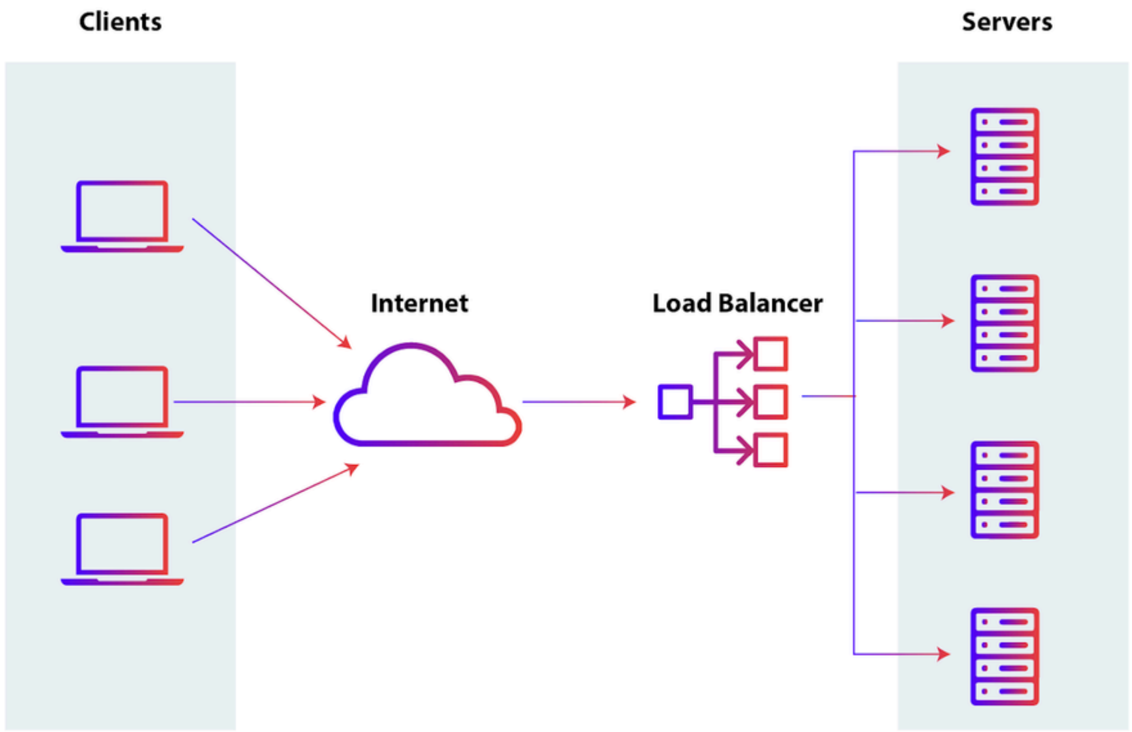
Load Balancer

🎯 **What is a Load Balancer?**

A **load balancer** is like a traffic cop for your application. It sits between users and your servers, ensuring that no single server is overwhelmed by requests.



🚦 **Function**:

- It receives incoming traffic and distributes it across multiple servers.
- This keeps things **smooth, fast, and available**.

✨ **Why is it important?**

1. **Improved Performance** – By balancing the load, it reduces response time and latency.
2. **High Availability** – If a server goes down, traffic is rerouted to healthy ones.
3. **Scalability** – You can add or remove servers without downtime.

🛠️ **How does it work?**

- **Health Checks**: Constantly monitors server health.
- **Algorithms**: Uses strategies like **Round Robin** or **Least Connections** to decide which server handles the next request.
- **Failover**: If one fails, traffic is redirected automatically.

🧩 **Types of Load Balancers**:

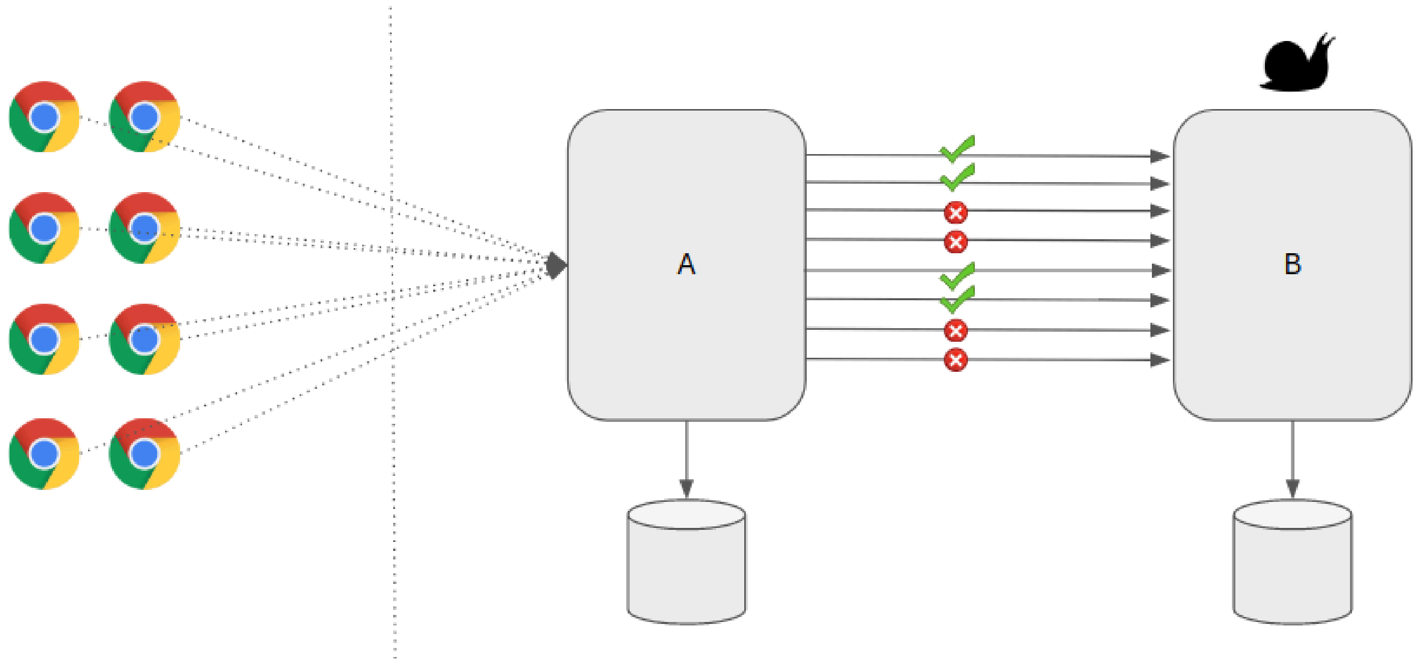
- **Hardware Load Balancer** – Physical device, handles massive traffic.
- **Software Load Balancer** – Lightweight, flexible, runs on VMs or servers.
- **Application Load Balancer** – Works at HTTP/HTTPS level.
- **Network Load Balancer** – Manages TCP/UDP traffic.
- **Database Load Balancer** – Routes traffic between DB replicas.

Rate Limiter

🎯 **What is a Rate Limiter?**

A rate limiter is a **mechanism that controls the frequency of requests or actions a user or client can make within a specific time period**, protecting systems from overload and abuse. It does this by limiting the number of requests allowed within a certain timeframe, preventing malicious actors from overwhelming the system

A **Rate Limiter** is like a security guard for your APIs.
It controls **how many requests** a user or client can make in a given time — helping your application stay **safe, stable, and fair**.



🛡️ **Why do we need it?**

- To **protect from abuse** or spam.
- To **prevent server overload**.
- To **ensure fair usage** for all users.
- And to help defend against **DoS attacks**.

⏱️ **How does it work?**

It tracks requests and applies limits like:

- **100 requests per minute** per user.
  If the limit is exceeded, it can:
  🚫 Block the request,
  ⏳ Ask the user to retry later,
  or return a **429 Too Many Requests** error.

📊 **Common Algorithms**:

1. **Fixed Window**:
   Limits per fixed time period (e.g., 100/minute).
   🧠 Simple, but not super accurate at boundaries.
2. **Sliding Window**:
   Counts requests over a rolling window (e.g., past 60 seconds).
   ⚖️ More accurate and smooth.
3. **Token Bucket**:
   Tokens are added at a fixed rate; a request consumes a token.
   Great for **burst traffic**.
4. **Leaky Bucket**:
   Like a queue — requests are processed at a fixed rate.
   🎯 Good for **smoothing spikes**.

