SDE: System Design and Engineering

Lecture – 6

Introduction to

Caching, Performance and OS

From Zero to Google: Architecting the Invisible Infrastructure

by

Aatiz Ghimire

Sections

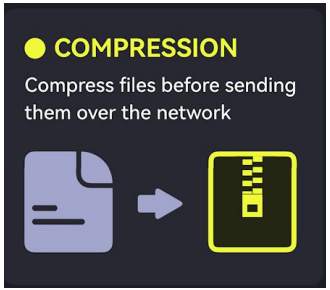
- Performance Metrics
- Content Delivery Networks (CDNs)
- Caching Fundamentals and Strategies
- Cache Eviction and Performance Pitfalls
- In-Memory Caching with Redis and Memcached
- Few OS Concepts

Performance Optimization in Frontend Codes

Boost your website's speed with proven optimization strategies.

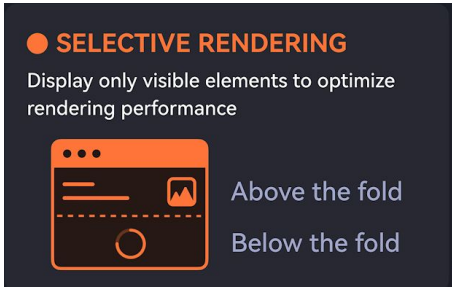
1. Compression

- Minimize file sizes (HTML, CSS, JS, images)
- Use Gzip or Brotli to reduce network load



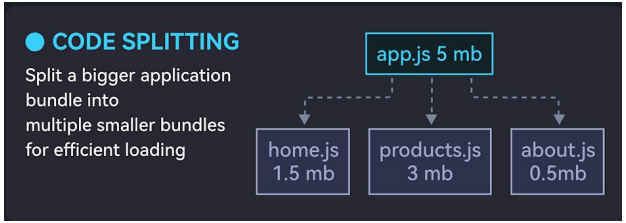
2. Selective Rendering / Windowing

- Render only visible elements in dynamic lists
- Great for long scrollable content (e.g., chat apps, product listings)



3. Modular Architecture with Code Splitting

- Break large bundles into smaller, manageable chunks
- Improves load time and reduces memory usage



Performance Optimization in Frontend Codes

4. Priority-Based Loading

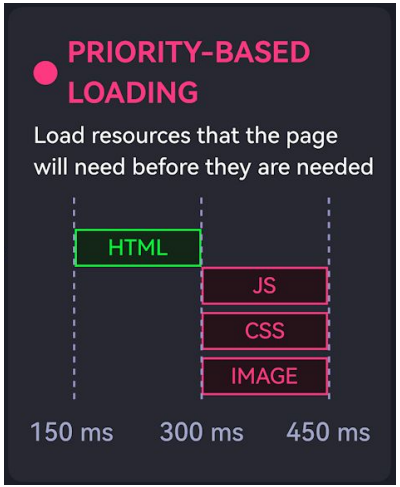
- Load critical (above-the-fold) content first
- Defer non-essential scripts and styles

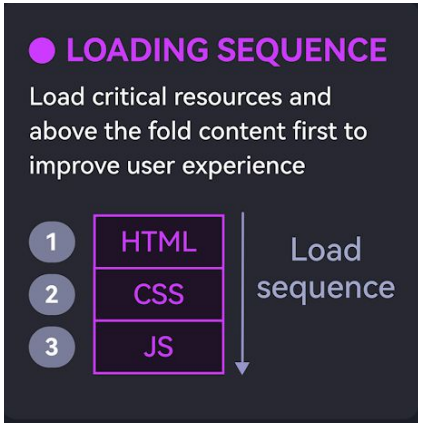
5. Pre-loading

- Fetch important resources early
- Improves user-perceived performance

6. Tree Shaking / Dead Code Elimination

- Remove unused JavaScript code from final build
- Keeps bundles lean and efficient

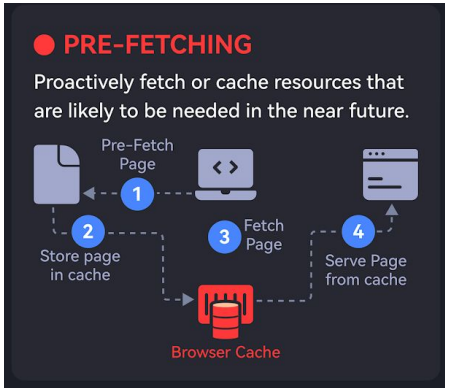




Performance Optimization in Frontend Codes

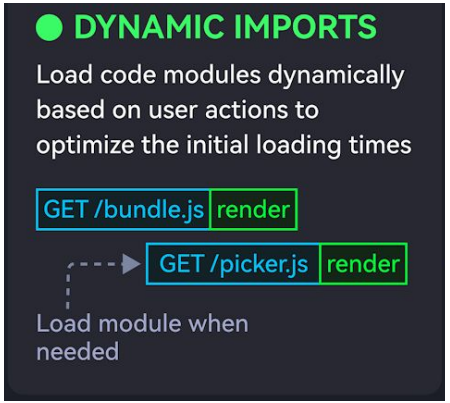
7. Pre-fetching

- Predict and load likely next resources in the background
- Enhances navigation speed and responsiveness



8. Dynamic Imports

- Load modules only when needed
- Reduces initial page load time



Website Performance Metrics

1. Load Time

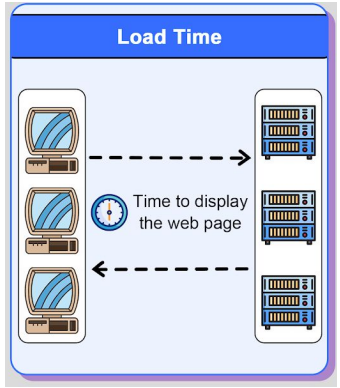
- Total time for a browser to download and render a webpage
- Measured in **milliseconds**
- Key metric for user perception of site speed

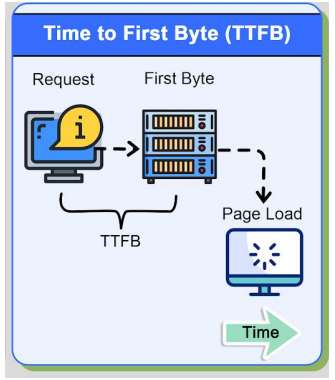
2. Time to First Byte (TTFB)

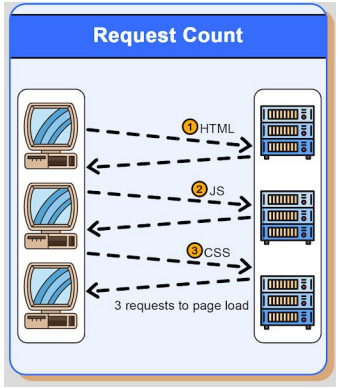
- Time until the **first byte** is received from the server
- Reflects **server responsiveness** and **backend efficiency**

3. Request Count

- Number of **HTTP requests** made to fully load a page
- Fewer requests → faster load time



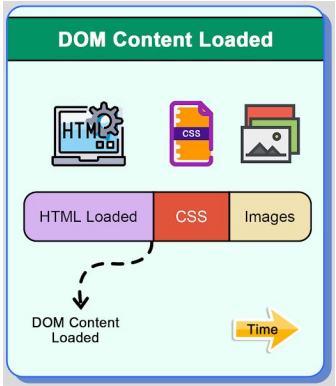


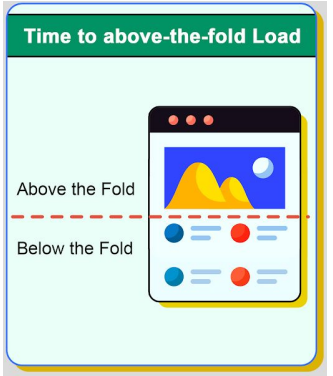


Website Performance Metrics

4. DOMContentLoaded (DCL)

- Time taken to load and parse the full **HTML** document
- Doesn't include external assets (e.g., CSS, JS)
- Key to enabling user interaction



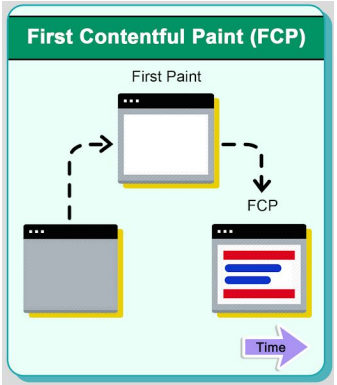


5. Time to Above-the-Fold Load

- Time to render **visible content** without scrolling
- Crucial for **first impressions** and **engagement**

6. First Contentful Paint (FCP)

- Time when **first visible content** (text, image, etc.) appears
- A good FCP improves user-perceived performance



Website Performance Metrics

7. Page Size

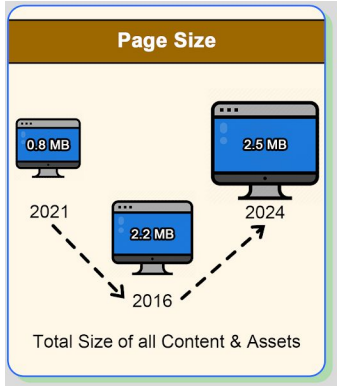
- Total size of all content/assets on the page
- Larger pages take **longer to load**, especially on mobile

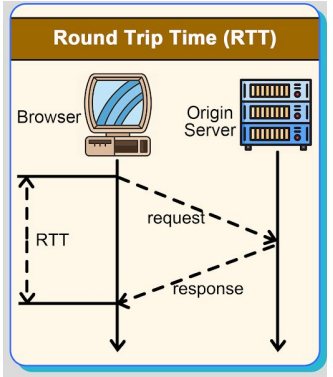
8. Round Trip Time (RTT)

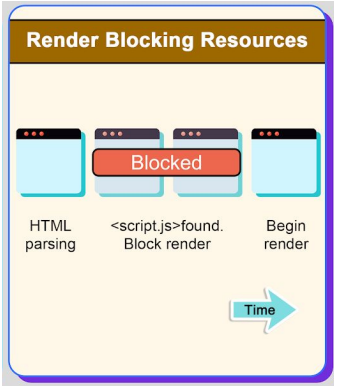
- Time for a request to travel to the server and back
- Lower RTT = faster **data retrieval**

9. Render-Blocking Resources

- Scripts/styles that **delay rendering** of visible content
- Should be **minimized or deferred**



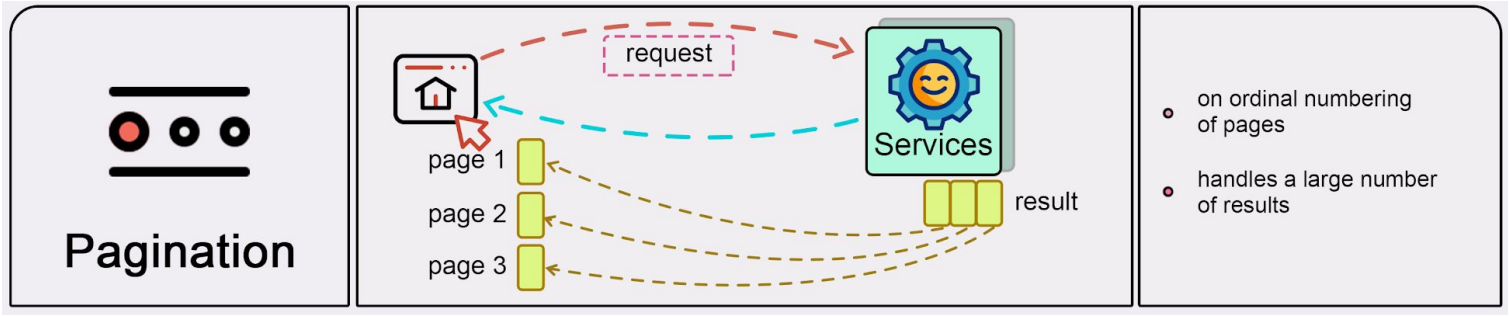




Ways to Improve API Performance

1. Result Pagination

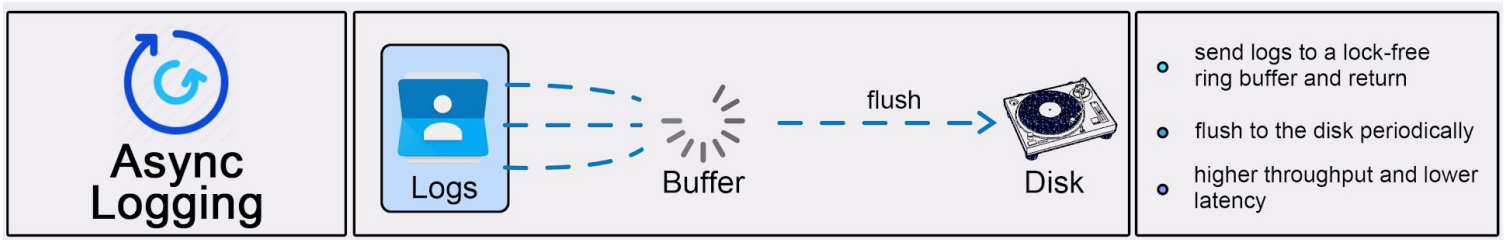
- Stream large datasets in **smaller, manageable chunks**
- Improves **response time** and **user experience**
- Ideal for list views, feeds, search results, etc.



