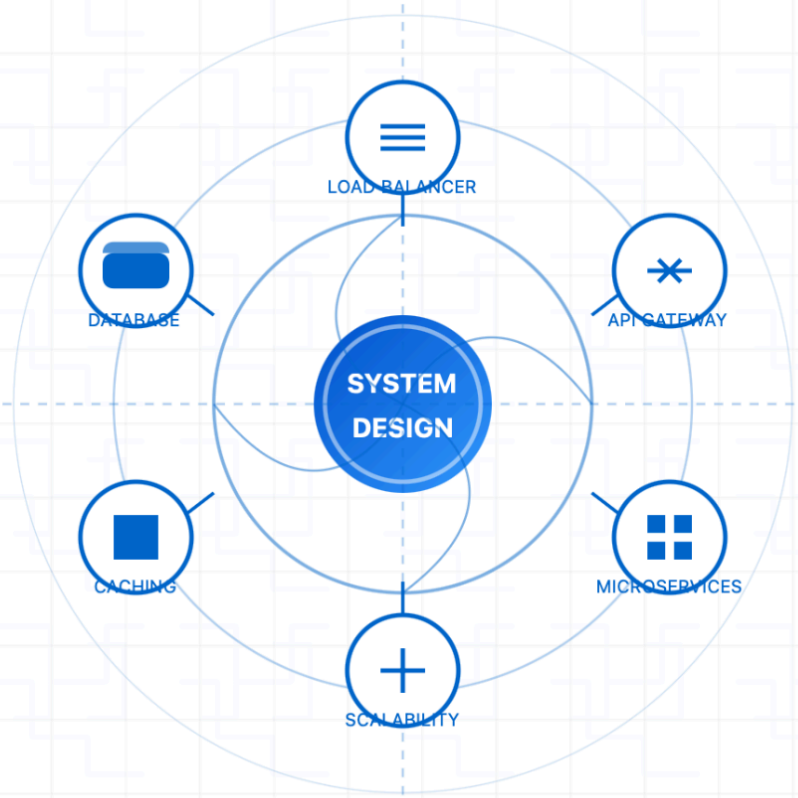
SYSTEM DESIGN INTERVIEW ROADMAP



FUNDAMENTALS COMPONENTS FRAMEWORKS CASE STUDIES MASTERY

THE COMPLETE GUIDE TO ACING TECHNICAL INTERVIEWS

System Design Interview Roadmap

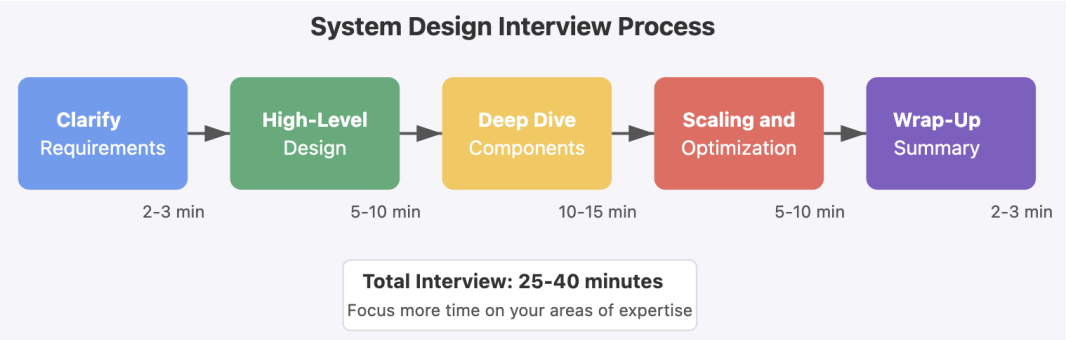
Subscribe : https://systemdr.substack.com

Part 1 : System Design Fundamentals

1. System Design Interviews. A Visual Roadmap

What Is a System Design Interview?

A system design interview evaluates your ability to design scalable, reliable, and efficient systems that solve real-world problems. Unlike coding interviews that test algorithm skills, system design interviews assess your architectural thinking and engineering judgment.



Why This Matters

System design interviews are critical for senior roles because they.

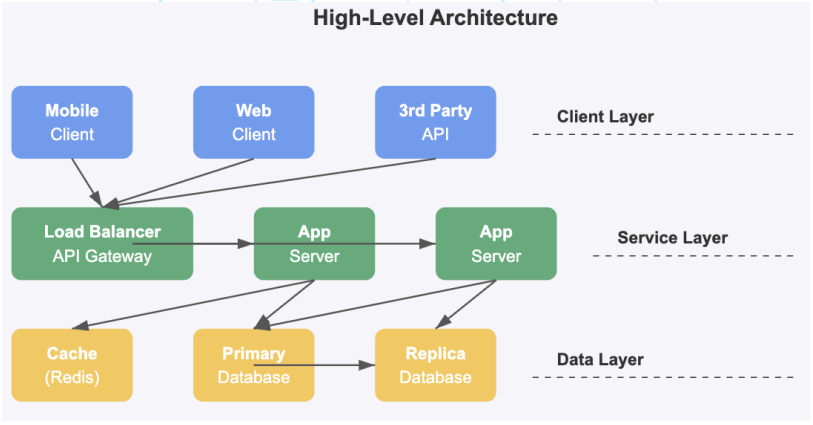
- Demonstrate your ability to make technical decisions with business impact
- Show how you handle ambiguity and requirements gathering
- Reveal your communication skills when discussing complex technical topics
- Highlight your experience with large-scale, distributed systems

The 5-Step Interview Framework

1. **Clarify Requirements (2-3 minutes)**
   - **Ask probing questions** to understand scope and constraints
   - **Identify users** and their primary use cases
   - **Establish scale** (users, traffic, data volume)
   - **Define functional and non-functional requirements**

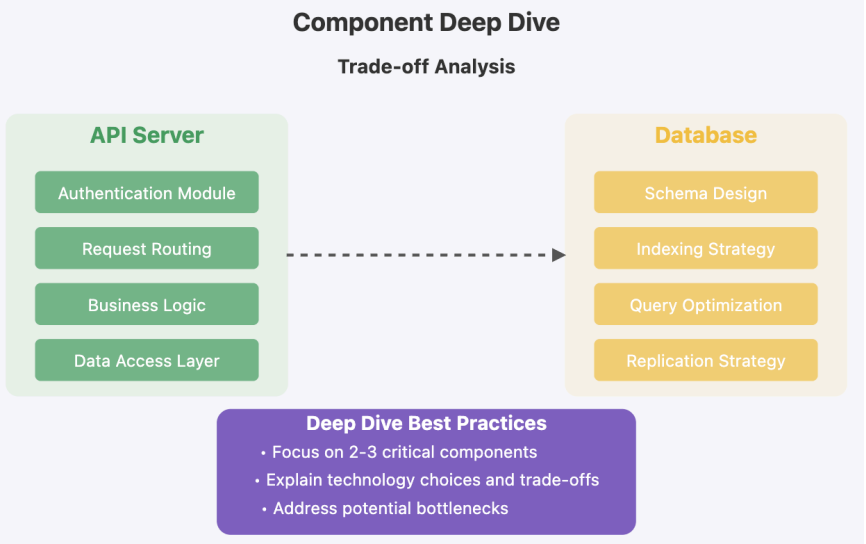
2. High-Level Design (5-10 minutes)

- **Sketch core components** (clients, servers, databases, caches)
- **Define API contracts** between components
- **Outline data models** and relationships
- **Create a basic system flow** for primary use cases



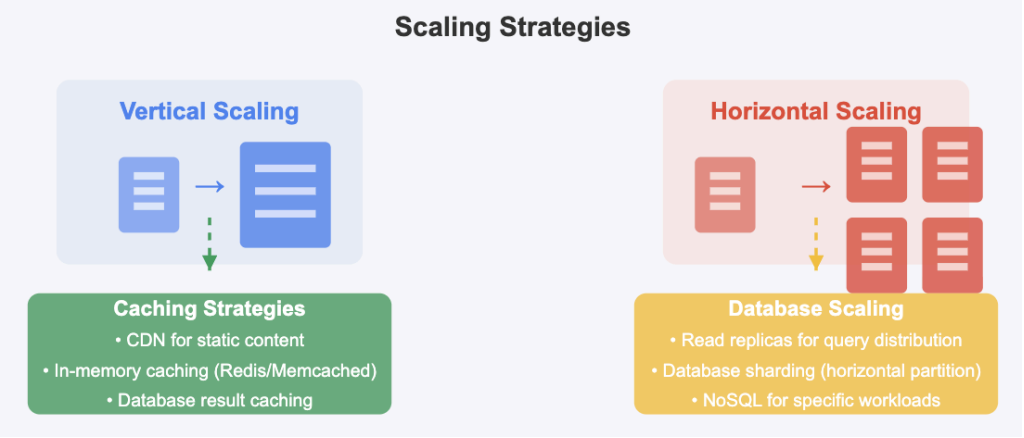
3. Deep Dive Into Components (10-15 minutes)

- **Select critical components** to explore further
- **Address potential bottlenecks**
- **Explain technology choices**
- **Consider tradeoffs** between different approaches



4. Scaling & Optimization (5-10 minutes)

- **Identify scaling challenges**
- **Apply relevant patterns** (sharding, replication, caching)
- **Address edge cases** and potential failures
- **Consider performance optimizations**



5. Wrap-Up (2-3 minutes)

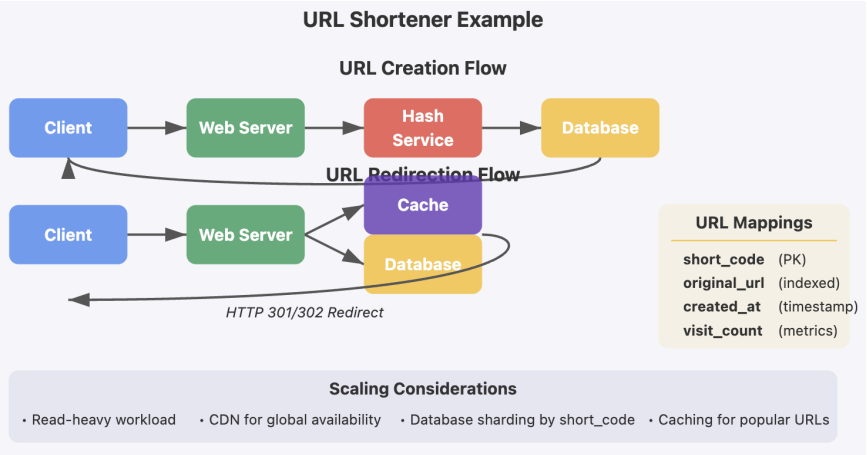
- **Summarize your design**
- **Acknowledge limitations**
- **Suggest future improvements**
- **Demonstrate understanding of evolution path**

Common Pitfalls to Avoid

- **Starting too detailed** - Begin broad, then narrow
- **Jumping to solutions** without understanding requirements
- **Focusing solely on one aspect** (like just the database)

- **Not considering scale** from the beginning
- **Silent thinking** without communicating your process

Practice Example: URL Shortener



Let's apply our framework to design a URL shortening service like bit.ly:

1. **Requirements:**
    - Create short URLs from long ones
    - Redirect users to original URL when short URL is accessed
    - Scale to millions of URLs
    - Fast response times
2. **High-Level Design:**
    - Web servers to handle URL creation and redirection
    - Database to store URL mappings
    - Hashing service to generate short codes
3. **Deep Dive:**
    - URL encoding algorithm (Base62 vs. MD5+Base62)
    - Database schema and indexing strategy
    - Caching layer for popular URLs
4. **Scaling:**
    - Read-heavy workload → Caching and read replicas
    - Database sharding based on short code
    - CDN for global availability
5. **Wrap-Up:**
    - System handles millions of URLs with sub-100ms response time
    - Future improvements: Analytics, custom URLs, expiration policies

Over to You

Think about a recent app you've used. How would you design its backend system? What questions would you ask first? What components would be in your high-level design?

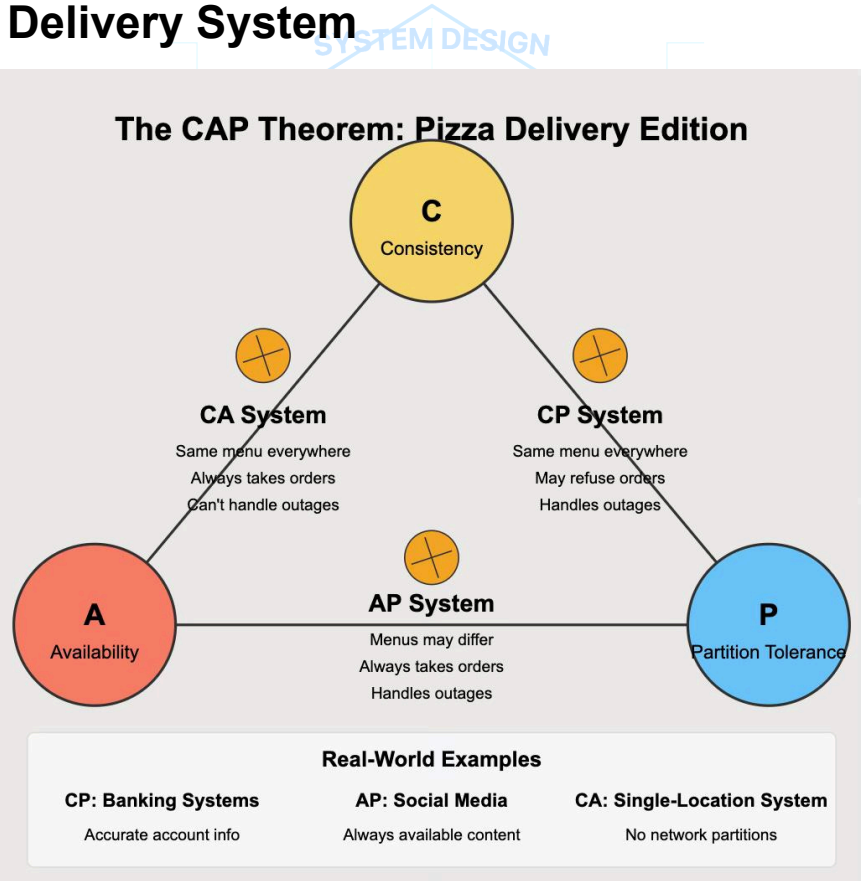
2. The CAP Theorem Explained with Pizza Delivery Analogies

In the world of distributed systems, the CAP theorem stands as a fundamental principle that shapes how we design and build scalable applications. Coined by computer scientist Eric Brewer in 2000, this theorem states that it's impossible for a distributed data store to simultaneously provide more than two out of three guarantees: Consistency, Availability, and Partition tolerance.