📦 **Where is it used?**

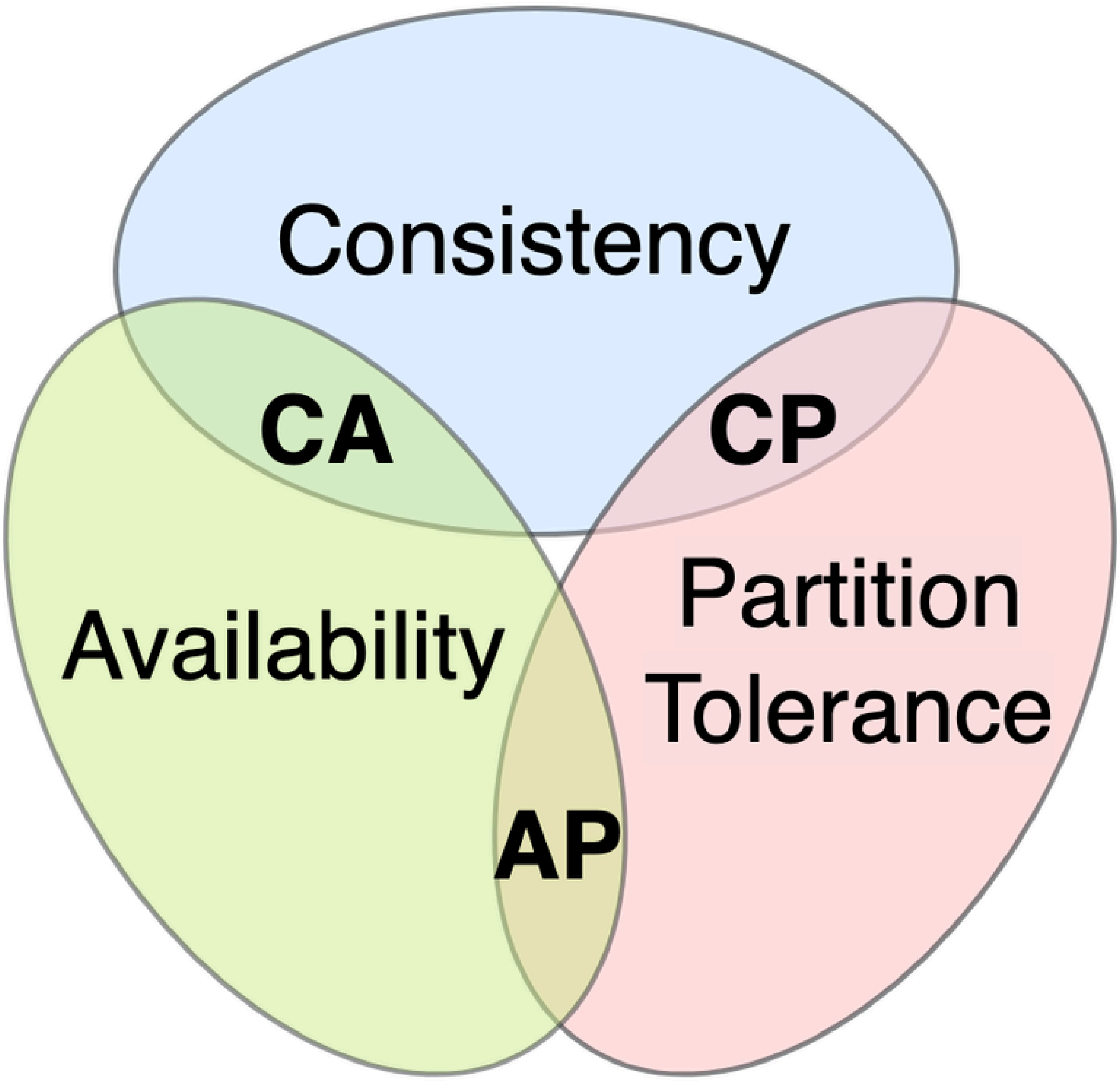
- APIs
- Login endpoints
- Payment gateways
- And anywhere user actions need throttling.

🧠 **TL;DR**:
Rate limiting ensures **performance, security**, and **fairness** by controlling the rate of incoming requests.

CAP Theorem

🎯 **What is the CAP Theorem?**

In distributed systems, the **CAP Theorem** says:
You can only guarantee **two** out of **three**:



🔹 **C – Consistency**
All nodes see the **same data at the same time**.
Like reading the latest update — always.

🔹 **A – Availability**
The system is always **responsive** — even if some nodes fail.

🔹 **P – Partition Tolerance**
The system **keeps working** even when there's a **network failure** between nodes.

