

# DNS in System Design

what is dns in system design?

explain this in details so that i am able to understand this well and do not have doubt in this.

Great question, Piyush 👍

DNS is one of those “invisible but critical” pieces of system design. If you truly understand DNS, a lot of backend + distributed system concepts will click automatically.

I'll explain it from absolute basics → real system design usage → scaling, performance, and failures, so you won't have doubts later.

## 1 What is DNS? (Very simple definition)

DNS (Domain Name System) is like the phonebook of the internet.

- Humans remember names → google.com
- Computers understand IP addresses → 142.250.195.14

👉 DNS converts a domain name into an IP address

google.com → 142.250.195.14

Without DNS, you would have to remember IP addresses for every website 😱

## 2 Why DNS is important in System Design

In system design, DNS helps with:

- 🌎 Service discovery
- ⚖️ Load balancing
- 🚀 High availability
- 🛡️ Traffic routing (geo-based, failover)
- 🔒 Security & isolation

Almost every distributed system starts with DNS resolution.

### 3 Real-life analogy (Best way to remember)

Imagine you want to call “Pizza Shop”:

- You ask a phone directory → “What is Pizza Shop’s number?”
  - Directory gives you the number
  - You call that number
- DNS = Phone directory
  - Domain name = Shop name
  - IP address = Phone number

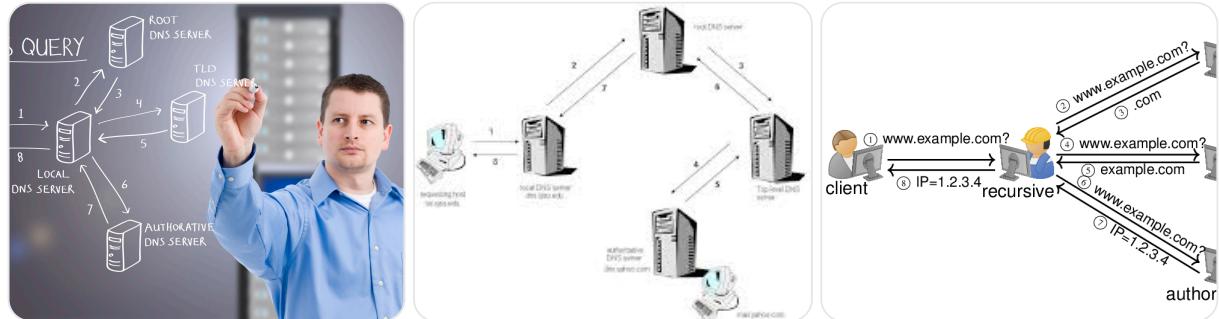
### 4 How DNS works (Step-by-step flow)

Let's say you open:

arduino

<https://www.amazon.com>

#### DNS Resolution Flow



#### Step 1: Browser cache

- Browser checks:  
“Do I already know amazon.com’s IP?”
- If yes → done ✓

## Step 2: OS cache

- OS (Windows/Linux) checks its DNS cache
- 

## Step 3: Recursive DNS Resolver (ISP / Public DNS)

If not found locally:

- Request goes to **Recursive Resolver**
    - Example:
      - Google DNS → 8.8.8.8
      - Cloudflare → 1.1.1.1
- 

## Step 4: Root DNS Server

Resolver asks:

| "Who knows about .com domains?"

Root server replies:

| "Ask the .com TLD server"

---

## Step 5: TLD (Top-Level Domain) Server

TLD server says:

| "Ask Amazon's authoritative DNS server"

---

## Step 6: Authoritative DNS Server

Authoritative server replies:

| "amazon.com → 176.32.103.205"

---

## Step 7: Response back to browser

- IP address returned

- Browser sends HTTP request to that IP 
- 

## 5 DNS hierarchy (Very important for interviews)

DNS is **hierarchical** (tree structure):

mathematica



**Levels:**

1. Root DNS servers
2. TLD servers (.com, .org, .in)
3. Authoritative servers

 This hierarchy allows DNS to **scale to billions of domains**

---

## 6 DNS Records (Must-know)

DNS doesn't store just IPs. It stores **records**.

Record	Purpose
A	Domain → IPv4 address
AAAA	Domain → IPv6 address
CNAME	Alias to another domain
MX	Mail servers
NS	Name servers

Record	Purpose
TXT	Verification, security (SPF, DKIM)

### Example:

CSS

```
api.myapp.com → CNAME → lb.myapp.com
lb.myapp.com → A → 13.234.56.78
```

## 7 DNS in System Design (Real-world usage)

Now the important part 

- ◆ 1. Load Balancing using DNS

DNS can return multiple IPs:

CSS

```
myapp.com → [IP1, IP2, IP3]
```

Browser picks one → distributes traffic

- ✓ Simple
- ✗ No real-time health check

- ◆ 2. Geo-based routing

DNS can return different IPs based on location:

