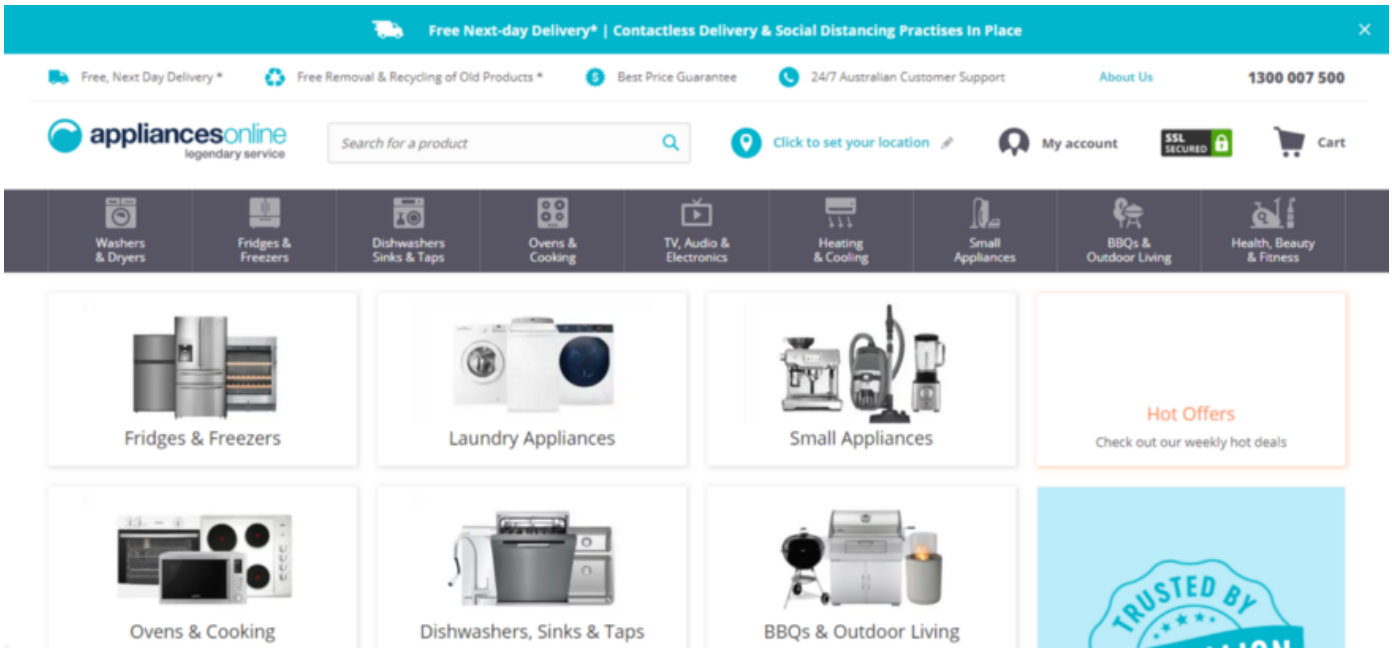
Design an E-Commerce System



How do you build an online shopping system that stores an inventory of products under different categories, and allows customers to search through them, and make purchases? Furthermore, how do you handle the expanding load on the website and prevent it from crashing down, particularly on busy days, such as the Black Friday Sale?

Table of Contents:

- How to handle peak traffic load during Black Friday Sales?

Functional Requirements

Here are the basic features an e-commerce website should include:

- Sellers should be able to add, delete and modify products they want to sell.
- The website should include a catalog of products.
- Buyers can search products by name, keyword, or category.
- Buyers can add, delete or update items in a cart.
- Buyers can purchase items in the cart and make payments.
- Buyers can view their previous orders.
- Buyers can review and rate purchased products.

Non Functional Requirements

The platform is expected to hold 100 million unique SKUs, 300 million registered customers, 1.5 million sellers, and handle 1 million orders on an average day.

payment service and inventory management will need to be highly consistent. Availability is of low priority in this case. Search service, on the other hand, should be highly available and have low latency, even at the cost of consistency. In other words, it's acceptable if the search service is eventually consistent.