Ways to Improve API Performance

2. Asynchronous Logging

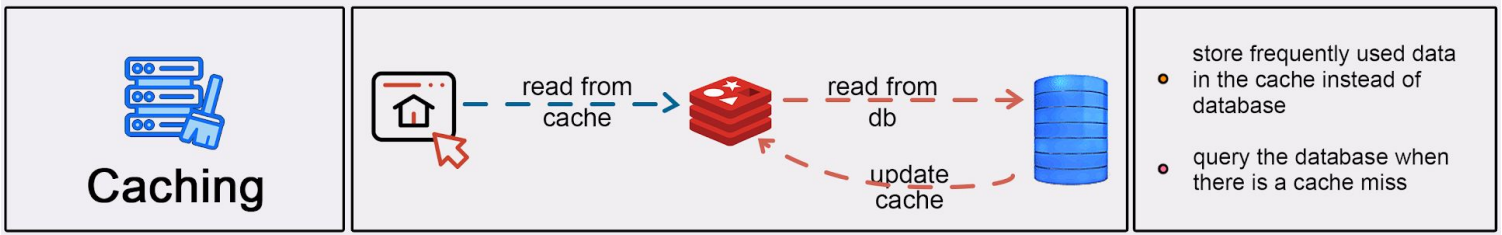
- Logs are written to a **lock-free buffer** instead of the disk immediately
- Disk writes are **batched and delayed**, reducing **I/O overhead**
- Helps maintain high throughput during heavy API usage



Ways to Improve API Performance

3. Data Caching

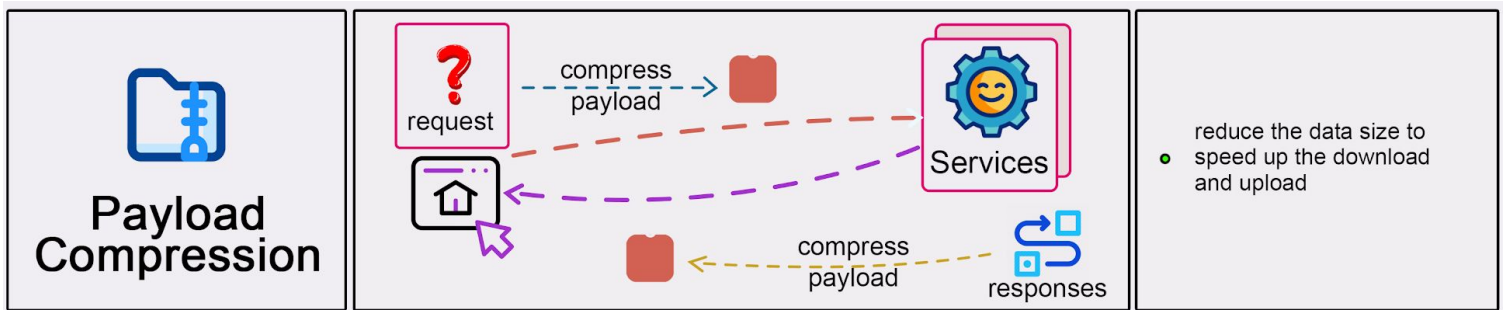
- Store frequently requested data in **memory** (e.g., Redis)
- Clients **check cache** before querying the backend
- Reduces **database load** and boosts response time



Ways to Improve API Performance

4. Payload Compression

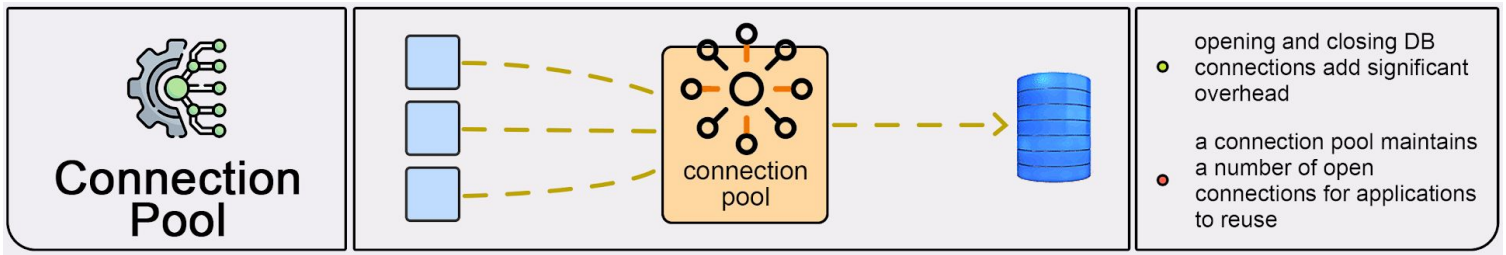
- Use compression (like **gzip**) to shrink request/response size
- Minimizes **network latency** and **bandwidth usage**
- Especially useful for large JSON/XML responses



Ways to Improve API Performance

5. Connection Pooling

- Reuse a **pool of database connections** rather than creating new ones
- Reduces latency and system load
- Ensures better **resource utilization** and **scalability**



Content Delivery Networks (CDNs)

A **CDN** is a globally distributed network of **edge servers** that deliver web content (images, videos, scripts, etc.) to users from **locations nearest to them**.

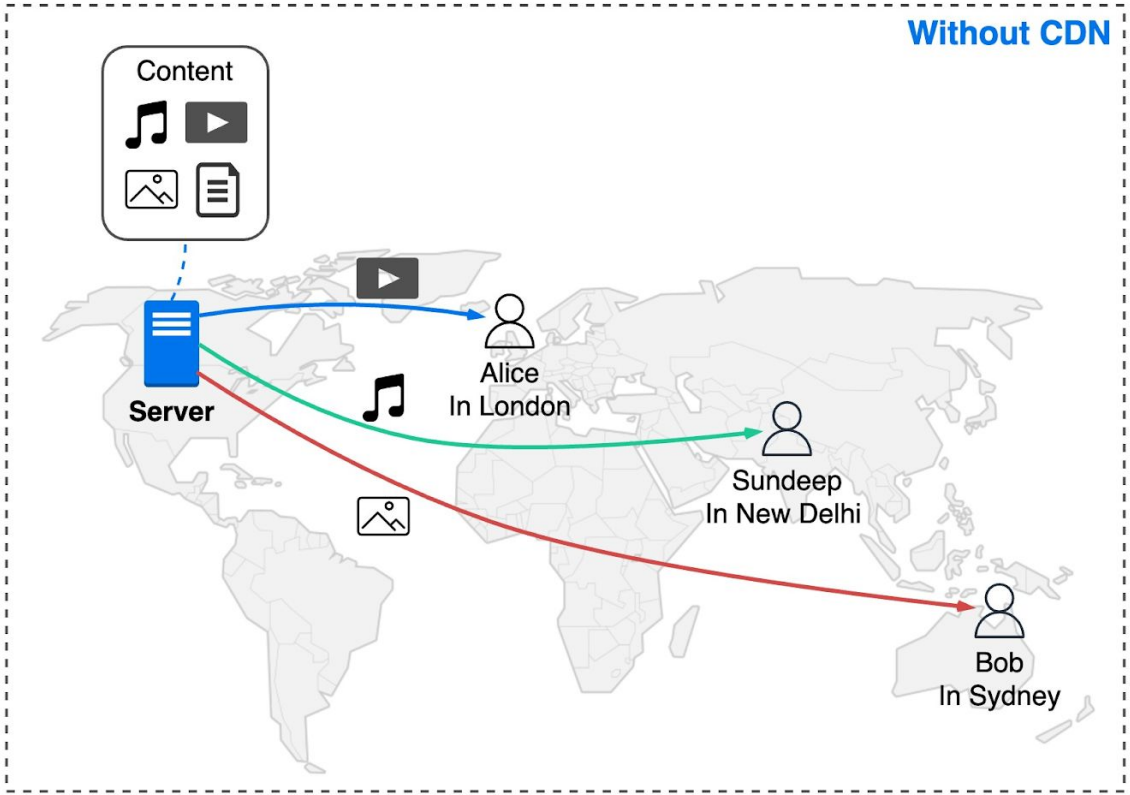
How It Works

- Instead of retrieving data from a single **origin server**, users access **cached content** from nearby **CDN nodes**.This reduces the **physical distance** data must travel, ensuring faster delivery.

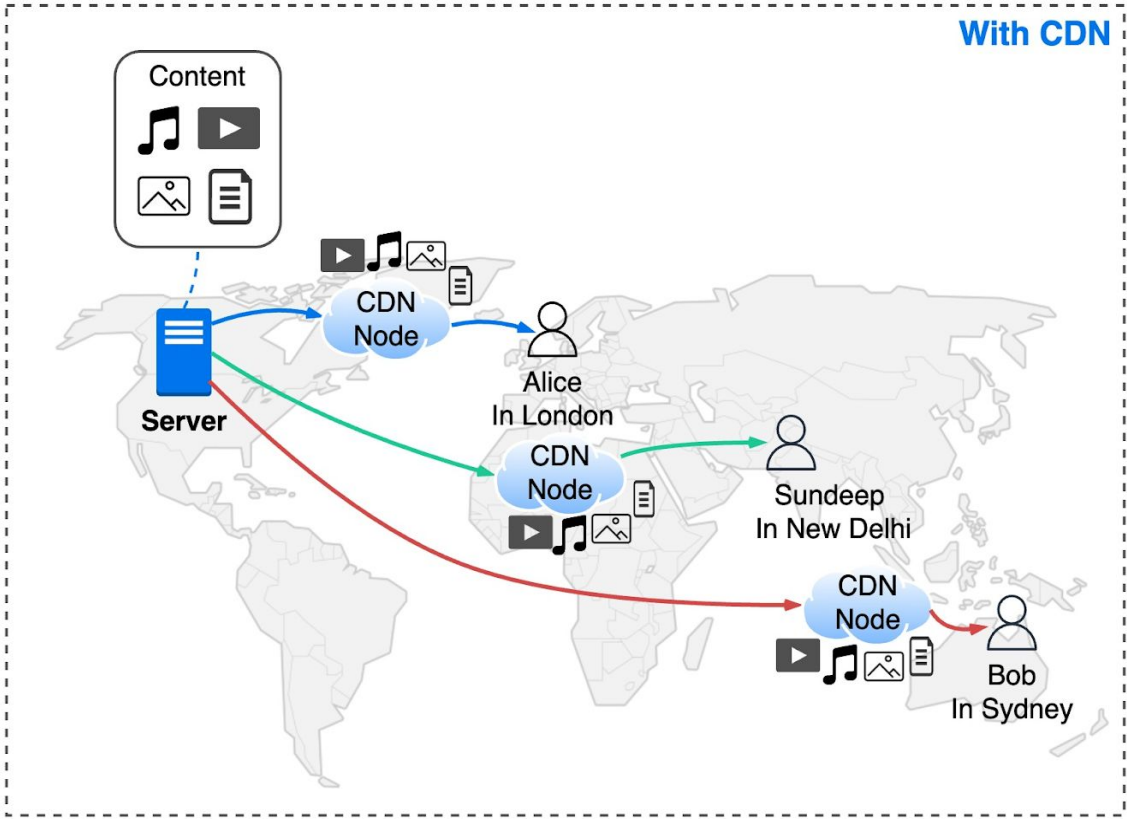
Key Benefits

- **Reduced Latency:** Content loads faster by minimizing the travel time between server and user.
- **Lower Bandwidth Usage:** Cached content reduces repeated server hits and conserves network bandwidth.
- **Improved Security:** CDNs help **mitigate DDoS attacks**, offering a buffer between origin servers and malicious traffic.
- **Higher Content Availability:** Content is **replicated globally**, ensuring high uptime and availability during high traffic or server failures.

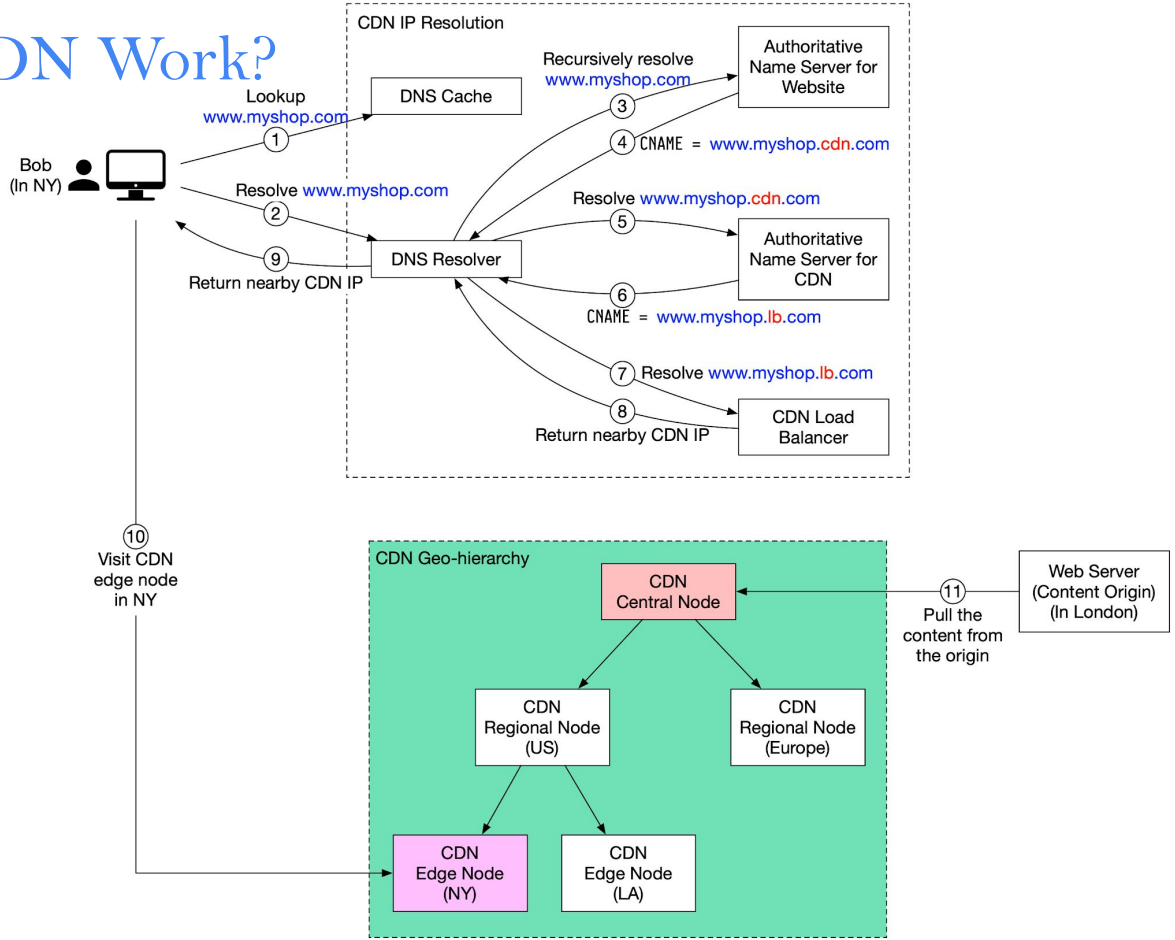
Without Content Delivery Networks (CDNs)



With Content Delivery Networks (CDNs)



How Does CDN Work?



How Does CDN Work?

Step-by-Step Workflow

1. **DNS Lookup Starts**
   - Bob types www.myshop.com
   - Browser checks **local DNS cache**

2. **DNS Resolution**
   - If not cached, browser queries **ISP DNS resolver**
   - Resolver queries the **authoritative DNS server**

3. **CDN Redirect via CNAME**
   - Instead of returning origin IP, DNS returns an **alias**:
     - www.myshop.com → www.myshop.cdn.com

How Does CDN Work?

4. CDN Load Balancing

- Resolver asks CDN's authoritative server for www.myshop.lb.com
- Load balancer selects the **best CDN edge server**:
  - Based on user IP, ISP, content type, and load conditions

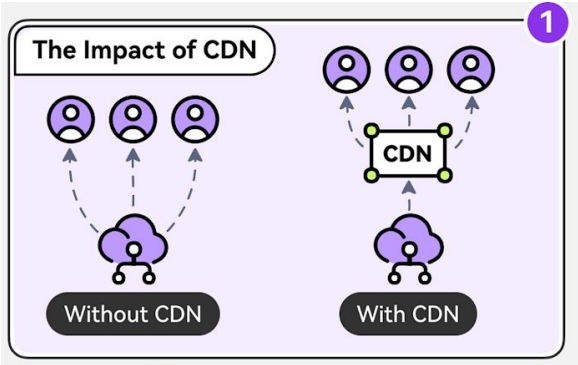
5.Edge Server Response

- IP of edge server returned to browser
- Browser fetches content from edge node

Why Are Content Delivery Networks (CDNs) So Popular?