But what does this actually mean in practice? Let's break it down using something we're all familiar with. pizza delivery.

Thanks for reading System Design Roadmap! Subscribe for free to receive new posts and support my work.

The Pizza Delivery System



Imagine a pizza chain with multiple locations across a city. Each location has its own kitchen, delivery drivers, and order management system. This is our distributed system.

- **Consistency**: When you place an order, all locations know about it immediately and have the exact same information.
- **Availability**: The pizza chain can always take your order and tell you about the status of existing orders.

- **Partition Tolerance**: The system continues to operate even if communication between locations breaks down (due to traffic jams, phone outages, etc.).

Consistency (C): Everyone Has The Same Menu

Let's say this pizza chain releases a new specialty pizza. With perfect consistency, the moment it's added to the system, every location would instantly update their menus. If you called any location, they would all tell you the same thing about this new pizza.

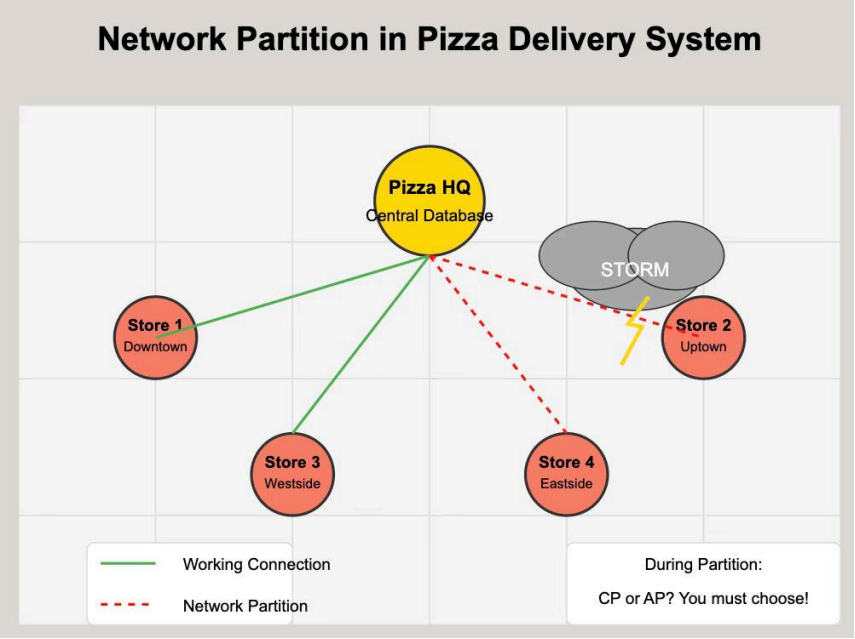
In pizza terms, consistency means that no matter which location you contact, you get the same menu, same prices, and same information about your order.

Availability (A): Always Taking Orders

Availability means that the pizza chain always accepts your order. If you call or use their app, you'll always get a response—they'll take your order, tell you about specials, or give you your delivery status.

A highly available pizza system never says, "Sorry, we can't take your order right now because we're checking with headquarters about menu updates."

Partition Tolerance (P): Surviving Communication Breakdowns



Imagine a major storm hits the city, cutting off phone and internet connections between some pizza locations. Partition tolerance means the system can still function despite these communication failures.

Each location can still take orders, make pizzas, and handle deliveries in their area, even if they can't talk to other locations or headquarters.

The Impossible Triangle: Pick Two

Now, here's where the CAP theorem comes in—you can only have two of these three guarantees at any given time:

CP (Consistency and Partition Tolerance)

In a CP system, our pizza chain prioritizes having the same, correct information at all locations, even during communication failures.

If a new pizza is added to the menu, but one location can't communicate with headquarters due to a network issue, that location would stop taking orders for any pizza rather than risk providing outdated information.

Example: "I'm sorry, we can't take any orders right now because our system is updating the menu. Please try again later."

AP (Availability and Partition Tolerance)

In an AP system, the pizza chain prioritizes always taking orders, even if they can't synchronize information between locations.

During a network partition, a location might take your order for a pizza that was just removed from the menu because they haven't received the update yet.

Example: "Yes, we can take your order for the Summer Special Pizza!" (Even though that pizza was discontinued an hour ago at other locations)

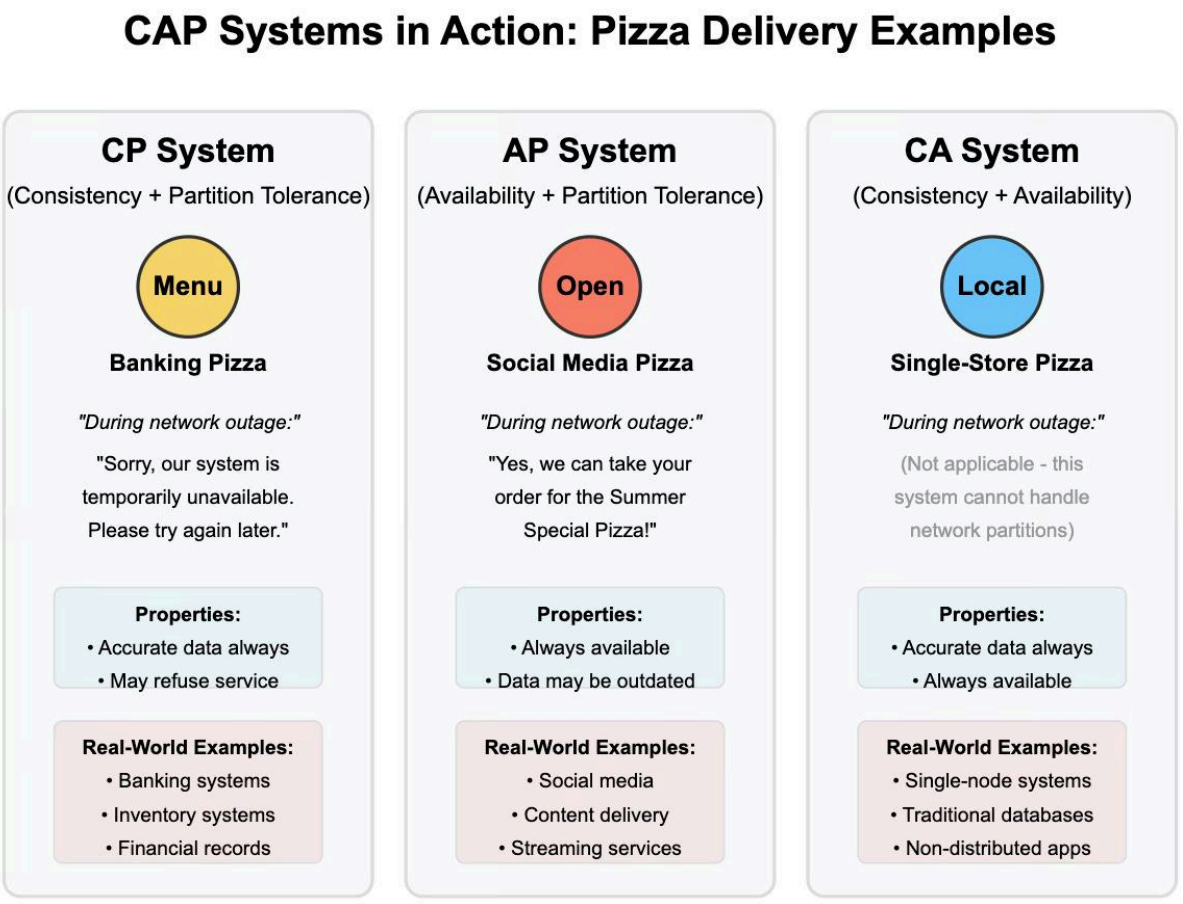
CA (Consistency and Availability)

In a CA system, the pizza chain guarantees that all locations have the same information and are always able to take orders—but only if there are no communication failures.

As soon as any communication issues arise between locations, the system can no longer maintain both consistency and availability.

This is why many distributed systems experts say that in practical terms, you must design for partition tolerance—making the real choice between consistency and availability when partitions occur.

Real-World Applications



Understanding the CAP theorem helps us make informed design decisions:

- **Banking Systems** typically choose consistency over availability (CP). You'd rather have your ATM tell you "System Unavailable" than give you incorrect information about your balance.
- **Social Media Platforms** often choose availability over consistency (AP). It's better for users to see slightly outdated content than to show an error message.
- **E-commerce Inventory Systems** might choose consistency for checkout (to prevent overselling) but availability for browsing (showing slightly outdated inventory is better than showing no products).

The Pizza Delivery Lessons

Next time you're designing a distributed system, think about your pizza delivery priorities:

1. Do you need to guarantee everyone has the exact same menu information (consistency)?
2. Do you need to guarantee you're always taking orders (availability)?
3. Do you need to handle communication breakdowns between locations (partition tolerance)?

Remember, you can only pick two—and in real-world distributed systems, you'll almost always need partition tolerance, narrowing your choice to CP or AP.

By understanding these trade-offs through familiar analogies, you can make better architectural decisions that align with your business requirements and user expectations.

3. Latency vs. Throughput: Understanding the Trade-offs

In the world of system design, two performance metrics consistently dominate discussions: latency and throughput. These concepts are fundamental to understanding how systems perform under various conditions, yet they're often misunderstood or incorrectly used interchangeably. This article will demystify these critical concepts and explore the inherent trade-offs between them that every system designer must consider.

Defining the Fundamentals

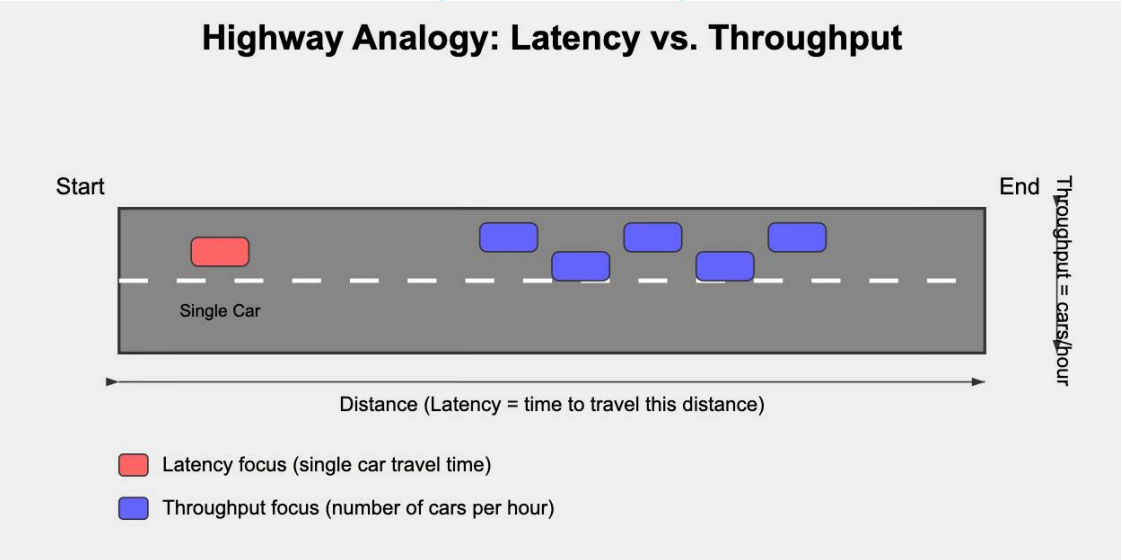
Latency is the time delay between initiating an action and seeing its effect. In simpler terms, it's how long it takes for a single piece of data to travel from source to destination. Latency is typically measured in units of time—milliseconds (ms) being the most common.

Throughput is the rate at which a system can process data. It represents the amount of work done per unit of time, often measured in operations per second, requests per second, or bits per second (bandwidth).

To illustrate this difference, consider a highway:

- **Latency** would be how long it takes for a single car to travel from one end to the other.
- **Throughput** would be how many cars can pass through a point on the highway per hour.