💥 But during a network partition, you must choose:

- **C + P** 👉 Consistency over availability
- **A + P** 👉 Availability over consistency

You **can't** have all **three** — that's the catch.

🧠 **TL;DR**
In distributed systems:
👉 You get **two out of three** — **Consistency**, **Availability**, and **Partition Tolerance**.

Choose wisely based on your app needs! ⚖️

Eventual Consistency

🎯 **What is Eventual Consistency?**

In distributed systems, **Eventual Consistency** means:
All nodes **will** have the same data... **eventually** — but **not instantly**.

🧵 Imagine you're updating your Instagram bio...
One friend sees the new bio instantly,
Another sees it after a few seconds.

That's **eventual consistency** in action!

🧠 Why?
Because syncing across servers **takes time** — especially if they're far apart or there's network delay.

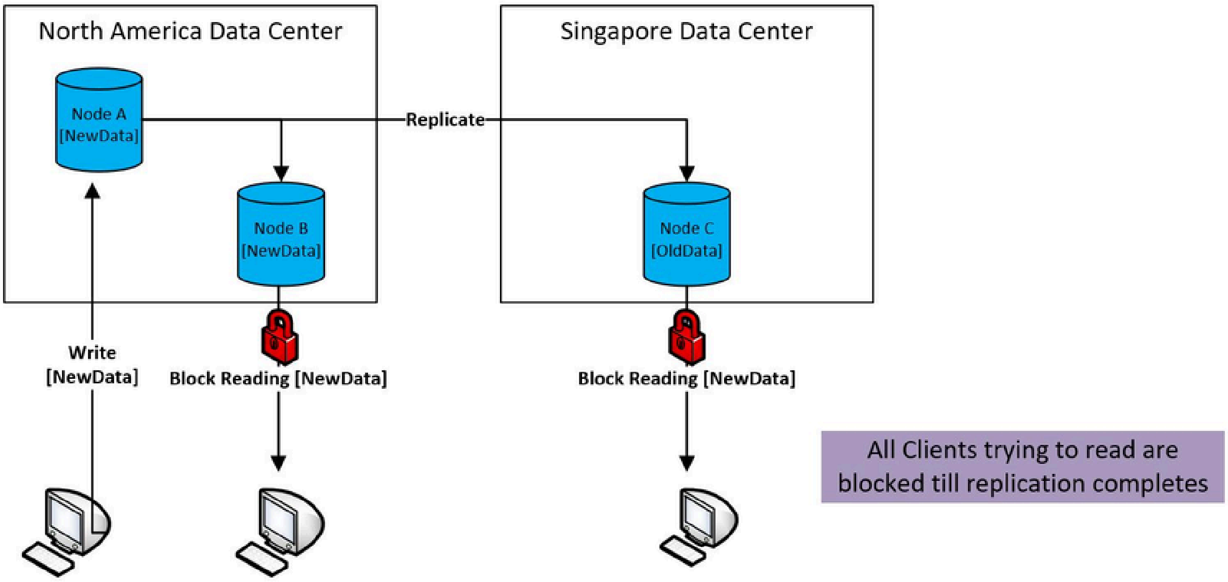
⚖️ It trades **strong consistency** for:

- **High availability**
- **Better performance**
- **Partition tolerance**

📦 Used in:

- NoSQL databases like **DynamoDB**, **Cassandra**, and **MongoDB**.

🧠 **TL;DR**:
Eventual Consistency = Sooner or later, all nodes agree — and that's okay for many real-world apps.



Webhooks vs Polling

🎯 **Webhooks vs Polling** — what's the difference?

