

# **SDE: System Design and Engineering**

**Lecture – 01 (A)**

**Introduction to**

**System Design and Engineering.**

**From Zero to Google: Architecting the Invisible Infrastructure**

*by*

**Aatiz Ghimire**

# Learning Objectives

- Use real-world architecture to reproduce system design frameworks
- Understand how large-scale systems scale and fail
- Gain experience applying architecture patterns from industry leaders

# Motivation

Thursday, 5 June 2025

Gorkhapatra | E-paper | f | □ | ▶

NATION'S 1<sup>ST</sup> ENGLISH BROADSHEET

## THE RISING NEPAL

ALL BE HAPPY, ALL BE WELL

HOME NATION WORLD OPED PROVINCES SPORTS BUSINESS HEALTH SOCIETY SCIENCE & TECH MORE

### Passport distribution stopped for technical problem



File Photo

#### Latest Updates



TRN Online Thu, 5 June 2025

**Vice President Yadav emphasizes on adopting environment-friendly technology**

TRN Online Thu, 5 June 2025

**Related bodies from Nepal, India promise to protect biodiversity, wildlife**

# Motivation

The screenshot shows the Khabarhub website interface. At the top, there's a dark blue header with the Khabarhub logo and name, the date "Thursday, June 5th, 2025", and social media icons. Below the header is a navigation bar with links for HOME, NEWS, NATIONAL, INTERNATIONAL, BUSINESS, INTERVIEW, OPINION, MORE, UTILITIES, and a language selector set to "नेपाली" (Nepali). The main headline reads "DoTM'S server down, all services closed indefinitely". Below the headline, the article is dated "23 March 2022" and includes a "Time taken to read : < 1 Minute" indicator. The article features a photo of a building with a sign that says "DOCTORAL TRAINING MANAGEMENT" and "DEPARTMENT OF TRANSPORT MANAGEMENT". To the right of the article, there's a "JUST IN" section with three news items: "Ex-PM Madhav Kumar Nepal faces corruption charges, Rs 185 million claimed", "KATHMANDU: The Commission for the Investigation of Abuse of Authority", and "Gyanendra Shahi questions legality of new NRB Governor's appointment".

**Khabarhub**  
23 March 2022  
Time taken to read : < 1 Minute

**DoTM'S server down, all services closed indefinitely**

**JUST IN**

- Ex-PM Madhav Kumar Nepal** faces corruption charges, Rs 185 million claimed
- KATHMANDU:** The Commission for the Investigation of Abuse of Authority
- Gyanendra Shahi** questions legality of new NRB Governor's appointment
- KATHMANDU:** Rastriya Prajatantra Party (RPP) Chief Whip Gyanendra Bahadur Shahi

**Snakes's meeting delayed as**

# Motivation



## DoTM to halt its services due to issues on server



# Motivation


NATION'S 1<sup>ST</sup> ENGLISH BROADSHEET

## THE RISING NEPAL

ALL BE HAPPY, ALL BE WELL


HOME NATION WORLD OPED PROVINCES SPORTS BUSINESS HEALTH SOCIETY SCIENCE & TECH MORE

### Int'l flights affected due to server down at TIA Immigration



TRN Online Sat, 28 January 2023

#### Latest Updates



TRN Online Thu, 5 June 2025

Vice President Yadav emphasizes on adopting environment-friendly technology

TRN Online Thu, 5 June 2025

Related bodies from Nepal, India promise to protect biodiversity, wildlife

# Motivation

TECHMANDU

FOLLOW

CLICK HERE FOR

TECHMANDU

HOME

DIGITAL NEWS

AUTO

TELECOM

NEPALI DATE CONVERTER TOOL

ABOUT

CONTACT US

Q

Home > News > 4 lakh Loksewa Data Lost, Public Service Commission Opening New Application?

News

4 lakh Loksewa Data Lost, Public Service Commission Opening New Application?

By Dinesh April 27, 2023

लोक सेवा आयोग

नेपाल

लग-इन

दर्ता गर्नुहोस्

प्रयोगकर्ता नाम/ईमेल \*

प्रयोगकर्ता नाम/ईमेल आवश्यक छ

पासवर्ड \*

पासवर्ड आवश्यक छ

कृपया ध्यान दिनुहोस्

लग-इन गर्न समस्या भयो? यहाँ Click गर्नुहोस्।

पासवर्ड रिसेट माग्नुहोस्

MOST POPULAR

IME Pay is Offering 2500 Cashback on NLIC Premium Payment

June 30, 2022

eSewa celebrates its 14th anniversary, find all the offers

May 13, 2023

27 Years Old Binay Khadka Appointed The New CEO of Khalti

November 17, 2021

Transport Office to Issue Driving License from Chitwan

April 27, 2022

Load more >

LATEST POSTS

Zeekr X EV price in Nepal | Latest June 2025 Updated

June 2, 2025

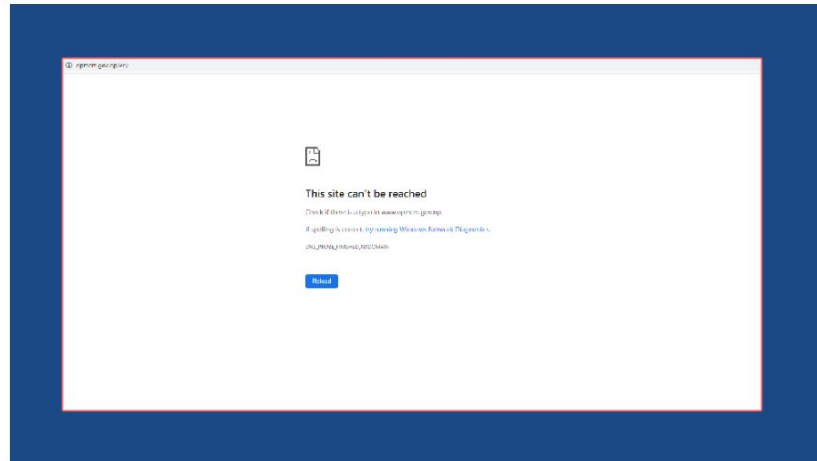
# Motivation

## All govt websites down due to server glitches

Nepalkhabar

17:17PM Jan 28,2023 | Kathmandu

35  
Shares





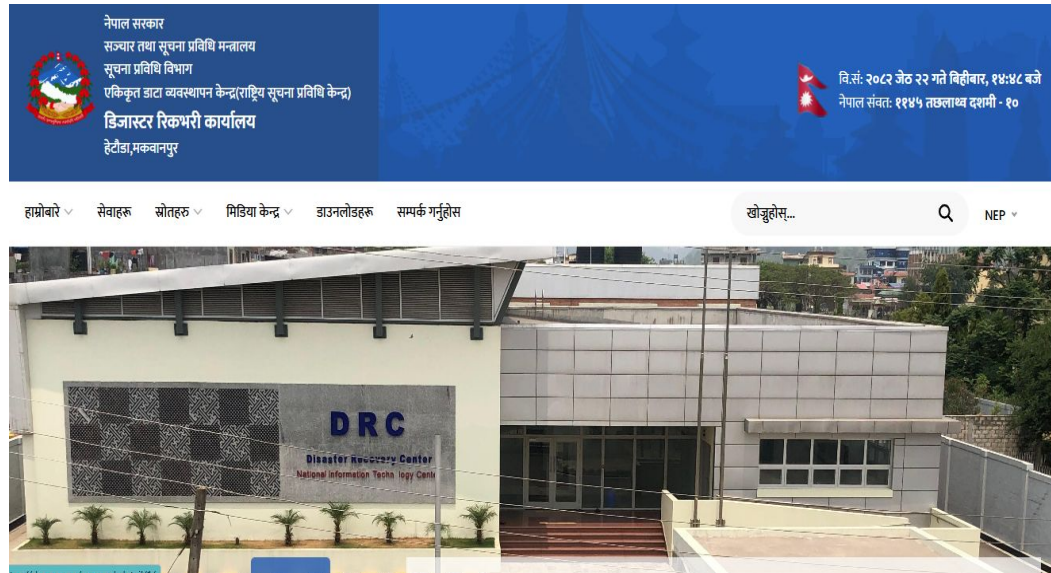
# are those server issues?



- Integrated Data Management Centre (IDMC) – Government of Nepal
- Sole official government data center, disaster recovery facility located in Hetauda
- Established: 2009 (with support from the Government of Korea)
- Fund \$26.59 million (USD)
- provides hosting, email, domain registration, intranet operation, disaster recovery, e-governance backend, technical consultancy, digital archiving, ICT infrastructure, and system interoperability for the Government of Nepal

Ref: <https://www.adb.org/sites/default/files/project-documents/38347/38347-022-pcr-en.pdf>

# are those server issues?



- Disaster Recovery Center in Hetauda ensures data backup, service continuity, and emergency recovery for government digital infrastructure across Nepal
- Established: 2009 (with support from the Government of Korea)
- Fund: \$ 3.2 million (USD)

Ref: <https://www.adb.org/sites/default/files/project-documents/38347/38347-022-pcr-en.pdf>

# Does this happen in all Datacenter? (NO)





## Government of Nepal – Data Center Uptime and Tier Classification

Source: Data Center and Cloud Service and Management Directives, 2081 (2024 AD)





Tier Level	Availability (Uptime/year)	Max Downtime/year	Redundancy	Fault Tolerance	Power Backup
Tier I (तह एक)	99.671%	≤ 28.8 hours/year	No UPS required, minimal backup	✗ Not fault tolerant	12 hours backup
Tier II (तह दुई)	99.749%	≤ 22 hours/year	UPS required, single active component	✗ Not fault tolerant	12–24 hours backup
Tier III (तह तीन)	99.982%	≤ 1.6 hours/year	N+1 redundancy, concurrently maintainable	✓ Yes	24–48 hours backup
Tier IV (तह चार)	99.995%	≤ 26.3 minutes/year	2N+1 full redundancy, fault tolerant	✓ Yes	≥ 48 hours backup with dual UPS and generators

Ref: <https://mocit.gov.np/content/13003/data-center-and-cloud-service--operations-and/>





# Then, who fault is to blame?

Failure Domain	Specific Issue / Cause	Responsible Party	Remarks
 <b>Infrastructure Fragility</b>	Single-point-of-failure, no backup links, non-Tier III compliance	DoIT, NITC, Hosting Providers	Design-level fault; violates high-availability principles
 <b>Power &amp; Connectivity Issues</b>	Grid instability, limited ISP redundancy, data center blackout	NEA, ISPs, Data Center Operators	Lack of failover, poor network engineering
 <b>Institutional Weakness</b>	Untrained staff, high turnover, slow decision-making	Executing Agencies, Line Ministries	Project delays, mismanagement of IT assets
 <b>Cybersecurity Lapses</b>	No firewalls, no MFA, exposed admin panels	System Admins, DevOps Engineers	Technical negligence; violates zero-trust and defense-in-depth norms

# Then, who fault is to blame?

Failure Domain	Specific Issue / Cause	Responsible Party	Remarks
 <b>Vulnerable Software</b>	Code injection, hardcoded passwords, no encryption	Developers, QA Teams	Secure coding and testing failure
 <b>Poor Configuration</b>	Open ports, default credentials, lack of SSL	System Admins, Deployment Engineers	Misconfiguration is one of top cloud threats (CSPM issue)
 <b>Insider Threats</b>	Malicious or negligent employee actions	Internal Users, Departmental Staff	Requires logging, auditing, RBAC, and regular reviews
 <b>No Backup/DR Plan</b>	No offsite backup, no restoration tested	Data Custodians, CIOs, DoIT	Violation of ISO 27001/ISMS baseline

# Then, who fault is to blame?

Failure Domain	Specific Issue / Cause	Responsible Party	Remarks
 <b>Weak Policies &amp; Oversight</b>	No cybersecurity framework, no audits or enforcement	MoCIT, DoIT, Supreme Audit Institutions	Policy gap; lack of accountability for digital assets
 <b>Delayed Procurement</b>	Slow tendering, dependency on foreign vendors	Procurement Units, Ministries	System integration delayed due to non-technical bottlenecks
 <b>Lack of Monitoring Tools</b>	No SIEM, no intrusion detection, no uptime logs	NITC, DevOps, National SOC	Systems are "blind" to ongoing threats and failures
 <b>End-User Negligence</b>	Clicking phishing emails, using weak passwords	Civil Servants, Local Administrators	Awareness training and password policy enforcement needed