Highway Analogy for Latency vs. Throughput



Measuring Performance in Real Systems

Let's translate these concepts to common computing scenarios:

Web Servers

- **Latency**: The time from when a user clicks a link to when the page appears (e.g., 200ms)
- **Throughput**: How many requests the server can handle per second (e.g., 5,000 req/sec)

Databases

- **Latency**: How long it takes to retrieve a single record (e.g., 10ms)
- **Throughput**: How many queries can be processed per second (e.g., 1,000 queries/sec)

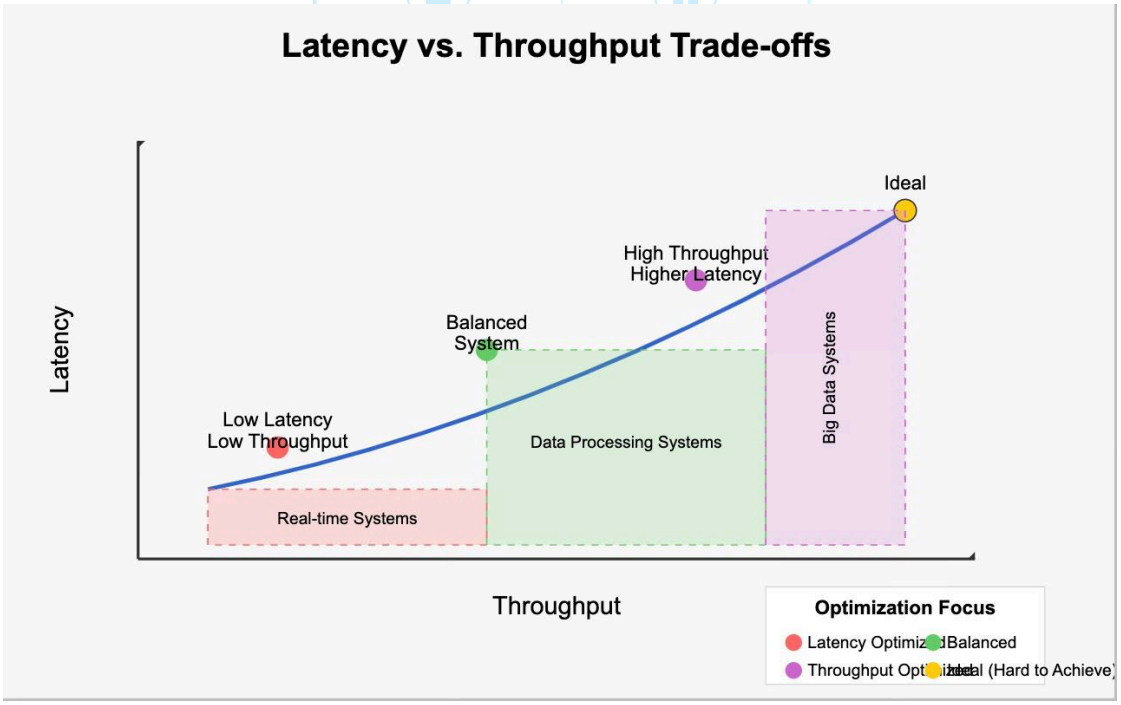
Networks

- **Latency**: The time it takes for a packet to travel from source to destination (e.g., 50ms)
- **Throughput**: The amount of data that can be transferred per second (e.g., 100 MB/sec)

The Inherent Trade-offs

The relationship between latency and throughput is complex and often involves trade-offs. While it's technically possible to optimize for both, real-world constraints frequently force designers to prioritize one over the other.

Latency vs. Throughput Trade-off Visualization



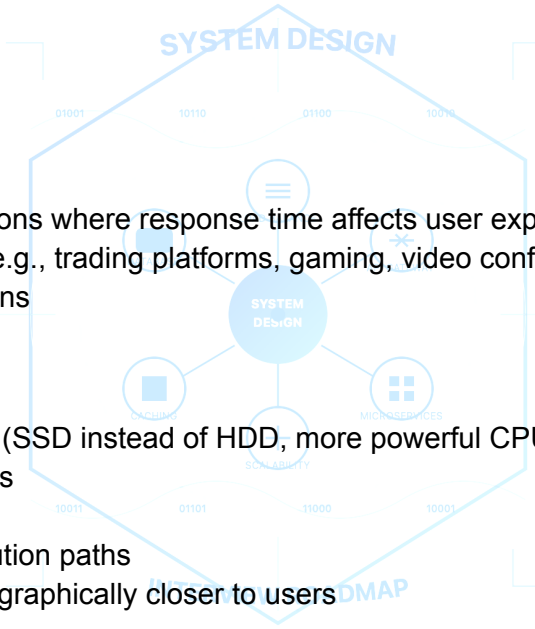
Why Trade-offs Exist

Several factors create these trade-offs:

1. **Resource Constraints**: Systems have finite resources (CPU, memory, network bandwidth).
2. **Batching**: Processing requests in batches increases throughput but adds latency as requests wait for the batch to fill.
3. **Queuing**: High throughput often requires queuing, which adds waiting time (latency).
4. **Parallelization**: Breaking down work for parallel processing improves throughput but introduces coordination overhead that can increase latency.
5. **Hardware Limitations**: Faster processors reduce latency but may consume more power, making them unsuitable for high-density deployments needed for high throughput.

Optimization Strategies

Depending on your system requirements, you might optimize for latency, throughput, or attempt to balance both:



Latency Optimization

When to prioritize latency:

- User-facing applications where response time affects user experience
- Real-time systems (e.g., trading platforms, gaming, video conferencing)
- Critical path operations

Strategies:

- Use faster hardware (SSD instead of HDD, more powerful CPUs)
- Reduce network hops
- Implement caching
- Optimize code execution paths
- Place resources geographically closer to users

Throughput Optimization

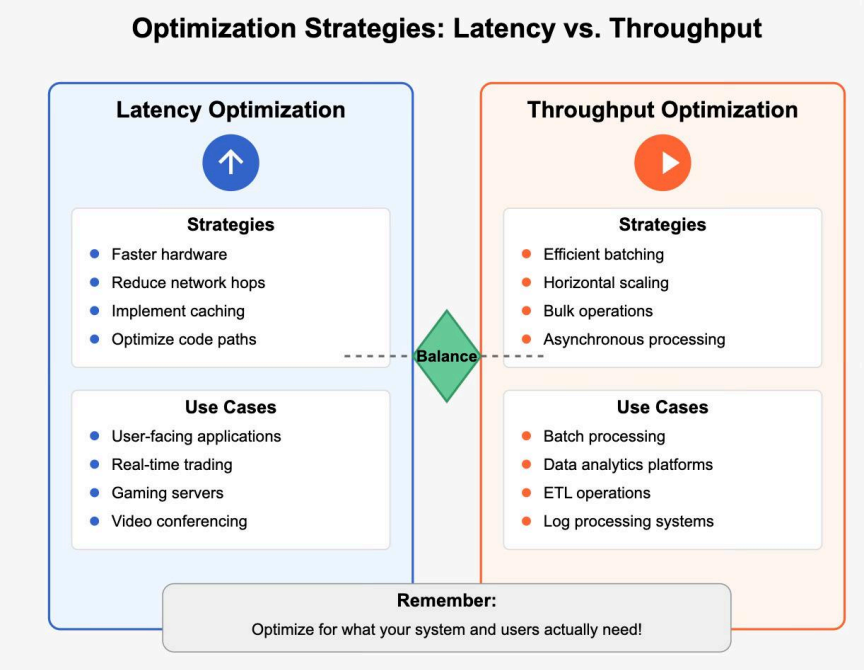
When to prioritize throughput:

- Batch processing systems
- Data analytics platforms
- High-volume transaction systems
- Systems where overall work completion time matters more than individual operation speed

Strategies:

- Implement efficient batching
- Use bulk operations
- Scale horizontally (add more nodes)
- Optimize for consistent, sustainable performance
- Use asynchronous processing

Optimization Strategies for Latency vs. Throughput



Real-World System Profiles

Different systems naturally gravitate toward different points on the latency-throughput spectrum:

Latency-Focused Systems

- **Stock Trading Platforms**: Sub-millisecond latency requirements, relatively low transaction volume
- **Online Gaming**: Low latency critical for user experience (ideally < 100ms)
- **Autonomous Vehicles**: Need real-time processing of sensor data

Throughput-Focused Systems

- **Big Data Processing**: Processing massive datasets where individual record processing time is less critical
- **Video Streaming Services**: Need to deliver petabytes of data efficiently
- **Payment Processors**: Handling millions of transactions per second during peak times

Balanced Systems

- **E-commerce Platforms**: Need reasonable page load times (latency) while handling many simultaneous users (throughput)
- **Social Media Feeds**: Quick loading essential, but must also handle billions of content views

Measurement and Improvement

To effectively optimize your system, follow these steps:

1. **Establish Baselines**: Measure current latency and throughput under various conditions.

2. **Identify Bottlenecks**: Use profiling tools to find what's limiting performance.
3. **Set Clear Goals**: Define acceptable latency percentiles (e.g., 99% of requests under 200ms) and required throughput levels.
4. **Test Incrementally**: Make one change at a time and measure its impact.
5. **Monitor in Production**: Real-world performance often differs from test environments.

Queuing Theory and Little's Law

One fundamental relationship between latency and throughput is expressed in Little's Law, which states:

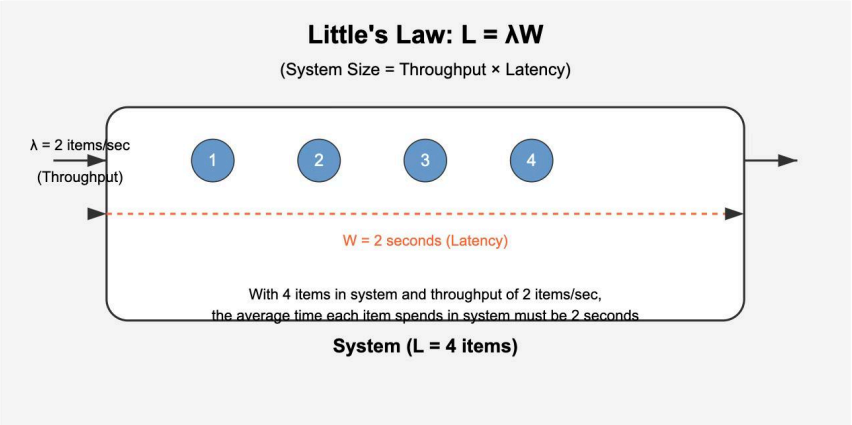
$$L = \lambda W$$

Where:

- L is the average number of items in the system
- λ (lambda) is the average arrival rate (throughput)
- W is the average time an item spends in the system (latency)

This elegant formula demonstrates that if you keep the system capacity (L) constant, you cannot increase throughput without decreasing latency, and vice versa.

Little's Law Visualization



Conclusion

Understanding the relationship between latency and throughput is essential for effective system design. While both metrics are important, most real-world systems must make intentional trade-offs based on their specific requirements. For latency-sensitive applications like gaming or financial trading, design decisions should prioritize minimizing response times. For throughput-oriented systems like batch processors or analytics platforms, maximizing work completed per unit time takes precedence.

The most important approach is to clearly understand your specific use case, measure what matters, and optimize accordingly. Remember that premature optimization is the root of many design problems—start with a clear understanding of your actual requirements before diving into optimizations.

As you build and scale systems, constantly revisit these fundamental concepts, measuring and analyzing how changes affect both latency and throughput. This balanced approach will help you create systems that deliver the right performance characteristics for your specific needs.

4. 5 Key Non-Functional Requirements Every System Designer Should Know

In the world of system design, we often focus heavily on functional requirements—what the system should do. However, equally important are the non-functional requirements (NFRs)—how the system should perform its functions. NFRs define the quality attributes that can make or break your system's success, regardless of how well it fulfills its core functionality.

This article explores the five most critical non-functional requirements that every system designer should understand, prioritize, and implement.

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.

Understanding Non-Functional Requirements

Before diving into specific NFRs, let's clarify what they are and why they matter:

Non-functional requirements define quality attributes and constraints that specify criteria for evaluating the operation of a system, rather than specific behaviors. While functional requirements describe what a system does, non-functional requirements describe how well it does it.

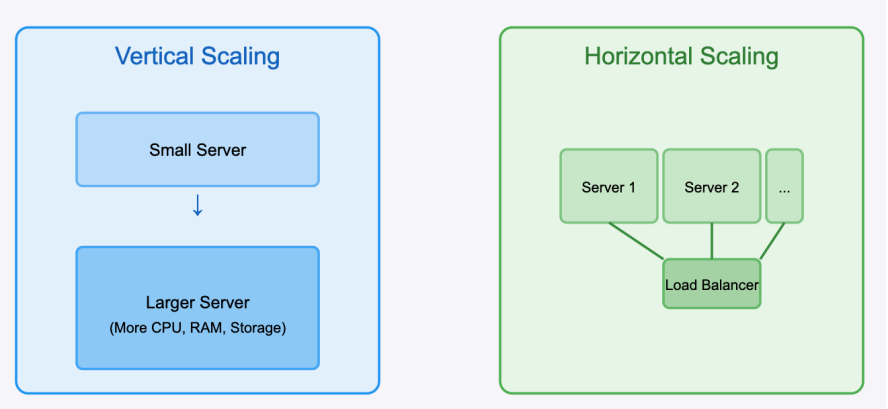
Think of functional requirements as the features of a car (steering, braking, acceleration), while non-functional requirements are qualities like fuel efficiency, safety ratings, and comfort.

The 5 Essential Non-Functional Requirements

1. Scalability

Definition: A system's ability to handle growing amounts of work by adding resources to the system.

Types of Scalability:



- **Vertical Scaling (Scaling Up)**: Adding more resources (CPU, RAM, storage) to existing servers.
- **Horizontal Scaling (Scaling Out)**: Adding more server instances to distribute the load.

Real-World Example:

Netflix handles massive traffic spikes when popular shows are released by implementing auto-scaling groups that can add or remove servers based on demand. Their architecture enables them to serve millions of concurrent streams worldwide.