Capacity Planning

- 300 m users * 10kb/user metadata on average = ~3 TB
- 300m customers * 25kb/customer metadata on average = ~7 TB
- 1 m seller * 50kb/seller metadata on average = ~ 70 GB
- 100m products * 100kb/product metadata = 9.3TB
- 100m * 2 Merchandisable Items/product on average * 10kb = ~2TB
- 1m orders/day * 365 days * 25kb = 8.5 TB order data/year

High Level Design:

Any E-Commerce website in its simplified form should have the following components :

1. **User Service:** Guests may browse items and add them to their shopping carts after searching for them and seeing them. They must first register in order to place an order. User Service will handle all the registered user info.

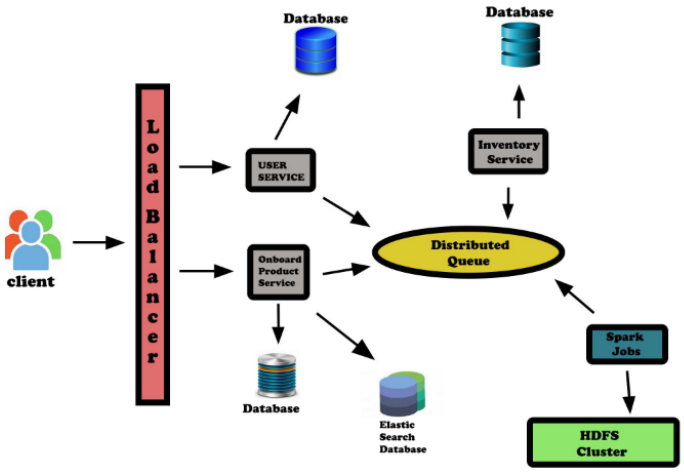
2. **Seller Service/On-Board Product Service:** Seller on board the product. Individual sellers offer items one at a time, but Professional sellers can market them in big batches via bulk uploading or inventory management with third-party platforms.
   The product will be available to both B2C and B2B clients after successfully listed.

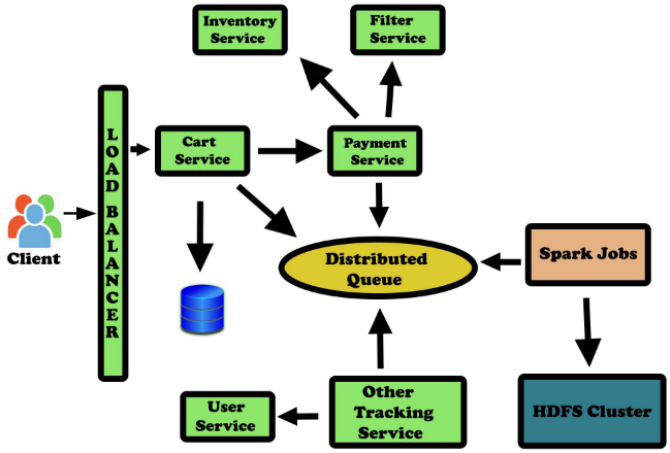
3. **Cart Service:** A shopping cart is a piece of software that makes purchasing a product or service easier. It receives the customer's payment and arranges for the information to be distributed to the merchant, payment processor, and other stakeholders.
   Users may use the shopping cart to check out and purchase products.

4. **Inventory Service:** Manage availability and warehouse location.