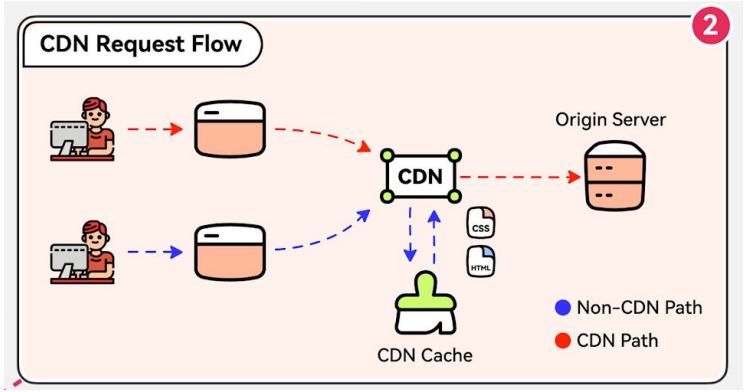
Impact of CDN on Web Performance

- **Improved Performance**: Reduced content access time
- **Increased Availability**: Redundant and distributed architecture
- **Lower Bandwidth Costs**: Offloading traffic from origin servers
- **Reduced Latency**: Users connect to **nearest edge server**



CDN Request Flow

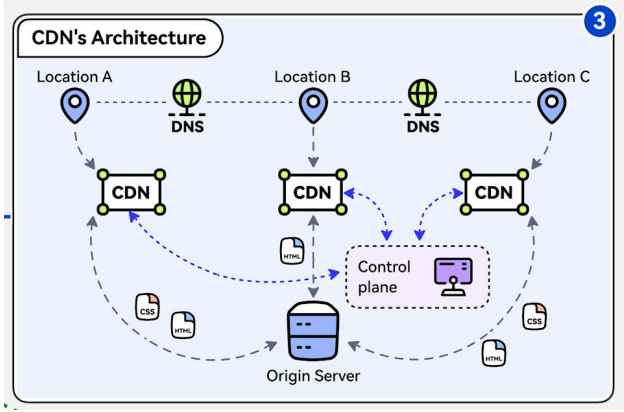
1. **DNS resolves** to nearest edge server
2. Edge server checks its **local cache**
3. If **cache hit** → serve content directly
4. If **cache miss** → fetch from **origin server**, cache it, and serve user



Why Are Content Delivery Networks (CDNs) So Popular?

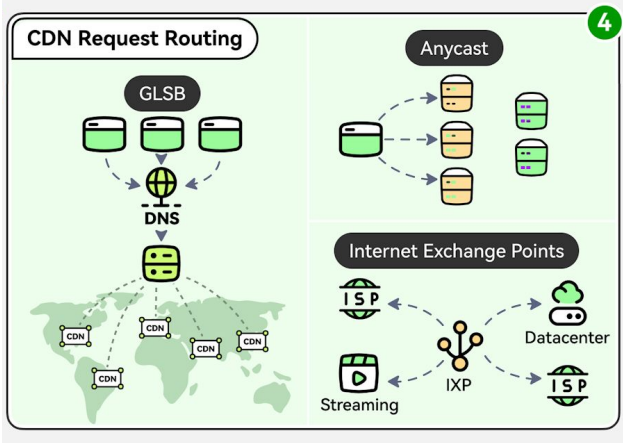
CDN Architecture Overview

- **Origin Server**: Master content source
- **Edge Servers**: Globally distributed caches
- **DNS**: Resolves domain to nearest CDN edge
- **Control Plane**: Manages and configures edge servers



CDN Request Routing Technologies

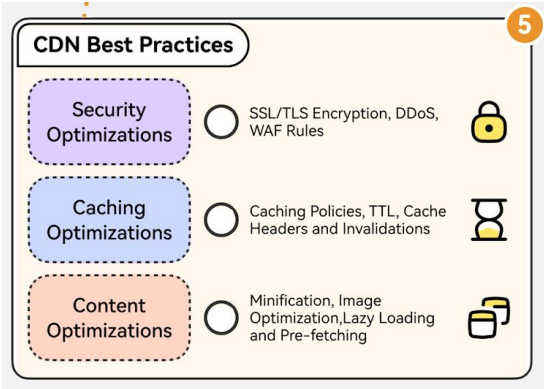
- **GSLB (Global Server Load Balancing)**
  - Routes based on proximity, load, and network health
- **Anycast DNS**
  - Multiple servers share same IP → traffic routed to **nearest**
- **Internet Exchange Points (IXPs)**
  - Direct peering with ISPs to reduce latency



Why Are Content Delivery Networks (CDNs) So Popular?

Best Practices for Optimizing CDN Usage

- Use **HTTPS with CDN** for security
- Enable **caching policies** for static assets
- Apply **content minification & compression**
- Monitor **cache hit ratios** and optimize TTLs



Caching Fundamentals and Strategies

What is a Cache?

- In-memory storage system
- Purpose: speed up data retrieval
- Concepts: Cache hit & miss

Why Do We Need to Cache?

- Improve performance & reduce latency
- Amdahl's Law: Optimizing the bottleneck
- Pareto Distribution: 80/20 rule in caching

Where is Caching Used?

1. **Hardware-Level Caching**
- L1, L2, L3 CPU caches
- Page cache in OS

3. Software-Level Caching

- Buffer cache in databases
- CDN for global content delivery
- Application-level tools: Memcached, Redis

Deployment Options

- In-process cache (local memory)
- Inter-process cache (shared in VM/container)
- Remote cache (e.g., Redis cluster)

Caching Fundamentals and Strategies

Caching Patterns

- Cache-aside
- Read-through
- Write-around
- Write-through
- Write-back

Cache Replacement Policies

- LRU (Least Recently Used)
- LFU (Least Frequently Used)
- Allowlist policy

Invalidation Techniques

- Active invalidation on modify/read
- TTL (Time-to-Live)
- Cache busting

Sharding Methods

- Modulus-based sharding
- Range-based sharding
- Consistent hashing

Common Pitfalls

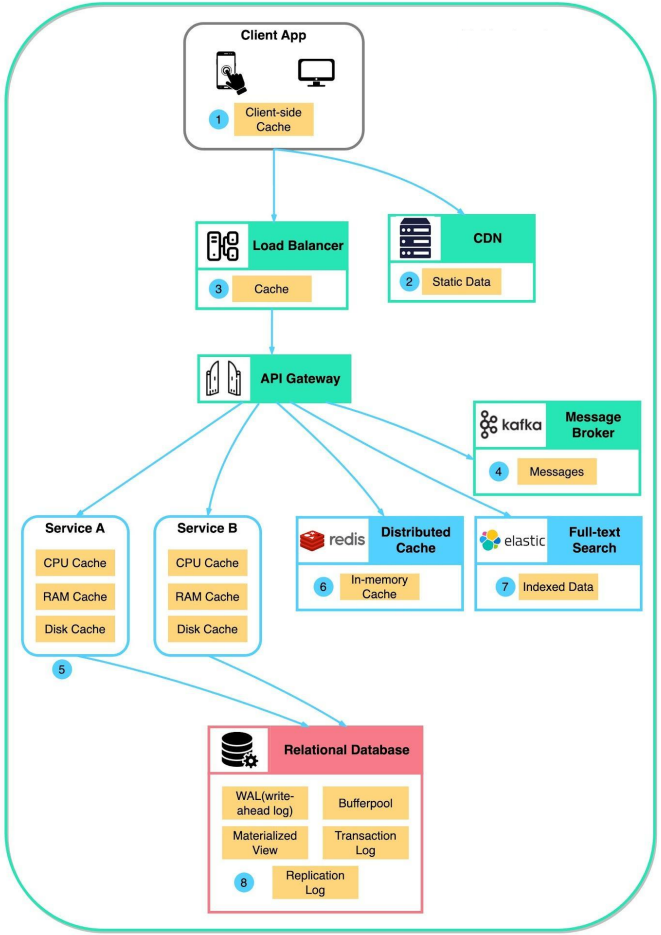
- Thundering herd
- Cache penetration
- Big key / Hot key problems
- Data consistency issues

Observability in Caching

- Metrics to track: hit/miss ratio, eviction rate
- Alerts for stale/invalid data
- Integration with Prometheus, Grafana

Caching Layers

1. **Client Apps**: Browsers cache HTTP responses. Server responses include caching directives in headers. Upon subsequent requests, browsers may serve **cached data** if still fresh.
2. **Content Delivery Networks**: CDNs cache **static content** like images, stylesheets, and JavaScript files. They serve cached content from locations **closer to users**, reducing **latency** and **load times**.
3. **Load Balancers**: Some load balancers cache **frequently requested data**. This allows serving responses without engaging **backend servers**, reducing load and response times.
4. **Message Brokers**: Systems like **Kafka** can cache messages on disk per a **retention policy**. Consumers then pull messages according to their own schedule.



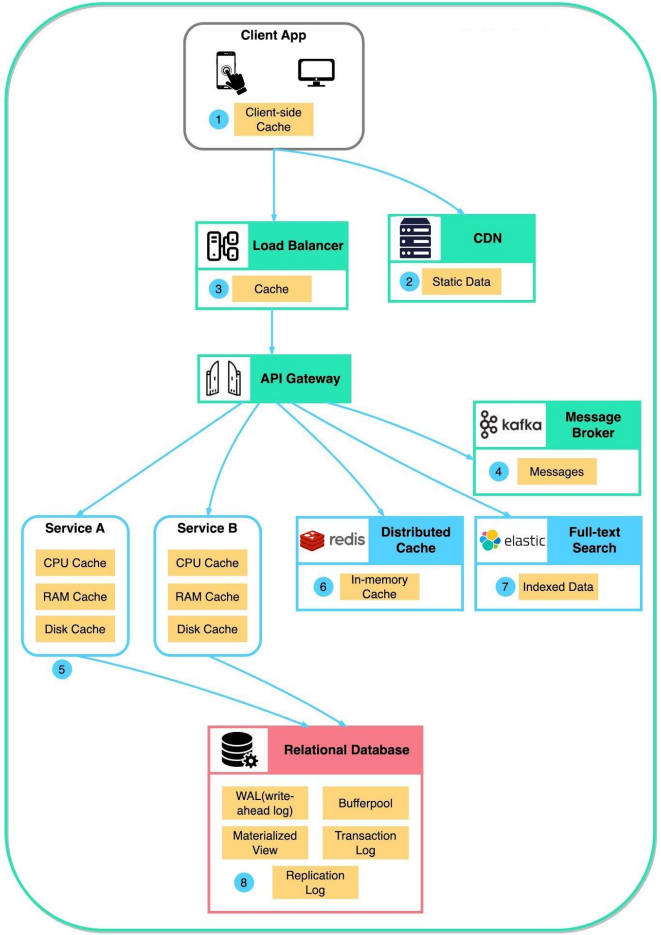
Caching Layers

5.Services: Individual services often employ caching to improve **data retrieval speeds**, first checking **in-memory caches** before querying databases. Services may also utilize **disk caching** for larger datasets.

6. Distributed Caches: Systems like **Redis** cache **key-value pairs** across services, providing faster **read/write** capabilities compared to traditional databases.

7. Full-text Search Engines: Platforms like **Elasticsearch** index data for efficient text search. This index acts as a cache, optimized for **quick retrieval**.

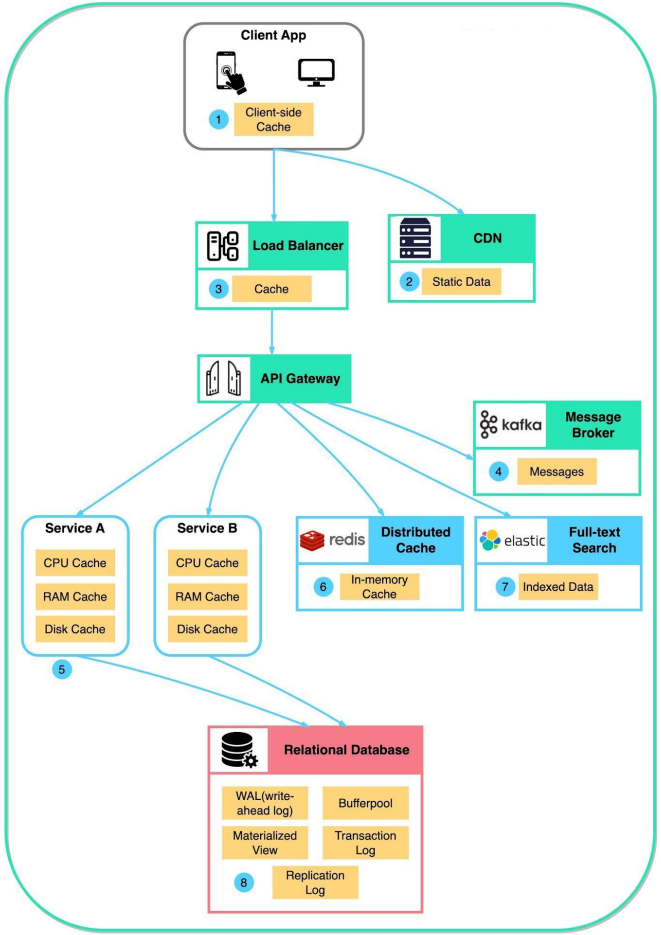
8. Databases: There are specialized mechanisms to enhance performance, some of which include **caching**:



Caching Layers

i.Bufferpool: A cache within the database that holds **data pages**. Enables **fast reads/writes** in memory, reducing disk access.

ii.Materialized Views: Store results of **expensive queries**. Allow quick returns of **precomputed results** without recalculating.



Caching Strategies

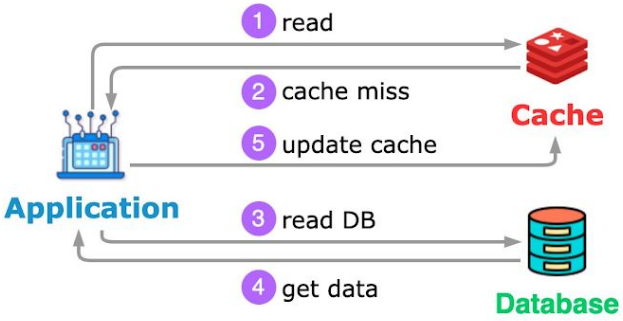
Cache Aside

Pros

1. update logic is on application level, easy to implement
2. cache only contains what the application requests for

Cons

1. each cache miss results in 3 trips
2. data may be stale if DB is updated directly