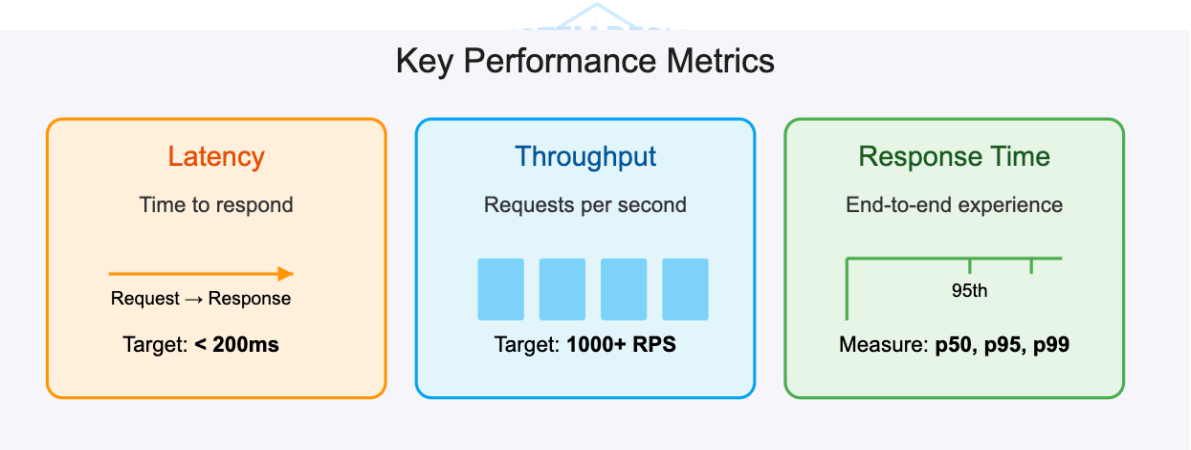
Design Considerations:

- **Stateless Services**: Easier to scale horizontally
- **Database Sharding**: Distributing data across multiple database instances
- **Microservices**: Breaking down applications into independently scalable services
- **Caching**: Reducing load on backend systems

2. Performance

Definition: How quickly a system responds to user actions or processes data.

Key Performance Metrics



Key Performance Metrics:

- **Latency**: Time taken to respond to a request
- **Throughput**: Number of requests a system can handle per unit time
- **Response Time**: Total time from request to response completion
- **Resource Utilization**: How efficiently system resources are used

Real-World Example:

Google's search engine delivers results in milliseconds despite processing billions of searches daily. They achieve this through distributed processing, efficient algorithms, and extensive caching mechanisms.

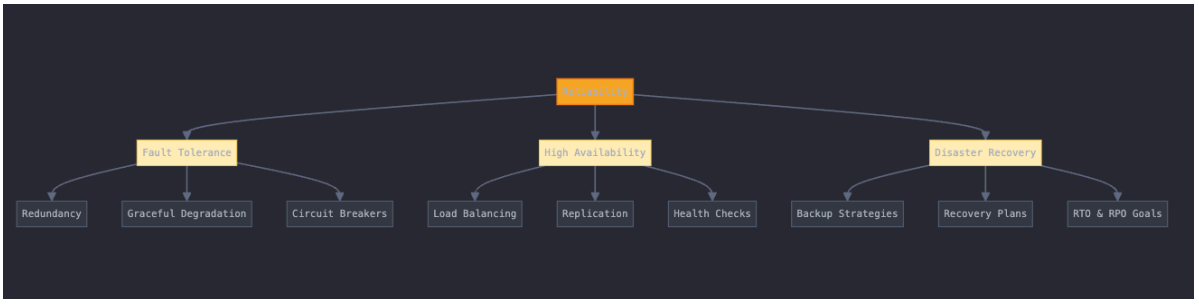
Design Considerations:

- **Caching Strategies**: Implementing various caching layers
- **Asynchronous Processing**: Using message queues for non-critical operations
- **Optimized Database Queries**: Proper indexing and query optimization
- **Content Delivery Networks (CDNs)**: Distributing static content geographically

3. Reliability

Definition: A system's ability to perform its functions correctly and consistently over time.

Reliability Components



Key Components:

- **Fault Tolerance**: The ability to continue functioning when components fail
- **High Availability**: Minimizing downtime, often measured in "nines" (e.g., 99.99% uptime)
- **Disaster Recovery**: Plans and procedures to recover from catastrophic failures

Real-World Example:

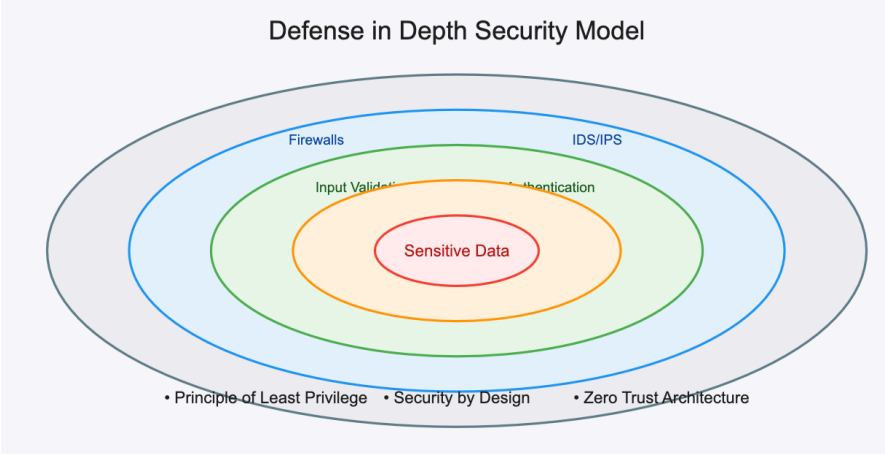
Amazon's shopping platform maintains high reliability during peak events like Prime Day through redundant systems, multiple availability zones, and graceful degradation strategies that prioritize core functionality.

Design Considerations:

- **Redundancy**: Eliminating single points of failure
- **Circuit Breakers**: Preventing cascading failures
- **Chaos Engineering**: Proactively testing resilience
- **Monitoring and Alerting**: Detecting issues before they impact users

4. Security

Definition: Protection of system data, functionality, and infrastructure against unauthorized access or attacks. : Defense in Depth Security Model



Key Security Considerations:

- **Authentication & Authorization**: Verifying identity and access rights
- **Data Encryption**: Protecting data at rest and in transit
- **Input Validation**: Preventing injection attacks
- **Security Monitoring**: Detecting and responding to threats

Real-World Example:

Financial institutions like PayPal implement multiple security layers including encryption, fraud detection algorithms, and strict access controls to protect sensitive financial transactions.

Design Considerations:

- **Defense in Depth**: Multiple security layers
- **Principle of Least Privilege**: Minimum necessary access
- **Regular Security Audits**: Identifying vulnerabilities
- **Compliance Requirements**: Meeting industry standards (GDPR, PCI-DSS, etc.)

5. Maintainability

Definition: The ease with which a system can be modified, updated, or improved over time.

Maintainability Metrics

Key Maintainability Factors:

- **Code Quality**: Well-written, readable, and consistently formatted code
- **Documentation**: Clear system documentation at all levels
- **Testability**: Ease of testing changes and new features
- **Modularity**: Well-defined components with clear interfaces

Real-World Example:

Spotify's engineering culture emphasizes maintainable systems through microservices architecture, automated testing, and clear documentation, allowing them to continuously deploy updates with minimal disruption.

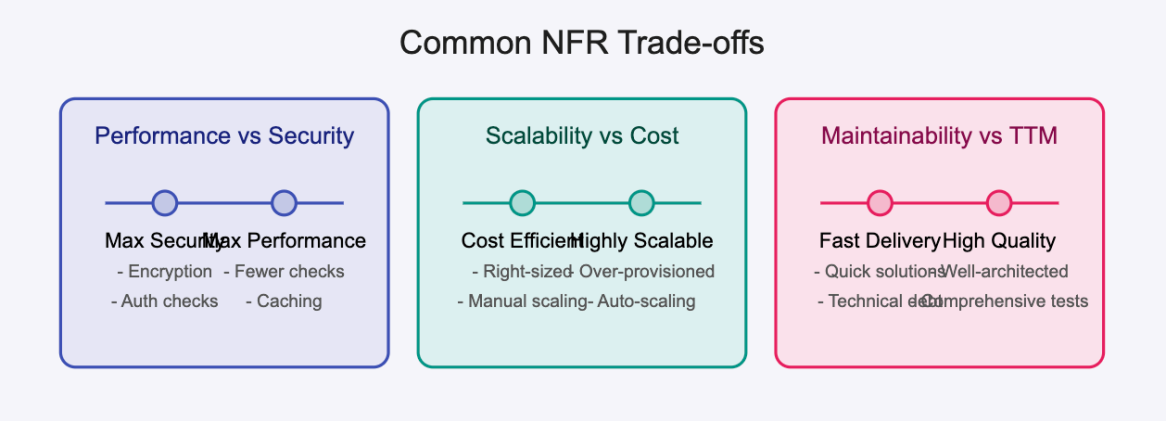
Design Considerations:

- **Clean Code Practices**: Readable, well-structured code
- **Comprehensive Documentation**: Architecture diagrams, API docs, etc.
- **Automated Testing**: Unit, integration, and end-to-end tests
- **Continuous Integration/Deployment**: Streamlined release processes

Balancing and Prioritizing NFRs

Non-functional requirements often involve trade-offs. For example:

NFR Trade-offs



How to Prioritize NFRs:

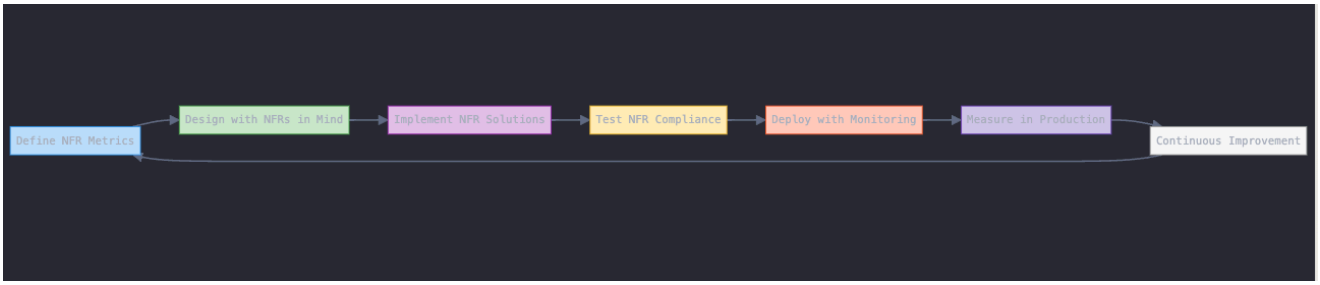
- **Understand Business Requirements**: Align NFRs with business goals
- **Consider User Expectations**: What quality attributes matter most to users?
- **Industry Standards**: Different domains have different NFR priorities
- **Risk Assessment**: Focus on potential failure points
- **Cost-Benefit Analysis**: Evaluate implementation costs against benefits

Implementing and Measuring NFRs

For each non-functional requirement, define:

- **Clear Metrics**: Quantifiable measures of success
- **Testing Strategies**: How to validate each requirement
- **Monitoring Plan**: How to track compliance in production
- **Improvement Process**: How to address shortfalls

NFR Implementation Process



Real-World Case Study: E-Commerce Platform

Let's examine how these NFRs might be applied to an e-commerce platform:

Scalability Requirements:

- **Metric**: Support 10x normal traffic during sales events
- **Implementation**: Auto-scaling infrastructure, CDN for static content, database read replicas
- **Testing**: Load testing with gradually increasing virtual users

Performance Requirements:

- **Metric**: Page load time under 2 seconds, API responses under 200ms
- **Implementation**: Image optimization, frontend caching, database query optimization
- **Testing**: Synthetic monitoring from multiple geographic locations

Reliability Requirements:

- **Metric**: 99.99% uptime for checkout functionality
- **Implementation**: Multiple availability zones, circuit breakers, graceful degradation
- **Testing**: Chaos engineering to simulate component failures

Security Requirements:

- **Metric**: PCI-DSS compliance for payment processing
- **Implementation**: Data encryption, input validation, regular security audits
- **Testing**: Penetration testing, vulnerability scanning

Maintainability Requirements:

- **Metric**: New features deployed within 2 weeks of development completion
- **Implementation**: Microservices architecture, CI/CD pipeline, comprehensive test suite
- **Testing**: Code reviews, static analysis, test coverage metrics

Conclusion

Non-functional requirements are critical to system success but are often overlooked in favor of functional features. By understanding, prioritizing, and implementing these five key NFRs—scalability, performance, reliability, security, and maintainability—system designers can create robust solutions that not only work correctly but also provide an excellent user experience.

Remember that NFRs should be:

- Specific and measurable
- Realistic and achievable
- Balanced against each other
- Continuously monitored and improved

By giving NFRs the attention they deserve early in the design process, you'll avoid costly rework and ensure your systems can meet both current and future demands.

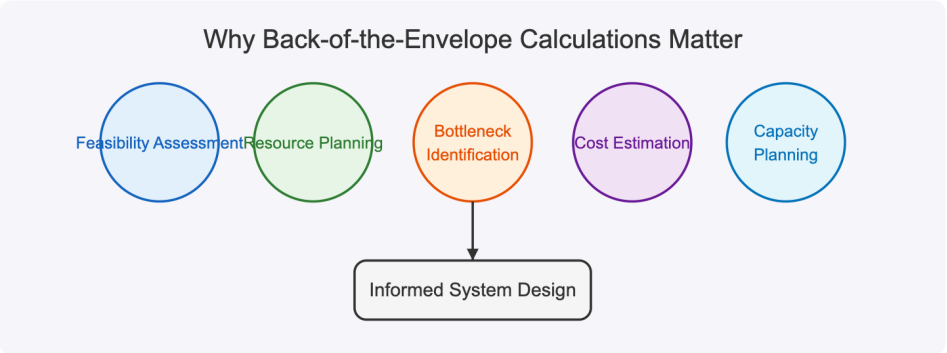
5. Back-of-the-Envelope Calculations for System Design

In system design interviews and real-world architecture planning, the ability to make quick, reasonably accurate estimations—often called "back-of-the-envelope" calculations—is an invaluable skill. These calculations help you determine whether a proposed design is feasible, identify potential bottlenecks, and make informed architectural decisions without getting lost in implementation details.

