**. If all instances are shown as healthy, it can distribute evenly.

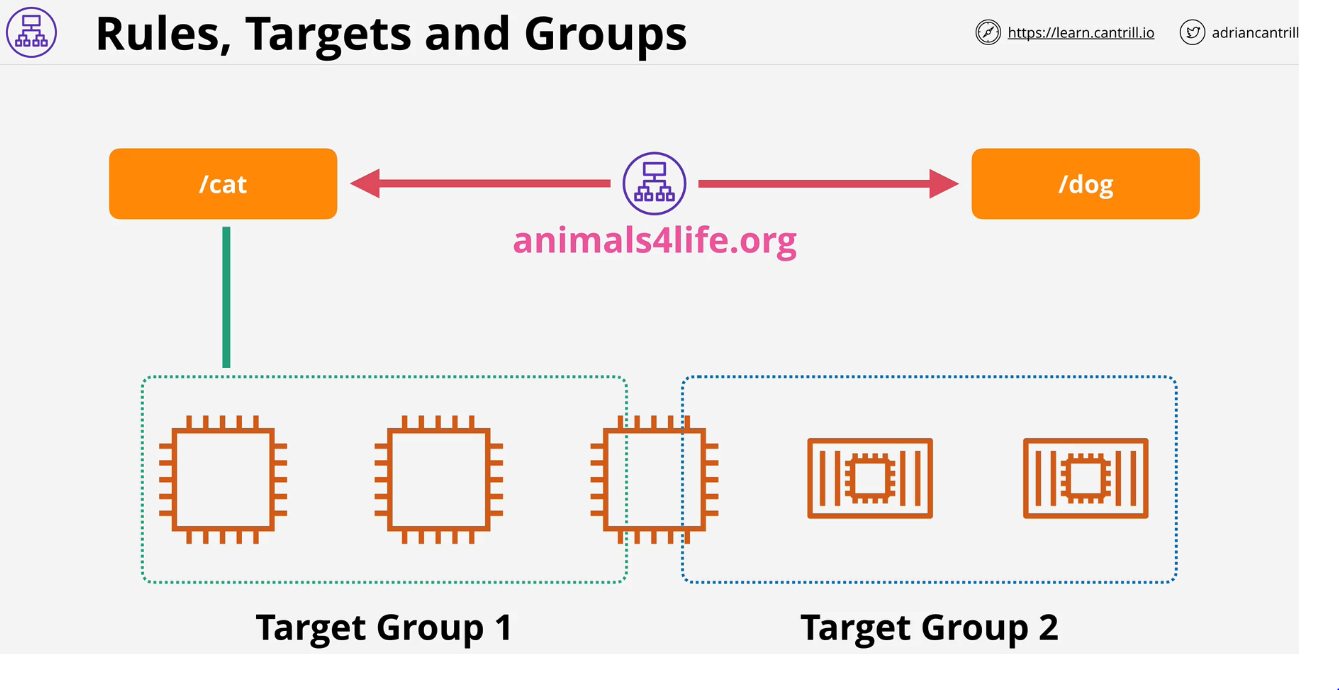
**Target Groups:**

ALB can support a wide array of targets**(EC2 instances, ECS Containers or Lambda)**. Targets are grouped within target groups and an individual target can be a member of multiple groups.

It's the groups which ALBs distribute connections to.

You could create **rules** to direct traffic to different Target Groups based on their DNS.

There can be host rules or path rules.

* **Path based Rules:** Can interpret paths. [www.a4l.org/cat](http://www.a4l.org/cat) is redirected to Target Group 1, [www.a4l.org/cat](http://www.a4l.org/cat) is redirected to Target Group 2.
* **Host based Rules:** Used when multiple domains are used within the same Load Balancer. For example – [www.dogs.a4l.org](http://www.dogs.a4l.org) is redirected to Target Group 1, [www cats.a4l.org/](http://www.a4l.org/cat) is redirected to Target Group 2.

**ALB Exam PowerUp**

* **Targets are one single compute resource** that connections are directed towards.
* **Target groups are groups of targets** which are addressed using rules.
* Rules are
  + **path based** /cat or /dog
  + **host based** if you want to use different DNS names.
* **Supports** EC2, EKS, Lambda, HTTPS, HTTP/2 and WebSockets.
* **ALB can use SNI(Server Name Indication)** for **multiple SSL certs** attached to that LB.
  + LB can direct **individual domain names using SSL certs at different target groups**.
* AWS does not suggest using Classic Load Balancer (CLB), these are legacy.
  + This can only use one SSL certificate.