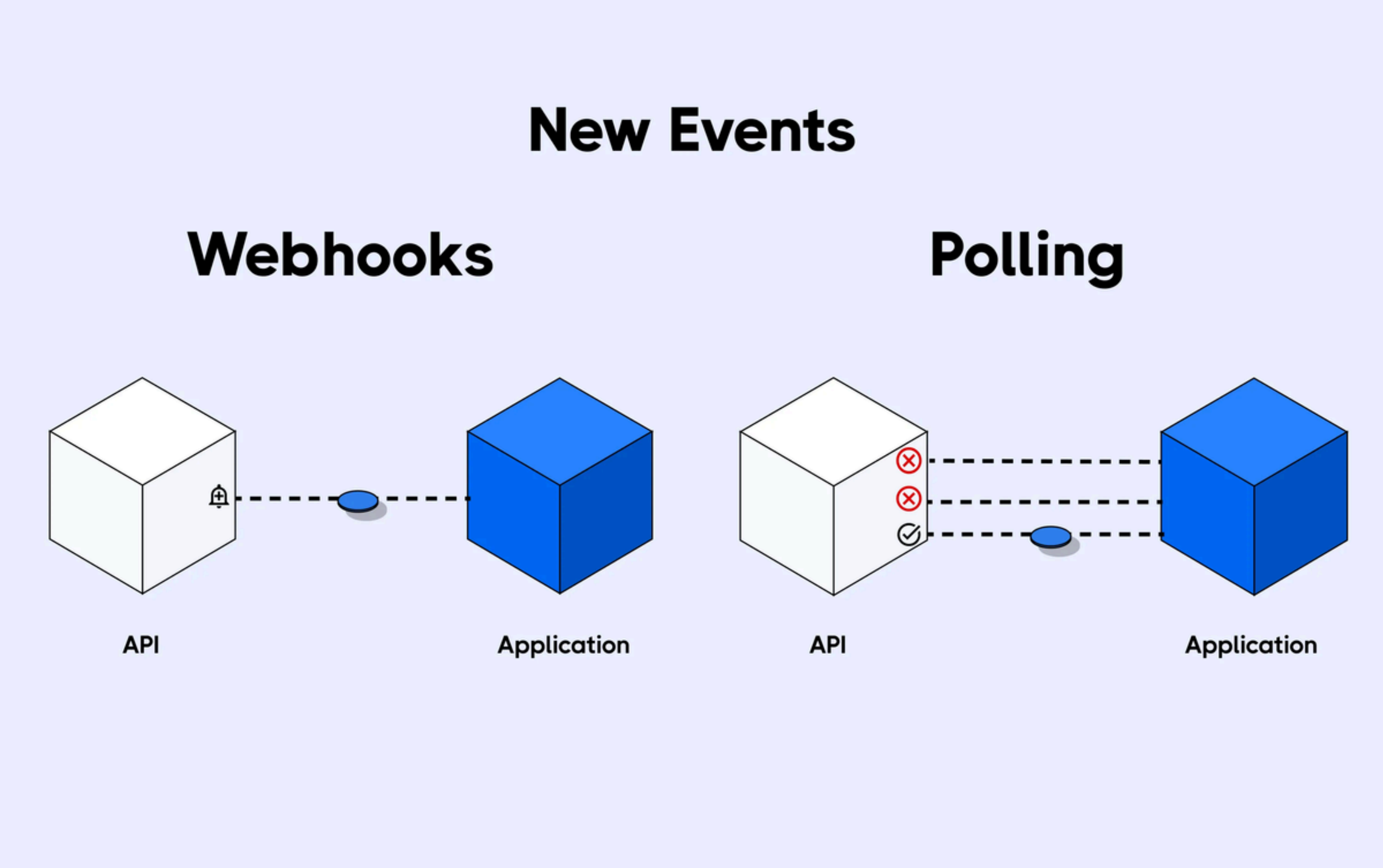
📬 **Polling** is like **you calling the pizza shop every 5 mins**:
"Is my pizza ready?"
"Is it ready now?"
Annoying, right? 😅
That's what your app does — keeps asking the server if there's new data.

📦 **Webhook** is smarter!
You order the pizza... and when it's ready, the shop **calls YOU**.
No extra effort, just chill and wait. 😌
That's **Webhook** — the server **pushes data** to your app when there's something new.

🧠 **TL;DR**

- **Polling** = You keep asking
- **Webhook** = You get notified when it matters

💡 Webhooks are **real-time & efficient**
Polling is **easy to implement**, but can waste resources.



Idempotency

🎯 **What is Idempotency?**

Imagine pressing the **"Buy Now"** button on an app... and your internet lags 🐢
So you click it **again**... and again... 😅

Without idempotency — you might get charged **3 times!** 💸💸💸

💡 But with **Idempotency**, no matter how many times you make the same request...
👉 the result is the **same** — **just one order**, **just one payment**.

🛡️ It's a **safety net** for APIs — especially for **POST**, **PUT**, or **DELETE** requests.

📦 Real-life example:
Payment gateways like **Stripe** use an idempotency-key to avoid duplicate charges.

🧠 **TL;DR**
Idempotent = Same request, same result — no matter how many times you send it.