User Location	DNS Response
India	India server IP
USA	US server IP

Used by:

- Netflix
  - Amazon
  - Cloudflare
- 

### ◆ 3. Failover & High Availability

If primary server goes down:

- DNS switches to backup IP

⚠ Controlled using TTL

---

## 8 TTL (Time To Live) – very important concept

TTL = How long DNS response is cached

Example:

ini

TTL = 300 seconds (5 minutes)

Impact:

- ⬇ Low TTL → Faster failover, more DNS traffic
- ⬆ High TTL → Better performance, slower changes

👉 In system design, TTL is a trade-off between availability & performance

---

## 9 DNS Caching (Why DNS is fast)

DNS uses caching at multiple levels:

1. Browser cache
2. OS cache
3. Resolver cache
4. CDN cache

That's why DNS lookup usually takes milliseconds.

---

## 10 DNS vs Service Discovery (System Design insight)

In microservices:

DNS	Service Discovery
External users	Internal services
Static-ish	Dynamic
Domain → IP	Service → instance

Example:

```
pgsql  
  
payment-service.default.svc.cluster.local
```

(Kubernetes uses DNS internally 🤖)

## 1 1 DNS Security (Brief but important)

Attacks:

- DNS spoofing
- Cache poisoning

Solutions:

- DNSSEC
- HTTPS + TLS
- Short TTLs

## 1 2 Where DNS fits in a system design interview

When asked:

“Design a scalable web application”

You should say:

1. DNS resolves domain
2. DNS points to Load Balancer

3. Load Balancer routes traffic
  4. Backend services handle request
- Mentioning DNS shows **strong fundamentals**
- 

## 1 3 One-line summary (remember this)

DNS is a globally distributed, hierarchical system that translates human-readable domain names into IP addresses and plays a critical role in scalability, availability, and traffic routing in system design.

---

## 1 4 Want to go deeper next?

If you want, I can explain:

- 🔥 DNS vs Load Balancer (when to use which)
- 🔥 DNS in Kubernetes (CoreDNS)
- 🔥 How AWS Route 53 works internally
- 🔥 DNS in payment systems & microservices

Just tell me what you want next 😊

what is vertical scaling and horizontal scaling system in system design?

explain this in details so that i am able to understand the well and do not have doubt in this.

Perfect topic to ask at this stage, Piyush 👍

**Vertical vs Horizontal Scaling** is a **CORE system-design concept** — if this is crystal clear, half of system-design interviews become easier.

I'll go **slow, visual, and practical**, with real-world examples, trade-offs, and interview angles so you don't have doubts later.

---



# Scaling in System Design (Big Picture)

**Scaling** = Ability of a system to handle more load (users, requests, data).

There are two fundamental ways to scale any system:

1. Vertical Scaling (Scale Up / Down)
  2. Horizontal Scaling (Scale Out / In)
- 

## 1 Vertical Scaling (Scale Up)

### ◆ What is Vertical Scaling?

Vertical Scaling means increasing the power of a single machine.

You keep one server, but you make it stronger:

- More CPU
- More RAM
- Faster Disk
- Better Network

### 📌 Example:

makefile

From: 4 CPU, 8 GB RAM  
To: 16 CPU, 64 GB RAM

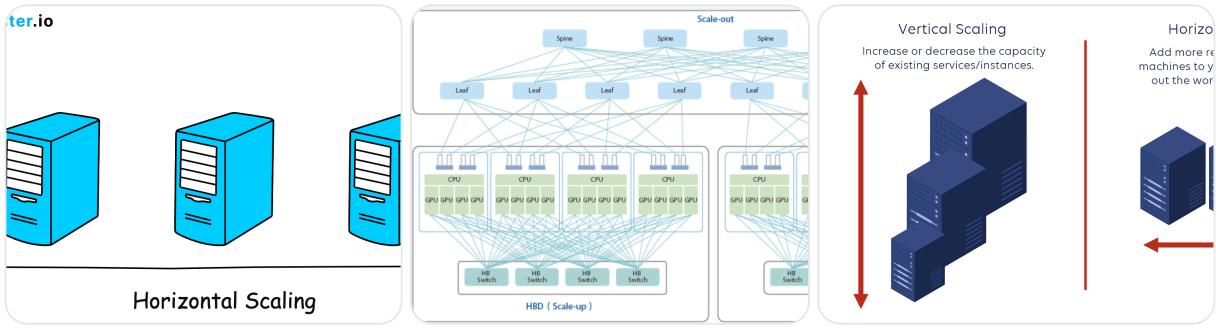
### ◆ Real-life analogy

Think of one worker doing work:

- Vertical scaling = Give that worker better tools
  - Faster laptop
  - More memory
  - Bigger screen

Same worker, stronger capabilities.