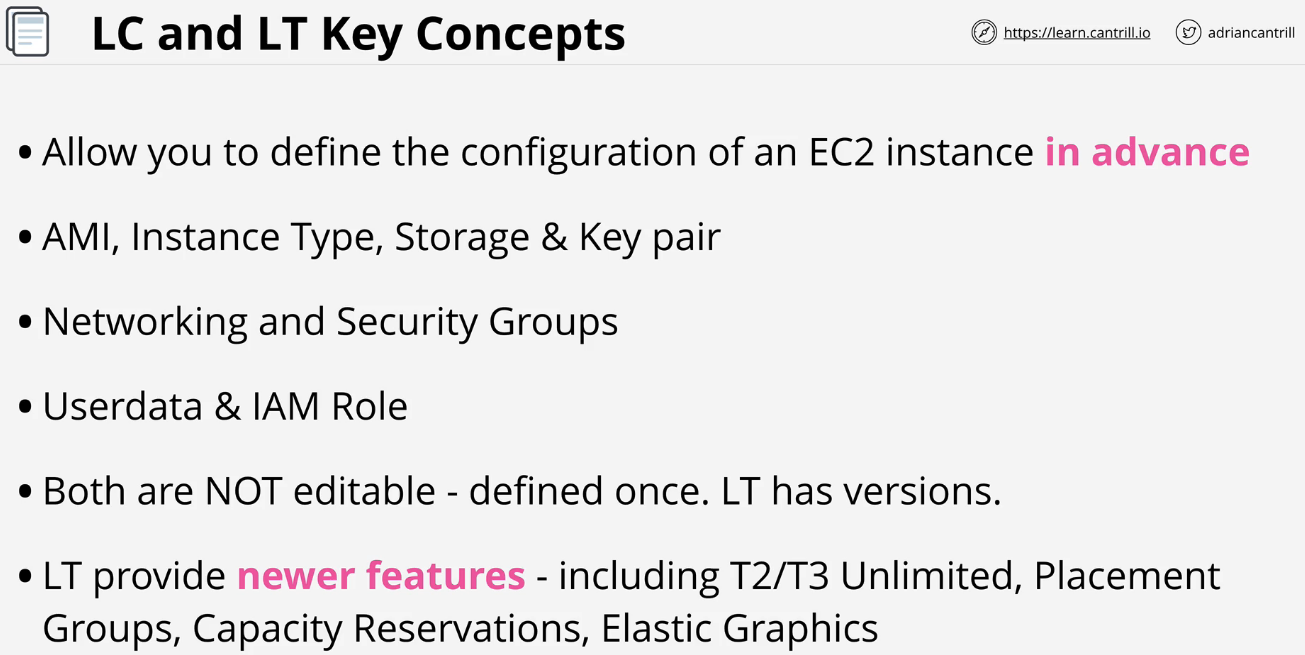
\*\*Single ALB is lower cost, we can achieve significant cost saving for large implementation involving load balancers, by consolidating lots of individual classic load balancers each using there own individual certificates onto a single application load balancer using **SNI** & **multiple host based rules.**

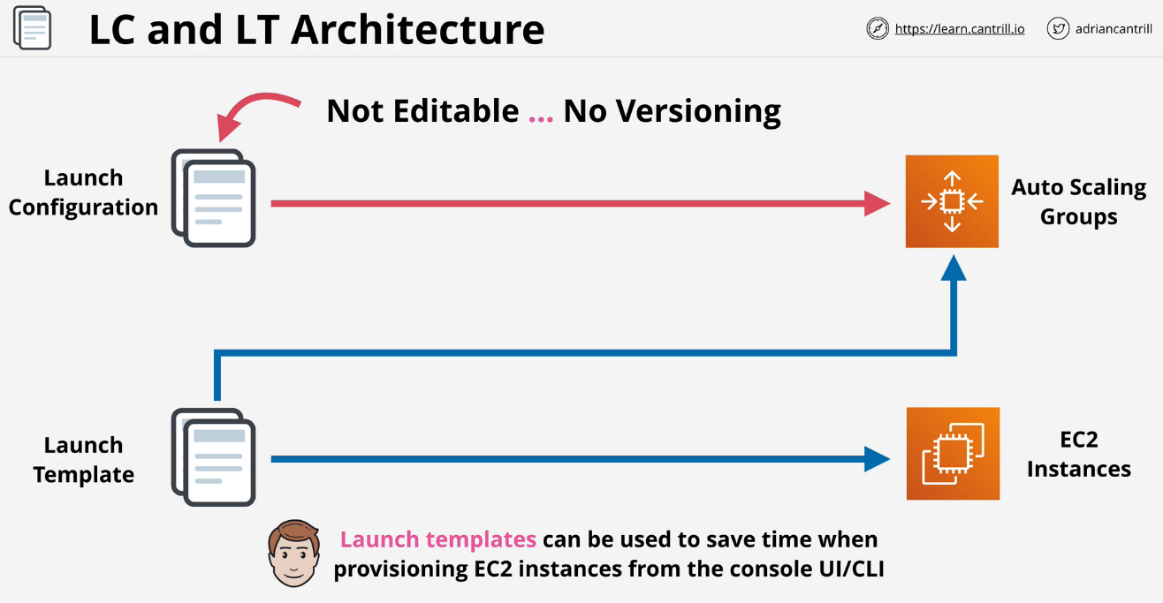
**Launch Configuration and Templates**

They are **documents which allow you to config an EC2 instance in advance**. Anything you usually define at the point of launching an instance can be selected with a Launch Configuration (LC) or Launch Template (LT).