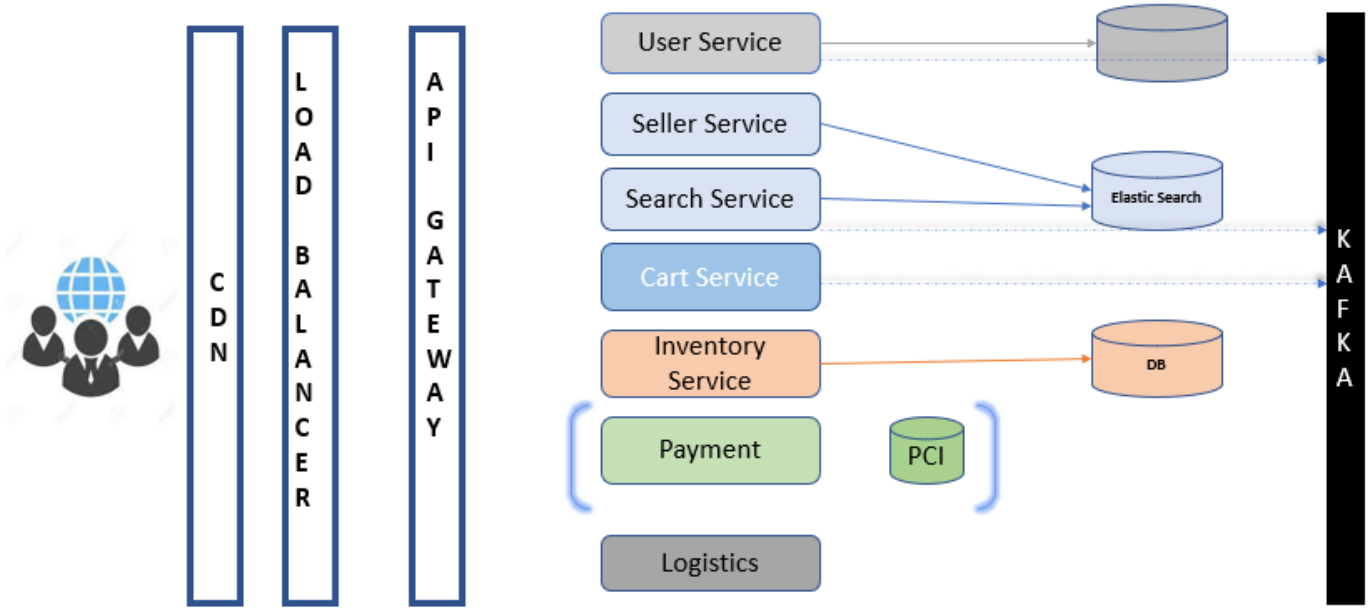
5. **Payment Service:** Payment instrument handling, charges. PCI.

6. **Logistic Service:** which identifies whether or not we deliver to a location, and calculates the ETA? It does so with the help of **Logistic Service** and **Warehouse Service**. Serviceability service can access warehouse service to get a repo of all products stored in a warehouse, or it can talk to logistic service to find out all the courier partners, or to fetch a list of pin codes that we can service, etc. Based on all this information, serviceability and TAT service will come up with the shortest route between the warehouse housing the product and the user and will also compute the ETA for delivery.





Overall High-Level Architecture:



Detail of Components:

CDN:

For optimizing latency, CDN such as **Cloudfront**, **Akamai,** or **Cloudflare** can be placed in front of static resources (Product images and videos) to provide a distributed network of nodes that cache your resources, serving user requests from the closest nodes, via the most optimal routes.

Load Balancer and API Gateway:

API gateway predominately does API management and provides various other key features such as

- IAM (Identity and Access Management),
- Rate limiting,
- circuit breakers.
- Hence, it mainly eliminates the need to implement API-specific code for functionalities such as security, caching, throttling, and monitoring for each of the microservice.

However, normally, the API Management does not include load balancing function, so it should be used in conjunction with a load balancer to achieve the same.

Databases:

Database for Product metadata storage:

Needs to be horizontally scalable and *highly available*; AP out of CAP.
High read/write ratio. Product attributes can diverge by product family, requiring a flexible structure.

Based on the above, Product metadata such as a number of units, name, description, localizations, and dimensions (excluding product media images/videos storage) can go into a document store that supports flexible query forms.

MongoDB is a document-oriented database that can be used for Product storage.
A MongoDB replica set ensures data redundancy and high availability by distributing data across many MongoDB servers.

To support text-based search **ElasticSearch** can be considered.