## ◆ Diagram (Vertical Scaling)



### ◆ Where Vertical Scaling is used

- Databases (MySQL, PostgreSQL)
- Legacy systems
- Early-stage startups
- Monolithic applications

Example:

rust

One MySQL server

- ↑ Increase RAM **for caching**
- ↑ Increase CPU **for queries**

### ◆ Advantages of Vertical Scaling

- Simple to implement
- No code changes
- No distributed system complexity
- Easier debugging

### ◆ Disadvantages of Vertical Scaling

- Hardware limit** (cannot scale forever)
- Single Point of Failure (SPOF)**
- Downtime during upgrades
- Expensive high-end machines

## ◆ Interview insight

"Vertical scaling is quick but not future-proof."

## 2 Horizontal Scaling (Scale Out)

### ◆ What is Horizontal Scaling?

Horizontal Scaling means adding more machines instead of making one machine bigger.

You distribute the load across **multiple servers**.

📌 Example:

pgsql

1 server → 5 servers → 100 servers

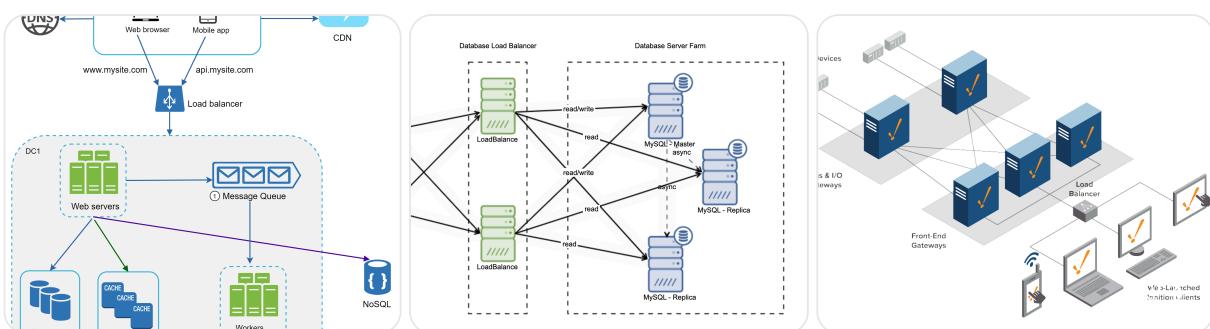
### ◆ Real-life analogy

Instead of one super-strong worker:

- Hire multiple workers
- Divide tasks among them

More people → more work done.

### ◆ Diagram (Horizontal Scaling)



## ◆ Where Horizontal Scaling is used

- Web servers
- Microservices
- Cloud-native apps
- High-traffic systems (Netflix, Amazon)

Example:

pgsql

User → DNS → Load Balancer → Server 1 / 2 / 3

## ◆ Advantages of Horizontal Scaling

- ✓ No single point of failure
- ✓ Scales almost infinitely
- ✓ High availability
- ✓ Fault tolerance
- ✓ Cost-effective (commodity hardware)

## ◆ Disadvantages of Horizontal Scaling

- ✗ More complex architecture
- ✗ Requires load balancing
- ✗ Data consistency issues
- ✗ Harder debugging
- ✗ Requires stateless design

## ◆ Interview insight

"Horizontal scaling is complex but scalable and resilient."

## 3 Vertical vs Horizontal Scaling (Side-by-Side)

Feature	Vertical Scaling	Horizontal Scaling
Scaling method	Bigger machine	More machines
Complexity	Low	High
Cost	Expensive at high end	Cheaper per node
SPOF	Yes	No
Downtime	Possible	Usually none
Scalability	Limited	Practically unlimited
Cloud friendly	✗	✓

## 4 Stateless vs Stateful (Very Important)

- ◆ **Vertical scaling works with stateful apps**

- Session stored in memory
- Single server handles everything

- ◆ **Horizontal scaling requires stateless apps**

❗ Why?

- Requests can go to **any server**

### Solutions:

- Store session in Redis
- Use JWT tokens
- Centralized database

## 5 Real System Example (Step-by-step)

### ● Early stage startup

1 server

Vertical scaling

### 🟡 Growing users

nginx

Bigger server + DB tuning

### 🔴 High traffic (millions of users)

pgsql

DNS

- Load Balancer
- Multiple App Servers
- Distributed Cache (Redis)
- Database Cluster

### ➡️ Mostly Horizontal Scaling

## 6 Scaling Databases (Tricky part)

### Vertical Scaling DB

- Increase RAM (cache)
- Increase CPU

### Horizontal Scaling DB

- Read replicas
- Sharding
- Partitioning

✖️ Databases often use both.

## 7 Hybrid Scaling (Real world)

Most real systems use both:

```
bash
```

Vertical scaling → until limit

Then → Horizontal scaling

Example:

- Scale DB vertically
- Scale API servers horizontally

---

## 8 Common Interview Questions & Answers

### ? Why not always vertical scaling?

| Because hardware has limits and creates SPOF.

### ? Why is horizontal scaling preferred?

| It improves availability, fault tolerance, and long-term scalability.