**LTs are newer and provide more features** than LCs like **versioning** but with LC you don’t have versioning.

Both of these are **not editable.** You define them once and that configuration is locked. If you **need to adjust a configuration, you must make a new one and launch it.**

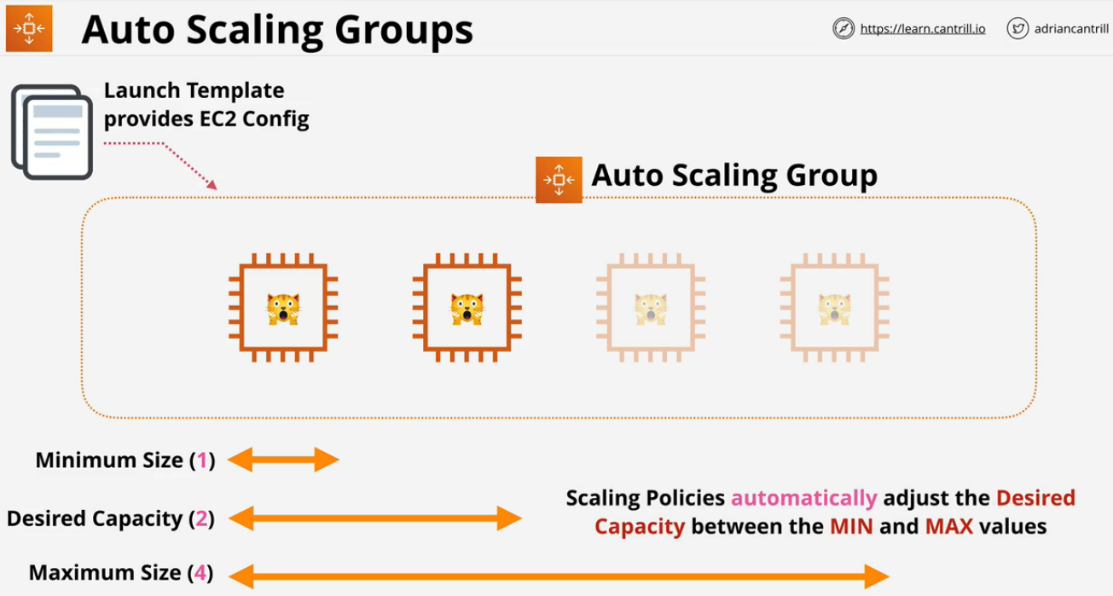
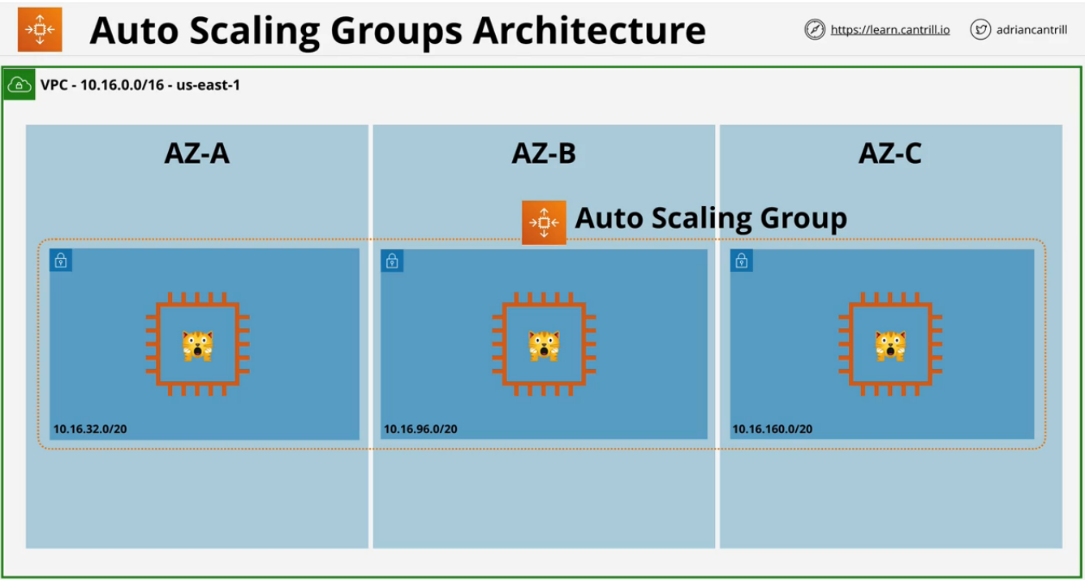