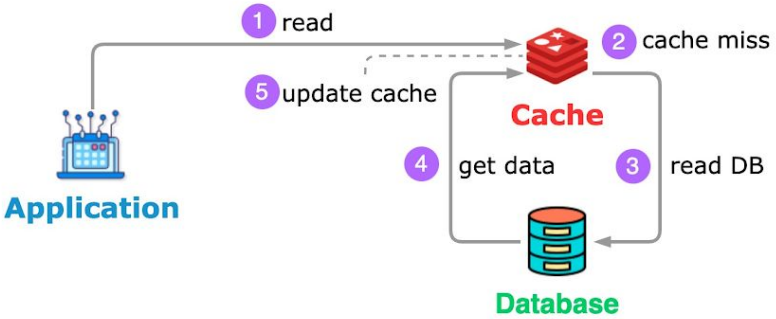
Read Through

Pros

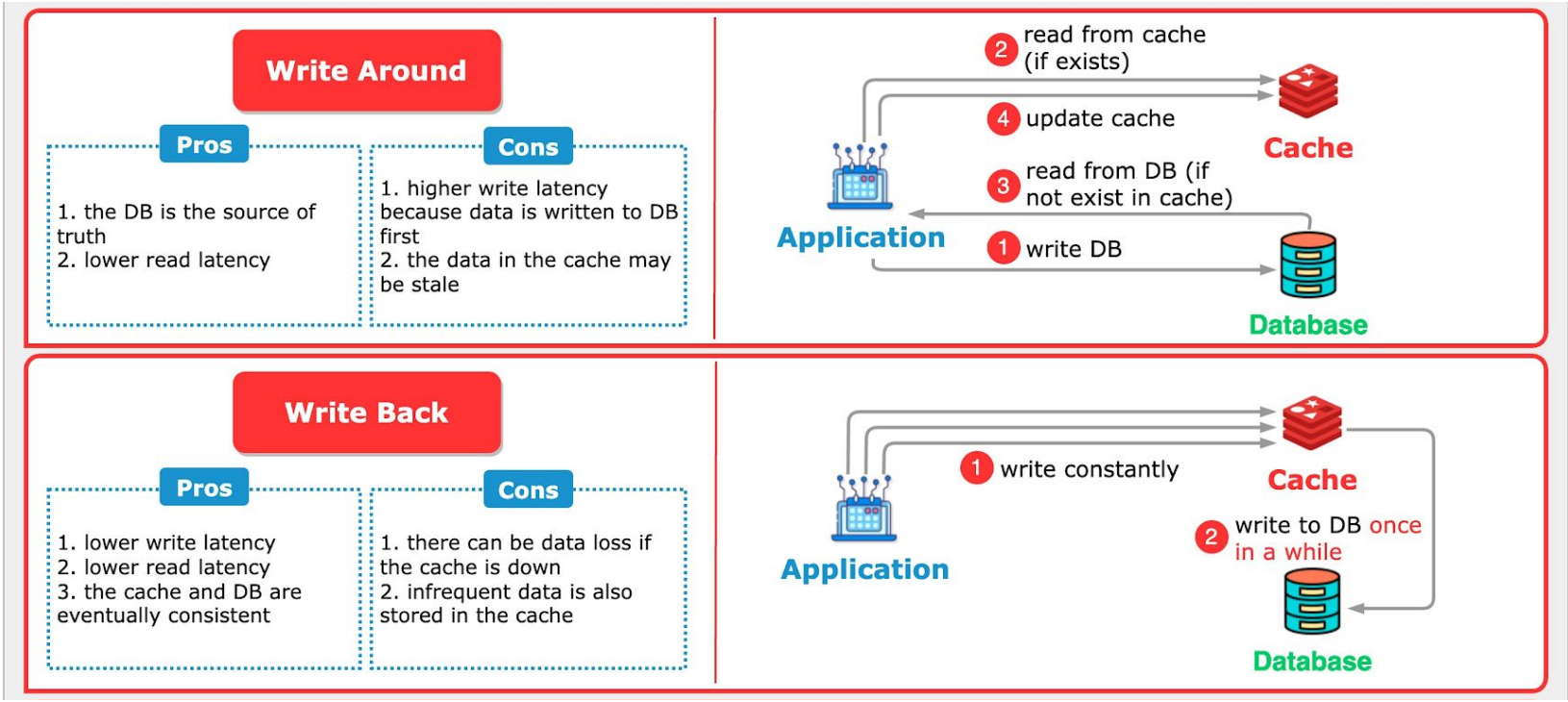
1. application logic is simple
2. can easily scale the reads and only one query hits the DB

Cons

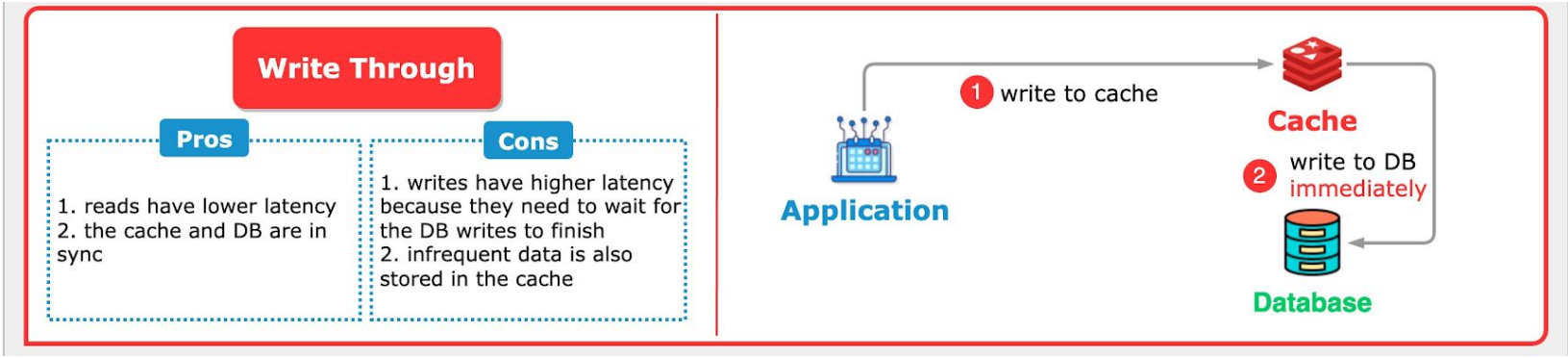
data access logic is in the cache, needs to write a plugin to access DB



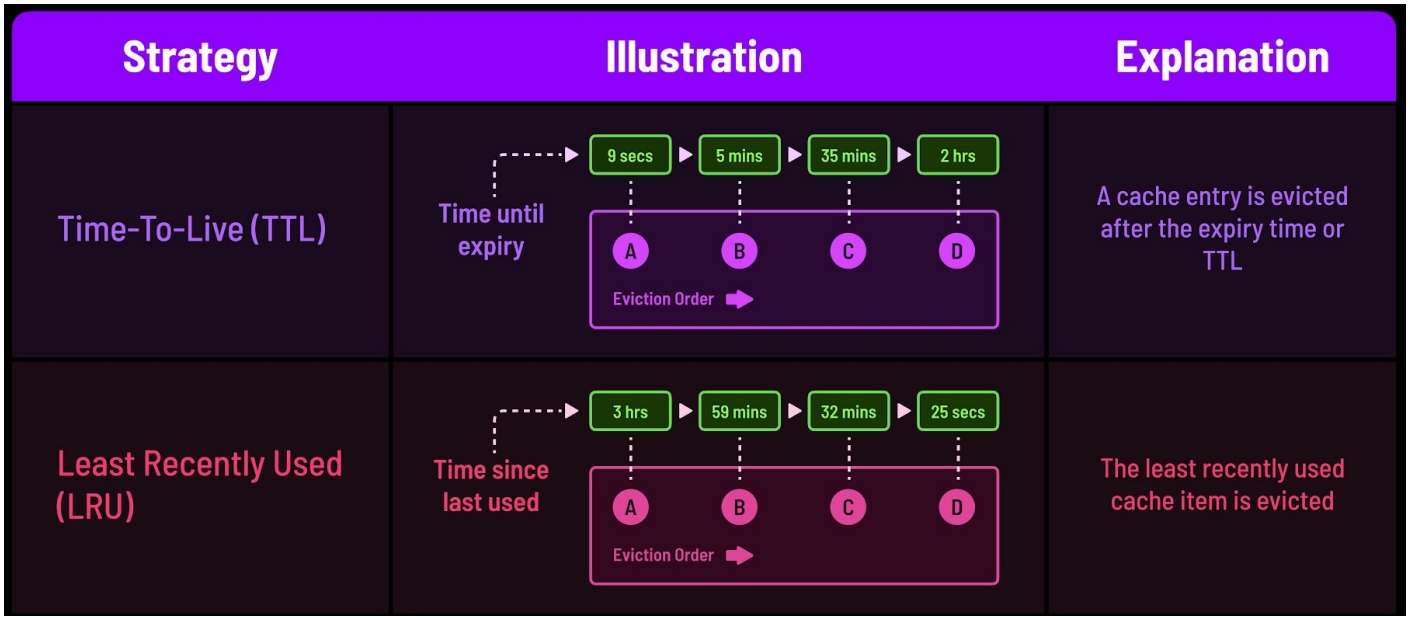
Caching Strategies



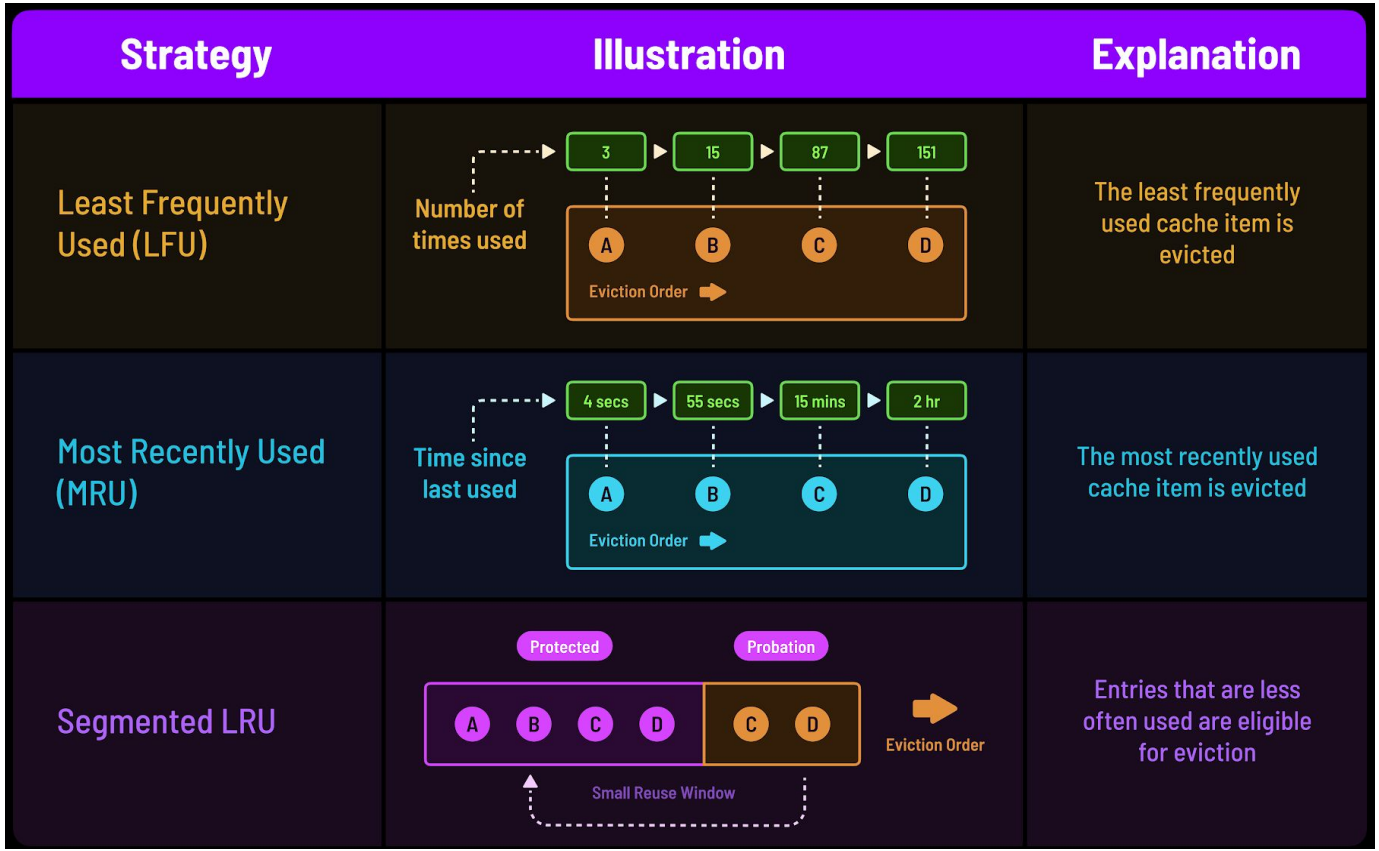
Caching Strategies



Cache Eviction Policies



Cache Eviction Policies



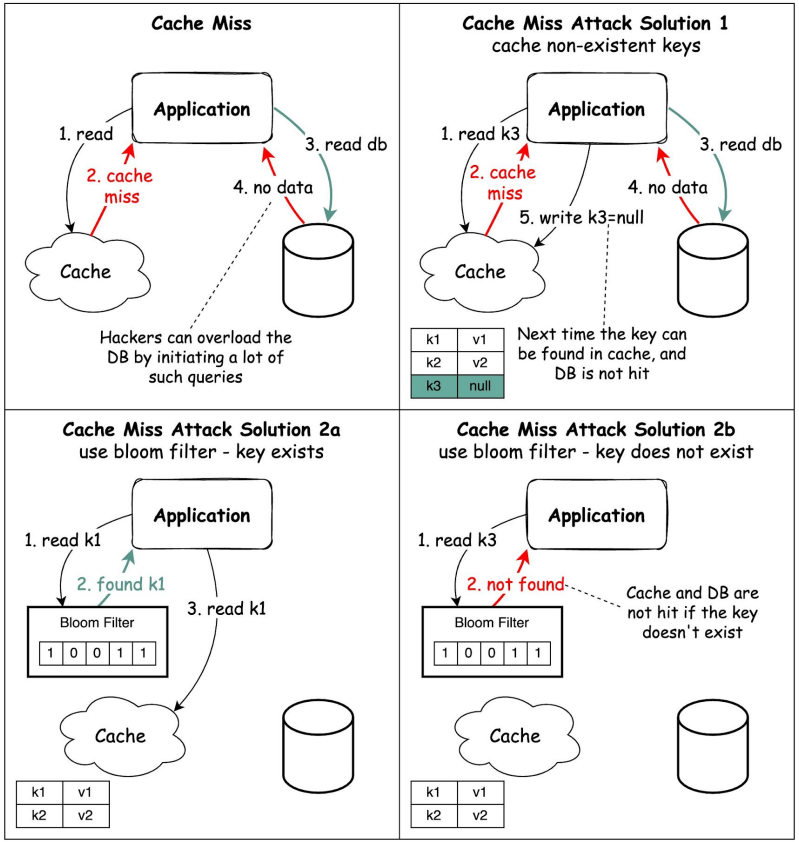
Cache Miss Attack

Caching is great—but it has its **vulnerabilities**.

A **Cache Miss Attack** occurs when:

- The requested **data doesn't exist in the database**
- It's also **not present in the cache**
- **Every request** results in a **direct database hit**

If a **malicious actor** triggers many such invalid or random requests, it can **overwhelm the database, defeating the purpose** of caching.