Product images and videos are relatively larger and would require a cost-efficient, highly-available store to persist those files. Cloud blob storage comes into play here, like **AWS S3**, **Google Cloud Storage**, or **Azure Blob Storage**. These services provide cheap, replicated, and hence HA storage for object storage.

Database for Inventory Storage:

Inventory Storage Manages availability of products, keeps track of warehouse locations, so the need is clear for *consistency* in how data is managed; CP out of CAP.

Inventory Sellable Items' data management requires ACID guarantees, which can be offered by an RDBMS like **Postgres**, **MySQL**, **Oracle**, or **MS SQL Server**.

Database for the Order Storage:

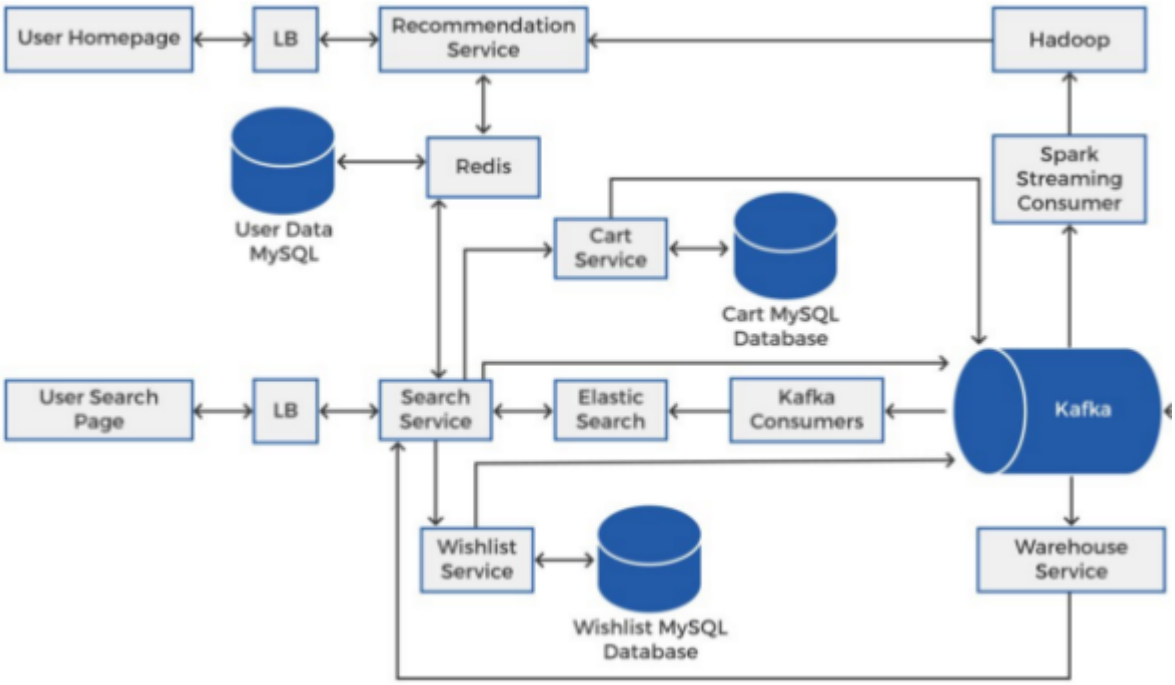
A submitted Order is an entity composed of different elements of data and references, i.e. relations. Orders carry state info, and data that would be required to handle, fulfill, pick, ship, or cancel the order.
Consistency is required, at least until the order is completed, reaching an end state (e.g. delivered or canceled); and hence, CP. As in the case of Inventory, an RDBMS would serve us very well for storing and managing orders.

Database for user information storage:

Customer and seller data like email hashed passwords and roles enjoy a well-defined structure, does not change often and is queried in well-defined ways. For scale and high availability, a NoSQL storage solution like **Cassandra** or **Google BigTable** can come in handy.

It also Maintains a **Redis** cache. So let's say our search service wants to fetch a user's pin code to communicate to the serviceability service, user service will first check in the Redis, if Redis doesn't have the information it will lookup in the MySQL database, get the user's information, store it in Redis for further use and return it to the search service.