LTs can be **used to save time when provisioning EC2 instances from the console UI / CLI.**

**Autoscaling Groups**

* **Automatic scaling and self-healing** for EC2
* They make use of **LCs** or **LTs** to know **what to provision.**
* Autoscaling group uses **one LC** or **one version of a LT which it's linked with.**
* Three values to control
  + **minimum**
  + **desired**
  + **maximum**

**Provision or terminate instances** to keep at the **desired level Scaling Policies can trigger this based on metrics.**

**Autoscaling Groups** will distribute EC2 instances to **try and keep the AZs equal.**

**Scaling Policies**

* **Manual Scaling -** manually adjust the desired capacity
* **Scheduled Scaling -** time based adjustments
* **Dynamic Scaling**
  + **Simple:** If CPU is above 50%, add one to capacity
  + **Stepped:** If CPU usage is above 50%, add one, if above 80% add three
  + **Target:** Desired aggregate CPU = 40%, ASG will achieve this

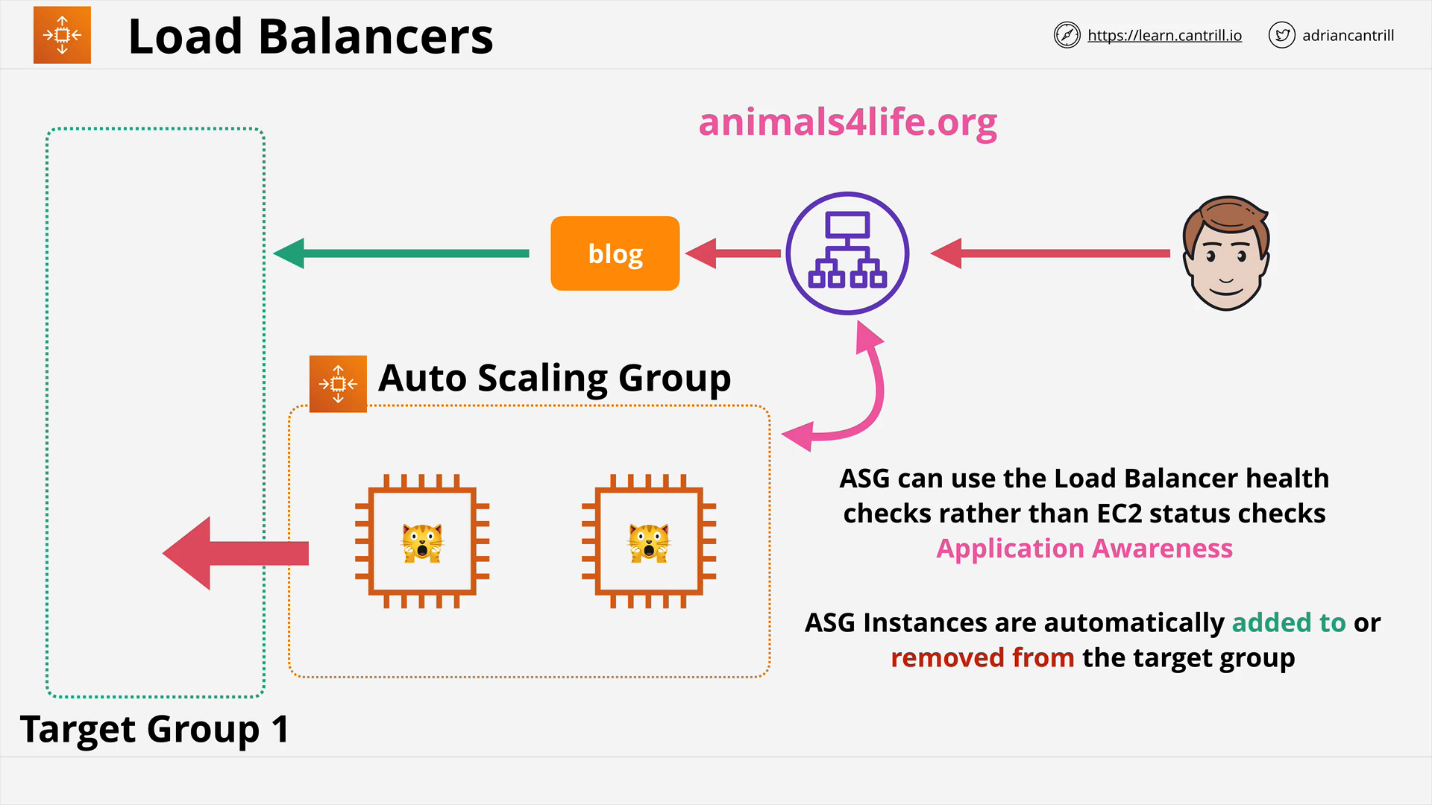
**Cooldown Period:**

Cool down period is **how long to wait at the end of a scaling action before scaling again.** There is a minimum billable duration for an EC2 instance. Currently this is 300 seconds.