# You are also responsible. (Software Developers and System Designer)

Failure Point	Cause	Resulting Impact
Poorly written code (e.g., SQL injection)	Failure to sanitize inputs, follow secure coding practices	Enables hacking, data theft, and unauthorized access
Weak authentication systems	Lack of password hashing, MFA, or session expiry	Account compromise and identity theft
No logging or audit trails	No code for event capture or traceability	Makes post-incident forensics and rollback impossible
Monolithic architecture	Hard coded logic, tight coupling, no scalability	System crashes under load, poor maintainability
No exception handling	Uncaught errors lead to service crashes or incorrect behavior	Inconsistent data or service unavailability
Insecure APIs	Unauthenticated endpoints or exposure of sensitive data	External abuse, denial of service, or data leakage

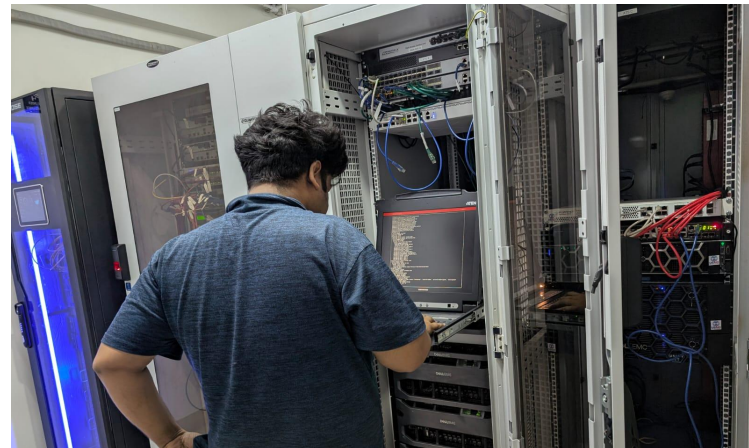
# \$whoami

## Aatiz Ghimire

- Lecturer, Herald College Kathmandu
- Research Engineer/ Fellow, Tribhuvan University Supercomputing Center
- Ex-CTO, Government Consultant

### Education:

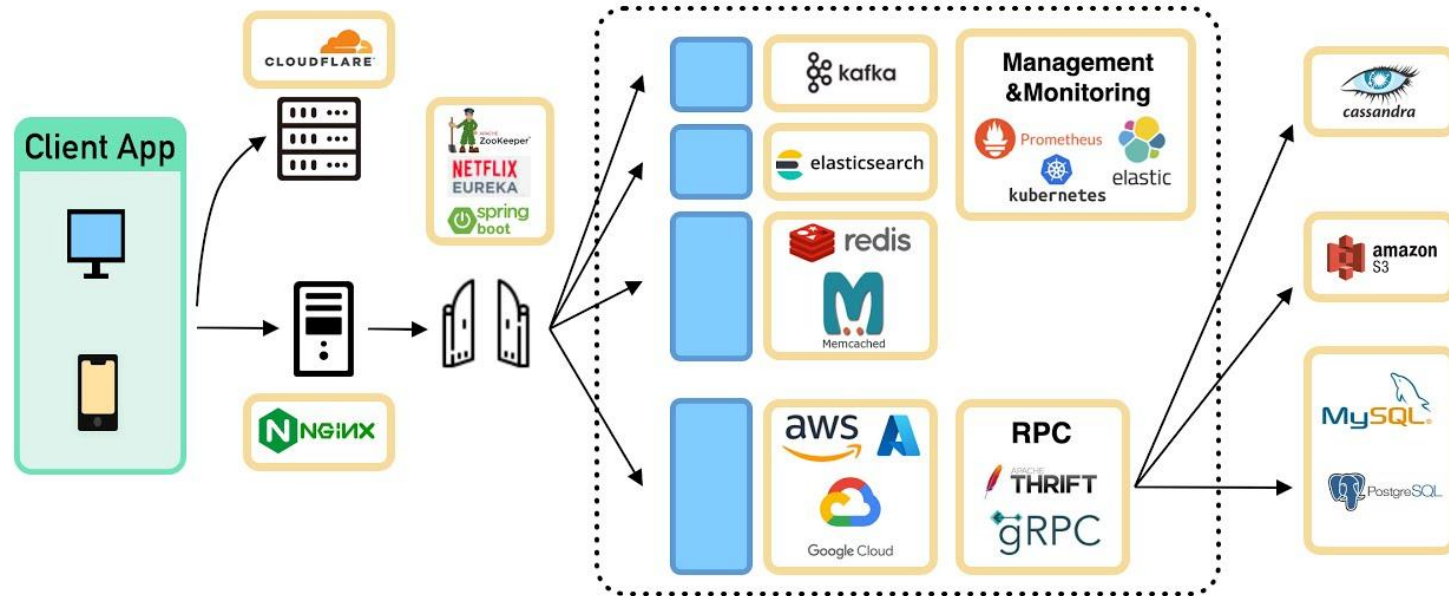
- Master in Data Science, Tribhuvan University
- Bachelor of Engineering, Tribhuvan University





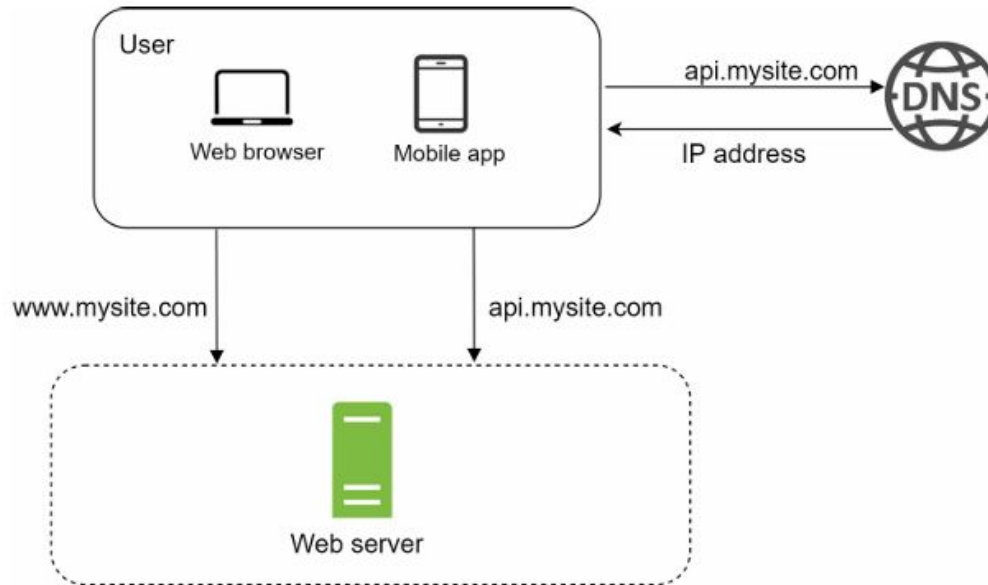
# What is System Design?

- Building a scalable system consists various components that can handle millions of users at once.
- Looks typically like this:



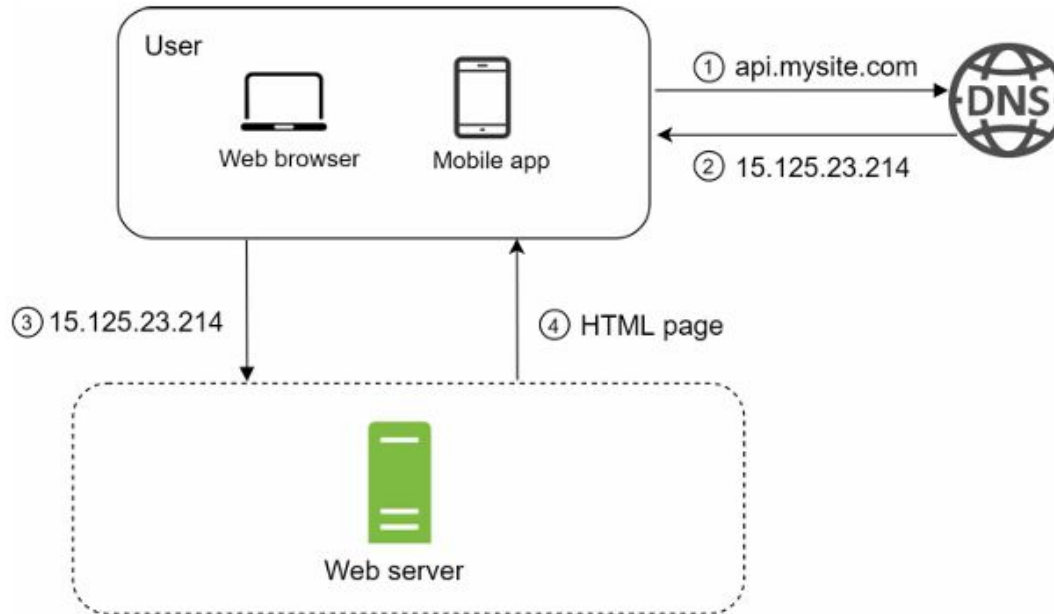
# Let's Get Started: Single server setup

- To start with something simple, everything is running on a single server.
- single server setup where everything is running on one server: web app, database, cache, etc.



# Single server setup

- To comprehend the architecture of this setup, it is essential to analyze both the request flow and the sources of traffic.



# Single server setup: Request Flow

- **Domain Resolution:**

Users typically access services via domain names, such as `api.mysite.com`. These domain names are resolved through the Domain Name System (DNS), which is often managed by third-party providers and not hosted directly on our servers.

- **IP Address Retrieval:**

Upon resolution, the DNS returns an Internet Protocol (IP) address—for instance, `15.125.23.214`—to the client (browser or mobile application).

- **HTTP Request Dispatch:**

Once the client obtains the IP address, it initiates an HTTP request directed at the corresponding web server.

- **Server Response:**

The web server processes the request and returns either an HTML page (for browser clients) or a JSON-formatted API response (for mobile or web apps) for rendering or data handling.

# Single server setup: Traffic Sources

Traffic to the web server generally originates from two principal sources:

- **Web Applications:** These applications employ server-side programming languages (e.g., Java, Python) to manage business logic and data persistence. On the client side, technologies such as HTML, CSS, and JavaScript are used for user interface rendering.

- **Mobile Applications:**

Mobile clients communicate with the server using the HTTP protocol. The server typically responds with data in JSON (JavaScript Object Notation) format, favored for its lightweight and easily parseable structure in modern API communication.

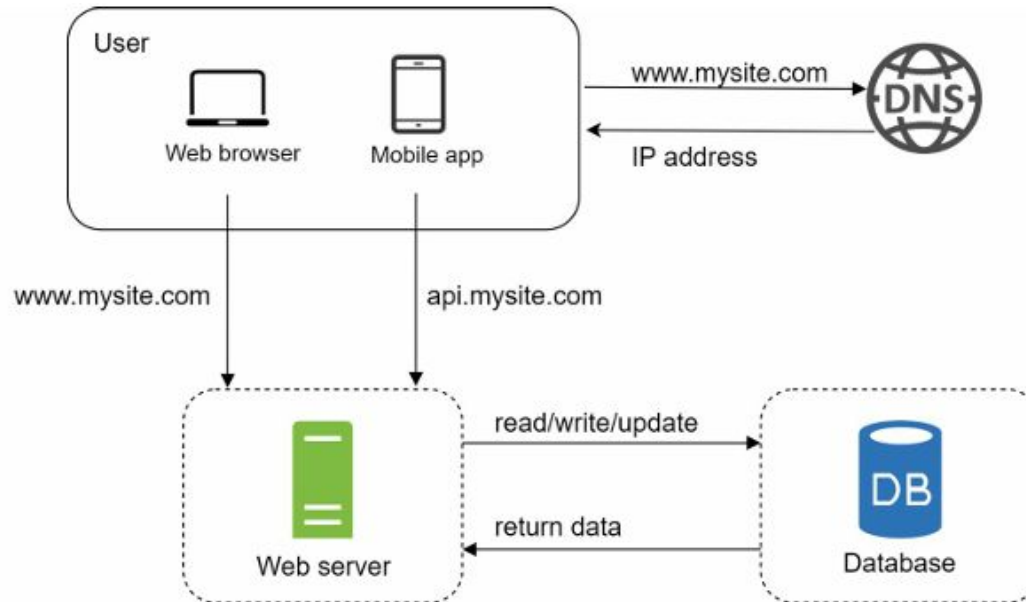
- An example of the API response in JSON format is shown below:

GET /users/45 – Retrieve user object for id = 45

```
{
  "id": 45,
  "firstName": "Aatiz",
  "lastName": "Ghimire",
  "address": {
    "streetAddress": "1 Naxal",
    "city": "Kathmandu",
    "state": "Bagmati",
    "postalCode": 44700
  },
  "phoneNumbers": [
    "xxxx-xxx-xxx",
    "xxxx-xxx-xxx"
  ]
}
```

# Database

- As the user base expands, **relying on a single server becomes insufficient**.
- To manage the **increasing load effectively**, the architecture must be **scaled to include multiple servers**—typically **one dedicated to handling web and mobile traffic (the web tier)**, and **another dedicated to managing data storage and retrieval (the data tier)**, optimizing resource utilization and system performance.