Home and Search Flow:



There will be two UIs that we will offer, a home screen that will have some recommendations, personalized or general depending on if it is a returning user or a new user, and a search page where users can see results based on some search text.

One of the consumers in our system is an **OnBoard service** that will listen to Kafka to onboard new items and will expose APIs to add, update, and fetch items. It sits on **MongoDB** because this item-related data will be unstructured. Unstructured in the sense that various types of items will have different attributes. For example, a shirt will have size, fabric, and color attributes, whereas a TV would have attributes like screen size, color technology, weight, resolution, etc.

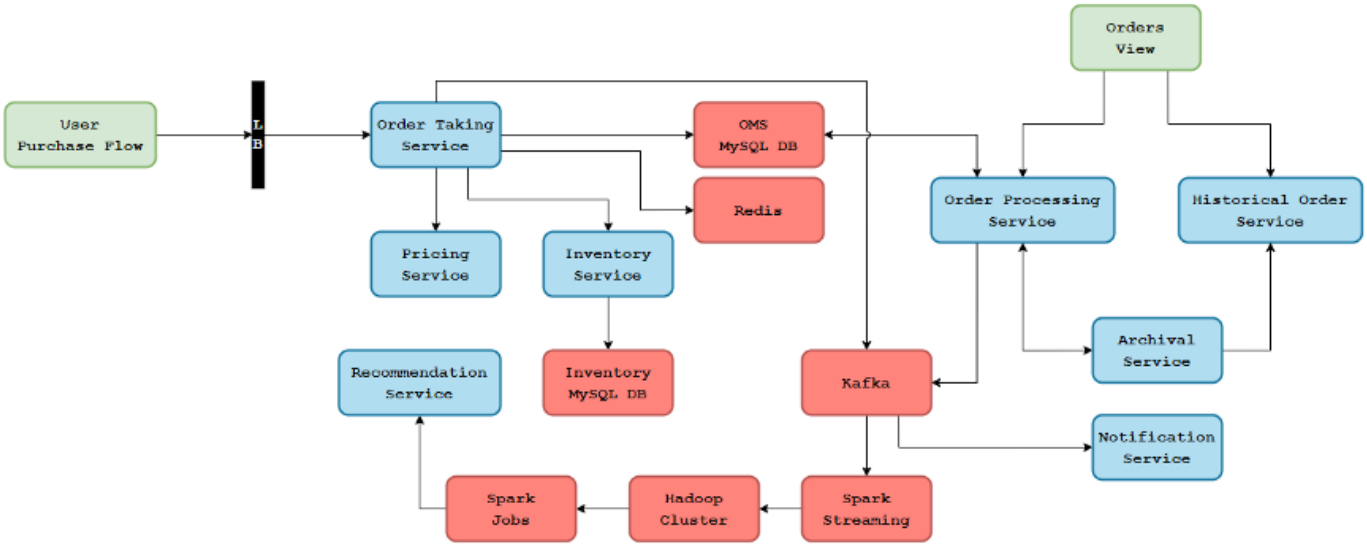
As soon as a new item is onboarded, a **search consumer** will make sure that this item can be queried by the users. It will read and process all the new items being onboarded and format them such that they can be stored in the database and the **search system** can understand them. After formatting, the search consumer will put this data in an **ElasticSearch** database. We use an ElasticSearch here as it is very efficient for text-based search and also supports fuzzy search which we need for a seamless user experience. Which have discussed this in more detail in another tutorial where we go over which databases might be best for what scenarios.

From the search screen, the user should be able to wishlist a product or add it to the cart. This happens via **Wishlist Service** and **Cart Service**. Wishlist service is a repo of all wish

lists in our system and cart service is a repo of all carts. Both these services will be built in the exact same way in that they each provide APIs to fetch, update, add to and delete items from a Wishlist or cart, and they will both sit on **MySQL DBs**.