**Self-Healing:**

Self-healing occurs when an instance has failed and AWS provisions a new instance in its place. This fixes most problems that are isolated to one instance.

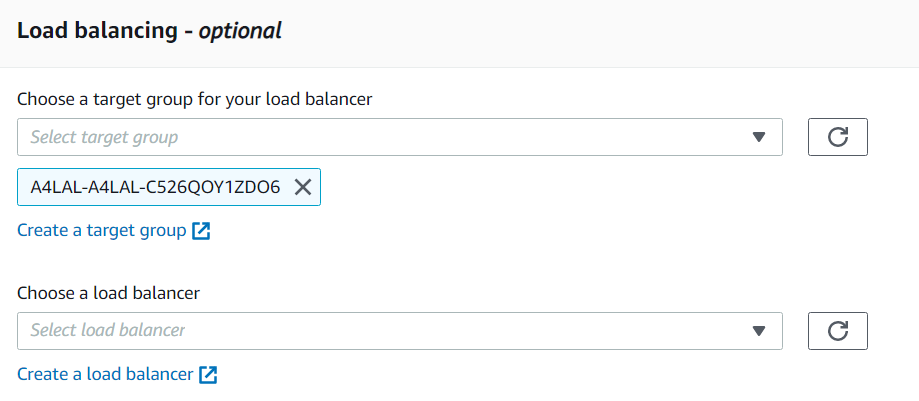
**ALB Status Checks:**

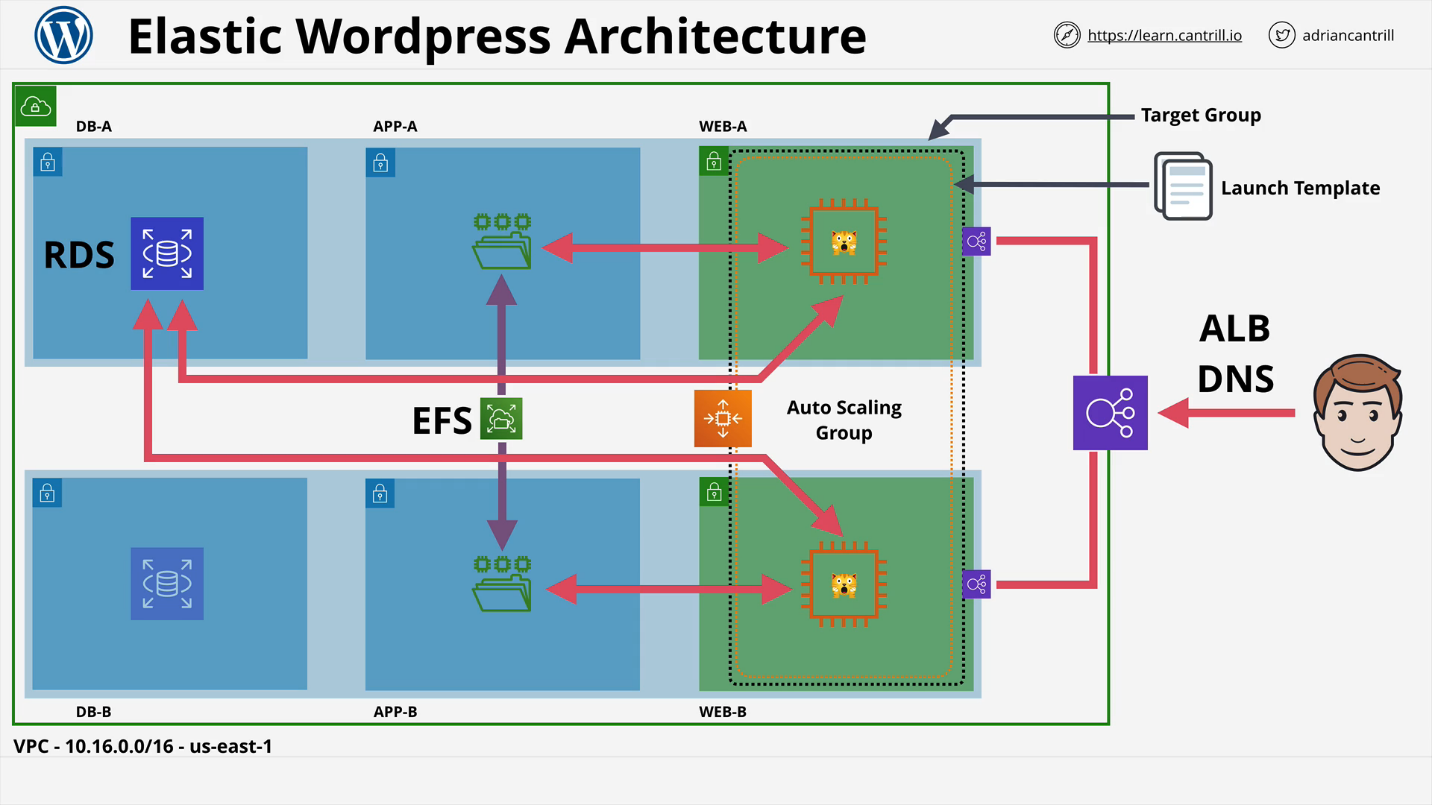
AGS can use the load balancer health checks rather than EC2. ALB status checks can be much richer than EC2 checks because they can monitor the status of HTTP and HTTPS requests. This makes them more application aware.