Cache Miss Attack

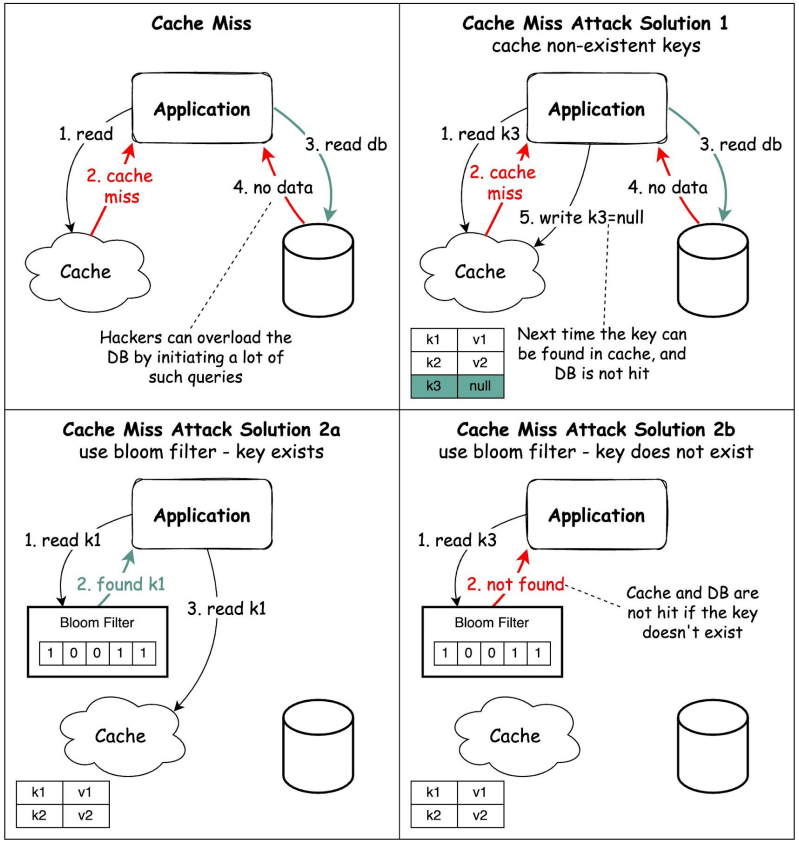
Mitigation Strategies

1. Cache Null Values

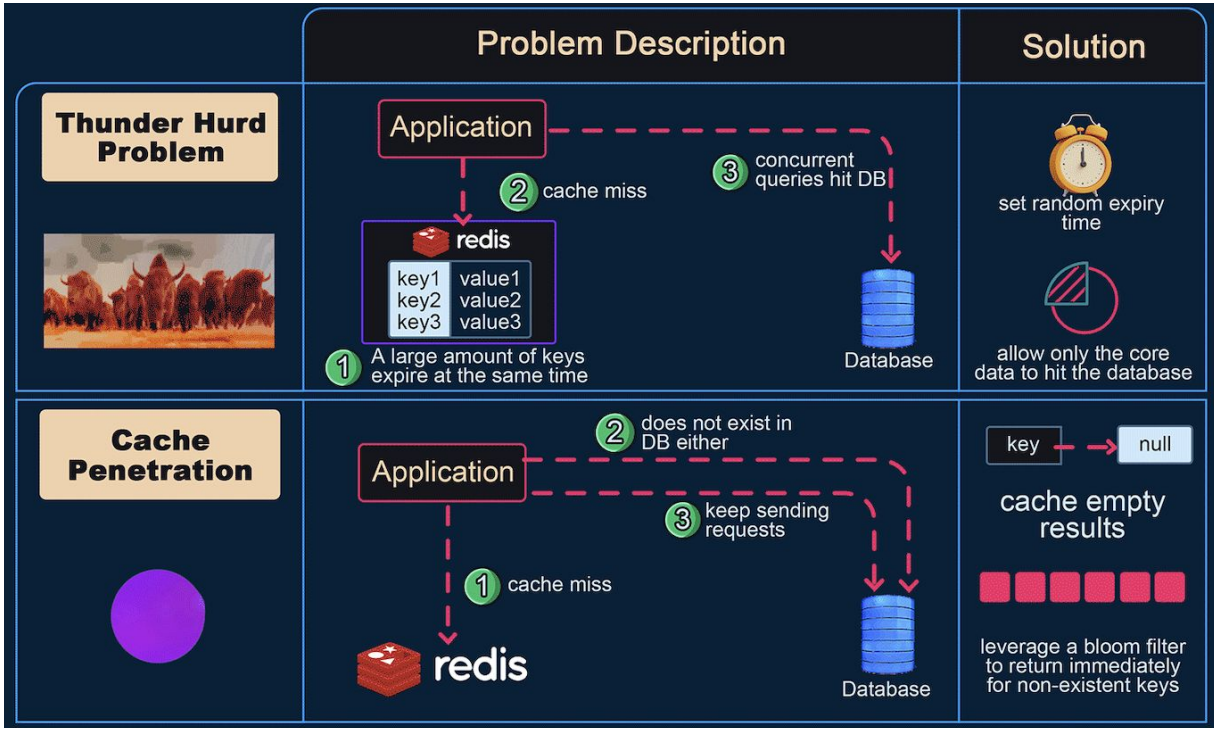
- Cache keys even when the value is **null**
- Apply a **short TTL (Time to Live)** to avoid stale caching

2. Use a Bloom Filter

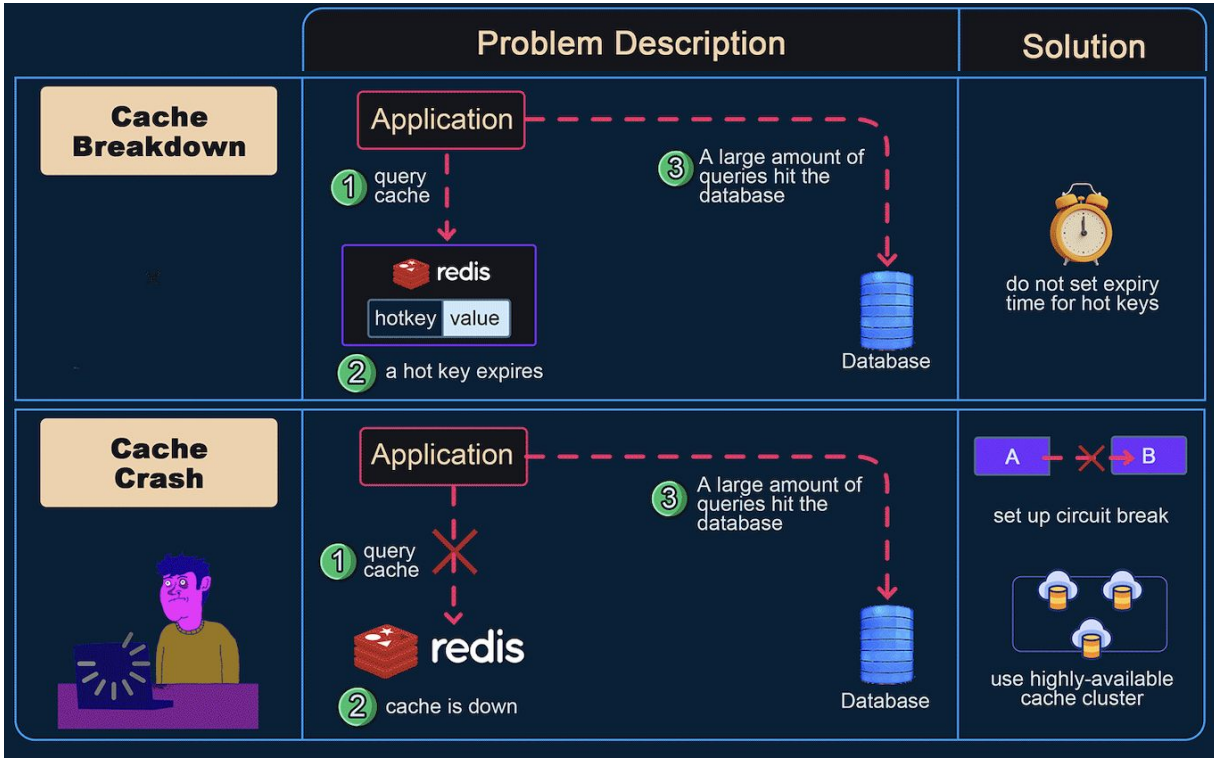
- A **probabilistic data structure** that checks if a key might exist
- Helps **filter out invalid keys** before they hit cache or DB
- If Bloom filter says "not present" → **skip cache/DB access**



How Can Cache Systems Go Wrong?



How Can Cache Systems Go Wrong?



Redis

What is Redis?

- Redis = **Remote Dictionary Server**
- A **multi-modal, in-memory database** with **sub-millisecond latency**
- Acts as both a **cache** and a **persistent data store**

Redis Adoption

- Widely used by **Airbnb**, **Uber**, **Slack**, and others
- A critical component of high-traffic, real-time systems

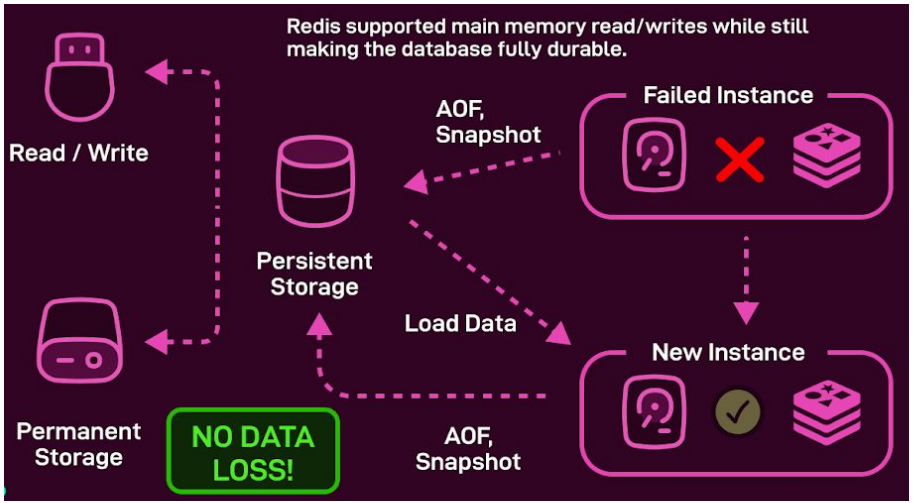
Redis

How Redis Changed the Database Game

- Performs **reads/writes from main memory**
- Ensures durability via:
  - **RDB (Snapshotting)**
  - **AOF (Append Only File)**

Redis Data Structures

- Stores data in **key-value** format
- Supports diverse structures:
  - **Strings**, **Lists**, **Sets**, **Sorted Sets**, **Hashes**
  - **Bitmaps**, **JSON**, etc.



Redis

Basic Redis Commands

- Common operations:
  - SET, GET, DELETE
  - INCR, HSET, etc.

Redis Modules

- Extend Redis capabilities with:
  - **RediSearch**, **RedisJSON**, **RedisGraph**
  - **RedisBloom**, **RedisAI**, **RedisTimeSeries**, **RedisML**, etc.

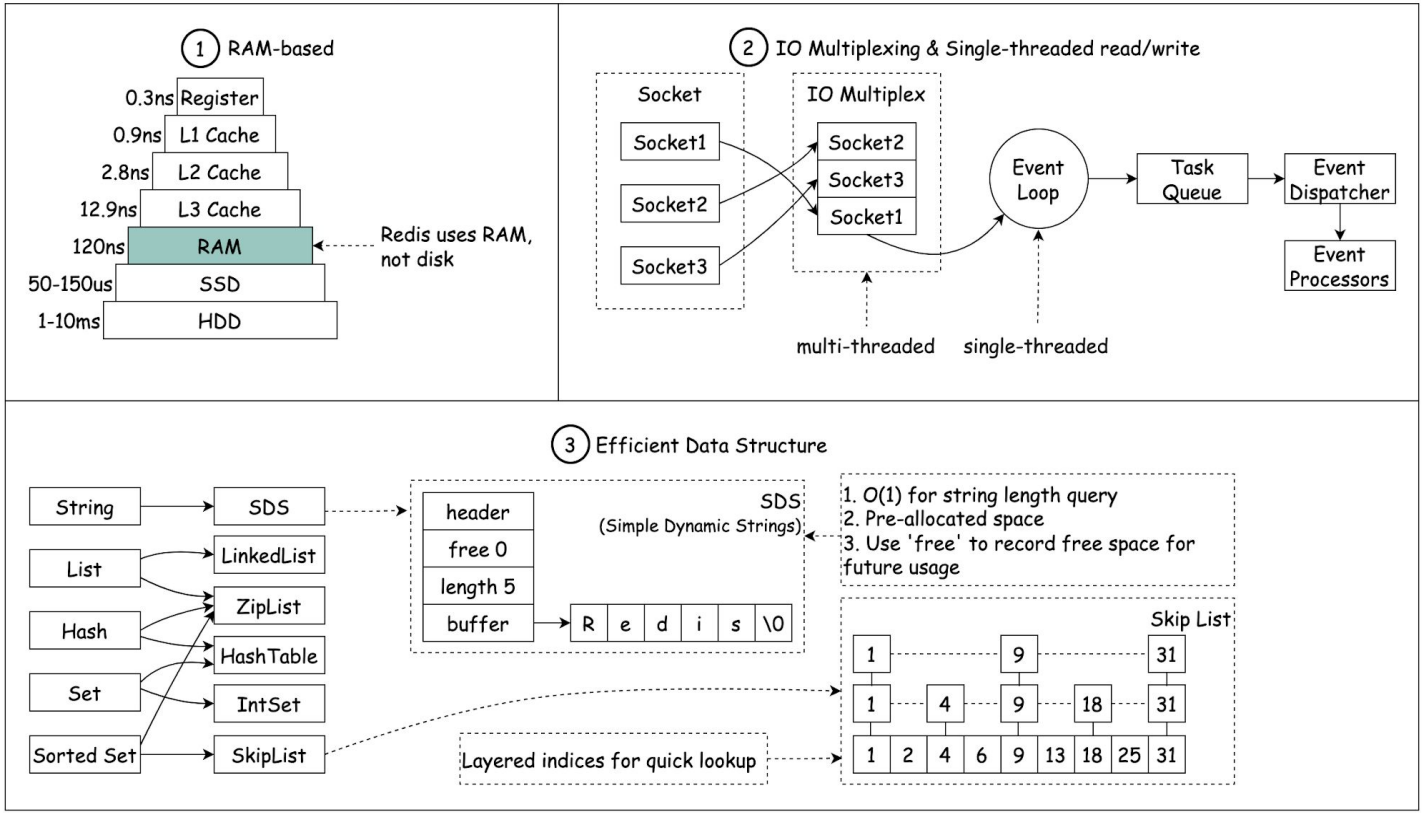
Redis Pub/Sub

- Enables **event-driven architecture**
- Supports **publish-subscribe model** for real-time messaging

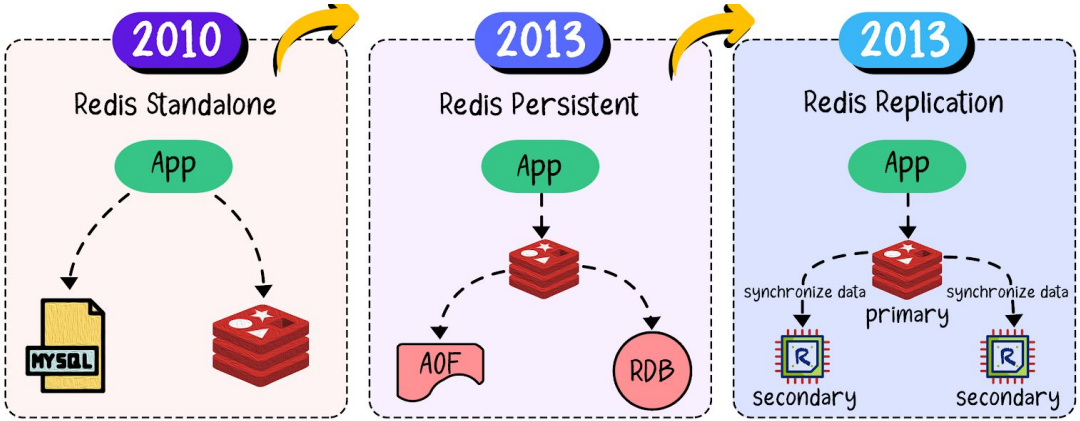
Redis Use Cases

- **Distributed Caching**
- **Session Storage**
- **Message Queues**
- **Rate Limiting**
- **High-speed Database**

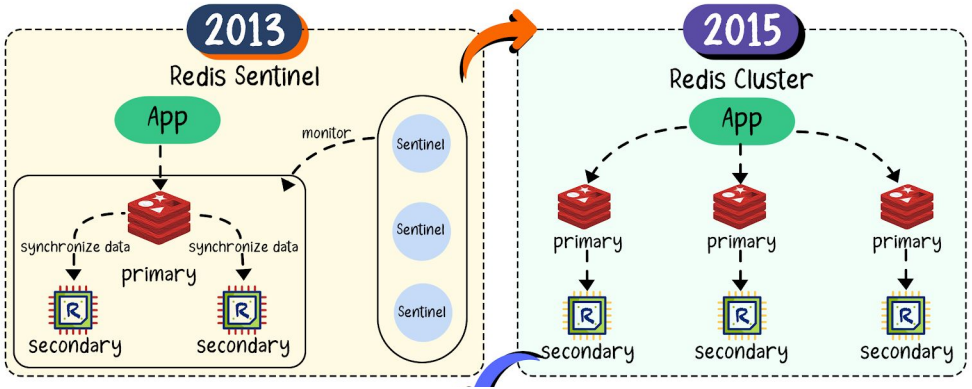
Why is Redis so Fast?