Now, each time a search happens, an event is fed to **Kafka**. This helps us build a recommendation system personalized to the user's interests and also a general recommendation based on the most popular products. Similarly, Wishlist and cart services will also send similar events to Kafka.

Purchase and Checkout flow



When a user wants to place an order, the request will go to an **Order taking service**, which is a part of the larger **order management system**. The order management system sits on a **MySQL** database. As expected we have multiple tables like customer table, item table, order table, etc, and a lot of transactions going on across these tables. Now we don't want two users to be able to order the last piece of the latest AirPods in our warehouse, just because our database couldn't reflect the change promptly enough. This means we need ACID properties of relational databases, hence MySQL.

Now as soon as the order-taking service is called, a record will be created in a **Redis** with an order id, date, and time at which the order was created, and an expiry time for the order id. Along with these details, there will also be a status against that order id, let's say initially this status will be "created". Now the next step will be to call the **inventory service**. For instance, there were 5 sets of Sony 65" Smart TV in stock before the order was created. After

placing the order the inventory count for the product will be reduced to 4, only after this the user will be redirected to payments.

Now, once the inventory is successfully updated, the order-taking service will talk to the **payments service**, Now there can be three possible outcomes from this payment flow — success response, failure response, and no response.

Success Response:

Let's start with the happy flow. Say order O1 was placed at 10:01 with an expiry time of 10:05 and at 10:03 we got a payment success response. In this case, we will update the order status to "placed" and an event will be fired to Kafka saying that an order is placed against this product with so and so order details

Failed Response:

Remember before starting the payment flow we have already updated the inventory. This means as far as the inventory system is concerned the order has already been placed, and now that the payment has failed, we need to cancel this order. This means the order status will be updated to "canceled". Since the order is canceled we need to revert the count in the inventory database, so we will again talk to the inventory service.

No Response Flow:

Now suppose the user started the payment flow but without completing the payment closed the browser. In this case, there will be no response from the payment service. That means we still have the Sony 65" Smart TV in our stock but it isn't reflected in our inventory system. Now, this is where the Redis comes in. At 10:05 the Redis record for O1 will get expired and will implement an expiry callback. At this point, the order-taking service will talk to the payment service to time out the payment and mark it as failed. From here out the flow will be the same as that in case of a failed payment case, the order will be canceled and the inventory count will be rolled back.

How Amazon handles concurrency write issues like when multiple users try to buy the same product whose available quantity is only one.

Considering the "inventory" table has below fields :

ID	NAME	PRICE	QUANTITY	STATUS
1	AMAZON ALEXA	10000	1	AVAILABLE

Let's say when 2 users have added the same product to their cart and one user proceeds to PAY :

```
BEGIN TRANSACTION

SELECT * FROM PRODUCT WHERE ID = 1 AND STATUS = "AVAILABLE" FOR UPDATE;

update the quantity field to "quantity — 1" and (if quantity = 0 then
status = "SOLD OUT") in product table

Call the payment gateway

If PAYMENT TIMEOUT ROLLBACK;

COMMIT

END TRANSACTION
```

1) While still above transaction is going on, other users can still see the PRODUCT available as it is still not committed and they may also proceed to PAY.

2) Also, if a sufficient quantity of product is available, taking a lock on the "product" table row may stop other users from trying to buy the same product at the same time.

How to handle these issues 🤔 ??

There are 2 broad ways to handle these situations:

1. Two/ Three-phase Commit
2. SAGA.

There are pros and cons to both approaches. However, **SAGA is more accepted** in Distributed Systems world because it is faster than 2PC.
Let's go through each approach.

👉**2-Phase Commit**.

In this approach, there would be a coordinator which coordinates the entire transaction. which will ask different microservices to carry out transactions.
Say we first secure the payment and then confirm buying of the product. 2PC works in 2 phases, as the name suggests:

1. Prepare
2. Commit

Prepare:
The coordinator asks Service1(Payments Service) if this is a valid payment? Payment Service answers -YES.
Then We ask Service2(Order Service) if this product is available? Answers — YES or NO
If the answer is YES, the service writes a READY log in the database.