* Autoscaling Groups are free, **only billed for the resources deployed.**
* Always **use cool downs to avoid rapid scaling.**
* Try and **use more smaller instances to allow granularity.**
* You should **use ALB with autoscaling groups.**
* **ASG** defines **when and where**, **Launch Template** defines **what.**

\*\*While integrating the the **Application Load Balancer**, we specify the **Target Group** associated with the load balancer only.

\*\*While integrating the **Network Load Balancer**, we specify the **Load Balancer** directly.





**Network Load Balancer (NLB)**

Part of **AWS Version 2 series of load balancers**.

**NLBs are Layer 4**, only **understand TCP and UDP.**

**Can't interpret HTTP or HTTPs,** but this makes it much **faster in latency**. If you see anything about **latency** and HTTP and HTTPS are not involved, this should default to a NLB.

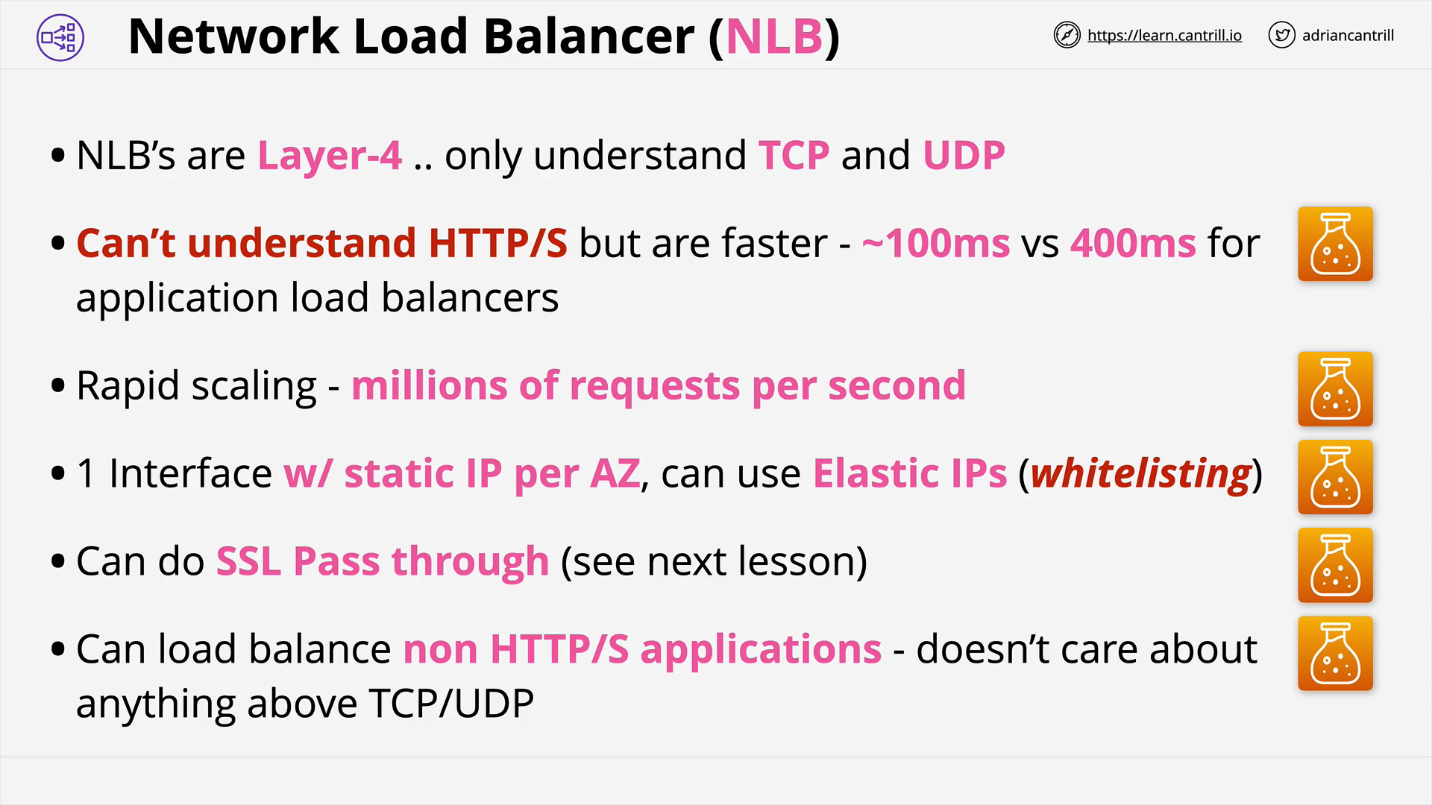
There is nothing stopping NLB from load balancing on HTTP just by routing data. They would do this really fast and can **deliver millions of requests per second.**