### ? Can we horizontally scale everything?

| No. Stateful systems like DBs need careful design.

---

## 9 One-line summaries (Remember these 🔥)

- Vertical Scaling → “Make one server stronger”
- Horizontal Scaling → “Add more servers”

---

## 10 Final mental model (No confusion)

```
mathematica
```

Vertical Scaling = Power ↑

Horizontal Scaling = Count ↑

If you want, next we can go deeper into:

- 🔥 Horizontal scaling in microservices
- 🔥 Load balancers (L4 vs L7)
- 🔥 Auto-scaling in AWS
- 🔥 Scaling databases (sharding vs replication)

Just say the word 😊

what is api gateways in system design?

explain this in details so that i am able to understand the well and do not have doubt in this.

Awesome question, Piyush 🌟

**API Gateway** is a **MUST-KNOW** concept in modern system design, especially if you're dealing with **microservices, Spring Boot, cloud, or payments**.

I'll explain this **from zero → real-world architecture → interview mindset**, so you won't have confusion later.

## 💡 What is an API Gateway in System Design?

### 👉 Simple definition

An API Gateway is a **single entry point** that sits **between clients (frontend/mobile) and backend services (microservices)**.

Instead of the client calling **many services directly**, it calls **one gateway**, and the gateway handles everything else.

arduino

**Client → API Gateway → Multiple Backend Services**

### 🗣 Real-life analogy (Very important)

Think of a hotel receptionist 🏨

- Guests don't talk directly to:
  - Housekeeping
  - Kitchen
  - Maintenance
- They talk to **one person (receptionist)**

The receptionist:

- Authenticates you
  - Routes request to correct department
  - Handles rules & policies
- API Gateway = Receptionist  
● Microservices = Departments

## ✖ Why API Gateway exists (The core problem)

### ✗ Without API Gateway (Bad design)

sql

```
Frontend
├── User Service
├── Payment Service
├── Order Service
└── Notification Service
```

Problems:

- Frontend must know **all service URLs**
- Authentication repeated in every service
- No centralized security
- Hard to scale
- Tight coupling

## ✓ With API Gateway (Good design)

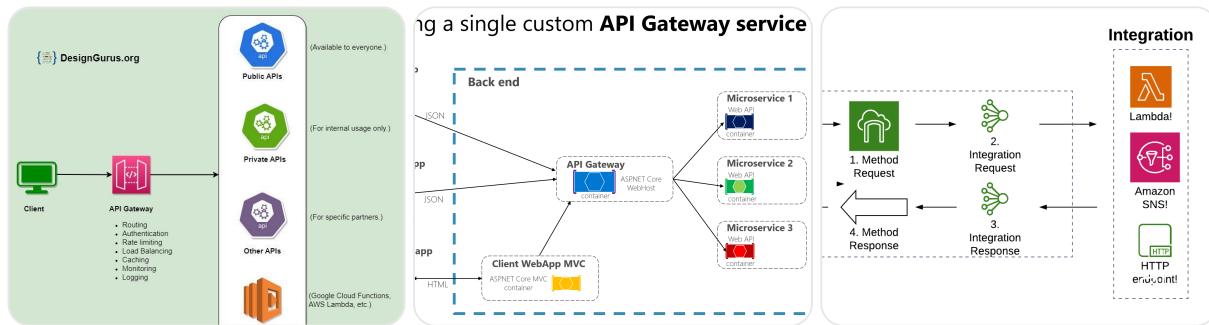
sql



Benefits:

- Single endpoint
- Centralized control
- Cleaner frontend
- Easier scaling

## High-level Architecture Diagram



## What does an API Gateway do? (Core Responsibilities)

An API Gateway is NOT just routing. It does a LOT.

### 1 Request Routing

Gateway decides which service should handle the request.

bash

```

/api/users      → User Service
/api/payments  → Payment Service
/api/orders     → Order Service
  
```

Frontend doesn't care *where* services live.

## 2 Authentication & Authorization

Instead of every service handling auth:

- API Gateway:
  - Validates JWT
  - OAuth2
  - API keys
- Services trust the gateway

arduino

Client → **Gateway** (Auth) → Backend

Centralized security

Less duplicate code

## 3 Rate Limiting & Throttling

Prevents abuse and DDoS.

Examples:

sql

100 requests / minute / user

1000 requests / second / IP

If limit exceeded →  429 Too Many Requests

## 4 Load Balancing

Gateway distributes traffic across instances:

nginx

Payment Service

|— Instance 1

```
└─ Instance 2  
└─ Instance 3
```

This enables **horizontal scaling**.

---

## 5 Request / Response Transformation

Gateway can:

- Modify headers
- Convert formats
- Aggregate responses

Example:

```
sql
```

```
Client wants → User + Order + Payment data  
Gateway calls 3 services → Combines response
```

This is called **API Aggregation**.