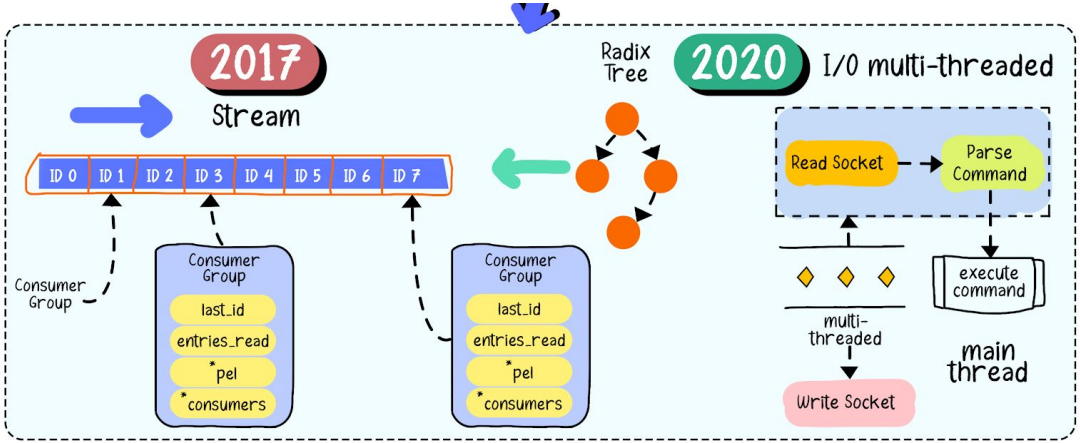
Redis Evolution



2010: Released as a simple in-memory cache with no persistence.
2013: Introduced RDB & AOF for persistence, plus replication and Sentinel for high availability.
2015: Added clustering in Redis 3.0 for horizontal scalability via slot-based sharding.



Redis Evolution



2017: Redis 5.0 introduced stream data type for real-time, event-driven workloads.

2020: Redis 6.0 implemented multi-threaded I/O to eliminate network bottlenecks.

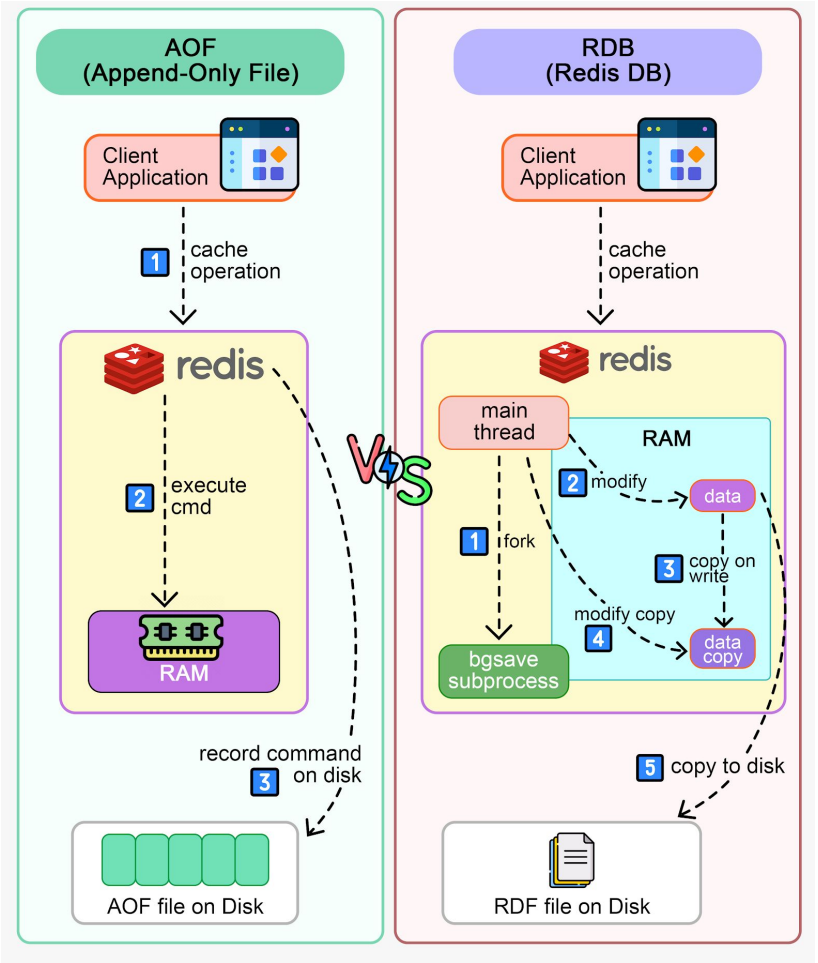
Beyond: Evolved into a multi-model platform with RedisAI, RedisTimeSeries, RedisBloom, and more for analytics, AI, and serverless use cases.

How Does Redis Persist Data?

Redis is an **in-memory database**, so it needs to **persist data to disk** to avoid data loss on crashes.

AOF (Append-Only File)

- Logs **write commands** after they execute in memory (write-after log)
- Records **commands**, not data
- Enables **event-based recovery**
- Does **not block** write operations
- Recovery requires **replaying the full log**, which can be slow if large



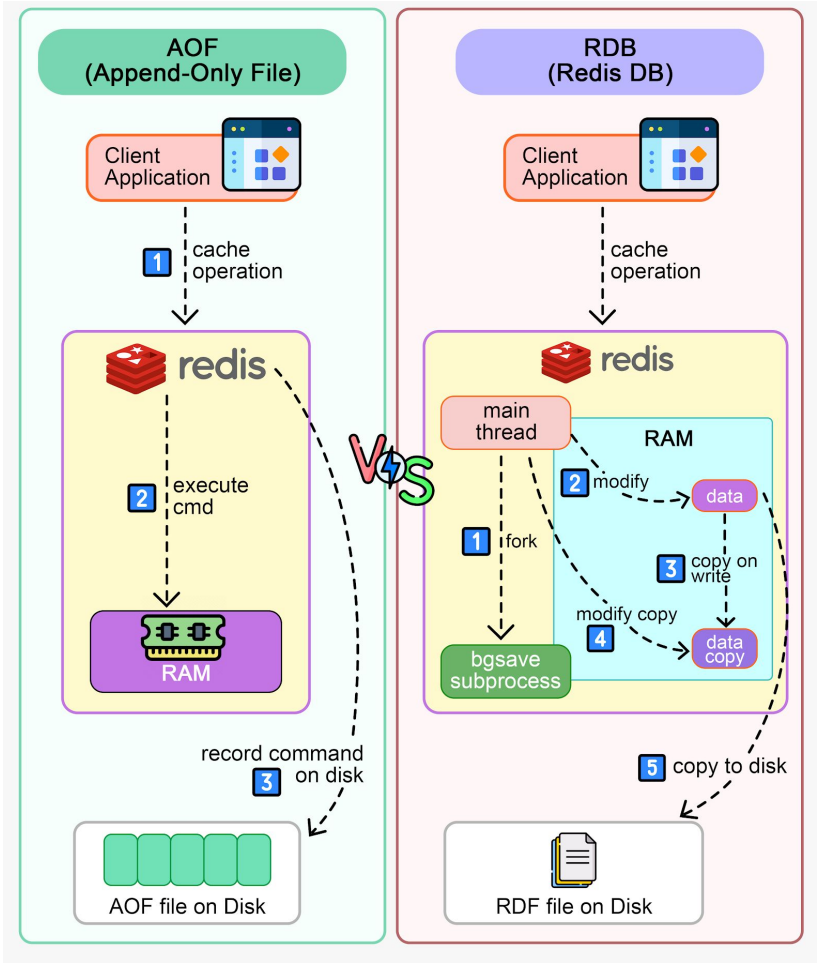
How Does Redis Persist Data?

RDB (Redis Database)

- Stores **point-in-time snapshots** of the dataset
- Uses a **forked bgsave subprocess** to write data to disk
- Offers **faster recovery** than AOF by loading snapshot directly
- Follows **copy-on-write**: changes during snapshot are written to a memory copy
- Ideal for **low-frequency but fast recovery** scenarios

Mixed Approach (Recommended)

- Combines **RDB for periodic snapshots**
- Uses **AOF for changes since last snapshot**
- Balances **recovery speed** and **data integrity**



Redis Use Cases You Should Know

- **Session Management**
  Use Redis to **share user session data** across services in distributed systems.

- **Cache**
  Cache **objects or pages**, especially **hotspot data**, to reduce database load.

- **Distributed Locking**
  Use Redis **string keys** to implement locks across **distributed services**.

- **Counter**
  Track **likes, views**, or other counts using atomic increment operations.

- **Rate Limiting**
  Apply **rate limits** for user IPs or actions using Redis key expiration and counters.

Redis Use Cases You Should Know

- **Global ID Generator**
  Use Redis **integer keys** to generate **globally unique identifiers**.

- **Shopping Cart**
  Use **Redis Hashes** to store product IDs and quantities as **key-value pairs**.

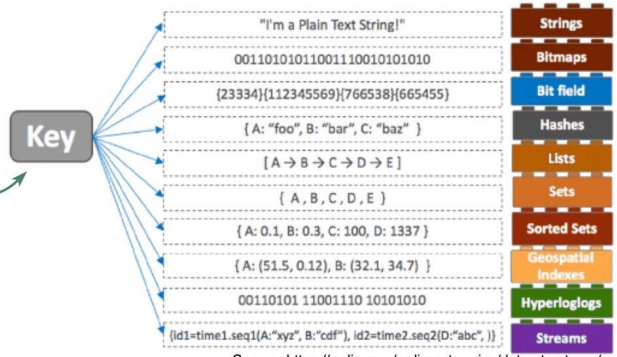
- **User Retention Analysis**
  Represent daily logins with **Bitmaps** to compute **user retention** efficiently.

- **Message Queue**
  Implement simple queues using **Redis Lists (LPUSH + RPOP)**.

- **Ranking System**
  Use **Sorted Sets (ZSet)** to maintain article scores and build **leaderboards**.

Memcached vs Redis

	Memcached	Redis
Data Structure	plain string values	lists, sets, sorted sets, hashes, bit arrays, and hyperloglogs
Architecture	multi-threaded	single thread for reading/writing keys
Transaction	x	support atomic operations
Snapshots/ Persistence	x	keep data on disks, support RDB/AOF persistence
Pub-sub Messaging	x	supports Pub/Sub messaging with pattern matching
Geospatial Support	x	Geospatial indexes that stores the longitude and latitude data of a location
Server-side Scripts	x	support Lua script to perform operations inside Redis
Supported Cache Eviction	LRU	noeviction, allkeys-lru, allkeys-lfu, allkeys-random, volatile-lru, volatile-lfu, volatile-random, volatile-ttl
Replication	x	leader-follower replication



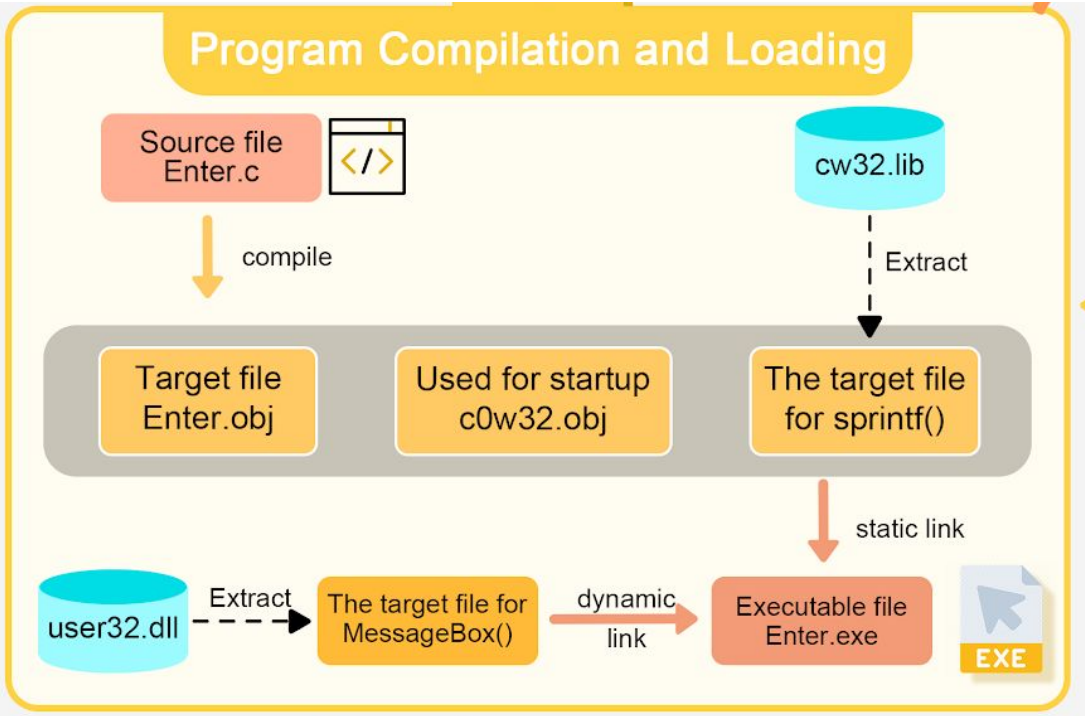
Source: https://redis.com/redis-enterprise/data-structures/

- **RDB** (Redis Database Backup) - a compact, point-in-time snapshot of the database at a specific time.
- **AOF** (Append Only File) - keep track of all the commands that are executed, and in a disastrous situation, it re-execute the commands to get the data back.