# Which databases to use?

You can choose between relational (SQL) and non-relational (NoSQL) databases based on your application needs.

- **Relational Databases (RDBMS):**

- Examples: MySQL, PostgreSQL, Oracle
- Store data in structured tables (rows and columns)
- Support complex queries and joins using SQL
- Proven, mature technology with over 40 years of reliability

- **Non-Relational Databases (NoSQL):**

- Examples: CouchDB, Cassandra, Neo4j, Amazon DynamoDB
- Store data using flexible formats: key-value, document, columnar, or graph
- Typically do not support joins
- Optimized for horizontal scaling and unstructured data

**Recommendation:** For most applications, relational databases are still the preferred choice due to their stability, consistency, and widespread support.

# Which databases to use?

- While relational databases are ideal for many scenarios, they may not suit every use case.
- It is essential to consider non-relational (NoSQL) databases when:
  - Your application demands **ultra-low latency access**.
  - Your data is **unstructured** or lacks clear relational structure.
  - You primarily work with **serialized formats like JSON, XML, or YAML**.
  - You need to manage and **store large-scale datasets** across distributed systems.



# Vertical scaling vs horizontal scaling

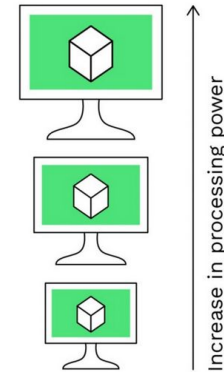
- **Vertical Scaling (Scale-Up):**

- Involves adding more resources (CPU, RAM) to a single server
- Simple to implement, suitable for low-traffic scenarios
- Limitations:
  - Hardware ceiling: finite capacity for upgrades
  - No redundancy—if the server fails, the entire system goes down

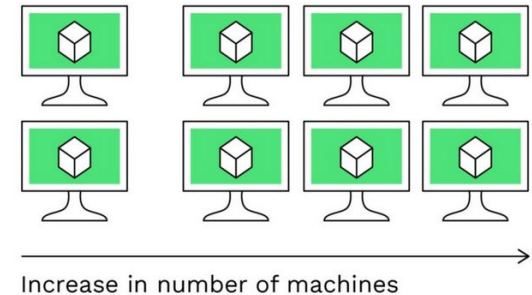
- **Horizontal Scaling (Scale-Out):**

- Adds multiple servers to distribute the load
- Enables failover, redundancy, and better handling of high traffic
- Preferred for large-scale, production-grade applications

**Vertical scaling**



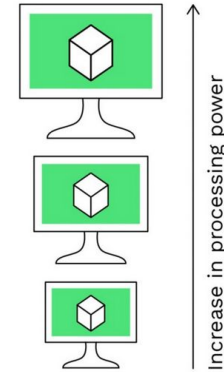
**Horizontal scaling**



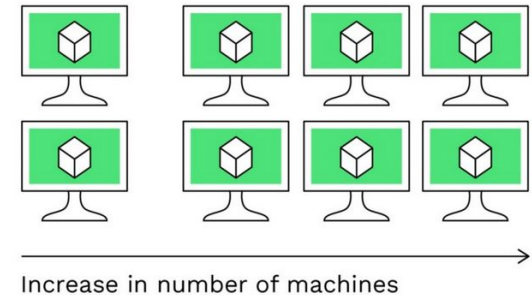
# Beyond Scaling

- Horizontal scaling is **generally preferred for large-scale applications** due to the inherent limitations of vertical scaling.
- In a traditional setup where users connect directly to a **single web server**, **system availability and performance become vulnerable**. If the server goes offline, the entire service becomes inaccessible.
- Moreover, under **high traffic**, the server may reach its **load capacity**, **resulting in latency or connection failures**.
- To mitigate these issues, implementing a **load balancer is essential**, as it distributes traffic across multiple servers, ensuring reliability and optimal performance.

**Vertical scaling**

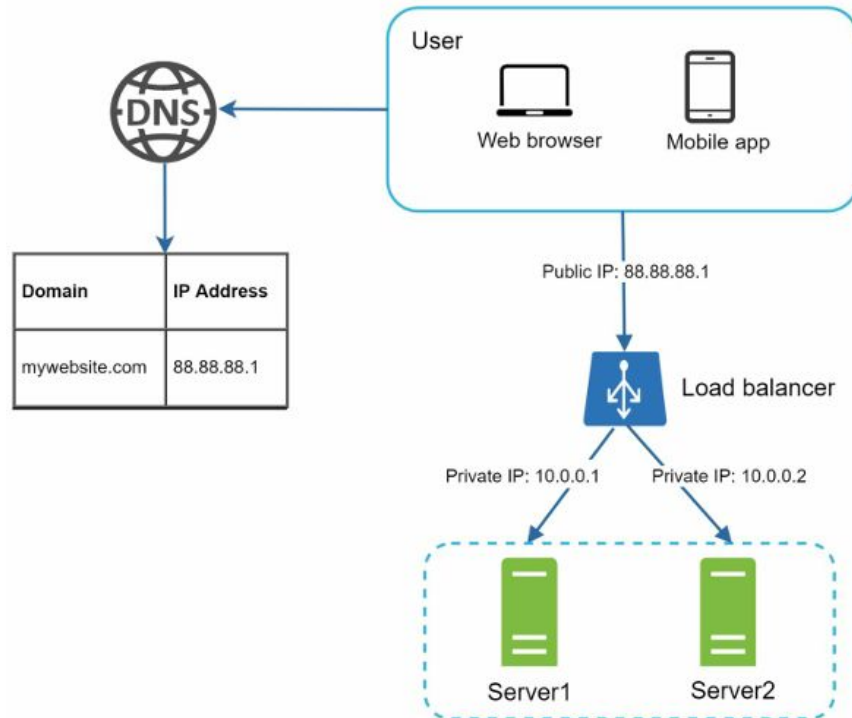


**Horizontal scaling**



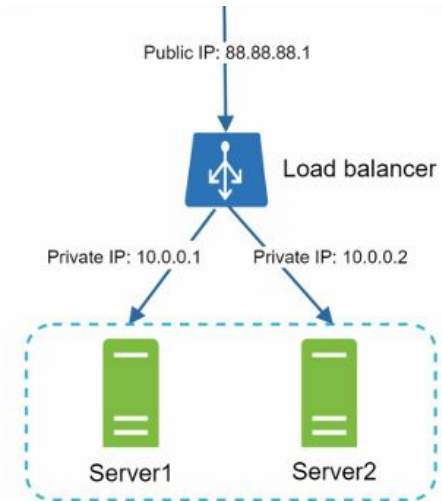
# Load balancer

- A load balancer evenly distributes incoming traffic among web servers that are defined in a load-balanced set.



# Load balancer

- In this setup, users connect directly to the **public IP of the load balancer**, not the web servers themselves. For enhanced security, web servers use **private IP addresses**, which are accessible only within the internal network and not exposed to the public internet.
- This architecture significantly improves the **availability and fault tolerance** of the web tier:
  - If **Server 1** goes offline, the load balancer redirects all traffic to **Server 2**, ensuring uninterrupted service. A replacement server can be added to maintain balance.
  - As traffic increases, the load balancer seamlessly distributes requests to **newly added servers**, scaling the system horizontally without disruption.

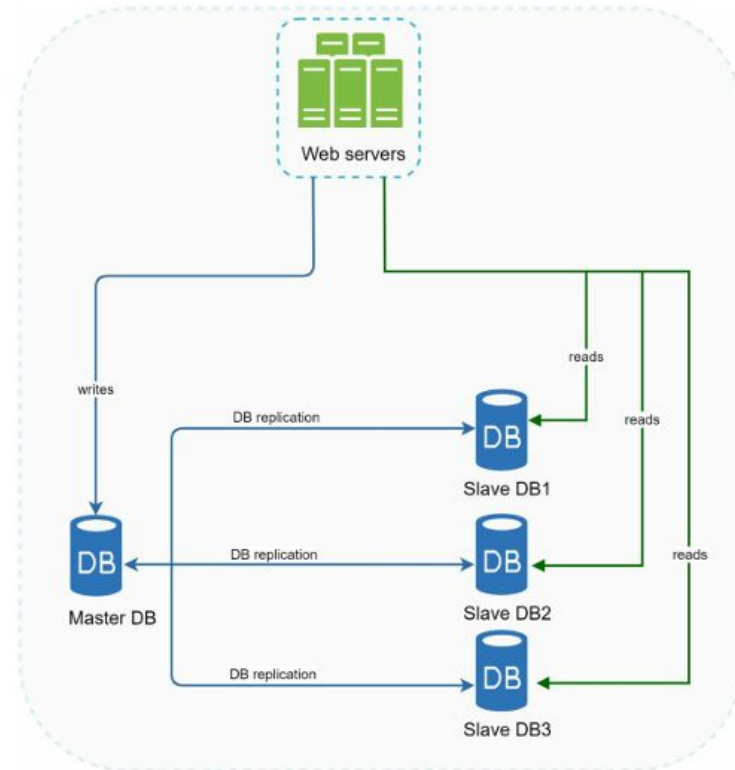


# Beyond Web Tier

- Now that the **web tier** is highly available and scalable, attention must shift to the **data tier**.
- In the current architecture, relying on a **single database** introduces a critical point of failure—there is no support for failover or redundancy.
- To mitigate this, **database replication** is a widely adopted technique that enhances both availability and fault tolerance by duplicating data across multiple database instances.

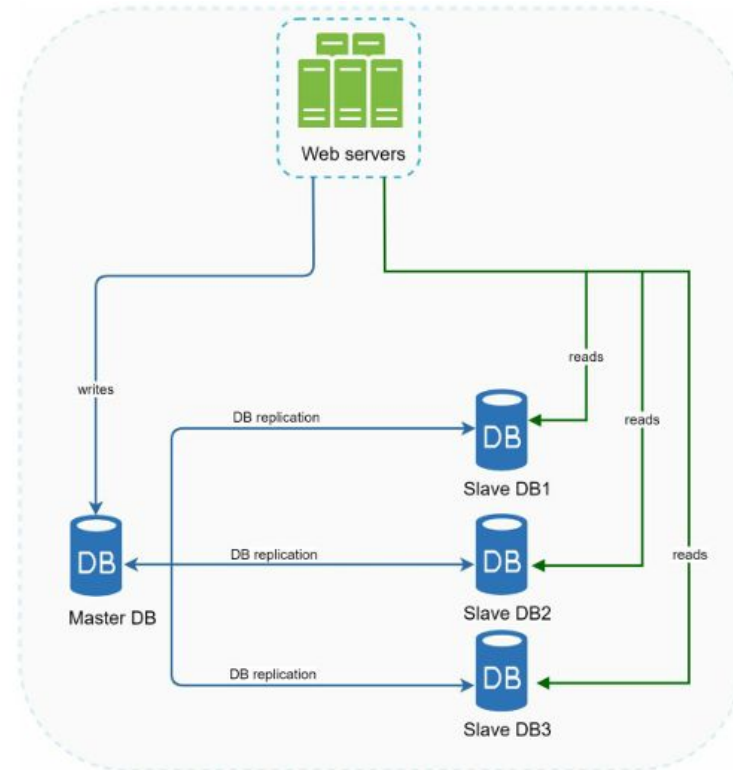
# Database replication

- **Database replication** is a common feature supported by many database management systems, typically following a **master-slave architecture**. In this model:
  - The **master database** handles all **write operations**, such as *INSERT*, *UPDATE*, and *DELETE*.
  - **Slave databases** replicate data from the master and are used exclusively for **read operations**.
- Since most applications perform significantly more reads than writes, systems are often designed with **multiple slave databases** for load distribution and performance optimization.
- This architecture not only improves **read scalability** but also introduces **redundancy**, enhancing system availability in the event of a failure.