---

## 6 Caching

Frequently requested data can be cached at gateway:

```
bash
```

```
GET /products
```

- Faster response
  - Reduced backend load
- 

## 7 Logging, Monitoring & Tracing

Gateway is the **best place** to log:

- Request latency
- Errors
- Traffic patterns

Used for:

- Observability
- Debugging
- Alerts

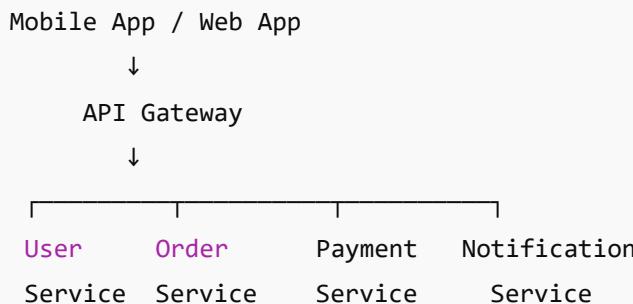
## API Gateway vs Load Balancer (Very common confusion)

Feature	API Gateway	Load Balancer
Level	Application (L7)	Network/App (L4/L7)
Routing	URL-based	IP/instance
Auth	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Rate limiting	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> Limited
Transformation	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No

- Load Balancer → "Where to send traffic"
- API Gateway → "How requests should behave"

## API Gateway in Microservices (Real World)

sql



Each service:

- Focuses only on business logic
- No auth / rate limit code

## 🔥 Types of API Gateways

### 1 Edge API Gateway

- Single gateway for all clients
- Simple but can become bottleneck

### 2 Backend-for-Frontend (BFF)

- Separate gateway per client

nginx

Web Gateway

Mobile Gateway

Admin Gateway

Very popular in large systems.

## 🛠️ Popular API Gateway Tools (Industry)

- [AWS API Gateway](#)
- [Spring Cloud Gateway](#)
- [Netflix Zuul](#)
- [Kong](#)
- [NGINX](#)

👉 As a Spring Boot learner, [Spring Cloud Gateway](#) is 🔥 for you.

## 💻 API Gateway in Payment Systems (Important!)

arduino

Client



API Gateway



Payment Service → Razorpay / Stripe

Gateway handles:

- Auth
- Idempotency keys
- Rate limits
- Webhook validation routing

This keeps payment service clean & secure.

---

## ⚠️ Disadvantages of API Gateway

- ✗ Single Point of Failure (if not HA)
- ✗ Added latency (extra hop)
- ✗ Complex configuration
- ✗ Needs careful scaling

👉 Solution:

- Run multiple gateway instances
  - Put Load Balancer before gateway
- 

## 🎯 Interview-Ready Explanation (Say this confidently)

"An API Gateway is a centralized entry point that manages routing, authentication, rate limiting, caching, and request transformation for backend services, especially in microservice architectures."

---

## 🧠 One-line memory trick

API Gateway = Brain + Gatekeeper of Microservices

---

## 🚀 What should you learn next?

Given your background in Spring Boot & backend, I strongly recommend next:

- 🔥 API Gateway vs Service Mesh

- 🔥 Spring Cloud Gateway internal flow
- 🔥 API Gateway + JWT + Redis
- 🔥 Design a payment system with API Gateway

Just tell me — we'll go step by step 😊

what is Short polling and long polling in system design?

explain this in details so that i am able to understand this well and do not have doubt in this.

Great question, Piyush 👍

**Short polling vs Long polling** is a classic system-design topic that sits right between **HTTP fundamentals** and **real-time systems**. I'll explain it from zero, with **flows, diagrams, trade-offs, and interview clarity**, so there's no doubt left.

---

## Polling in System Design (Big Picture)

**Polling** = *Client repeatedly asks the server*

"Do you have new data for me?"

This is used when:

- Server cannot directly push data
- Client wants updates (chat, notifications, order status)

There are **two main polling strategies**:

1. **Short Polling**
  2. **Long Polling**
- 

### 1 Short Polling

#### ◆ What is Short Polling?

**Short polling means the client sends requests to the server at fixed intervals, regardless of whether new data is available.**

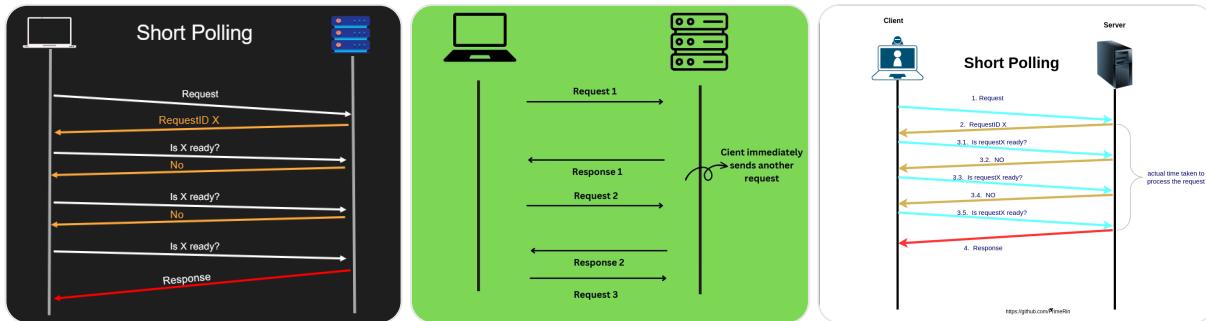
pgsql

```
Client → Server → Response (new data OR no data)
(wait some time)
Client → Server → Response
```