**Only member of the load balancing family that can be provided a static IP**.

There is **1 interface per AZ**. Can also use **Elastic IPs (whitelisting)** and should be used for this purpose.

Can perform SSL pass through.

NLB **can load balance non HTTP/S applications**, doesn't care about anything above TCP/UDP. This means it can **handle load balancing for FTP or things that aren't HTTP or HTTPS.**



**SSL Offload and Session Stickiness**

**Bridging - Default mode**

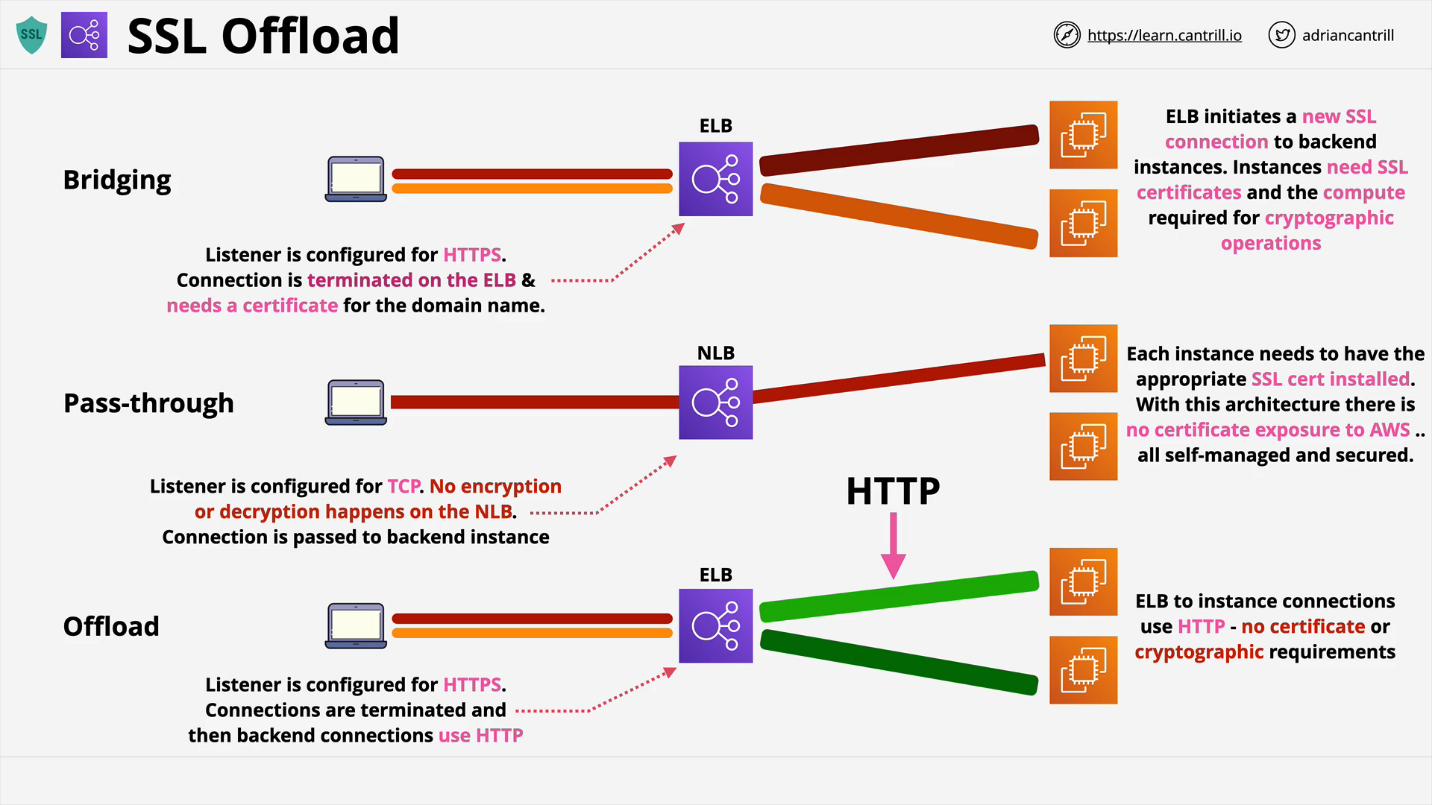
One or more clients makes one or more connections to a load balancer. The load balancer is configured so its listener uses HTTPS, SSL connections occur between the client and the load balancer.