Why Back-of-the-Envelope Calculations Matter

Back-of-the-envelope calculations serve several critical purposes in system design:

1. **Feasibility Assessment**: Quickly determine if a proposed solution is viable before investing in detailed design
2. **Resource Planning**: Estimate hardware, network, and storage requirements
3. **Bottleneck Identification**: Discover potential performance limitations early
4. **Cost Estimation**: Approximate infrastructure and operational costs
5. **Capacity Planning**: Ensure the system can handle expected load and growth'

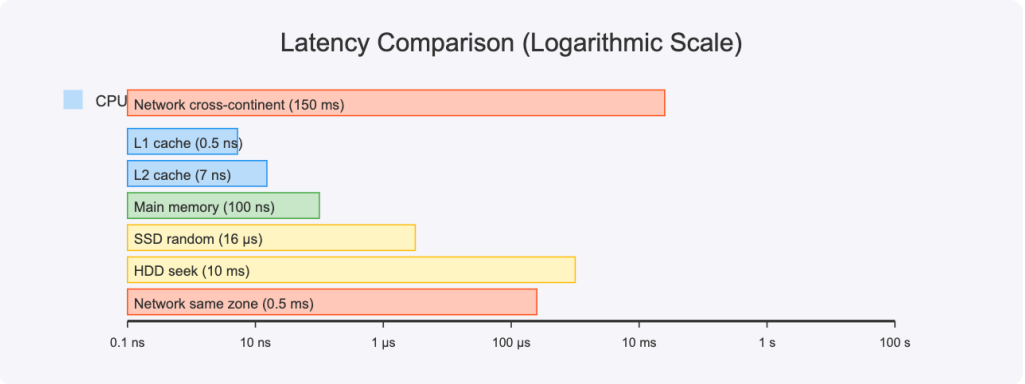


Essential Numbers to Memorize

To perform back-of-the-envelope calculations effectively, you should memorize certain key values. Here are the most important ones:

Time-Related Constants

Operation	Latency
L1 cache reference	0.5 ns
L2 cache reference	7 ns
Main memory access	100 ns
SSD random read	16,000 ns (16 µs)
SSD sequential read	1,000 ns/page
HDD seek	10,000,000 ns (10 ms)
Network: Same zone	500,000 ns (0.5 ms)
Network: Cross-region	150,000,000 ns (150 ms)
Internet: Cross-continent	100-150 ms



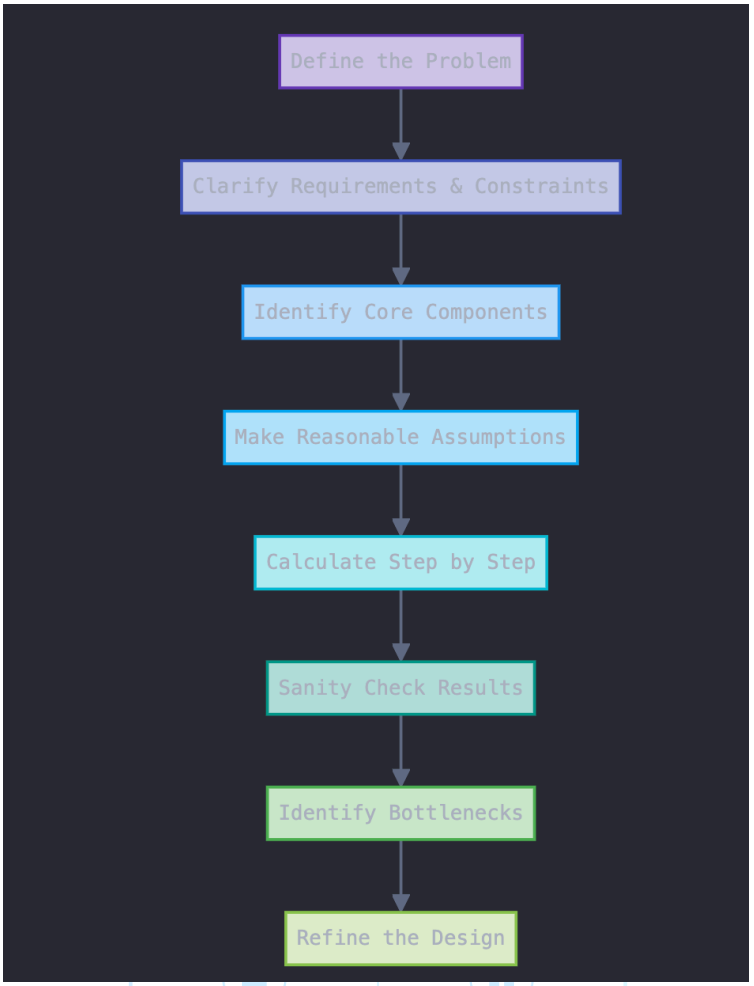
Storage and Data Transfer

Unit	Value
1 byte	8 bits
1 kilobyte (KB)	1,000 bytes or 10^3 bytes
1 megabyte (MB)	1,000 KB or 10^6 bytes
1 gigabyte (GB)	1,000 MB or 10^9 bytes
1 terabyte (TB)	1,000 GB or 10^12 bytes
1 petabyte (PB)	1,000 TB or 10^15 bytes

Common Web Application Numbers

Metric	Approximate Value
Daily active users (DAU) to monthly active users (MAU) ratio	0.1 - 0.3
Average reads to writes ratio	10:1 to 100:1
Average request size	1-10 KB
Average text content storage	1 KB per 1,000 characters
Average image size	200 KB - 5 MB
Average video storage	50 MB - 200 MB per minute
Average database record size	1-10 KB
CDN cache hit ratio	0.8 - 0.95

The Art of Estimation: A Step-by-Step Approach



1. Define the Problem Clearly

Begin by understanding what you're estimating. Is it storage requirements, QPS (queries per second), bandwidth, or something else?

2. Clarify Requirements and Constraints

Identify key metrics like:

- Number of users (daily/monthly active)
- Expected request rates
- Data storage needs
- Latency requirements
- Availability expectations

3. Make Reasonable Assumptions

Document your assumptions clearly. For example:

- Average user session duration
- Read-to-write ratio
- Data growth rate
- Peak-to-average traffic ratio

4. Break Down the Problem

Divide complex estimations into smaller, manageable calculations.

5. Apply Power-of-Ten Approximations

Round numbers to simplify calculations (e.g., use 10^6 instead of 1,234,567).

6. Verify with Sanity Checks

Double-check results against known benchmarks or real-world examples.

7. Identify Potential Bottlenecks

Use your calculations to spot system limitations.

Example 1: URL Shortener System

Let's estimate storage and QPS requirements for a URL shortener service like bit.ly.

Step 1: Define Requirements

We need to estimate:

- Storage requirements for URLs
- QPS the system must handle
- Bandwidth requirements

Step 2: Make Assumptions

1. **Traffic assumptions**:
    a. 100 million URLs shortened per month
    b. Read-to-write ratio: 10:1 (10 URL accesses for each new shortened URL)
2. **Storage assumptions**:
    a. Original URL average length: 100 characters (100 bytes)
    b. Shortened URL length: 7 characters (7 bytes)
    c. Metadata per URL: 50 bytes (timestamps, user ID, etc.)

Step 3: Calculate QPS

New shortened URLs per month: 100 million

- Per day: 100 million / 30 ≈ 3.33 million
- Per second: 3.33 million / 86,400 ≈ 39 URLs/second

URL redirections (reads) per second: 39 × 10 = 390 URLs/second

Total QPS: 39 + 390 = 429 QPS (writes + reads)

Peak QPS (assuming 2× peak factor): 429 × 2 ≈ 858 QPS

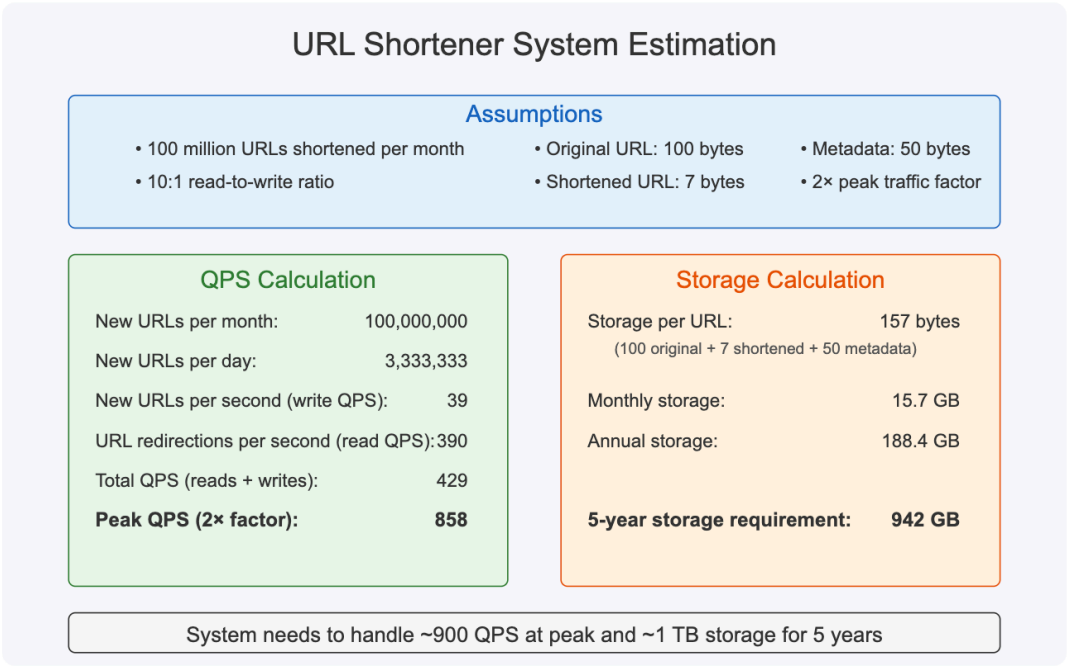
Step 4: Calculate Storage Requirements

Storage per URL: 100 bytes (original URL) + 7 bytes (shortened URL) + 50 bytes (metadata) = 157 bytes

Monthly storage growth: 100 million URLs × 157 bytes ≈ 15.7 GB/month

5-year storage requirement: 15.7 GB × 12 months × 5 years ≈ 942 GB

URL Shortener Estimation



Step 5: System Implications

Based on our calculations:

- QPS requirement (858 at peak) can be handled by a single modern application server
- Storage requirement (~1 TB) is modest and can be handled by a single database server
- We should consider sharding if we expect significant growth beyond our estimates

Example 2: Video Streaming Platform

Let's estimate storage and bandwidth requirements for a YouTube-like video streaming service.

Step 1: Define Requirements

We need to estimate:

- Daily storage requirements for new videos
- Bandwidth requirements for streaming

Step 2: Make Assumptions

1. **Traffic assumptions**:
    - 100 million daily active users (DAU)
    - 5% of users upload a video daily
    - Average user watches 10 videos per day
    - Average watch time: 70% of video length
2. **Video assumptions**:
    - Average video length: 5 minutes
    - Average video size: 50 MB per minute (multiple resolutions)
    - We store 3 resolutions of each video (HD, medium, low)

Step 3: Calculate Storage Requirements

Number of videos uploaded daily: 100 million × 0.05 = 5 million videos

Average video size: 5 minutes × 50 MB/minute × 3 resolutions = 750 MB per video

Daily storage required: 5 million × 750 MB = 3,750 TB (3.75 PB) per day

Annual storage growth: 3.75 PB × 365 = 1,368.75 PB (1.37 EB) per year

Step 4: Calculate Bandwidth Requirements

Views per day: 100 million users × 10 videos = 1 billion video views

Average data per view: 5 minutes × 70% × 3 MB/minute = 10.5 MB per view (assuming average streaming quality is about 3 MB/minute)

Daily bandwidth: 1 billion × 10.5 MB = 10.5 PB per day

Peak concurrent users: 10% of DAU = 10 million users

Peak bandwidth (assuming equal distribution): 10 million × 3 MB/minute = 30 TB/minute = 500 GB/second

Video Platform Requirements

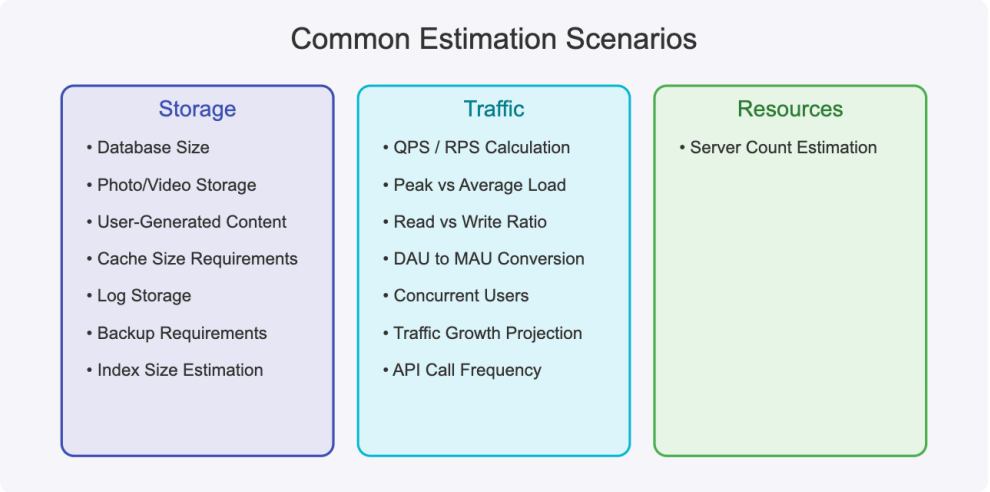
Step 5: System Implications

Based on our calculations:

- Storage requirements are massive, requiring a distributed storage solution
- Bandwidth requirements demand a global CDN infrastructure
- Cost considerations suggest the need for efficient video compression and storage optimization

Common Estimation Scenarios

Here are several common estimation scenarios you might encounter in system design interviews:



6. System Design Fundamentals: Load Balancing

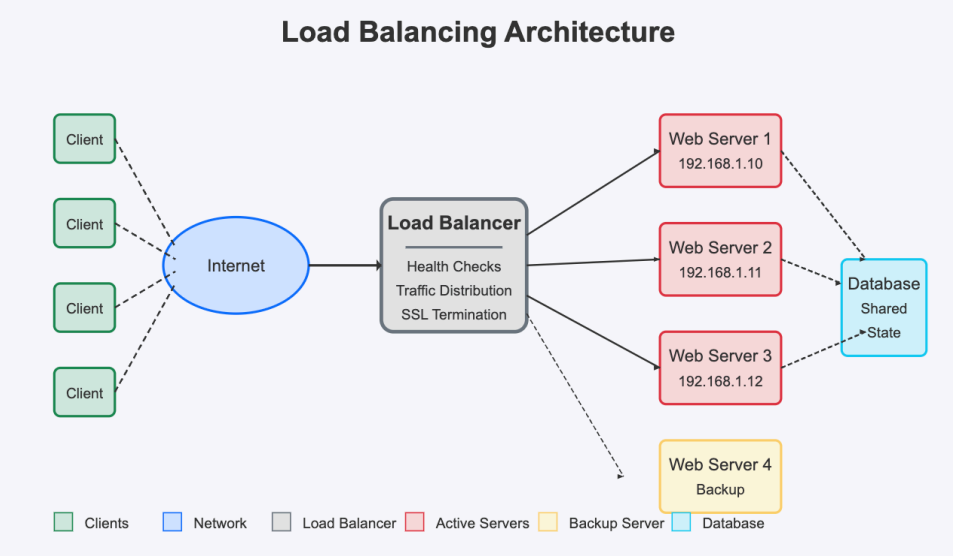
Introduction

Load balancing is a critical component in large-scale distributed systems. It's one of the fundamental building blocks that enables scalability, high availability, and reliability in modern applications. As traffic to an application increases, a single server often becomes insufficient to handle the load. Load balancers solve this problem by efficiently distributing incoming network traffic across multiple servers, ensuring no single server becomes overwhelmed.

What is Load Balancing?

Load balancing refers to the process of distributing network traffic across multiple servers to ensure no single server bears too much demand. By spreading the load, load balancing improves application responsiveness and availability.

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.



Why Load Balancing Matters

- **Scalability** - Allows systems to handle increasing workloads by adding more servers
- **High Availability** - Prevents system failure if one server goes down
- **Reliability** - Ensures consistent performance across varying levels of traffic
- **Efficiency** - Optimizes resource usage across a server fleet

Types of Load Balancers

Hardware vs. Software Load Balancers

Hardware Load Balancers

Hardware load balancers are physical devices optimized for network traffic management. They typically offer high performance but at a higher cost and with less flexibility.

Pros:

- High performance
- Purpose-built reliability
- Vendor support

Cons:

- Expensive
- Limited flexibility
- Physical maintenance required

Examples: F5 BIG-IP, Citrix ADC (formerly NetScaler)

Software Load Balancers

Software load balancers run on standard hardware and offer more flexibility at a lower cost.

Pros:

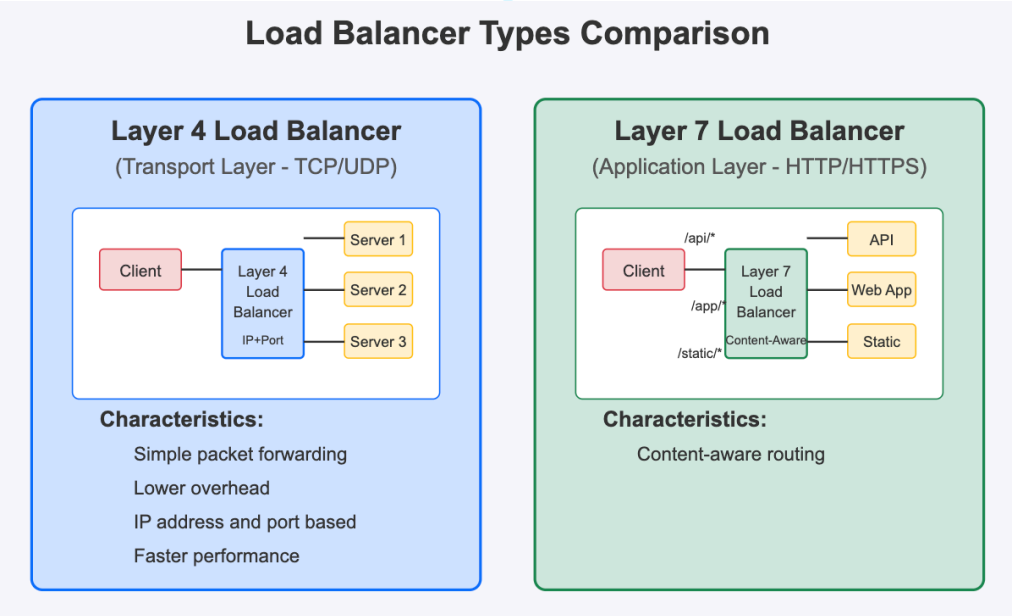
- Cost-effective
- Highly configurable
- Easy to scale

Cons:

- May have lower performance than hardware solutions
- Require operating system maintenance

Examples: NGINX, HAProxy, AWS ELB, Google Cloud Load Balancing

Layer 4 vs. Layer 7 Load Balancers



Layer 4 (Transport Layer) Load Balancers

These operate at the transport layer (TCP/UDP) and route traffic based on network information like IP addresses and ports without inspecting packet content.

Characteristics:

- Simple and fast
- Lower overhead
- Limited routing capabilities
- Makes decisions based on IP address and port

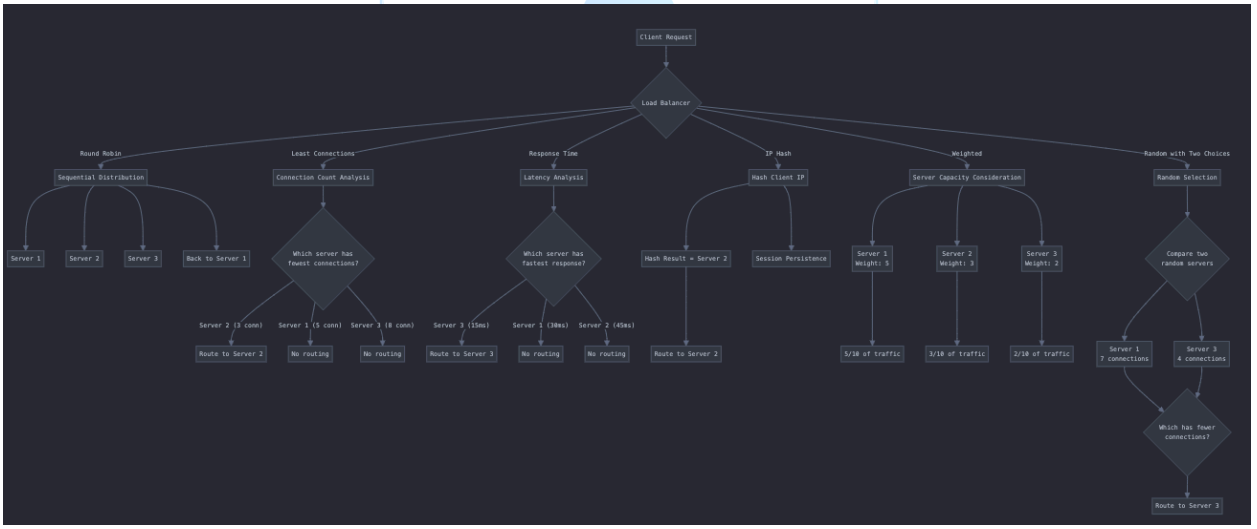
Layer 7 (Application Layer) Load Balancers

These operate at the application layer (HTTP/HTTPS) and can make routing decisions based on the content of the request (URL, headers, cookies, etc.).

Characteristics:

- Content-aware routing
- Can handle SSL termination
- More sophisticated distribution policies
- Higher computational overhead

Common Load Balancing Algorithms



1. Round Robin

Requests are distributed sequentially across the server group. Simple but doesn't account for varying server capacity or load.

2. Least Connections

Traffic is sent to the server with the fewest active connections. Better for applications with varying connection times.

3. Least Response Time

Directs traffic to the server with the lowest response time, which indicates the fastest server

4. IP Hash

Uses the client's IP address to determine which server receives the request. Ensures a client always connects to the same server, which is useful for session persistence.

5. Weighted Round Robin/Least Connections

Similar to their standard counterparts but with server weights that allow more powerful servers to handle more connections.

6. Random with Two Choices

Randomly selects two servers and then chooses the one with fewer connections, combining the speed of random selection with some load awareness.

Load Balancer Features and Capabilities

Health Checks

Load balancers regularly check server health by sending test requests. If a server fails to respond properly, it's temporarily removed from the pool.

Session Persistence (Sticky Sessions)

Ensures that a client's requests are directed to the same server for the duration of a session. Essential for applications that store session information locally.

SSL Termination

The load balancer handles SSL encryption/decryption, relieving backend servers from this computational overhead.

Content-Based Routing

Layer 7 load balancers can route requests to different server pools based on the request content (URL, headers, etc.).

Auto-Scaling Integration

Modern load balancers can integrate with auto-scaling services to dynamically adjust the server pool size based on traffic.

Real-World Implementation Examples

Example 1: Web Application with Nginx

A typical setup might use Nginx as a load balancer for a web application:

```
http {
    upstream web_backend {
        server backend1.example.com weight=5;
        server backend2.example.com weight=5;
        server backend3.example.com weight=3;
        server backup1.example.com backup;
        server backup2.example.com backup;
    }

    server {
        listen 80;
        server_name example.com;

        location / {
            proxy_pass http://web_backend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

In this configuration:

- Three active backend servers with different weights
- Two backup servers that only receive traffic if the primary servers fail
- HTTP headers are properly forwarded to the backend

Example 2: AWS Elastic Load Balancing

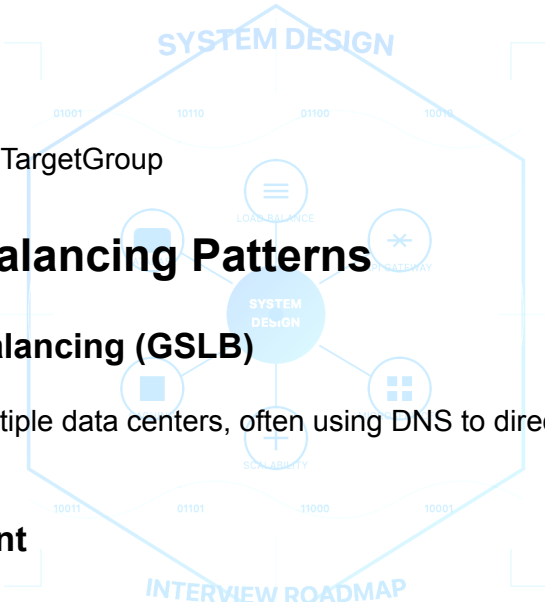
AWS offers multiple load balancing options:

1. **Application Load Balancer (ALB)** - For HTTP/HTTPS traffic, with content-based routing
2. **Network Load Balancer (NLB)** - For TCP/UDP traffic requiring ultra-low latency
3. **Classic Load Balancer** - Legacy option with basic functionalit

A typical AWS deployment might use ALB with auto-scaling:

```
# AWS CloudFormation snippet
Resources:
  ApplicationLoadBalancer:
    Type: AWS::ElasticLoadBalancingV2::LoadBalancer
    Properties:
      Subnets:
        - subnet-12345678
```

```
      - subnet-87654321
    SecurityGroups:
      - sg-12345678

  TargetGroup:
    Type: AWS::ElasticLoadBalancingV2::TargetGroup
    Properties:
      VpcId: vpc-12345678
      Port: 80
      Protocol: HTTP
      HealthCheckPath: /health
      HealthCheckIntervalSeconds: 30