# Advantages of Database Replication

- **Improved performance:** In a master-slave architecture, write operations are handled by the master, while read operations are distributed across slave nodes. This separation enables higher throughput by allowing concurrent query processing.
- **Enhanced Reliability:** Data replication ensures resilience against catastrophic failures. Even in the event of a disaster—such as a typhoon or earthquake—data remains intact across geographically distributed replicas.
- **High Availability:** Replicating data across multiple servers ensures continuous service. If one database instance becomes unavailable, others can seamlessly serve requests, minimizing downtime and maintaining operational continuity.



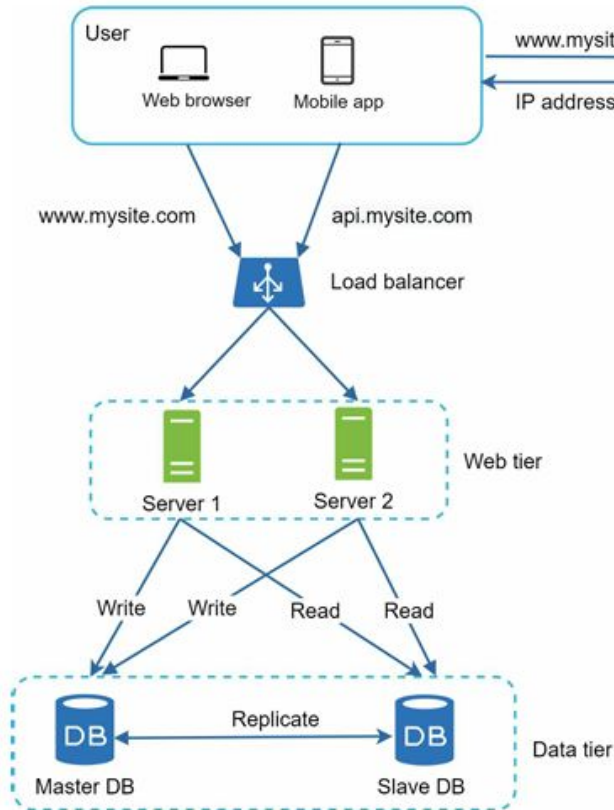
# What happens if a database goes offline?

 Note: In production environments, promoting a slave to master is non-trivial. The promoted slave may not be fully synchronized, and data recovery scripts might be required to reconcile any missing data.

- More advanced configurations such as **multi-master replication** and **circular replication** offer enhanced resilience, but involve greater complexity and are outside the scope of this course.
- Students interested in these topics are encouraged to explore the suggested reference materials.
- Ref: [https://en.wikipedia.org/wiki/Multi-master\\_replication](https://en.wikipedia.org/wiki/Multi-master_replication)
- Ref: <https://dev.mysql.com/doc/refman/8.4/en/mysql-cluster-replication-multi-source.html>



# After adding the load balancer and database replication



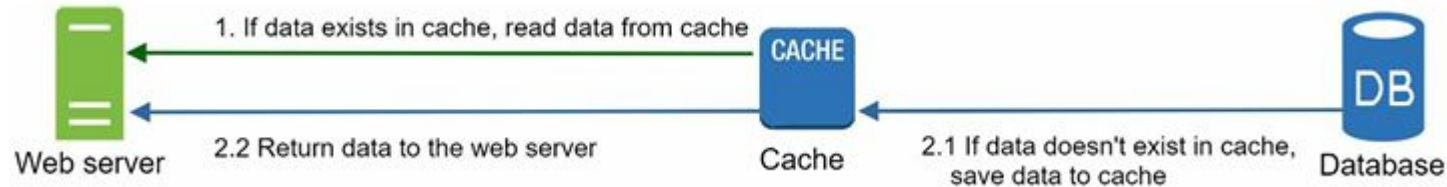
Let us examine the overall design flow:

- The user obtains the **IP address of the load balancer** via DNS resolution.
- The user connects to the **load balancer** using the resolved IP.
- The **load balancer** routes the HTTP request to one of the available web servers (e.g., Server 1 or Server 2).
- The **web server** retrieves user data from a **slave database** for read operations.
- Any **data-modifying operations** (e.g., **INSERT**, **UPDATE**, **DELETE**) are directed to the **master database**.

With this architecture, you now have a robust foundation for understanding the separation and coordination between the **web tier** and **data tier** in scalable system design.

# Cache: Speed(Load/Response Time) Through Temporary Storage

- A **cache** is a high-speed temporary storage layer that holds the results of expensive computations or frequently accessed data in memory. This allows subsequent requests to be served significantly faster.
- Without caching, each web page load may trigger multiple **database queries**, which degrades performance under heavy load. Caching mitigates this by minimizing redundant database access and lowering latency.



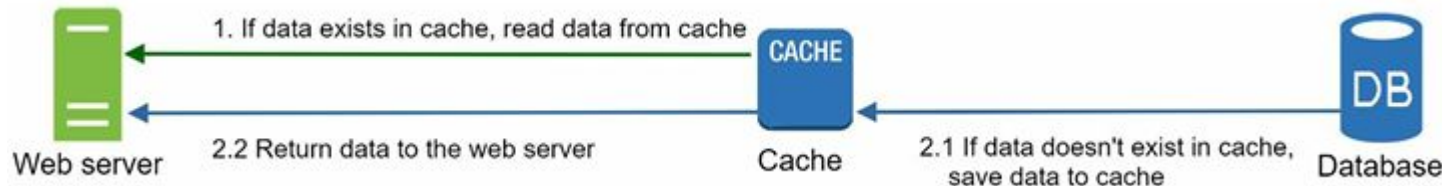
# Read-Through Caching Strategy

After receiving a request, the web server performs the following sequence:

1. **Check the cache** for the required response.
2. If found (*a cache hit*), it returns the cached response to the client.
3. If not found (*a cache miss*), it queries the database, stores the response in the cache, and then returns the result to the client.

This is known as a **read-through caching strategy**, which is effective for dynamic content with predictable access patterns.

Note: Other caching strategies (e.g., write-through, write-around, cache-aside) may be used depending on **data access frequency, size, and volatility**.



# Cache: Developer Integration

- Most cache servers—such as **Memcached** or **Redis**—provide client libraries or APIs for mainstream programming languages, enabling straightforward integration into application logic.
- Example (using Memcached API): ->
- This approach ensures faster page loads, better resource utilization, and a smoother user experience—especially under high traffic scenarios.

```
# Python example using pylibmc (Memcached client)
import pylibmc

cache = pylibmc.Client(["127.0.0.1"],
    binary=True)

# Read-through caching
user_id = 'user:123'
cached_data = cache.get(user_id)

if cached_data:
    return cached_data
else:
    data = fetch_from_db(user_id)
    cache.set(user_id, data, time=300) #
cache for 5 minutes
    return data
```

# Considerations for Using Cache

When implementing a caching layer, several architectural and operational factors must be considered to ensure reliability, consistency, and performance:

## 1. When to Use Cache

- Ideal for **read-heavy** data that is **infrequently modified**.
- Cache stores data in **volatile memory** (e.g., RAM), which is not persistent.
- Do **not** use caching for critical data persistence—if the cache server crashes or restarts, all stored data is lost.
- Ensure persistent storage (e.g., a database) is used for long-term data integrity.

## 2. Expiration Policy

- Cached data should **expire after a defined time** to prevent memory bloat or stale responses.
- Avoid setting TTL (time-to-live) too short—this increases load on the database.
- Avoid overly long TTL—stale or outdated data may be served to clients.
- Implement **smart expiration strategies** based on data volatility and access frequency.

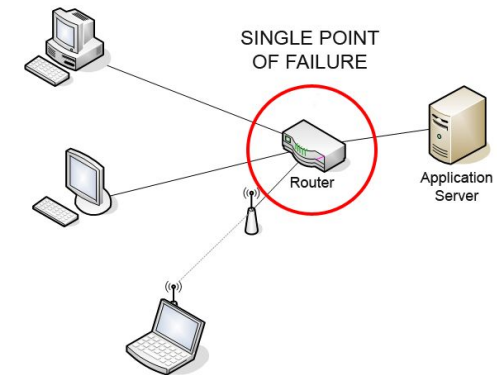
# Considerations for Using Cache

## 3. Data Consistency

- Maintaining **cache-store consistency** is non-trivial, especially for write-heavy applications.
- Writes to cache and database often occur **outside a single transaction**, leading to possible inconsistency.
- In **multi-region architectures**, synchronizing cache and DB can be challenging.
- Refer to advanced strategies in research such as *"Scaling Memcache at Facebook"*.

## 4. Mitigating Failures

- A **single cache server** is a potential **Single Point of Failure (SPOF)**.
- To avoid this:
  - Deploy **multiple cache servers** across regions or availability zones.
  - Use **replication or clustering** (e.g., Redis Cluster).
  - **Overprovision memory** by 20–30% as a buffer against sudden load surges.



# Considerations for Using Cache

## 5. Eviction Policy

- When the cache is full, new entries will **evict** old ones based on policy.
- Common eviction policies include:
  - **LRU (Least Recently Used)** – evicts items not accessed recently.
  - **LFU (Least Frequently Used)** – evicts items accessed least often.
  - **FIFO (First In, First Out)** – evicts the oldest inserted items.
- Choose the eviction policy based on application-specific access patterns.

Proper cache system design improves not only **system speed and efficiency**, but also **user experience and scalability** under real-world loads.

# Content delivery network (CDN)

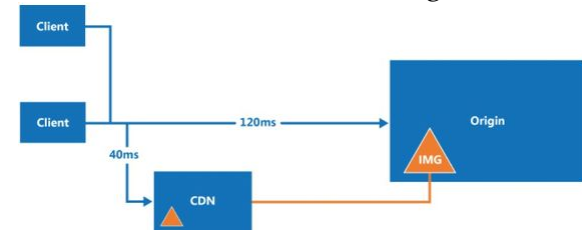
A **Content Delivery Network (CDN)** is a geographically distributed network of proxy servers designed to deliver **static content** to users with high availability and low latency. These servers cache and serve static resources such as: Images, Video files, CSS and JavaScript, Fonts and documents.

## How CDN Works – High-Level Overview

When a user accesses a website:

1. The request is routed to the **nearest CDN edge server** based on the user's geographic location.
2. The edge server checks its cache:
  - If the content is cached, it is returned immediately (*cache hit*).
  - If not, the CDN fetches the content from the origin server and caches it for future requests (*cache miss*).
3. This reduces **round-trip time (RTT)** and improves **page load performance**.

Example: If CDN servers are located in San Francisco, users in Los Angeles will experience faster load times compared to users in Europe due to physical proximity to the edge nodes.

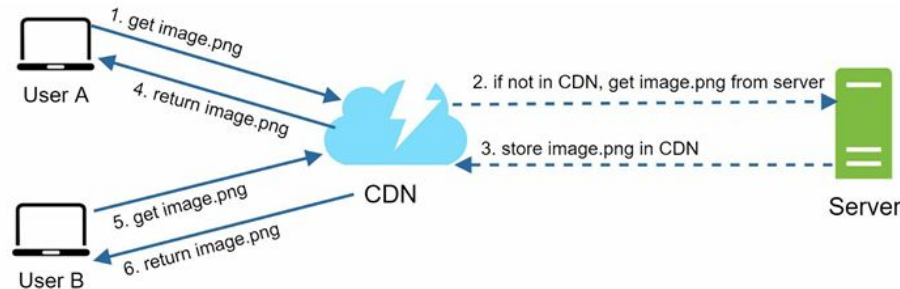




# CDN Workflow: Step-by-Step

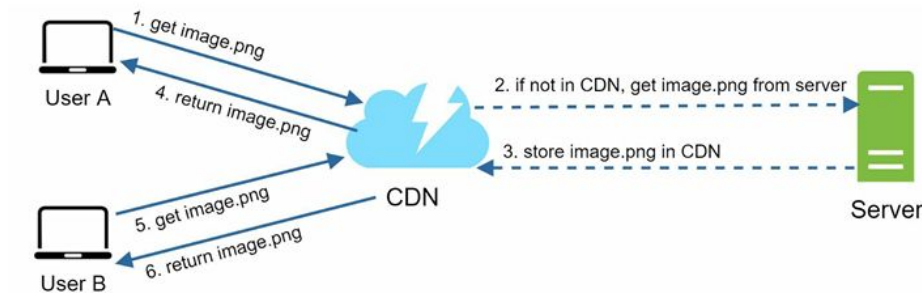
The following steps explain how a Content Delivery Network (CDN) efficiently delivers static assets to users:

1. **User A requests** a static asset such as `image.png` using a CDN-backed URL.  
Example URLs:
  - `https://mysite.cloudfront.net/logo.jpg` (Amazon CloudFront)
  - `https://mysite.akamai.com/image-manager/img/logo.jpg` (Akamai)
2. The **CDN edge server checks** whether the requested image exists in its local cache:
  - **If not cached**, it forwards the request to the **origin server** (e.g., your web server or storage like Amazon S3).