Commit
Once we get back both YES, we actually commit both transactions, for the order and the payment, if we get one NO, we Abort the transaction.

Problems with this:

1. It is slow since we are locking in multiple databases at the same time.
2. The coordinator is SPOF.

👉We need something called **3 phase commit** to tackling this issue, where we divide the commit phase into 2 further steps:

1. PRepare to commit: In this phase, it acquires the necessary locks.
2. Commit: Actually commit.

So if the coordinator fails, we can still find out if the prepare to commit phase was carried out or not. If yes, then GO AHEAD, else ABORT.

Problems with 3PC:
It is still not very fast.

👉**SAGA:**
SAGA Pattern relies heavily on queues, and it provides eventual consistency.
So even if your payment goes through, if the product is not available, then we will not CONFIRM the order.

The way it works is as follows:

We ask the payments service if the payment is valid-> if YES, put the order in a QUEUE(call it **OrderProcessQueue**) for further processing-> Send back 200 to the User saying Order has been placed and we are waiting for confirmation.

Now the **Order Service** listens to the **OrderProcessQueue**, and tries to place the order for the product. If the product is available, we use another queue and WebSockets to send this information to the user.

If the product is not available, we send the order to an **OrderCancelQueue**.
Now the payments service will listen to the OrderCancelQueue, undo the payment(refund, or if using an API, ask for a refund.), and then use another queue and WebSockets to inform the user.

Problem with this model: There are some inconsistencies as the user's money is debited even if the product is not confirmed.

Pros: This is much faster than 2pc and 3pc.

How to handle peak traffic load during Black Friday Sales?

As we have captured in the first diagram, we are using a load balancer to load balance across different replicas. We can use AWS-provided Elastic Load Balancer (ELB) there.

If we provision enough capacity to always handle your peak traffic load, Black Friday, for example, we are wasting money. It would be better to match compute power with demand.

That's what **Auto Scaling** lets you do, the automatic resizing of computing clusters. We have something in AWS called Auto Scaling which can serve this purpose for us.