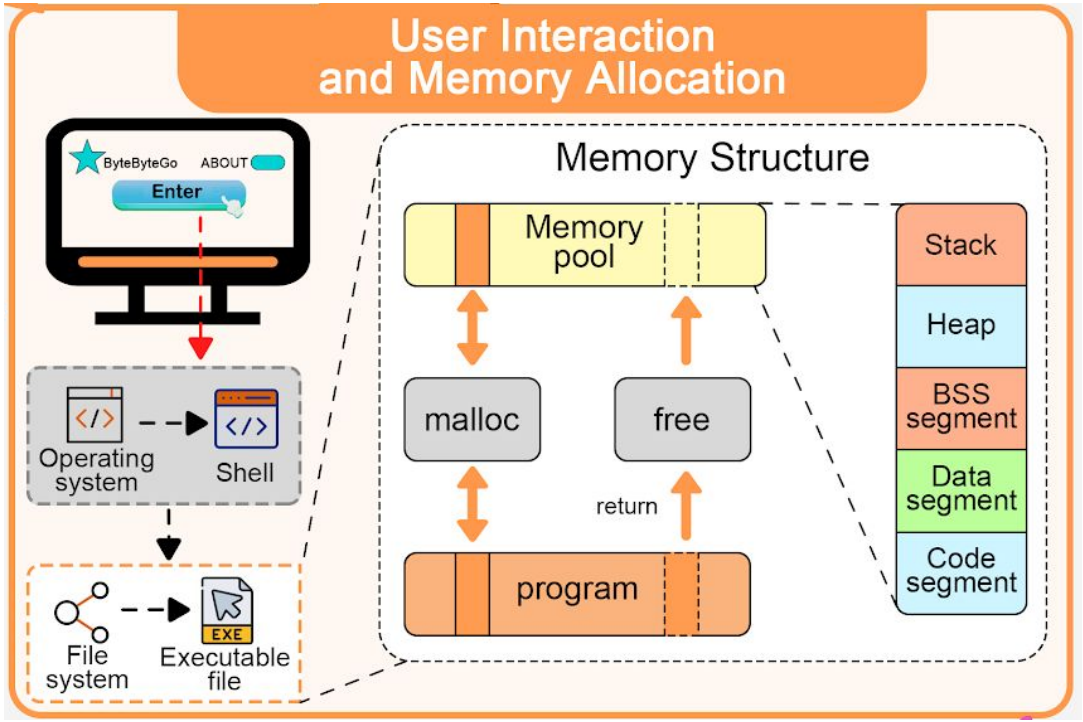
build a high performance chatroom

- find the distance between two elements (people or places)
- find all elements within a given distance of a point

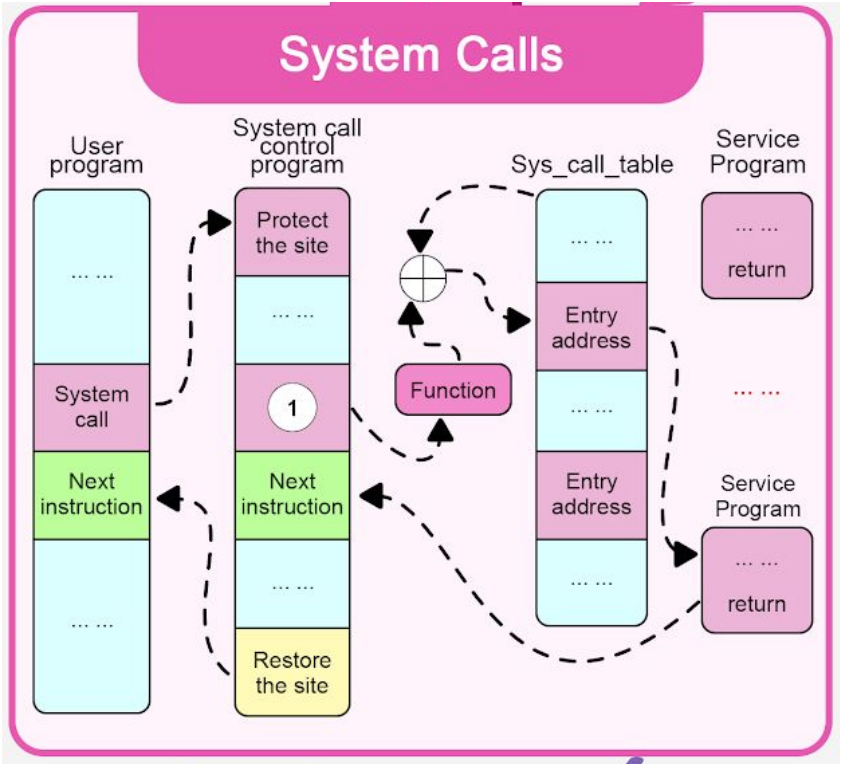
OS Concepts: How Do Computer Programs Run?



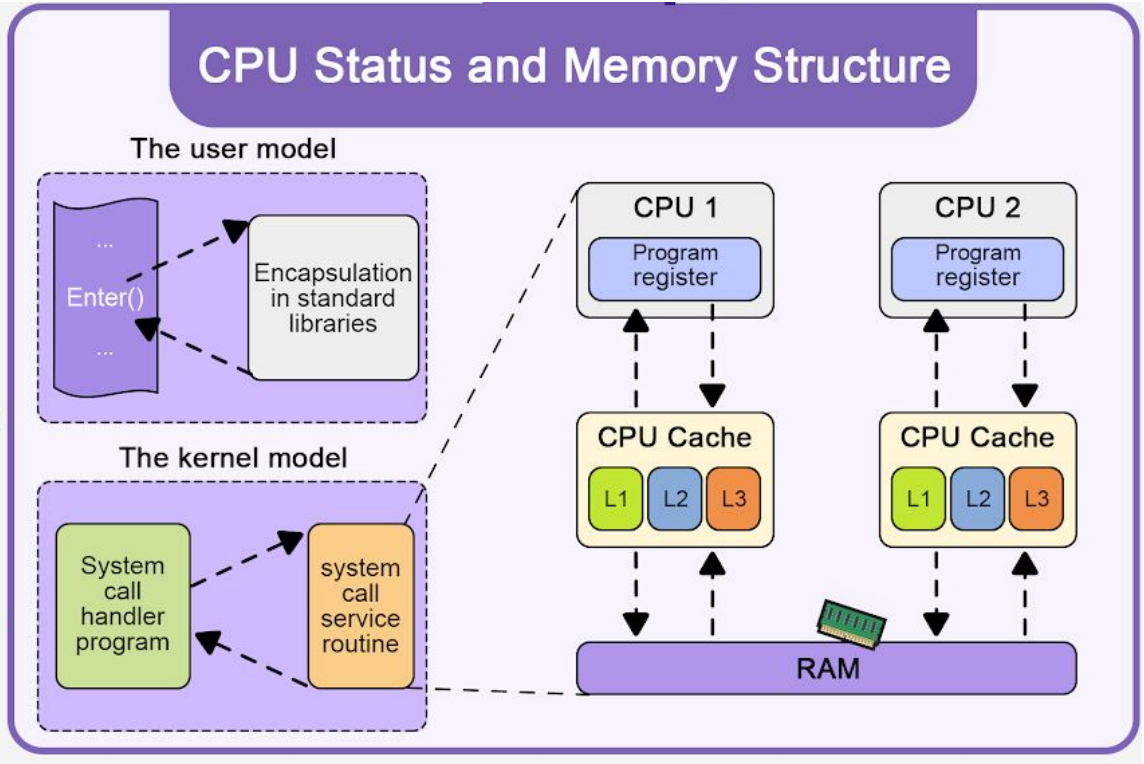
OS Concepts: How Do Computer Programs Run?



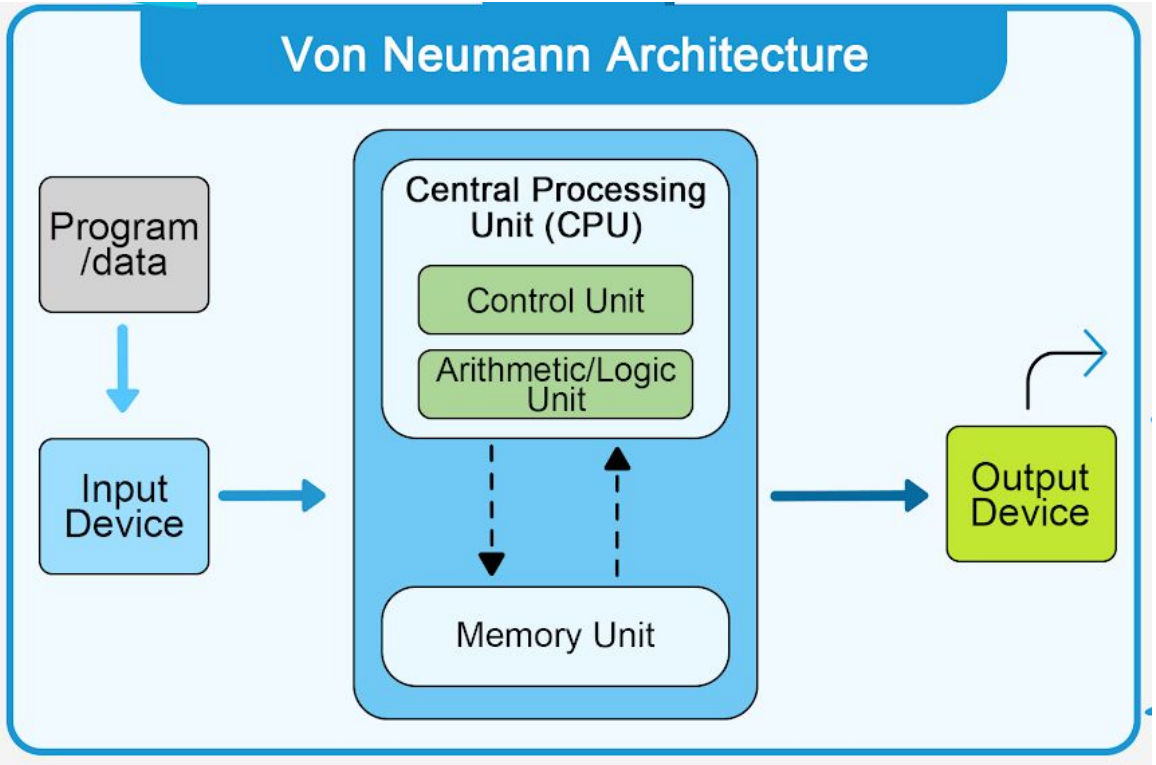
OS Concepts: How Do Computer Programs Run?



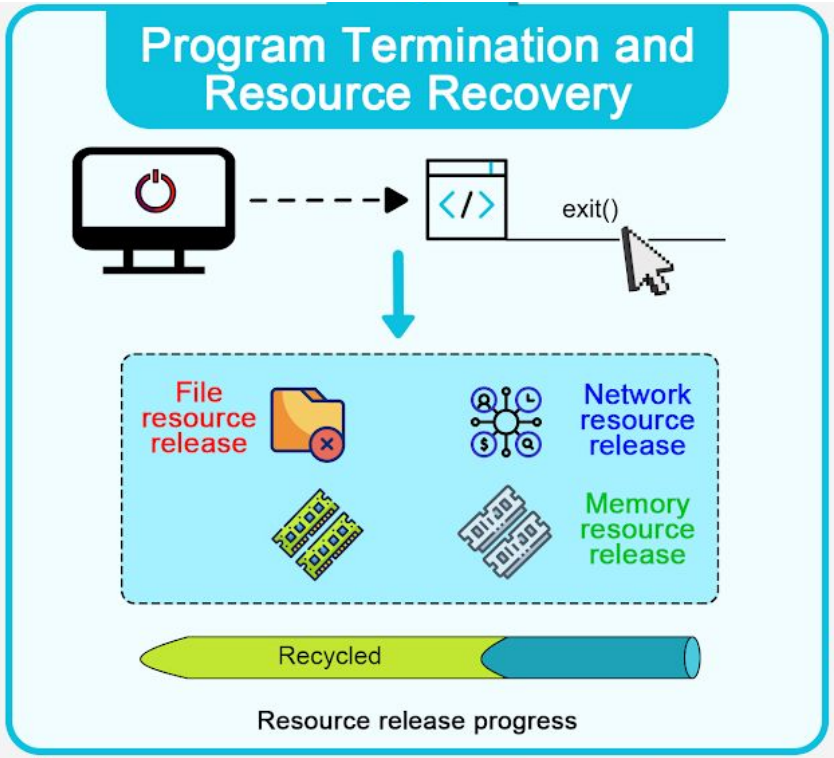
OS Concepts: How Do Computer Programs Run?



OS Concepts: How Do Computer Programs Run?

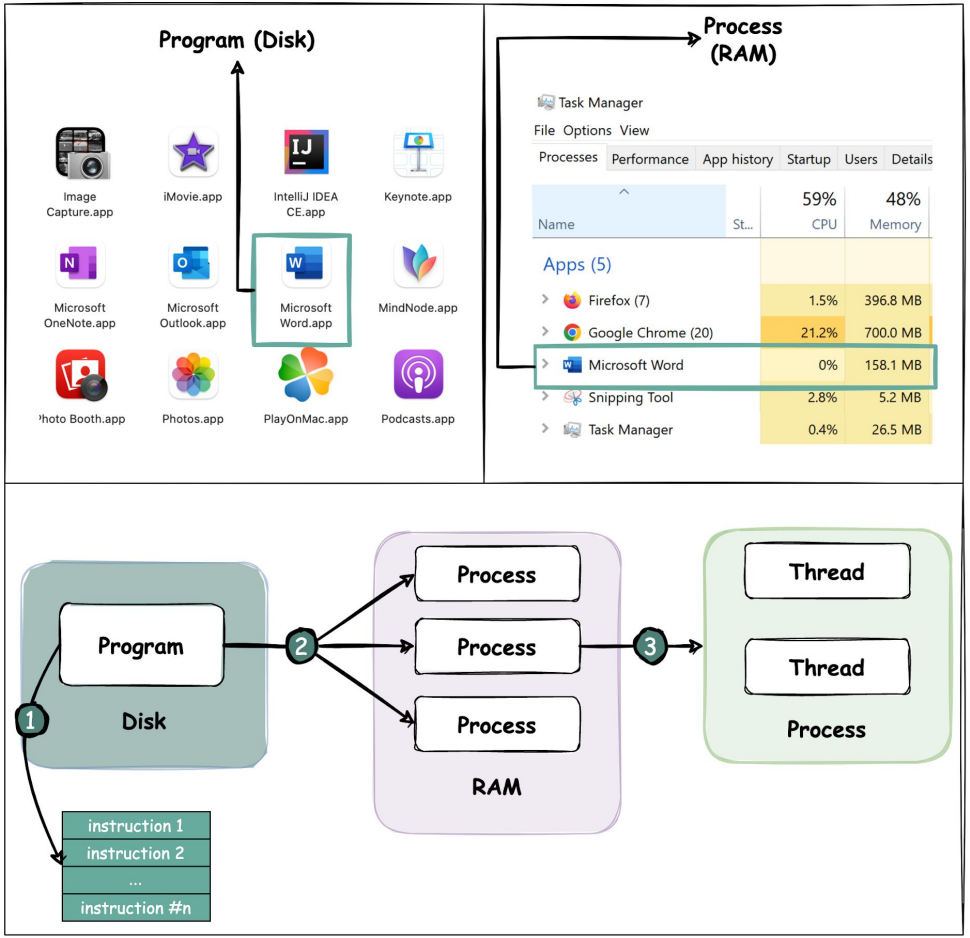


OS Concepts: How Do Computer Programs Run?



Process vs Thread

- **Processes** are usually **independent**, while **threads** exist as **subsets of a process**.
- Each **process has its own memory space**. **Threads within the same process share memory**.
- A **process is heavyweight** — it takes **more time to create and terminate**.
- **Context switching** is **more expensive** between **processes**.
- **Inter-thread communication** is **faster** compared to inter-process communication.