The load balancer then needs an SSL certificate that matches the domain name that the application uses. AWS has access to this certificate. If you need to be careful of where your certificates are stored, you may have a problem with this system.

ELB initiates a new SSL connection to backend instances with a removed HTTPS certificate. This can take actions based on the content of the HTTP.

The application local balancer requires a SSL certificate because it needs to decrypt any data that's being encrypted by the client. Once decrypted, it will interpret it then create new encrypted sessions between it and the back end EC2 instances. The EC2 instance will need matching SSL certificates.

Needs the compute for the cryptographic operations. Every EC2 instance must perform these cryptographic operations. This overhead can be significant.

The main benefit is the **elastic load balancer gets to see the unencrypted HTTP and can take actions based on what's contained in this plain text protocol.**

**Pass-through - Network Load Balancer**

The client connects, but the load balancer passes the connection along without decrypting the data at all. The instances still need the SSL certificates, but the load balancer does not. Specifically it's a **network load balancer** which is able to perform this style of connection.

The load balancer is configured for TCP, it can see the source or destinations, but it never touches the encrypted connection. The certificate never needs to be seen by AWS.

Negative is you don't get any load balancing based on the HTTP part because that is never exposed to the load balancer. The EC2 instances still need the compute cryptographic overhead.

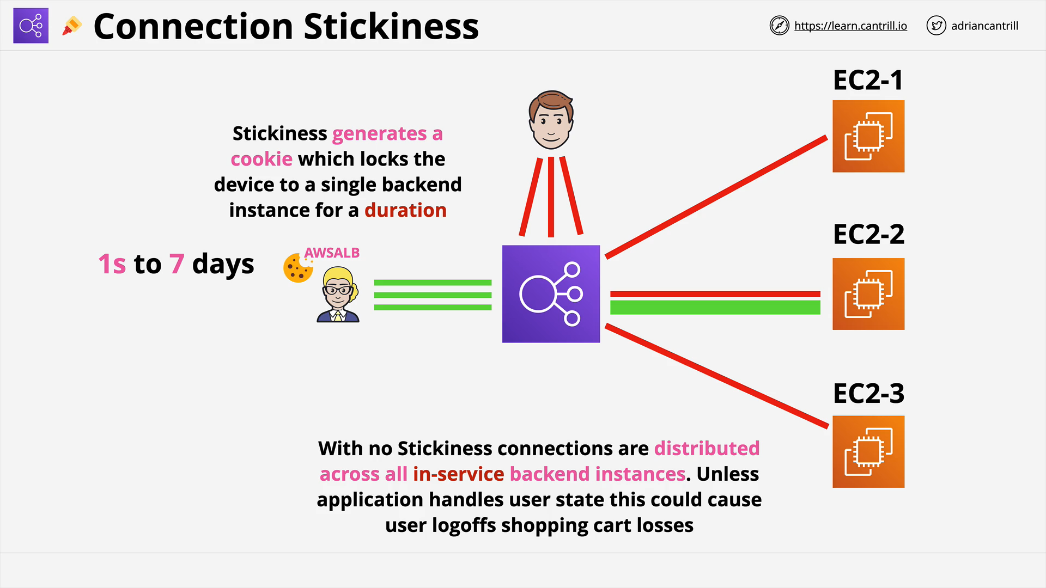
**Offload**

Clients connect to the load balancer using HTTPS and are terminated on the load balancer. The LB needs an SSL certificate to decrypt the data, but on the backend the data is sent via HTTP. While there is a certificate required on the load balancer, this is not needed on the Instances

Data is in **plaintext form across AWS's network.** Not a problem for most.

**Connection Stickiness**

If there is no stickiness, each time the customer logs on they will have a stateless experience. If the state is stored on a particular server, sessions can't be load balanced across multiple servers.

There is an option available within elastic load balancers called Session Stickiness. And within an application load balancer this is **enabled on a target group**. If enabled, the first time a user makes a request, the load balancer generates a cookie called **AWSALB** with a duration.

A valid duration is between one second and seven days. For this time, sessions will be sent to the same backend instance.

This will **happen until:**

* A **server failure**, then the user will be moved to a different server.
* The **cookie expires**, the whole process will repeat and will receive a new cookie

This could cause backend unevenness because one user will always be forced to the same server no matter what the distributed load is. Applications should be designed to hold **session stickiness somewhere other than EC2**.