👉Auto Scaling & ELB

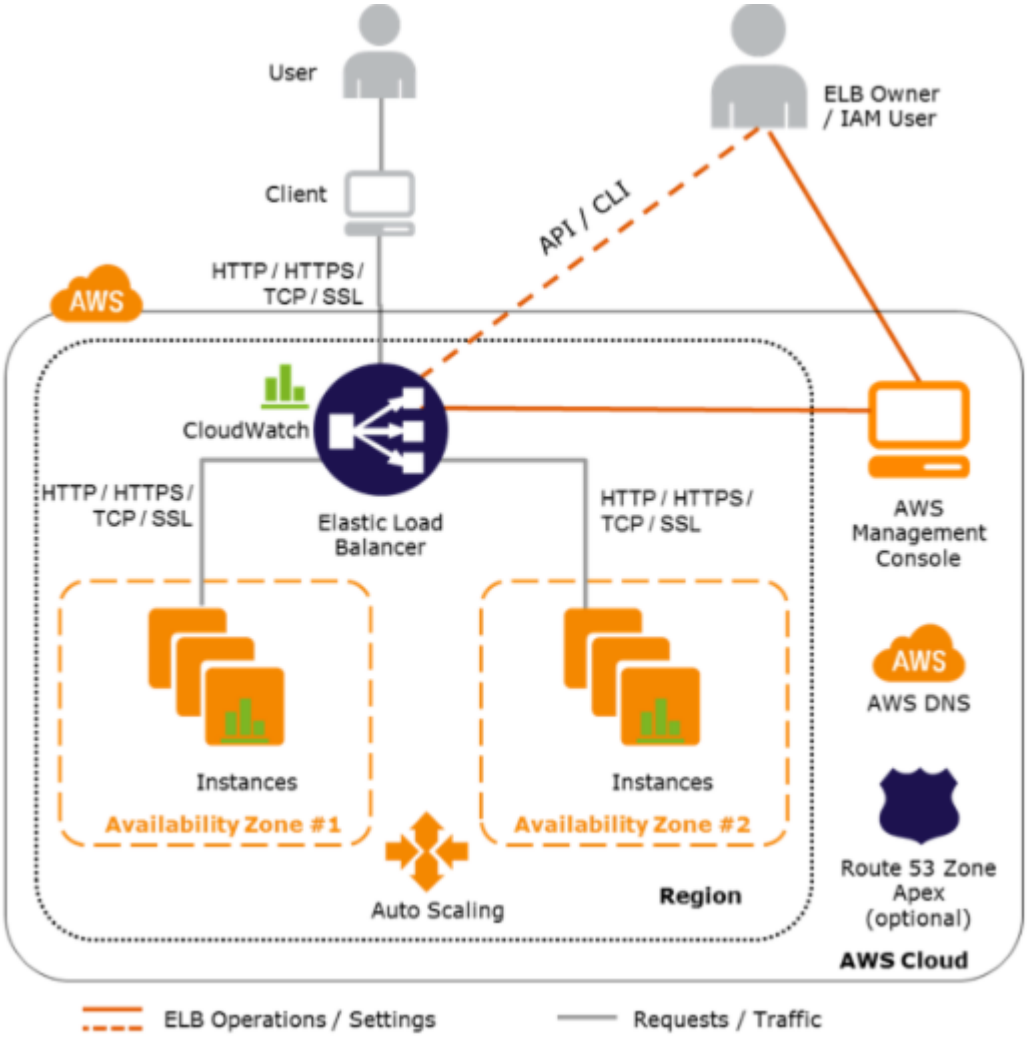
- The elastic load balancer can be set up to distribute incoming requests across EC2 instances in a single AZ or multiple AZs within a region.
- ELB acts as a single point of contact for all incoming traffic to the instances in an Auto Scaling group.
- Auto Scaling dynamically adds and removes EC2 instances, while Elastic Load Balancing manages incoming requests by optimally routing traffic so that no one instance is overwhelmed
- Auto Scaling helps to automatically increase the number of EC2 instances when the user demand goes up, and decrease the number of EC2 instances when demand goes down
- ELB helps to monitor traffic and handle requests that come through the Internet.
- Auto Scaling can span across multiple AZs, within the same region.
- When one AZ becomes unhealthy or unavailable, Auto Scaling launches new instances in an unaffected AZ.
- When the unhealthy AZ recovers, Auto Scaling redistributes the traffic across all the healthy AZ.
- Using Auto Scaling & ELB by **spanning Auto Scaling groups across multiple AZs within a region and then setting up ELB to distribute incoming traffic across those AZs helps take advantage of the safety and reliability of geographic redundancy.**
- Incoming traffic is load balanced equally across all the AZs enabled for ELB.
- Elastic Load Balancing sends data about the performance of your resources to CloudWatch.

- After registering one or more load balancers with the Auto Scaling group, the Auto Scaling group can be configured to use ELB metrics (such as request latency or request count) to scale the application automatically.

We can define the minimum and maximum size of your pools. As a user we decide what's the smallest number of instances in your cluster and the largest number of instances.

CloudWatch is a management service that's embedded into all applications.

- **CloudWatch events drive scaling.**
- Are you going to scale on CPU utilization? Are you going to scale on latency? On network traffic?
- You can also push your own custom metrics into CloudWatch. If you want to scale on something application-specific you can push that metric into CloudWatch and then tell Auto Scaling you want to scale on that metric.


User
Client
ELB Owner
/ IAM User
API / CLI
HTTP / HTTPS /
TCP / SSL
AWS
CloudWatch
HTTP / HTTPS /
TCP / SSL
Elastic Load
Balancer
HTTP / HTTPS /
TCP / SSL
AWS
Management
Console
AWS
AWS DNS
Instances
Availability Zone #1
Instances
Availability Zone #2
Auto Scaling
Region
Route 53 Zone
Apex
(optional)
AWS Cloud
ELB Operations / Settings
Requests / Traffic