# CDN Workflow: Step-by-Step

3. The **origin server responds** with `image.png`, often accompanied by an **HTTP Cache-Control header** or **Time-to-Live (TTL)** metadata specifying how long the content should remain cached.
5. The **CDN caches** the image and simultaneously **returns it to User A**.
6. **User B** later requests the **same image** (e.g., `image.png`).
7. If the **TTL has not expired**, the CDN returns the **cached version directly**, significantly reducing response time and origin server load.



# Considerations When Using a CDN

When implementing a Content Delivery Network (CDN), the following operational and architectural factors must be evaluated:

- **Cost Implications**

CDN services are typically provided by third-party vendors and incur costs based on **data transfer volume**. Hosting rarely-accessed or low-priority assets in the CDN may result in **unjustified expenses** without performance gains.

- **Cache Expiry Configuration**

It is essential to **set appropriate cache expiration times**, especially for time-sensitive content:

- Too **long**: Clients may receive **stale data**.
- Too **short**: Increased load on **origin servers** due to frequent cache revalidation or bypass.

# Considerations When Using a CDN

- **CDN Fallback Strategy**

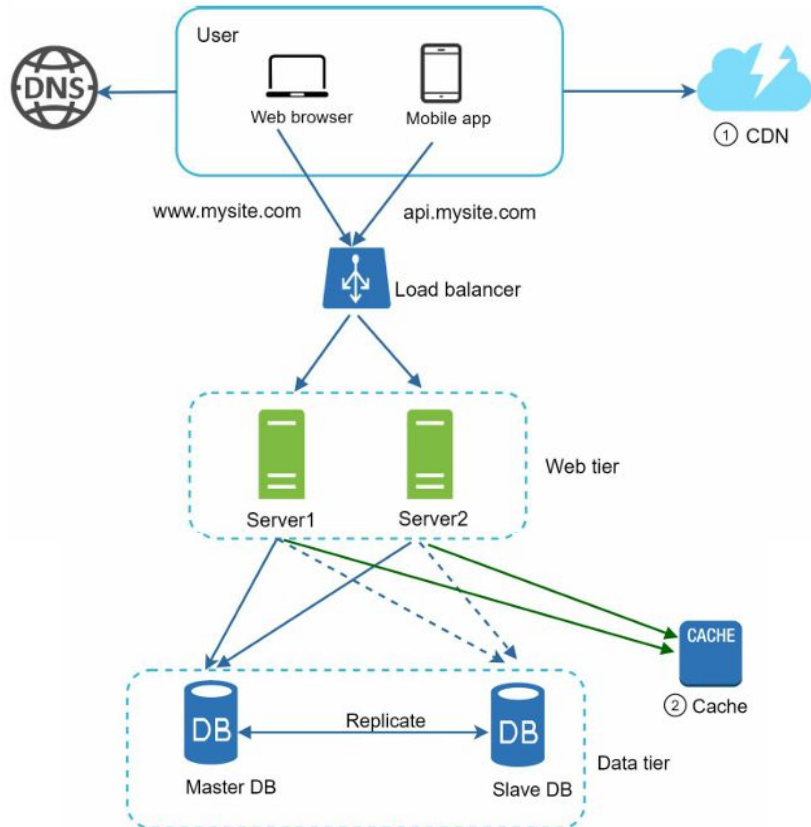
Applications should be resilient to **CDN outages**. In the event of temporary failures, clients should be able to detect unavailability and **fallback to the origin server** to ensure service continuity.

- **File Invalidation Mechanisms**

If a file must be removed or updated **before its TTL expires**, there are two common approaches:

- i. **API-based Invalidation:** Most CDN providers (e.g., Cloudflare, AWS CloudFront) offer APIs to **purge** cached objects.
- ii. **Object Versioning:** Alter the object URL using query parameters or path changes to **force a cache miss**.  
*Example: `image.png?v=2` serves a newer version, bypassing the cached `image.png`.*

## After adding the CDN and Cache



- Static assets (e.g., JavaScript, CSS, images) are no longer served directly by the web servers. Instead, they are delivered via the Content Delivery Network (CDN) to reduce latency and improve global performance.
- Database load is significantly reduced by employing caching strategies, allowing frequently accessed data to be served from in-memory stores like Redis or Memcached.

# Stateless Web Tier

To enable **horizontal scaling** of the web tier, it is essential to design it as **stateless**. This means **moving session data** (e.g., user authentication state) out of individual web servers and into **persistent storage** systems such as relational databases or NoSQL stores. All web servers in the cluster can then **uniformly access** the shared state, ensuring scalability and consistency.

## Stateful vs. Stateless Architectures

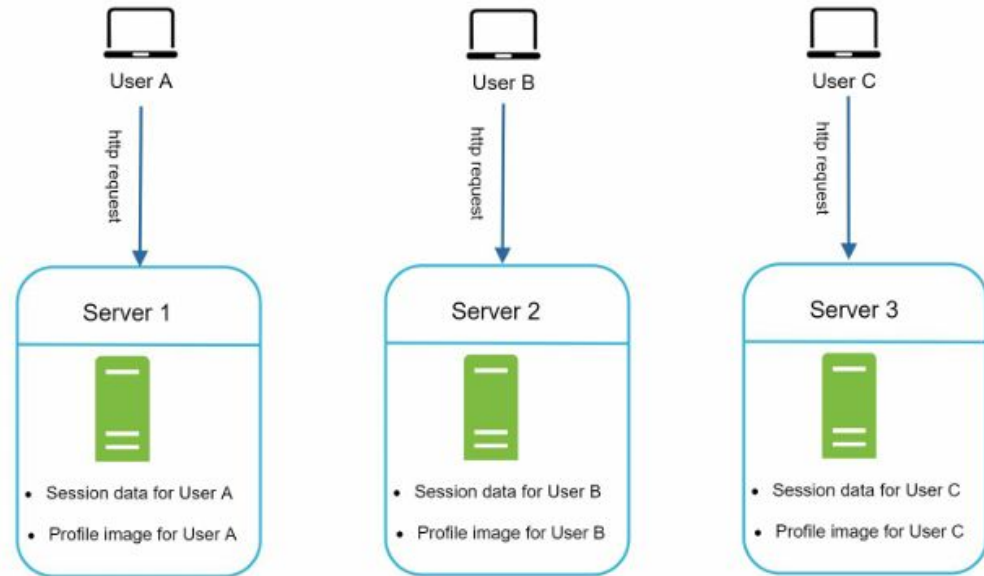
- **Stateful Server:** Maintains **client-specific data** (state) across multiple requests. If the server fails, the session is often lost unless state replication is implemented.
- **Stateless Server:** Does not retain any session information between requests. Each request is processed independently, enabling **simpler scaling and fault tolerance**.

# Stateful architecture

In a **stateful architecture** (as illustrated in Figure), session data—such as authentication tokens and profile images—is stored **locally** on the web server. For example:

- **User A's** session data resides on **Server 1**, so all HTTP requests from User A must be routed to Server 1.
- **User B** and **User C** must similarly be routed to Server 2 and Server 3, respectively.

This model requires **sticky sessions**, a configuration supported by most load balancers that ensures a user's requests are always routed to the same server.

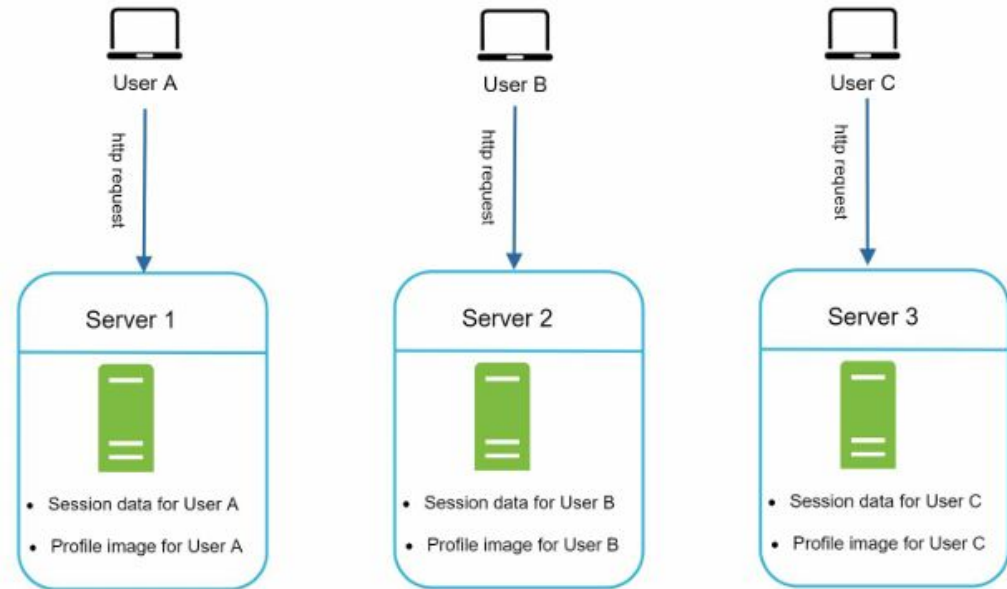


# Stateful architecture

However, this approach introduces significant limitations:

- **Increased complexity** in adding or removing servers.
- **Reduced fault tolerance**—if the designated server fails, the user session is lost.
- **Scalability constraints** due to session-server coupling.

Thus, a **stateless design**, where session data is stored in shared persistent storage, is more suitable for building resilient and horizontally scalable systems.





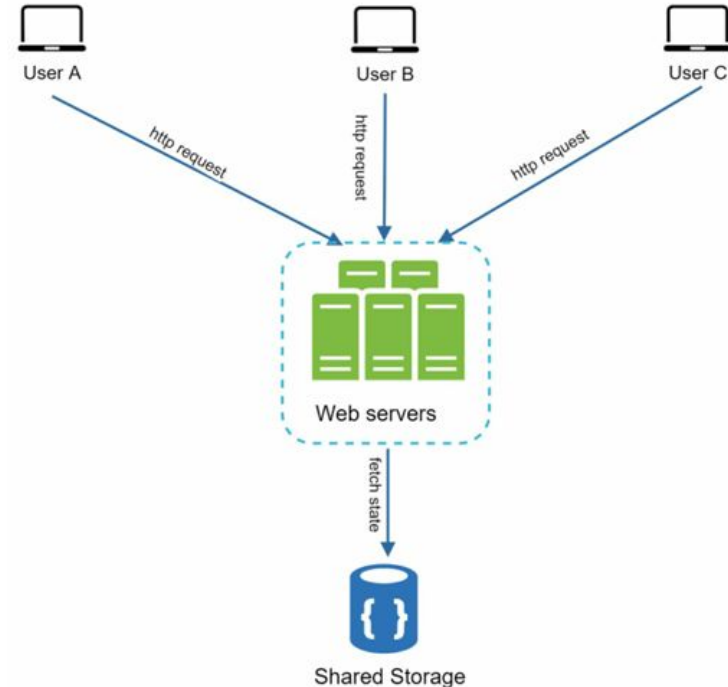
# Stateless Web Tier

In a **stateless architecture**, HTTP requests from any user can be routed to **any web server** in the cluster. This is possible because **session or state data is stored externally** in a **shared persistent data store**, not within the web server itself.