- ◆ Simple definition (remember this)

Short polling = "Are we there yet?" asked again and again 😅

- ◆ Short Polling Flow (Step-by-step)



1. Client sends request
2. Server responds immediately
  - With data ✘ or "no updates"
3. Client waits (e.g., 5 seconds)
4. Client sends request again

- ◆ Example (Chat application)

bash

```
GET /messages
```

Every 5 seconds, frontend calls backend:

- Most of the time → "No new messages"
  - Sometimes → New message arrives
- 

#### ◆ Pros of Short Polling

- ✓ Very simple to implement
  - ✓ Works with plain HTTP
  - ✓ Easy debugging
  - ✓ Stateless
- 

#### ◆ Cons of Short Polling

- ✗ Wastes network & CPU
  - ✗ High number of useless requests
  - ✗ Higher latency (wait until next poll)
  - ✗ Does NOT scale well
- 

#### ◆ When Short Polling is acceptable

- Small systems
  - Low traffic
  - Non-real-time updates
  - Admin dashboards
  - Simple cron-like checks
- 

## 2 Long Polling

#### ◆ What is Long Polling?

Long polling keeps the HTTP request open until new data is available or a timeout occurs.

arduino

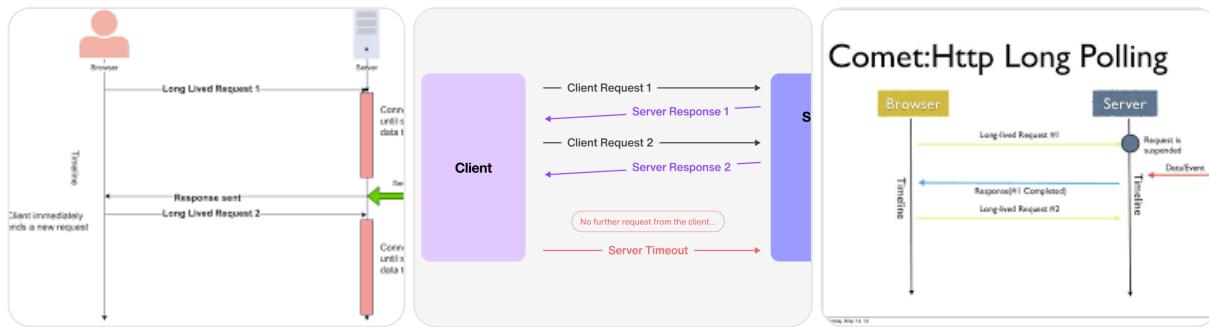
Client → Server (waits...)

**Server → Client** (only when data is ready)

## ◆ Simple definition

Long polling = "Wait here, I'll call you when something happens."

## ◆ Long Polling Flow (Step-by-step)



1. Client sends request
2. Server does not respond immediately
3. Server waits for:
  - New data OR
  - Timeout
4. Once data is available → Server responds
5. Client immediately sends next request

## ◆ Example (Notification system)

bash

```
GET /notifications
```

- User waits
- Server responds instantly when notification arrives
- Client reconnects again

Feels almost real-time.

## ◆ Pros of Long Polling

- Near real-time updates
  - Fewer useless requests
  - Better than short polling for scale
  - Works over HTTP
- 

## ◆ Cons of Long Polling

- Server resources tied up
  - More complex logic
  - Connection management required
  - Can overload servers if badly designed
- 

## ◆ When Long Polling is used

- Chat applications (old WhatsApp web)
  - Notifications
  - Live feeds (before WebSockets)
  - Payment status updates
  - Real-time dashboards (medium scale)
- 

## 3 Short Polling vs Long Polling (Side-by-Side)

Feature	Short Polling	Long Polling
Request frequency	Fixed interval	On data availability
Server response	Immediate	Delayed
Latency	Higher	Low
Network usage	High	Lower
Complexity	Simple	Moderate
Scalability	Poor	Better

Feature	Short Polling	Long Polling
Real-time feel	✗	✓

## 4 Timeline Comparison (Mental Model)

vbnets

**Short Polling:**

**Client:** —Req—Resp—wait—Req—Resp—wait—Req—Resp

**Long Polling:**

**Client:** —Req—————Resp—Req—————Resp

👉 Long polling reduces empty responses.