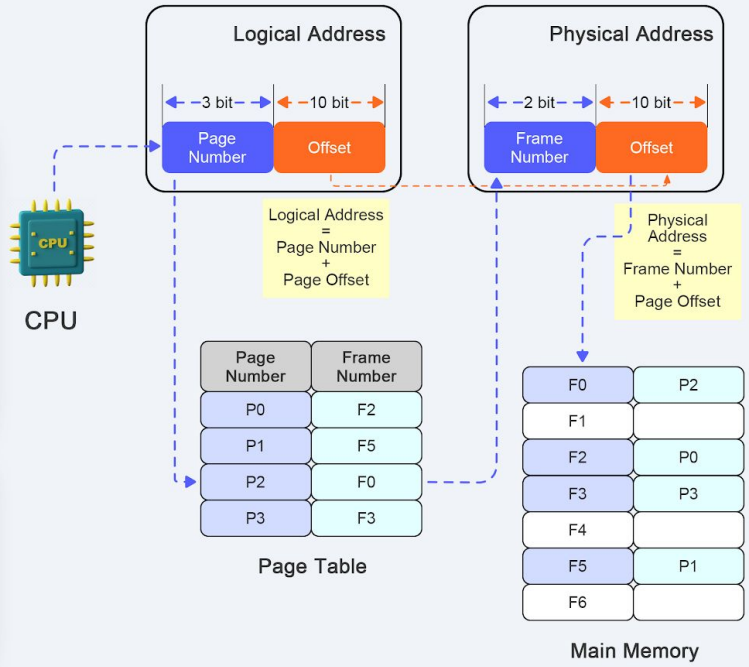
Memory Allocation: Paging vs Segmentation

Paging

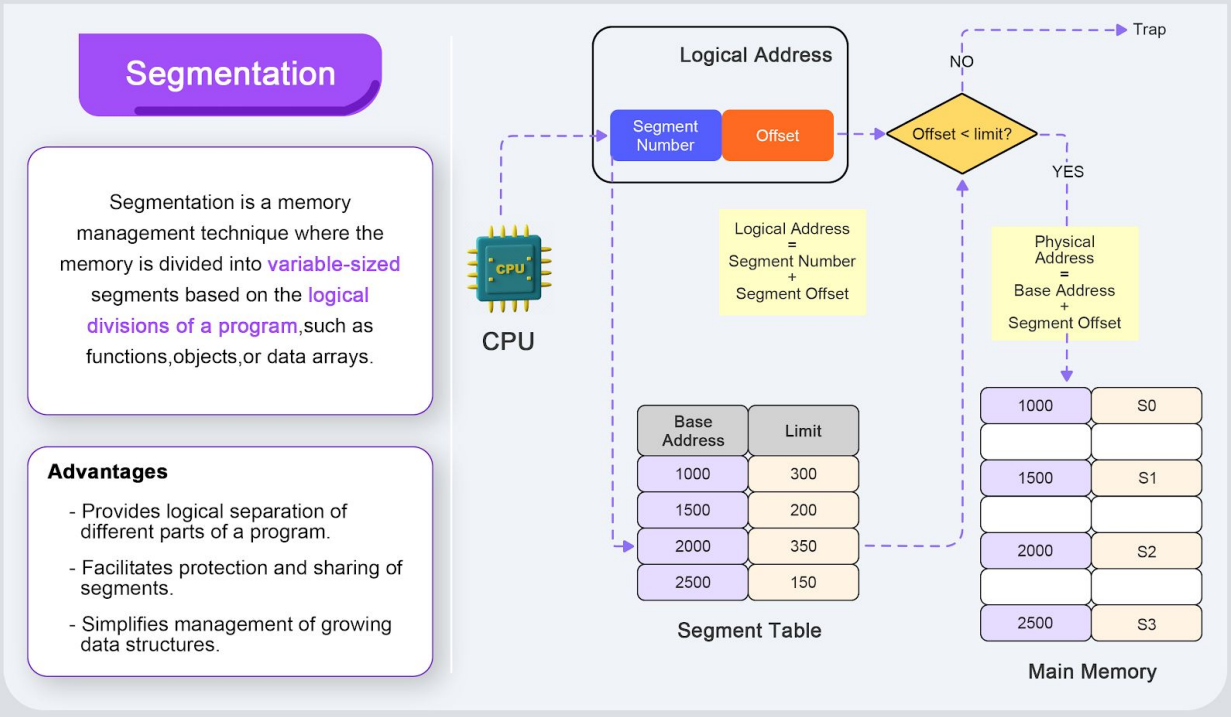
Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.The process's address space is divided into fixed-size blocks called pages,while physical memory is divided into fixed-size blocks called frames.

Advantages

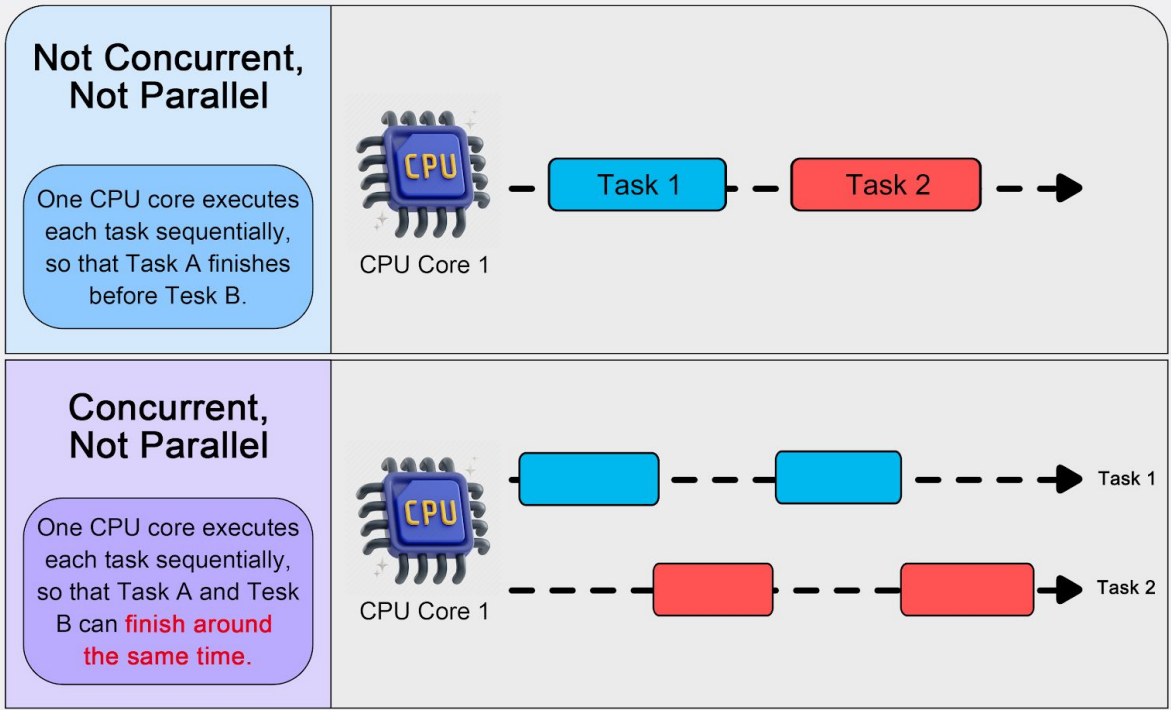
- Eliminates external fragmentation.
- Simplifies memory allocation.
- Supports efficient swapping and virtual memory



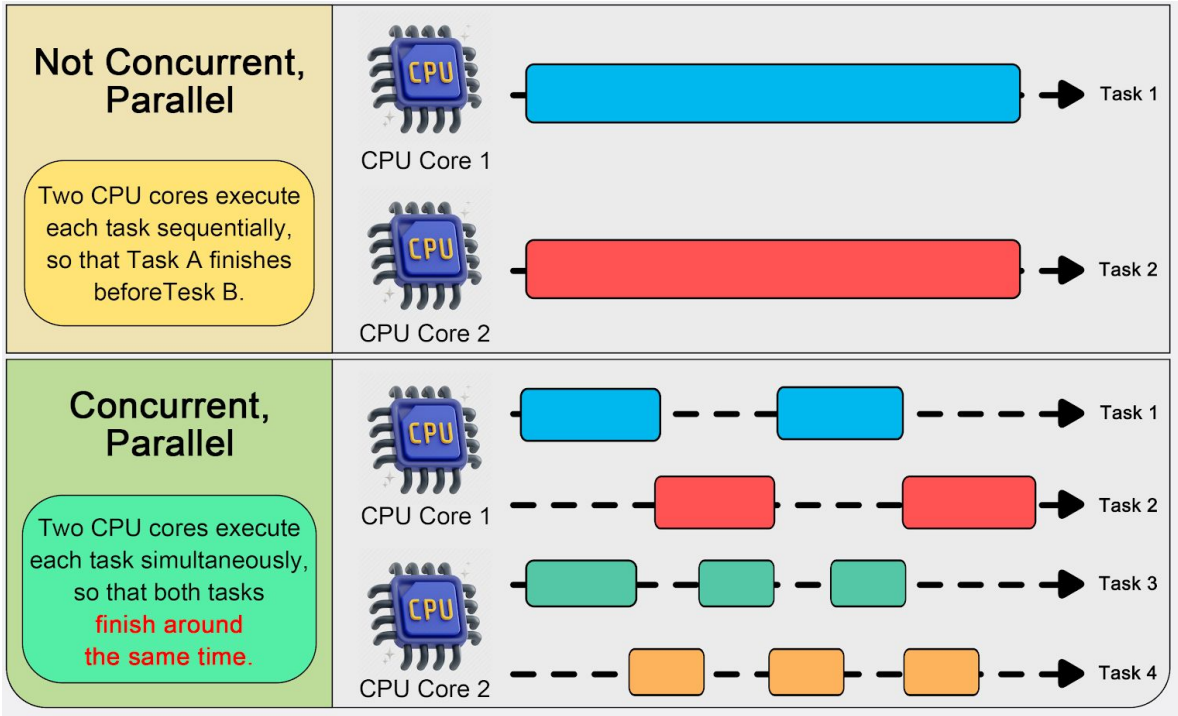
Memory Allocation: Paging vs Segmentation



Concurrency is NOT Parallelism



Concurrency is NOT Parallelism



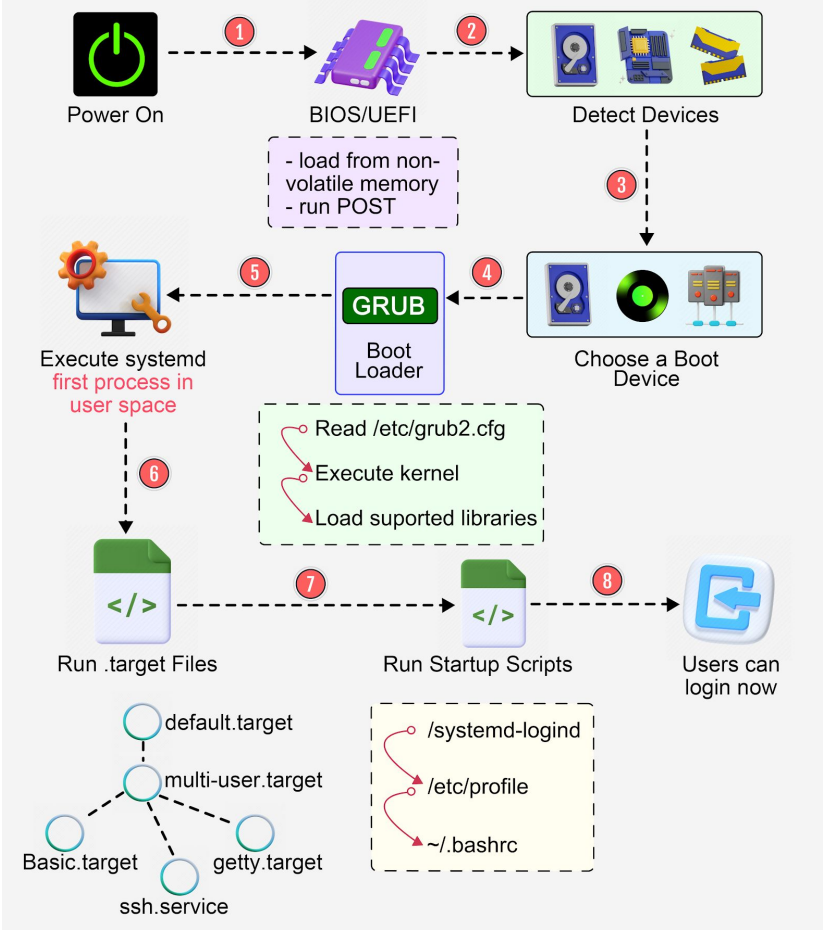
Linux Boot Process

Step 1:
When we turn on the power, **BIOS (Basic Input/Output System)** or **UEFI (Unified Extensible Firmware Interface)** firmware is loaded from **non-volatile memory** and executes **POST (Power On Self Test)**.

Step 2:
BIOS/UEFI detects connected devices, including **CPU, RAM, and storage**.

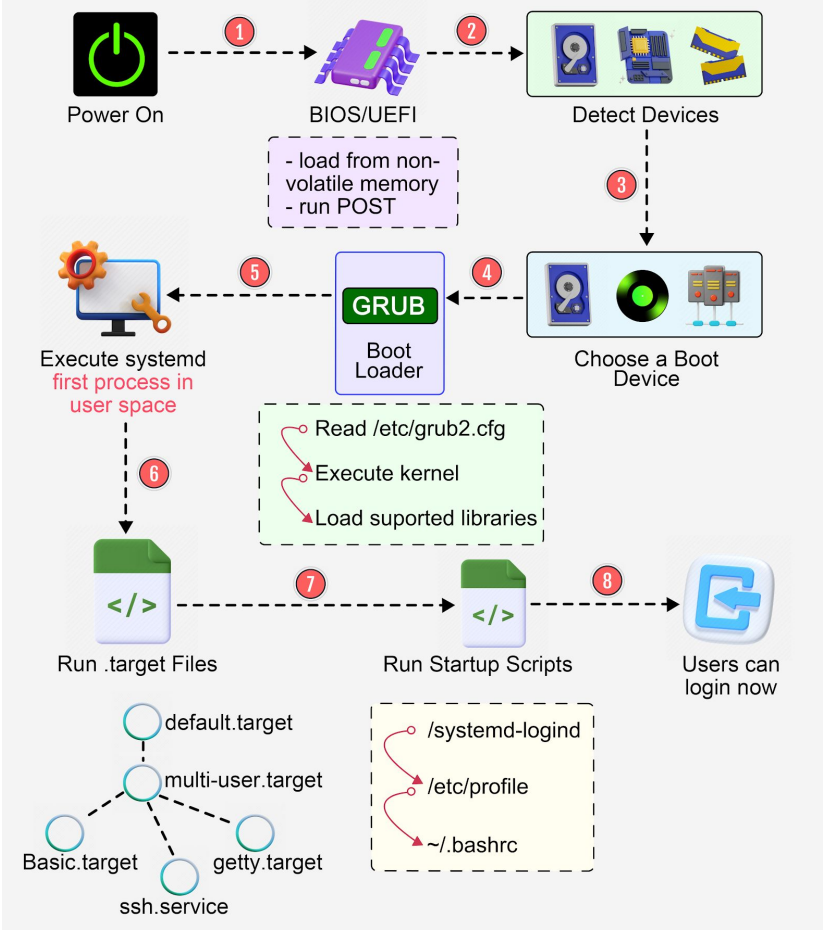
Step 3:
Selects a boot device to load the OS from—e.g., **hard drive, network server**, or **CD-ROM**.

Step 4
BIOS/UEFI runs the bootloader (e.g., **GRUB**), which presents a **menu to choose the OS or kernel functions**.



Linux Boot Process

- **Step 5**
  After the **kernel is loaded**, control switches to **user space**. The kernel starts **systemd**, which:
  - Manages **processes and services**
  - Probes hardware
  - Mounts **filesystems**
  - Launches the **desktop environment**
- **Step 6**
  systemd activates the default target (e.g., graphical.target or multi-user.target) and other **unit files**.
- **Step 7**
  The system executes **startup scripts** and **configures the environment**.
- **Step 8**
  The user is presented with a **login window** — the system is now **ready for use**.



Thank you!
Any Questions?