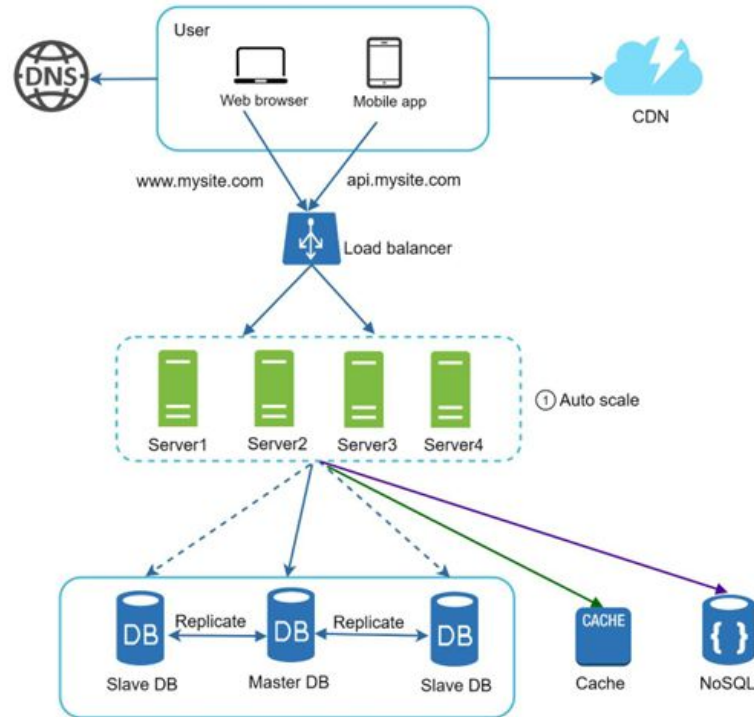
This architectural pattern offers several advantages:

- **Simplicity:** Eliminates the need for sticky sessions or session affinity.
- **Robustness:** No single point of failure tied to a specific user's session.
- **Scalability:** New servers can be added or removed without affecting session continuity.

As a result, stateless systems are **more fault-tolerant**, **easier to maintain**, and **better suited for horizontal scaling**.



# After adding the Stateless Web Tier



# After adding the Stateless Web Tier

To enable **auto-scaling** and **horizontal scalability**, we **move session data out of the web tier** and store it in a **shared persistent data store**, such as:

- Relational Databases (e.g., PostgreSQL)
- In-memory Caches (e.g., Redis, Memcached)
- NoSQL Databases (e.g., DynamoDB, MongoDB)

Among these, **NoSQL stores are often preferred** for session data due to their **scalability and performance**.

Once the web tier becomes stateless:

- **Auto-scaling** is simplified, allowing dynamic adjustment of server count based on traffic.
- **High availability** is easier to implement.

As user traffic expands globally, **deploying multiple data centers** becomes critical for:

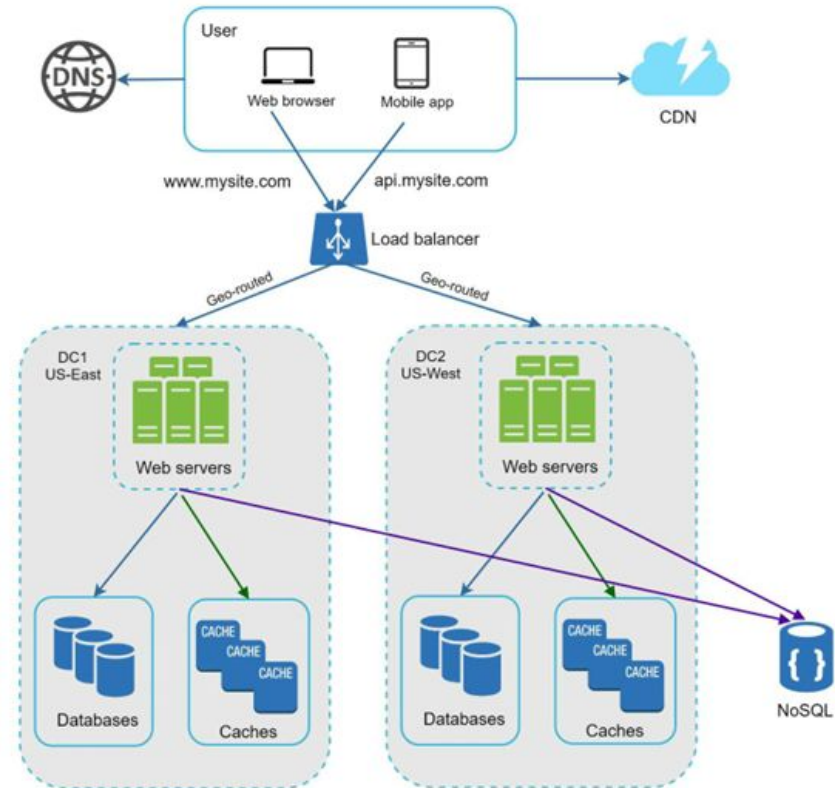
- **Reducing latency** for users across different regions
- **Improving fault tolerance**
- **Enhancing overall user experience** by serving content from geographically closer locations.

# Data centers

In a **multi-data center setup**, users are typically routed to the **nearest data center** using **GeoDNS**, a DNS service that resolves domain names based on the user's geographic location.

For example:

- Under normal conditions, traffic is **geo-routed** with a split—e.g., **x% to US-East** and **(100 – x)% to US-West**.
- **GeoDNS** ensures minimal latency by directing users to the geographically closest and most responsive data center.

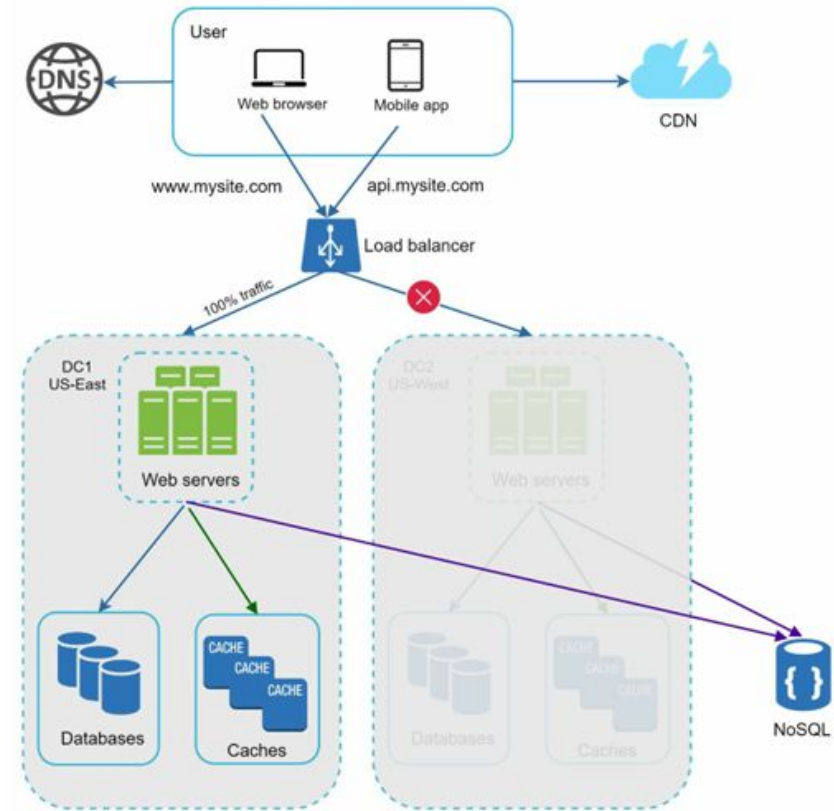


# Data centers Outage

In case of a **data center outage**:

- All traffic is **rerouted to a healthy data center**.
- As illustrated in Figure 1-16, when **US-West is offline**, **100% of the traffic is directed to US-East**, ensuring service continuity.

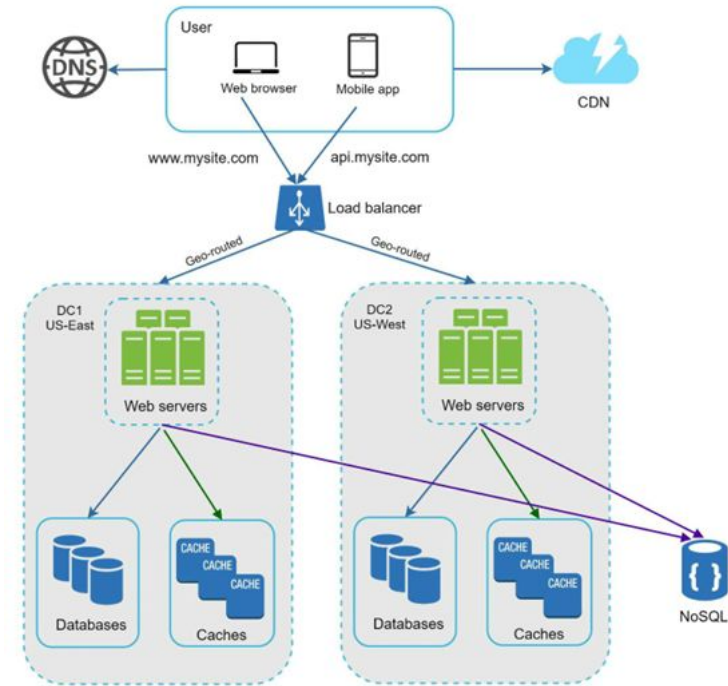
This design improves **availability**, **fault tolerance**, and **global user experience**.



# Technical Challenges in Multi-Data Center Architecture

Implementing a multi-data center setup introduces several **engineering challenges** that must be addressed to ensure reliability and scalability:

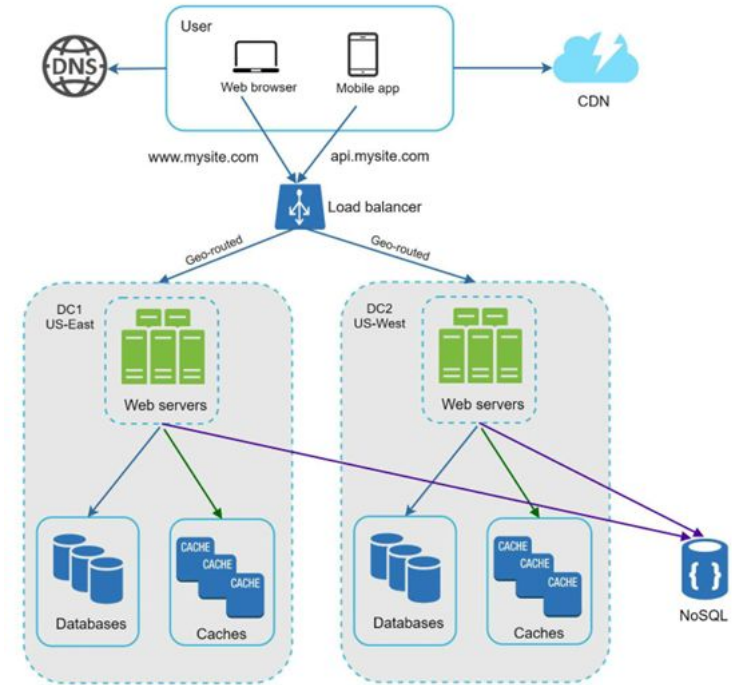
- **Traffic Redirection:**  
Efficient routing is crucial.
  - **GeoDNS** is commonly used to route users to the nearest data center based on geographic location.
  - Ensures optimal latency and performance.
- **Data Synchronization:**  
Users may interact with different **regional databases or caches**. During failover:
  - Requests could be redirected to a location **where the data is unavailable or stale**.
  - **Asynchronous replication** is widely used to keep data consistent across data centers.
  - For example, **Netflix** employs multi-region replication strategies to maintain data availability.



# Technical Challenges in Multi-Data Center Architecture

- **Testing and Deployment:**  
With services distributed across locations:
  - Comprehensive testing across **multiple geographic regions** is required.
  - **Automated CI/CD pipelines** help maintain consistency and reduce configuration drift across environments.

To support further scalability and modularity, the next architectural improvement involves **decoupling components** using **messaging queues**, which are essential in many large-scale distributed systems.



# Message Queue: Concept and Architecture

A **message queue** is a durable middleware component that enables **asynchronous communication** between services in a distributed system.

## ◆ Key Functions:

- Acts as a **buffer** for messages between **producers** and **consumers**
- Ensures **decoupling** of components by allowing them to operate **independently** and **at different speeds**
- Provides **fault tolerance** and **load leveling**

## ◆ Core Terminology:

- **Producer / Publisher:**  
The service that creates and sends messages to the queue.
- **Consumer / Subscriber:**  
The service that receives and processes messages from the queue.