## 5 Important System Design Considerations

### ◆ 1. Server Load

- Short polling → many short-lived requests
- Long polling → fewer but longer-held connections

### ◆ 2. Timeout Handling

In long polling:

- Server sets timeout (e.g., 30 seconds)
- If no data → returns empty response
- Client reconnects

This prevents infinite connections.

### ◆ 3. Scalability Concerns

Long polling needs:

- Async servers

- Non-blocking I/O
- Event-driven architecture

Bad idea with:

- Thread-per-request servers
  - Blocking I/O
- 

## 6 Polling vs WebSockets (Quick context)

Technique	Nature
Short Polling	Client pull
Long Polling	Client pull (optimized)
WebSocket	Full duplex (push + pull)
SSE	Server push only

👉 Polling is fallback, not ideal for heavy real-time systems.

---

## 7 Real-world Example (Payment Status)

### ✗ Short Polling

arduino

Client asks every 5 sec:  
"Payment done?"

### ✓ Long Polling

arduino

Client waits  
Server responds immediately after payment success

Better UX + fewer requests.

---

## 8 Interview-Ready Explanation (Say this confidently)

**Short polling** repeatedly requests the server at fixed intervals, often wasting resources.  
**Long polling** keeps the request open until new data is available, reducing latency and unnecessary requests while still using HTTP.

## 9 One-line Memory Trick 🔥

- Short polling → *Ask again and again*
- Long polling → *Wait until something happens*

## 10 What should you learn next?

Since you're into **backend + system design**, next logical topics are:

- 🔥 Long polling vs WebSockets vs SSE
- 🔥 How chat systems work internally
- 🔥 Event-driven systems with Kafka
- 🔥 Real-time payment status architecture

Just tell me what you want next — we'll build it step by step 😊

what is cdn in system design?

explain this in details so that i am able to understand the well and do not have doubt in this.

Excellent question, Piyush 🌟

**CDN (Content Delivery Network)** is one of those concepts that silently powers **fast, scalable, global systems**. If you truly get CDN, you'll understand *why Netflix loads instantly, why images feel "nearby"*, and *how large systems survive traffic spikes*.

I'll explain this **from basics → internal working → system design usage → interview mindset**, step by step, no gaps.



## What is a CDN in System Design?

## ◆ Simple definition

A CDN (Content Delivery Network) is a **globally distributed network of servers** that delivers content to users from the **nearest location** instead of a single central server.

👉 Goal: Reduce latency, improve performance, and increase availability

pgsql

User → Nearest CDN Server → Content

## 🧠 Real-life analogy (remember this forever)

Imagine a popular book 📚 :

✗ One bookstore in Delhi

- People from Mumbai, Chennai, Kolkata travel far → slow & crowded

✓ Many bookstores across India

- Everyone buys from the nearest store → fast & scalable

● Bookstores = CDN edge servers

● Book = content (images, videos, APIs)

## 🚫 Problem without CDN (Why CDN exists)

Without CDN:

SCSS

User (India)



Single Server (USA)

Problems:

- ✗ High latency
- ✗ Slow page loads
- ✗ Server overload

- ✗ Poor user experience
- ✗ Single point of failure

## ✓ With CDN (Correct design)

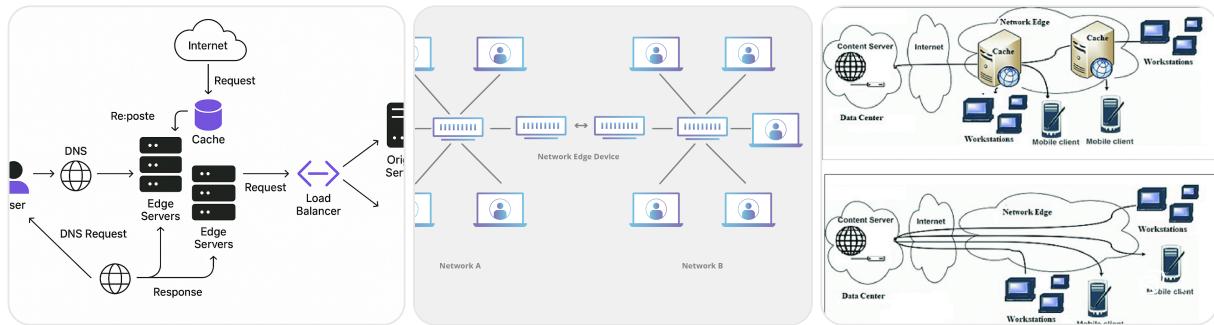
pgsql

```
User (India)
↓
CDN Edge Server (Mumbai)
↓
Origin Server (only if needed)
```

Result:

- 🚀 Faster
- 🌎 Scalable
- 💪 Reliable

## 🖼 High-Level CDN Architecture



## 🧱 Core Components of a CDN

### 1 Origin Server

- Your main backend
- Stores original content

Example:

api.myapp.com

## 2 Edge Servers (PoPs – Points of Presence)

- Distributed across the world
- Cache content close to users

Example locations:

- Mumbai
- Singapore
- Frankfurt
- New York

## 3 CDN DNS

- Routes users to nearest edge server
- Uses GeoDNS / Anycast

## 🔄 How CDN works (Step-by-step flow)

Let's say user opens:

arduino

<https://myapp.com/Logo.png>

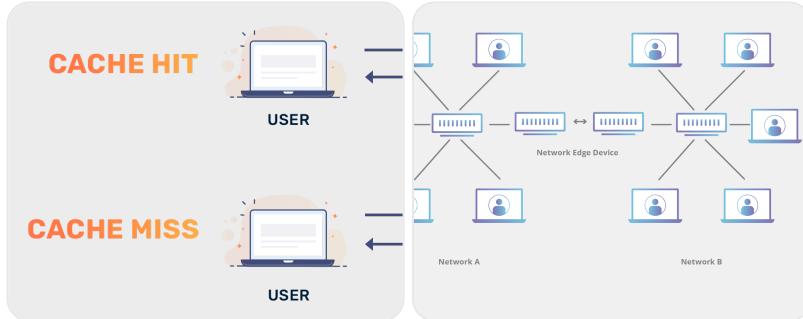
**Step-by-step:**

1. **DNS resolution**
  - Domain points to CDN
2. **User routed to nearest edge**
  - Based on location
3. **Cache check**
  - If content exists → return immediately
4. **Cache miss**
  - Edge fetches from origin

## 5. Cache & respond

- Stores content for future users
- 

### 🖼 CDN Cache Flow



### 📦 What content is served by CDN?

#### Static Content (Most common)

- Images
- CSS
- JavaScript
- Fonts
- Videos

#### Dynamic Content (Advanced CDNs)

- API responses
- HTML pages
- Authenticated content (with rules)

### ⌚ TTL (Time To Live) in CDN

TTL decides how long content stays cached.

Example:

logo.png → TTL = 24 hours

### Trade-off:

- High TTL → Faster, less origin load
- Low TTL → Fresher data, more origin calls

👉 TTL is a system design tuning knob

---

## ✳️ CDN in System Design (Very Important)

### 1 Performance Improvement 🚀

- Reduced latency
  - Faster TTFB (Time To First Byte)
- 

### 2 Scalability ⚖️

- Millions of users
- CDN absorbs traffic spikes

Example:

mathematica

Flash Sale / IPL Streaming / Product Launch

---

### 3 High Availability 💪

- Origin server goes down
  - Cached content still served
- 

### 4 DDoS Protection 🛡️

- CDN blocks malicious traffic
  - Rate limiting at edge
- 

### 5 Reduced Backend Cost 💰

- Fewer origin hits

- Lower compute & bandwidth usage
- 



## CDN vs Load Balancer (Common confusion)

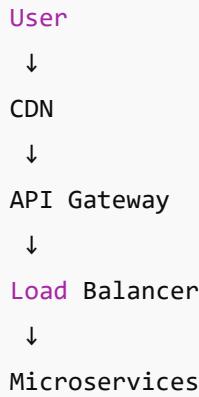
Feature	CDN	Load Balancer
Purpose	Content delivery	Traffic distribution
Location	Global	Usually regional
Caching	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Static content	Excellent	<input type="checkbox"/>
Dynamic APIs	Limited	<input checked="" type="checkbox"/>

- **CDN** → "Bring content closer to users"
  - **Load Balancer** → "Distribute traffic to servers"
- 



## CDN + API Gateway + Load Balancer (Real System)

pgsql



This is **industry-standard architecture**.

---



## CDN Security Features

- HTTPS termination

- WAF (Web Application Firewall)
  - Rate limiting
  - Bot protection
  - Token-based access
- 

## Popular CDN Providers (Industry)

- Cloudflare
  - Akamai
  - AWS CloudFront
  - Fastly
- 

## CDN in Payment & Backend Systems

Even in payments:

- Static assets → CDN
- Payment status pages → CDN + API
- Webhook endpoints →  CDN (direct to backend)

 **CDN is NOT for webhooks** (important interview point)

---

## Disadvantages of CDN

-  Cache invalidation complexity
  -  Slight delay for uncached content
  -  Cost at very large scale
  -  Not ideal for highly personalized data
- 

## Interview-Ready Explanation

"A CDN is a geographically distributed network of servers that cache and deliver content from locations closest to users, reducing latency, improving scalability, and protecting origin servers."

---

Say this confidently 😊

## 🧠 One-line Memory Trick 🔥

| CDN = Bring content closer to the user

---

## 🚀 What should you learn next?

Given your system-design journey, next perfect topics are:

- 🔥 CDN vs Reverse Proxy
- 🔥 Cache invalidation strategies
- 🔥 Designing Netflix / YouTube
- 🔥 CDN + Redis caching

Just tell me — we'll go deeper step by step 🌟