Click it once or a hundred times — no duplicates, no drama! 💥

Cache

🎯 **What is a Cache?**

A **cache** is like a **memory shortcut** for your app 🧠✨

Instead of fetching the same data again and again...
👉 It stores a **copy** nearby — so the next time, it's **super fast**! ⚡

📦 Example:
When you open Instagram, your feed loads instantly — that's because recent posts are **cached** on your device!

💡 Benefits:

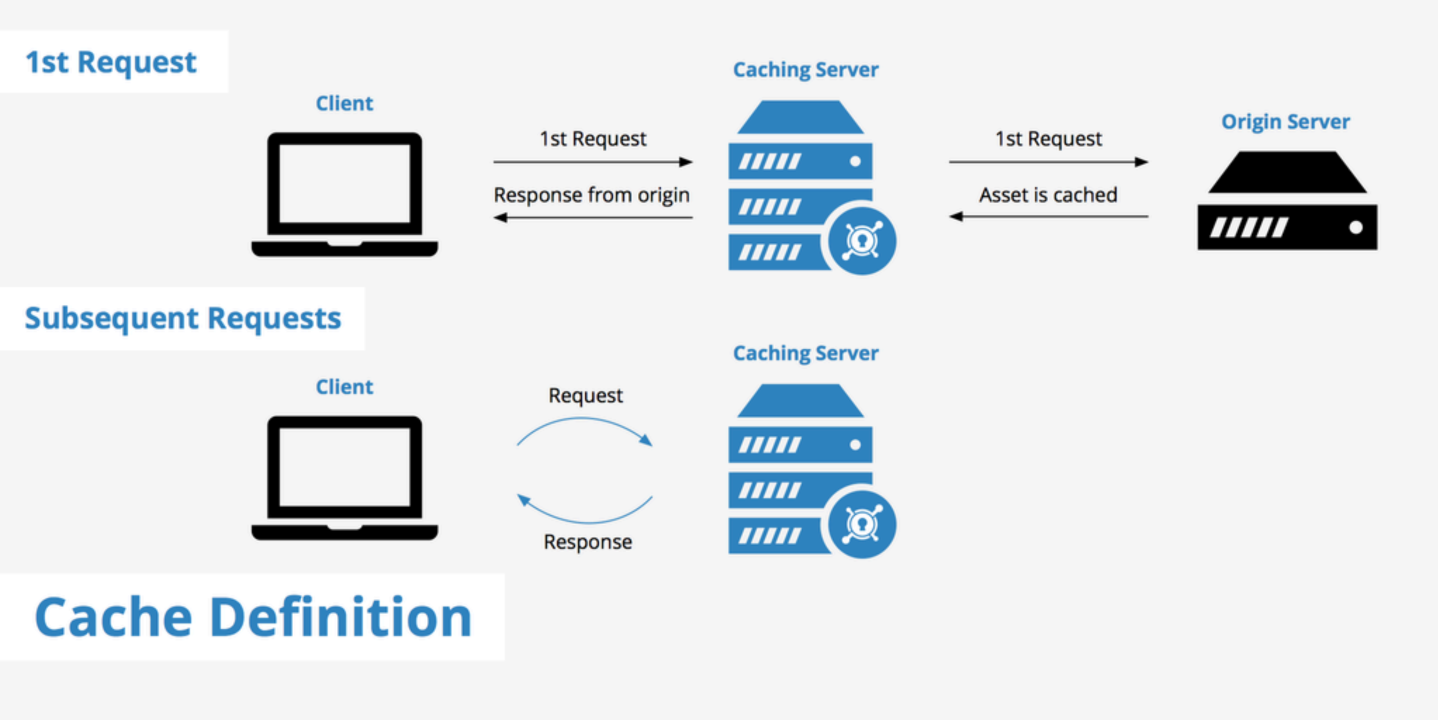
- 🔥 **Faster response time**
- 📉 **Reduces server load**
- 🌐 **Saves bandwidth**

🧠 **Types of Cache**:

- **Browser Cache** – Stores static assets (images, CSS)
- **In-memory Cache** – Like **Redis**, stores frequently used data
- **CDN Cache** – Stores content closer to users

⚠ But remember:
Cached data can become **stale** — so we need **expiry** or **invalidation** logic!

🧠 **TL;DR**
Cache = Fast access to repeated data 🚀
Use it wisely, and your app feels ✨instant✨!


1st Request
Client
Caching Server
1st Request
Response from origin
1st Request
Asset is cached
Origin Server
Subsequent Requests
Client
Caching Server
Request
Response
Cache Definition