# Why Message Queues?

Decoupling via message queues makes them ideal for building **scalable** and **reliable** systems.

- **Asynchronous Communication:**  
Producers can post messages even when consumers are offline.
- **Resilience:**  
Consumers can process queued messages later, even if producers are unavailable.

This decoupled architecture ensures **fault tolerance**, **independent scaling**, and **robust system behavior** under varying loads.



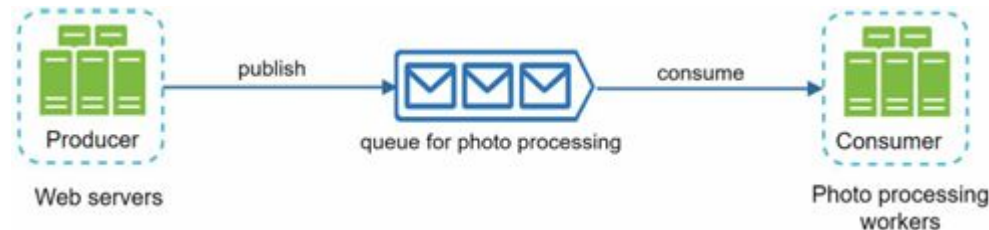
# Use Case: Photo Customization with Message Queues

**Scenario:** Your application allows users to perform photo edits (e.g., crop, sharpen, blur), which are time-consuming tasks.

**Architecture:** **Web Servers (Producers):** Submit photo processing jobs to a **message queue**. **Worker Nodes (Consumers):** Asynchronously pick up jobs from the queue and perform the edits.

**Benefits:**

- **Independent Scaling:**
  - If the queue grows → **Add more workers** to reduce latency.
  - If the queue is often empty → **Scale down workers** to save resources.
- **Improved Responsiveness:** Users get quick acknowledgment while processing continues in the background.
- **Resilience:** Decoupled design avoids blocking the main application workflow.



# Logging, Metrics, and Automation

As your system scales from a simple application to a business-critical platform, the following become essential:

## Logging

- Enables tracking and diagnosing errors across servers.
- Aggregated logging systems (e.g., ELK Stack, Graylog) simplify monitoring and querying logs centrally.

## Metrics

Track system and business health through:

- **Host-level metrics:** CPU usage, memory consumption, disk I/O.
- **Service-level metrics:** Performance of DB tier, cache tier, web tier.
- **Business metrics:** Daily active users (DAU), retention rate, revenue trends.

# Logging, Metrics, and Automation

## Automation

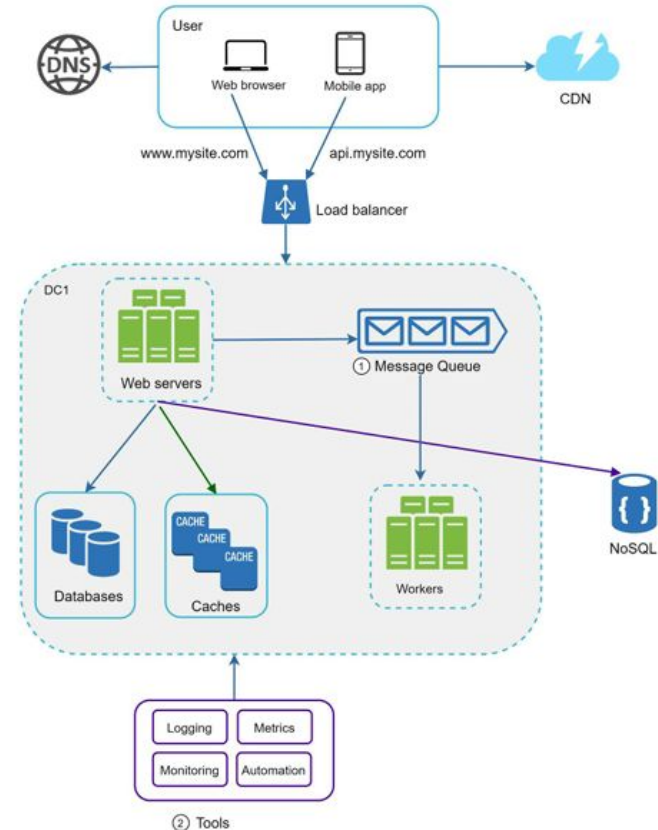
- Reduces human error and boosts productivity in large-scale deployments.
- **Continuous Integration (CI):** Every code commit is automatically built and tested.
- **Deployment Automation:** Streamlines build, test, and deployment pipelines (e.g., using Jenkins, GitLab CI/CD, Ansible).

**Conclusion:** Together, logging, metrics, and automation provide observability, stability, and scalability—crucial pillars of modern system design.

# After adding message queues and different tools

*Note: For clarity, the diagram shows a single data center, though the design supports multiple.*

1. The design includes a message queue, which helps to make the system more loosely coupled and failure resilient.
2. Logging, monitoring, metrics, and automation tools are included.



# Database Scaling

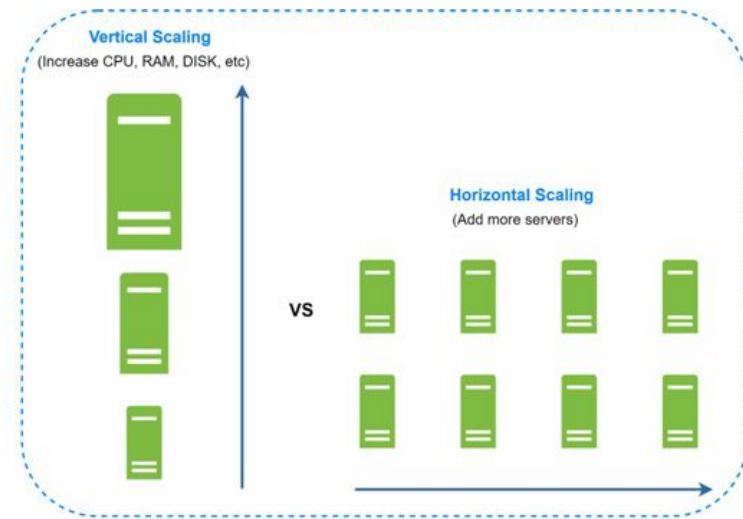
As data volume and user traffic grow, it becomes essential to scale the data tier. There are two primary approaches:

## Vertical Scaling (Scale-Up)

- Involves adding more resources (CPU, RAM, SSD) to a single server.
- Example: Amazon RDS offers instances with up to 24 TB RAM.
- **Drawbacks:**
  - Hardware limitations exist—cannot scale indefinitely.
  - Higher risk of single point of failure.
  - Expensive compared to distributed alternatives.

## Horizontal Scaling (Scale-Out / Sharding)

- Distributes data across multiple servers or shards.
- Improves scalability, availability, and fault tolerance.
- More cost-efficient and widely used in large-scale systems.



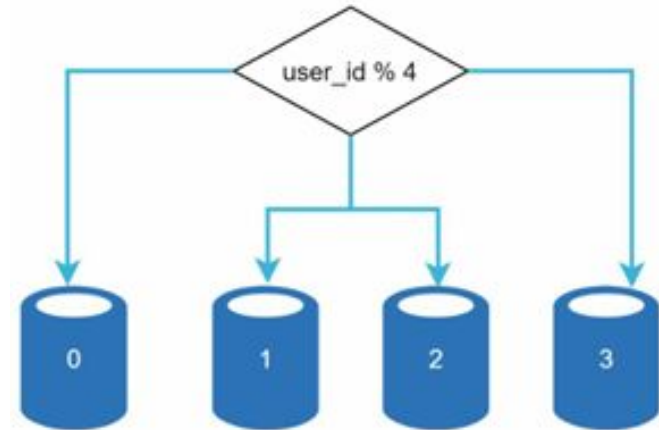
# Sharding (Horizontal Database Scaling)

**Sharding** is the practice of partitioning a large database into smaller, manageable pieces called **shards**. Each shard:

- Has the **same schema**,
- Contains **unique subsets of data**, based on a sharding key.

## Example: User-Based Sharding

- Data is distributed using a hash function like:  $\text{user\_id} \% 4$
- Result determines which shard stores or retrieves the user's data:
  - $\text{user\_id} \% 4 = 0 \rightarrow \text{Shard 0}$
  - $\text{user\_id} \% 4 = 1 \rightarrow \text{Shard 1}$
  - and so on...

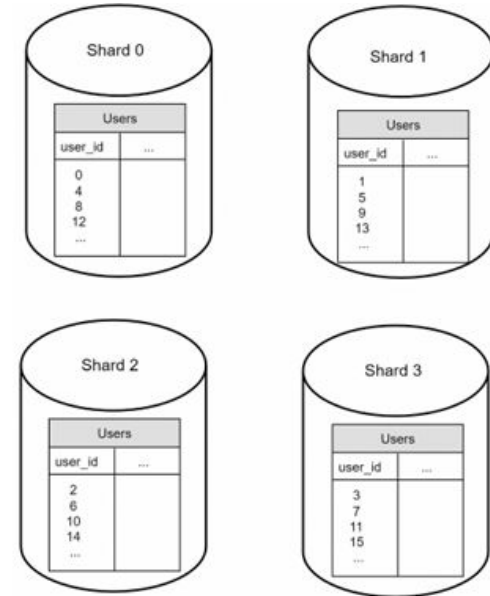


**Benefit:** Sharding improves performance, scalability, and fault isolation in large-scale systems.

# Choosing a Sharding Key

- The **sharding key** (or partition key) is a column (or set of columns) that determines how data is distributed across shards.
- A **well-chosen sharding key**:
  - Ensures **even data distribution**, avoiding overloaded shards (hotspots).
  - Allows **efficient routing** of queries to the appropriate shard.
  - Enables **scalability** and **performance** as the dataset grows.

**Example:** If `user_id` is the sharding key, then queries like `SELECT * FROM users WHERE user_id = 5` can be routed directly to the correct shard using a hash function like `user_id % number_of_shards`.





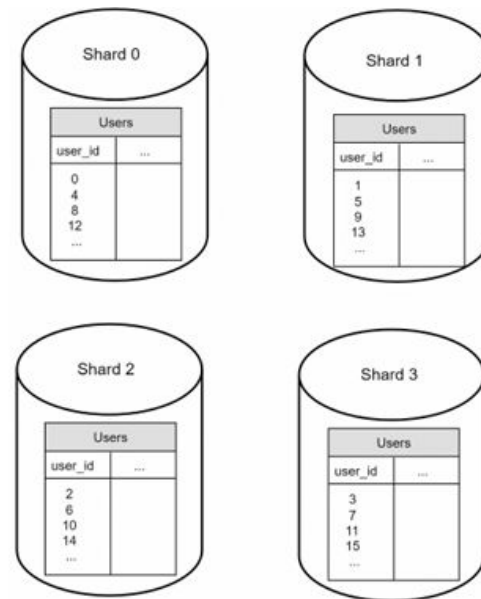
# Challenges of Sharding

## Resharding Data

- Needed when:
  - A shard exceeds its storage or performance limit.
  - Data is unevenly distributed (some shards grow faster).
- Requires:
  - Updating the sharding logic.
  - Migrating existing data.
- **Solution:** Use **consistent hashing** to reduce data movement and maintain balance.

## Celebrity Problem (Hotspot Key Issue)

- Excess traffic to a single shard due to popular entities (e.g., Katy Perry, Justin Bieber).
- Can lead to **server overload**.
- **Solution:**
  - Assign **dedicated shards** to high-traffic entities.
  - Consider **further partitioning** those shards if needed.



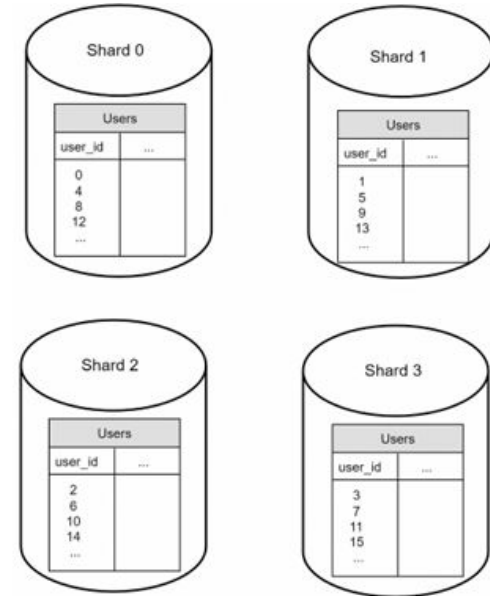
# Challenges of Sharding

## Join and Denormalization Issues

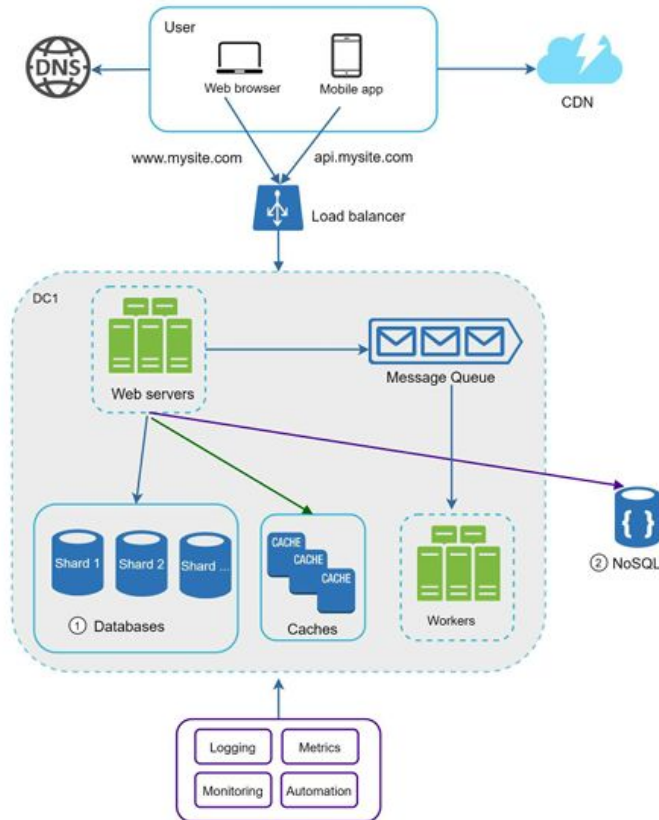
- **Joins** across shards are complex and expensive.
- **Workaround:**
  - Use **denormalization** to store related data together in a single table.
  - Reduce need for cross-shard queries.

## Hybrid Strategy

- Combine relational sharding with **NoSQL stores** for unstructured or high-volume data.
- Offloading such tasks can **reduce RDBMS load** and improve scalability.



# After adding Database Scaling



# Scaling to Millions of Users and Beyond

Scaling a system for millions of users is an ongoing, iterative process. As your system grows, fine-tuning and further modularization become essential.

## Key Principles for Large-Scale System Design:

- **Stateless Web Tier:**  
Decouple session state from web servers to enable auto-scaling and resilience.
- **Redundancy at Every Tier:**  
Ensure failover support for web, application, and database layers.
- **Aggressive Caching:**  
Cache frequently accessed or expensive data to reduce latency and database load.
- **Multi-Data Center Support:**  
Deploy across geographies for high availability and lower latency.
- **Content Delivery Network (CDN):**  
Offload static assets (JS, CSS, images, videos) to reduce server load and speed up delivery.
- **Sharded Data Tier:**  
Partition data to distribute load and scale horizontally.
- **Microservices Architecture:**  
Decouple monolith into manageable, independently scalable services.
- **Monitoring & Automation:**  
Employ observability tools and CI/CD pipelines to ensure reliability and efficiency.

# BACK-OF-THE-ENVELOPE ESTIMATION

# Back-of-the-Envelope Estimation in System Design

In system design interviews and real-world architecture planning, engineers are often required to **estimate system capacity or performance** through **quick, approximate calculations**—known as *back-of-the-envelope estimations*.

*“Back-of-the-envelope calculations are estimates you create using a combination of thought experiments and common performance numbers to get a good feel for which designs will meet your requirements.”*

— **Jeff Dean**, Google Senior Fellow

## Purpose:

To **quickly evaluate** whether a system design can meet scalability, latency, or availability targets before committing to detailed implementation.

The following concepts should be well understood: **power of two**, **latency numbers** every programmer should know, and **availability numbers**.

## Use Cases:

- Estimating **daily data volume** for a million users
- Determining if a **single server** can handle QPS (queries per second)
- Judging whether **network latency** will be a bottleneck

# Latency Numbers

Originally compiled by **Dr. Jeff Dean (Google)** in 2010, these latency values provide insight into the **relative speed** of common computing operations. While modern systems may show faster performance, the **order of magnitude remains relevant** for system design estimations.

## Time Unit Reference

- **1 ns** (nanosecond) =  $10^{-9}$  seconds
- **1  $\mu$ s** (microsecond) = 1,000 ns
- **1 ms** (millisecond) = 1,000  $\mu$ s = 1,000,000 ns

## Why It Matters

- Helps with **back-of-the-envelope estimations**.
- Informs architectural decisions like caching, batching, and replication.
- Highlights trade-offs in **latency-sensitive** systems such as real-time applications and distributed architectures.

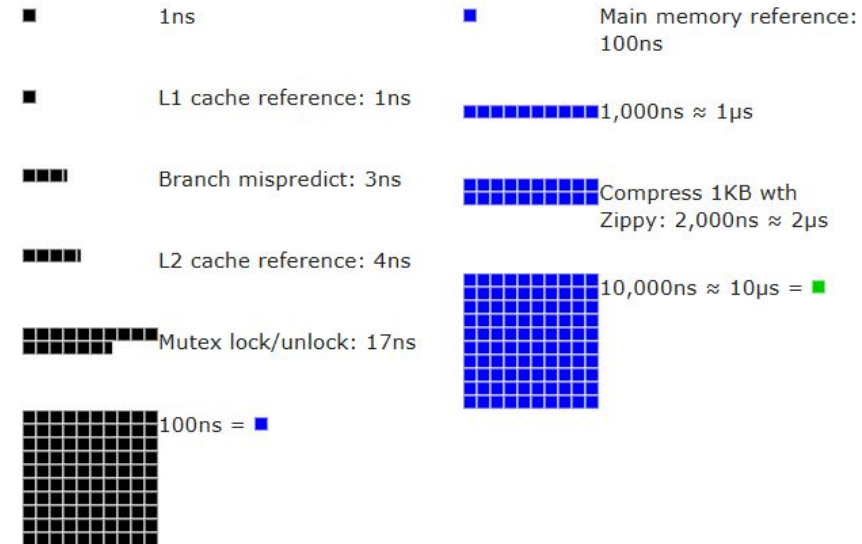
Operation	Approx. Latency
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1 KB using Zippy	10 $\mu$ s
Send 2 KB over 1 Gbps network	20 $\mu$ s
Read 1 MB sequentially from memory	250 $\mu$ s
Round trip within the same data center	500 $\mu$ s
Disk seek	10 ms
Read 1 MB sequentially from the network	10 ms
Read 1 MB sequentially from disk	30 ms
Send packet CA $\leftrightarrow$ Netherlands (round trip)	150 ms

# Visualizing Latency: Key Insights

A Google software engineer created a **visual tool** to illustrate **Dr. Jeff Dean's latency numbers** in relative scale, updated for modern hardware (circa 2020). The tool incorporates **time-based perspective**, allowing engineers to intuitively grasp the **relative cost of operations**.

## Insights from the Visualization

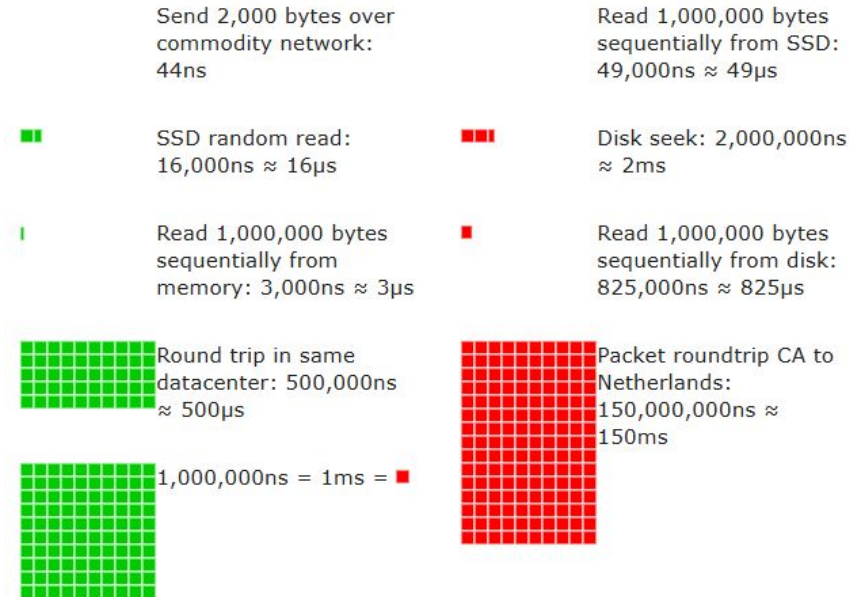
- **Memory operations are extremely fast**, while **disk I/O remains relatively slow**.
- **Disk seeks** are particularly expensive—**avoid them when possible**.
- **Simple compression algorithms** (e.g., Zippy) are fast enough to justify use in I/O-heavy tasks.
- **Compressing data** before transmission can significantly reduce network latency and bandwidth costs.
- **Data center latency** is non-negligible due to geographical distance—**inter-region traffic should be minimized** for real-time applications.



Ref: [Numbers Every Programmer Should Know By Year](#)



# Visualizing Latency: Key Insights



Ref: [Numbers Every Programmer Should Know By Year](#)

# Availability Numbers

**High availability (HA)** refers to a system's ability to remain operational and accessible for a **consistently long period**, minimizing downtime.

## What Is High Availability?

- Measured as a **percentage of uptime**.
- **100% availability** = zero downtime (theoretical ideal).
- In practice, most services aim for **99% and above**.

## Service Level Agreement (SLA)

An **SLA** is a contractual guarantee of service uptime provided by a vendor or service provider.

Major cloud providers typically offer SLAs of:

- **Amazon Web Services (AWS):**  $\geq 99.99\%$
- **Google Cloud Platform (GCP):**  $\geq 99.9\%$
- **Microsoft Azure:**  $\geq 99.9\%$

Availability	Downtime per Year	Downtime per Month	Downtime per Day
99%	~3.65 days	~7.3 hours	~14.4 minutes
99.9%	~8.76 hours	~43.8 minutes	~1.44 minutes
99.99%	~52.6 minutes	~4.38 minutes	~8.6 seconds
99.999%	~5.26 minutes	~26.3 seconds	~0.86 seconds

📌 *The more the nines, the higher the reliability—and the greater the cost and complexity to achieve it.*

Week 1: Introduction to System Design

# Example: Estimate Youtube QPS and Storage Requirements

*Note: These values are fictional and for exercise purposes only.*

## Assumptions:

- 1 billion monthly active users
- 60% of users are active daily
- Each user watches 5 videos and uploads 0.01 videos per day
- 15% of uploaded videos are in 4K
- Videos are stored for 10 years
- Average video durations:
  - Standard: 100 MB
  - 4K: 500 MB

## Estimations: Query Per Second (QPS):

- **Daily Active Users (DAU)** =  $1B * 60\% = 600 \text{ million}$
- **Video view QPS** =  $600M * 5 / (24 * 3600) \approx 34,700$
- **Peak QPS** =  $2 * 34,700 = \sim 70,000$

## Storage Requirements: Uploads per Day:

- **Uploads** =  $600M * 0.01 = 6 \text{ million videos/day}$
- **4K videos** =  $6M * 15\% = 900,000 \text{ videos}$
- **Standard videos** =  $6M * 85\% = 5.1 \text{ million videos}$

## Daily Storage:

- **Standard** =  $5.1M * 100 \text{ MB} = 510 \text{ TB/day}$
- **4K** =  $900k * 500 \text{ MB} = 450 \text{ TB/day}$
- **Total** =  $510 + 450 = 960 \text{ TB/day}$

## 10-Year Storage:

- $960 \text{ TB/day} * 365 * 10 \approx 3.5 \text{ EB (Exabytes)}$

## Summary:

- Estimated QPS:  $\sim 70,000$  (peak video views)
- Storage:  $\sim 960 \text{ TB/day}$
- Long-term storage (10 years):  $\sim 3.5 \text{ Exabytes}$

# Workshop + Linux Account

- Apply System Design Framework to replicate Big Giants.
- Linux account will be provided today.

Thank you!  
*Any Questions?*