  Listener:
    Type: AWS::ElasticLoadBalancingV2::Listener
    Properties:
      LoadBalancerArn: !Ref ApplicationLoadBalancer
      Port: 80
      Protocol: HTTP
      DefaultActions:
        - Type: forward
          TargetGroupArn: !Ref TargetGroup
```



Common Load Balancing Patterns

Global Server Load Balancing (GSLB)

Distributes traffic across multiple data centers, often using DNS to direct users to the nearest or most available data center.

Blue-Green Deployment

Two identical environments (blue and green) exist with only one active at a time. The load balancer switches traffic from blue to green during deployments.

Canary Deployment

The load balancer directs a small percentage of traffic to a new version of the application for testing before full deployment.

Best Practices for Load Balancing

1. **Implement proper health checks** - Customize health checks to verify actual application functionality, not just that the server responds.
2. **Design for failure** - Assume servers will fail & ensure the load balancer can handle these failures gracefully.
3. **Monitor load balancer performance** - Track metrics like latency, connection counts, error rates.

4. **Configure timeouts appropriately** - Set reasonable timeouts to prevent hanging connections from degrading performance.
5. **Use consistent hashing for session persistence** - When sticky sessions are needed, consistent hashing minimizes disruption when servers are added or removed.
6. **Consider SSL offloading** - Have the load balancer handle SSL to reduce backend server load.
7. **Implement rate limiting** - Protect backend services from traffic spikes or potential DDoS attacks.
8. **Plan for load balancer redundancy** - The load balancer itself can be a single point of failure if not made redundant.

Common Load Balancing Challenges and Solutions

Challenge: Session Persistence

Solution: Use sticky sessions, distributed caching (Redis/Memcached), or move session data to a shared database.

Challenge: SSL Certificate Management

Solution: Centralize SSL termination at the load balancer with automated certificate renewal.

Challenge: Uneven Load Distribution

Solution: Use more sophisticated algorithms like least connections or response time, and implement proper server weighting.

Challenge: Load Balancer Becoming a Bottleneck

Solution: Implement a hierarchical load balancing structure or use DNS-based load balancing to distribute traffic before it reaches your primary load balancer.

Conclusion

Load balancing is a fundamental technique for building scalable and reliable systems. The right load balancing strategy depends on your specific application requirements, expected traffic patterns, and infrastructure constraints. As systems grow more complex, sophisticated load balancing techniques become increasingly important for maintaining performance and reliability.

Whether you're using hardware appliances, software solutions, or cloud-based services, understanding the principles and best practices of load balancing will help you design more robust distributed systems that can handle growth and provide consistent user experiences even under high load.

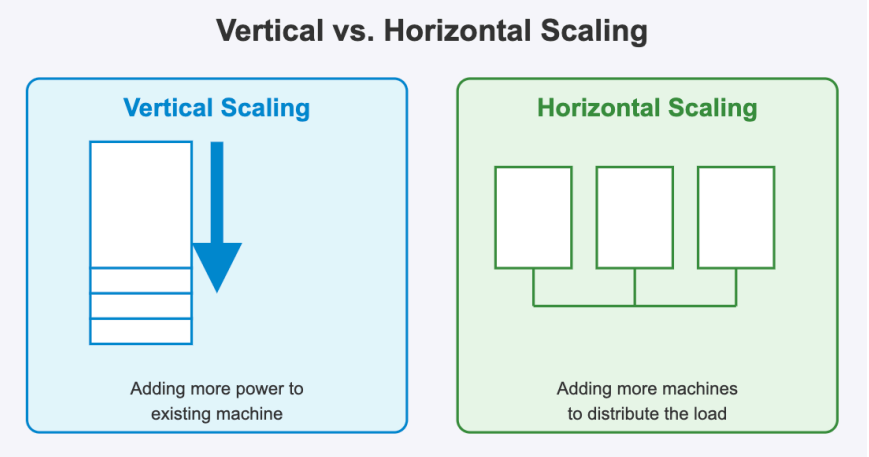
7. Vertical vs. Horizontal Scaling: When to Choose Each

Introduction

Scaling is a critical aspect of system design that determines how your application can handle growth in traffic, data, and complexity. In this article, we'll explore the two primary approaches to scaling: vertical and horizontal. We'll examine their characteristics, advantages, disadvantages, and provide guidance on when to choose each strategy.

Vertical vs. Horizontal Scaling Comparison

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.



What is Vertical Scaling?

Vertical scaling, also known as "scaling up," involves increasing the capacity of a single server by adding more resources such as CPU, RAM, or storage. Think of it as making your existing machine more powerful rather than adding more machines.

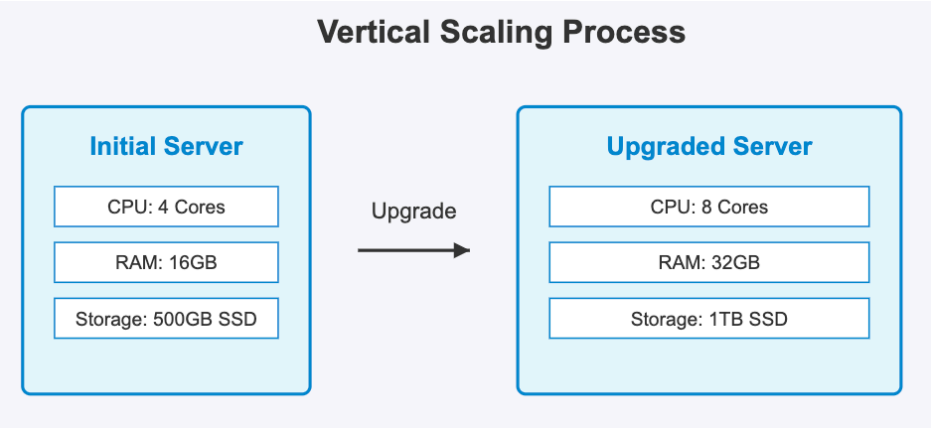
Characteristics of Vertical Scaling:

- **Single Server Focus**: Resources are added to a single machine.
- **Hardware Upgrades**: Typically involves replacing or upgrading hardware components.
- **Simplicity**: Maintains the same architecture but with more powerful hardware.
- **Limited Scalability**: Eventually hits hardware limitations and cost barriers.

Real-World Example:

Imagine a small e-commerce site running on a single server with 4 CPU cores and 16GB RAM. As traffic increases, the site becomes slow during peak hours. With vertical scaling, you would upgrade to a server with 8 CPU cores and 32GB RAM to handle the increased load.

Vertical Scaling Process



What is Horizontal Scaling?

Horizontal scaling, also known as "scaling out," involves adding more machines to your system and distributing the load across multiple servers. Rather than making one server more powerful, you add more servers of the same or similar capacity.

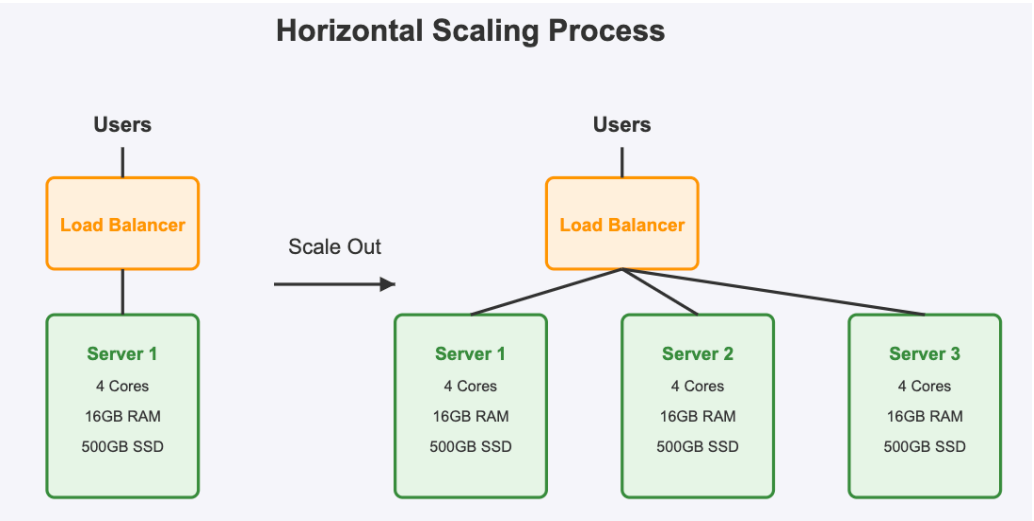
Characteristics of Horizontal Scaling:

- **Multiple Servers**: Adds more machines to distribute workload.
- **Load Distribution**: Requires load balancing to distribute traffic.
- **Resilience**: Improves fault tolerance as the system can survive single server failures.
- **Elasticity**: Can dynamically add or remove capacity based on demand.

Real-World Example:

Using the same e-commerce site example, instead of upgrading a single server, you would add two more servers with the same 4 CPU cores and 16GB RAM configuration. A load balancer would distribute incoming traffic across all three servers.

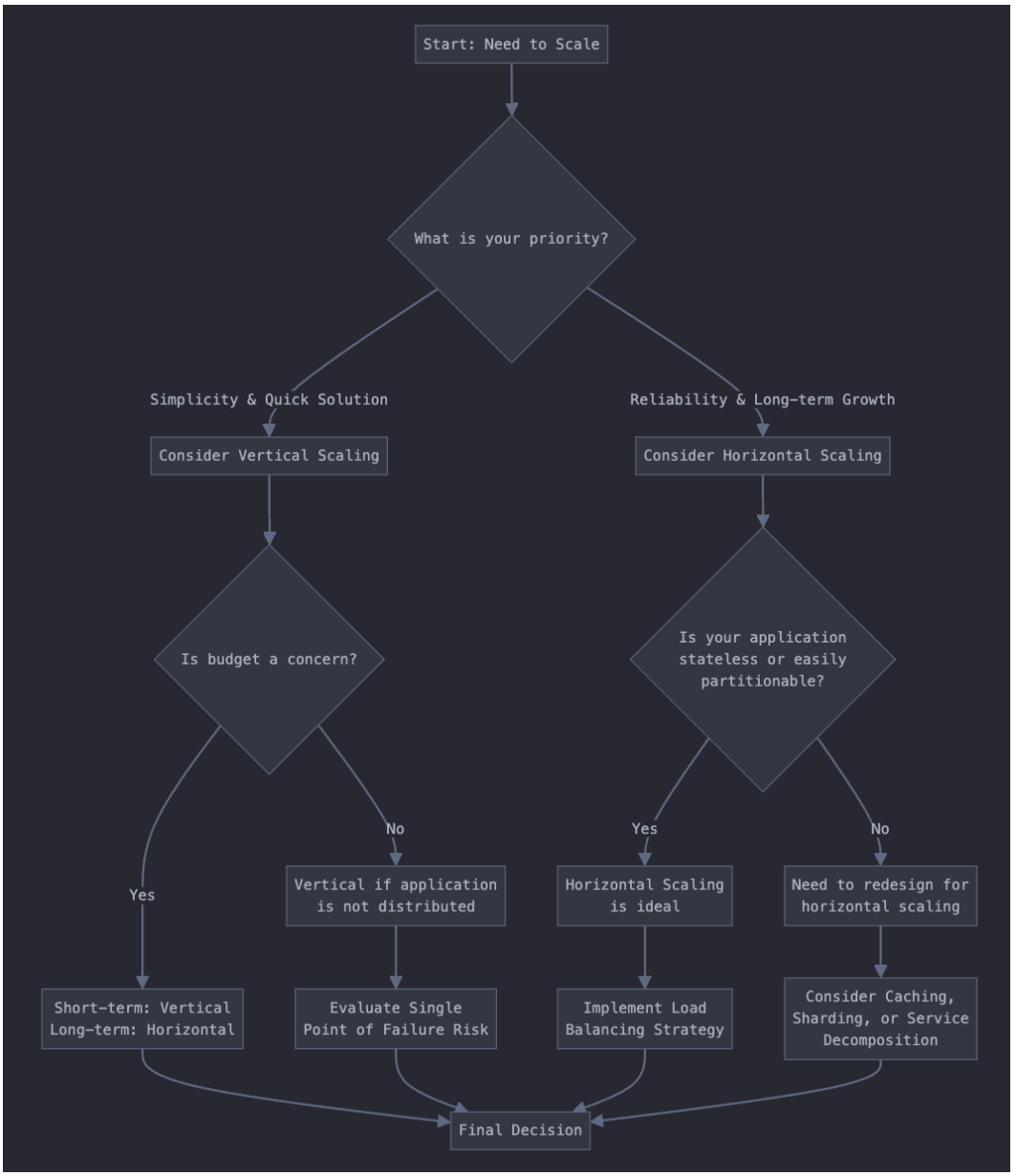
Horizontal Scaling Process



Comparing Vertical and Horizontal Scaling

To help you understand when to choose each approach, let's compare them across several key factors:

Decision Flowchart: Vertical vs. Horizontal Scaling



1. Implementation Complexity

- **Vertical Scaling**: ⭐⭐☆☆☆ (Low to Medium)
  - Relatively simple to implement
  - Often just involves hardware upgrades
  - Minimal code or architecture changes
- **Horizontal Scaling**: ⭐⭐⭐⭐☆ (Medium to High)
  - Requires load balancing
  - Applications must be designed to work across multiple servers
  - May require session management, distributed caching, etc.

2. Cost Considerations

- **Vertical Scaling**:
  - Higher per-unit cost (high-end hardware is expensive)
  - Cost increases non-linearly with capacity
  - May require specialized hardware
- **Horizontal Scaling**:
  - Lower per-unit cost (uses commodity hardware)
  - More linear cost scaling
  - Better resource utilization potential

3. Scalability Limits

- **Vertical Scaling**:
  - Limited by maximum hardware specifications
  - Practical limits to CPU, RAM, and storage
  - Diminishing returns as you approach hardware limits
- **Horizontal Scaling**:
  - Theoretically unlimited scalability
  - Can add machines as needed
  - Used by the largest systems in the world

4. Fault Tolerance

- **Vertical Scaling**:
  - Single point of failure
  - Hardware failure means complete system outage
  - Requires additional redundancy measures
- **Horizontal Scaling**:
  - Built-in redundancy
  - System can survive multiple server failures
  - Enables rolling updates and zero-downtime deployments

5. Performance Characteristics

- **Vertical Scaling**:
  - Better for computation-intensive workloads
  - Lower latency (no network overhead)
  - Simpler data consistency model
- **Horizontal Scaling**:
  - Better for I/O-bound or parallel workloads
  - Higher throughput potential
  - Might introduce network latency

When to Choose Vertical Scaling

Vertical scaling is typically the right choice when:

1. **You're in the early stages** of your application with lower traffic or data volumes
2. **Your application is monolithic** and not designed for distribution

3. **You need a quick solution** without significant architecture changes
4. **Your workloads are computation-intensive** and benefit from more powerful CPUs
5. **Your system has predictable, steady growth** rather than spiky traffic patterns
6. **You're dealing with systems that are difficult to distribute** (e.g., certain databases)
7. **Budget constraints require short-term solutions**

Case Study: Financial Trading System

A high-frequency trading system processes market data and executes trades within microseconds. In this scenario, the latency introduced by network communication between multiple servers would be unacceptable. Vertical scaling with high-performance CPUs and specialized hardware is the preferred approach despite the higher cost.

When to Choose Horizontal Scaling

Horizontal scaling is typically the better choice when:

1. **You expect significant growth** or highly variable traffic
2. **High availability is critical** to your business
3. **Your application is designed to be stateless** or has easily partitionable state
4. **Your workloads benefit from parallelization**
5. **You want to optimize for cost efficiency** at large scale
6. **You need geographic distribution** for lower latency or regulatory compliance
7. **You want more flexible capacity management**

Case Study: Social Media Platform

A social media platform needs to handle millions of users, with traffic spikes during major events. The platform stores user data, photos, videos, and processes a wide variety of activities. This scenario perfectly suits horizontal scaling, where the load can be distributed across hundreds or thousands of servers, and capacity can be added dynamically based on demand.

Scaling Decision Matrix

Scaling Decision Matrix

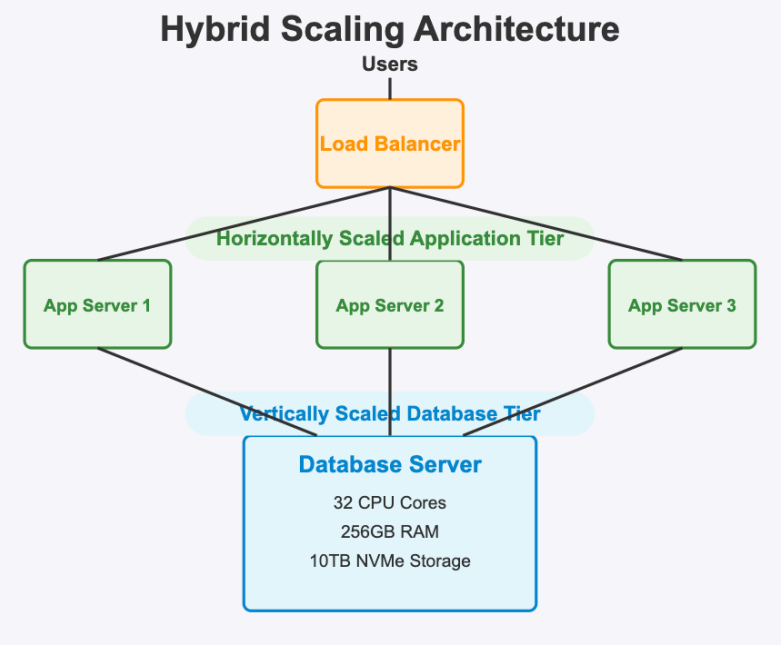
Factor	Vertical Scaling	Horizontal Scaling
Implementation Complexity	Low to Medium	Medium to High
Cost Structure	Higher per-unit cost Non-linear scaling	Lower per-unit cost Linear scaling
Scalability Limits	Limited by hardware maximums	Theoretically unlimited
Fault Tolerance	Single point of failure	Built-in redundancy
Ideal Use Cases	• Monolithic applications • Computation-intensive tasks • Early-stage applications	• Stateless applications • High availability needs • Variable traffic patterns

Hybrid Scaling Approach

In practice, many systems use a hybrid approach that combines both vertical and horizontal scaling:

1. **Vertical First, Horizontal Later**: Start with vertical scaling for simplicity, then transition to horizontal as your needs grow
2. **Tier-Specific Scaling**: Apply different scaling strategies to different tiers of your architecture
    a. Database tier: Often vertically scaled for consistency and performance
    b. Application tier: Horizontally scaled for fault tolerance and throughput
    c. Caching tier: Horizontally scaled for capacity and performance

Example Hybrid Architecture:



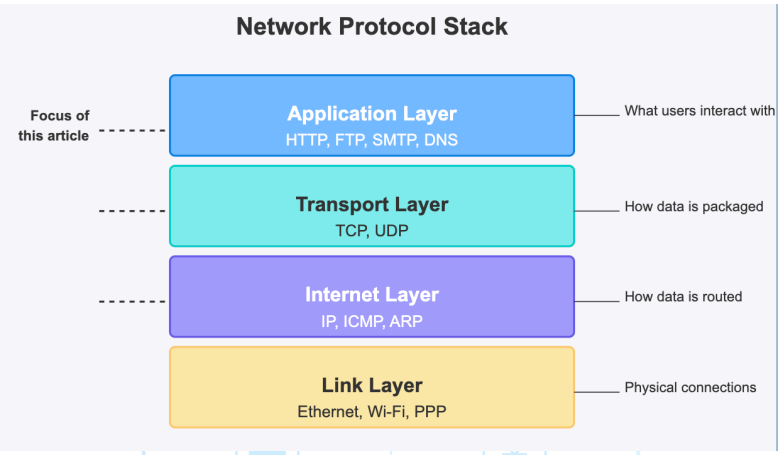
8. Understanding Network Protocols: HTTP, TCP/IP, UDP

Introduction

Network protocols are the fundamental rules that govern how data is transmitted across networks. They're like the languages that allow different devices to communicate effectively. This article dives deep into three of the most critical protocols in modern system design: HTTP, TCP/IP, and UDP, explaining how they work, their key differences, and when to use each.

Network Protocol Stack

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.



TCP/IP: The Foundation of Internet Communication

What is TCP/IP?

The Transmission Control Protocol/Internet Protocol (TCP/IP) is the fundamental suite of protocols that powers the internet. It's not a single protocol but rather a collection of protocols that work together in layers. The two main components are:

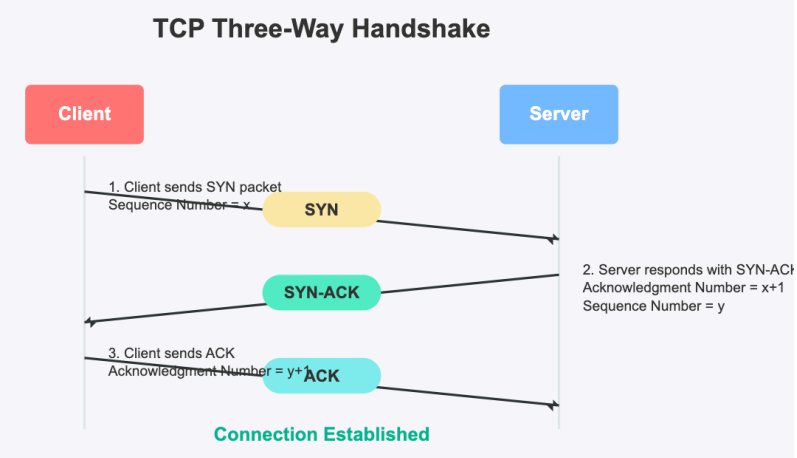
1. **TCP (Transmission Control Protocol)**: Handles the reliable delivery of data
2. **IP (Internet Protocol)**: Manages addressing and routing of data packets

How TCP/IP Works

TCP/IP follows a four-layer model:

1. **Link Layer**: Handles physical connections (Ethernet, Wi-Fi)
2. **Internet Layer**: Manages addressing and routing (IP)
3. **Transport Layer**: Ensures reliable data delivery (TCP, UDP)
4. **Application Layer**: Interfaces with applications (HTTP, FTP, etc.)

TCP Three-Way Handshake



The TCP Three-Way Handshake

One of TCP's most important features is its connection establishment process, known as the "three-way handshake":

1. **SYN**: Client sends a SYN (synchronize) packet with an initial sequence number (x)
2. **SYN-ACK**: Server responds with a SYN-ACK packet, acknowledging the client's sequence number (x+1) and providing its own sequence number (y)
3. **ACK**: Client sends an ACK packet, acknowledging the server's sequence number (y+1)

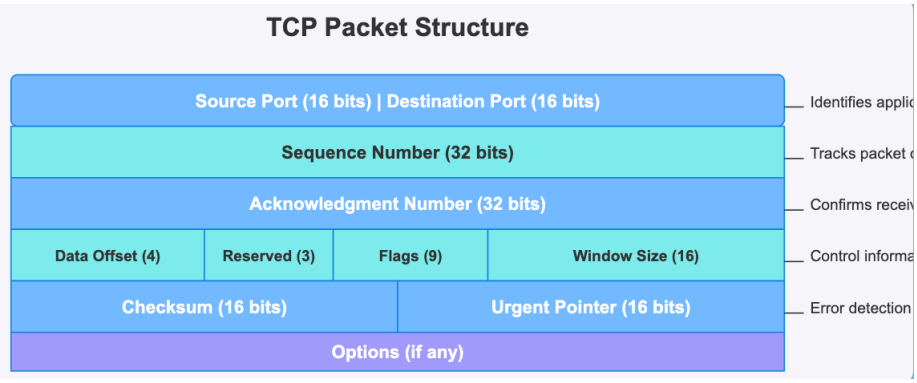
This process ensures both parties are ready to communicate and establishes sequence numbers for tracking packets.

Key Features of TCP

1. **Connection-oriented**: Establishes a dedicated connection before data transfer
2. **Reliable delivery**: Guarantees that data arrives intact and in order
3. **Flow control**: Prevents overwhelming receivers with too much data
4. **Congestion control**: Adjusts transmission rates based on network conditions
5. **Error detection and recovery**: Detects and retransmits lost packets

TCP Packet Structure

TCP Packet Structure



Each TCP packet includes:

- **Source and Destination Ports**: Identify the applications at both ends
- **Sequence Number**: Orders packets and tracks how much data has been sent
- **Acknowledgment Number**: Indicates the next expected sequence number
- **Flags**: Control bits like SYN, ACK, FIN for connection management
- **Window Size**: Flow control mechanism indicating buffer capacity
- **Checksum**: Ensures data integrity
- **Options**: Additional features like maximum segment size

UDP: The Lightweight Alternative

What is UDP?

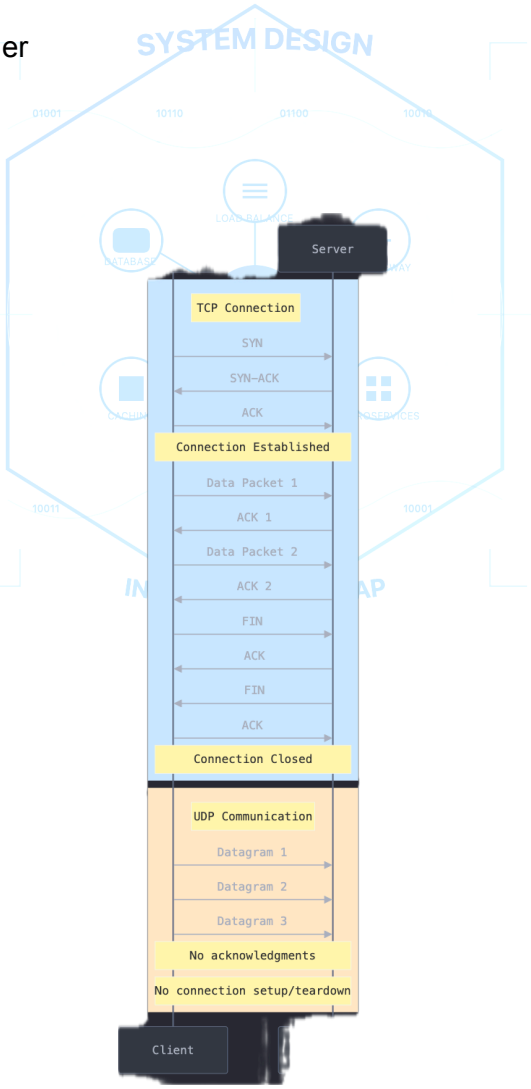
User Datagram Protocol (UDP) is a connectionless transport protocol that provides a simpler, faster alternative to TCP. While TCP prioritizes reliability, UDP prioritizes speed.

How UDP Works

UDP simply sends packets (called datagrams) to the recipient without:

- Establishing a connection
- Guaranteeing delivery
- Maintaining packet order
- Managing flow control

TCP vs UDP Communication

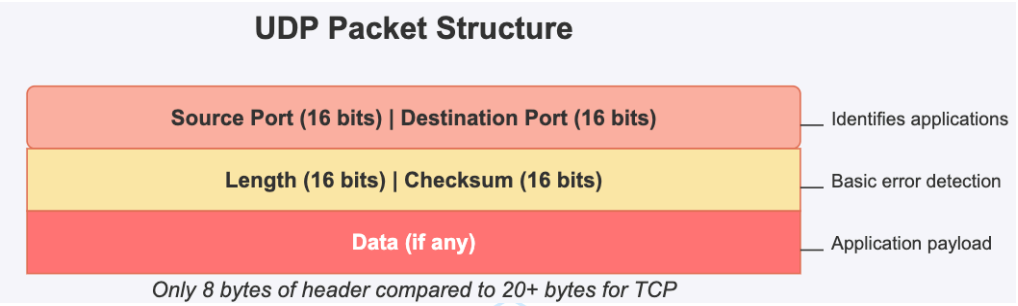


Key Features of UDP

1. **Connectionless**: No handshake required before sending data
2. **Low overhead**: Smaller header size and no connection management
3. **Fast transmission**: No waiting for acknowledgments
4. **No guarantees**: Packets may be lost, duplicated, or arrive out of order
5. **No congestion control**: Sends at a constant rate regardless of network conditions

UDP Packet Structure

UDP Packet Structure

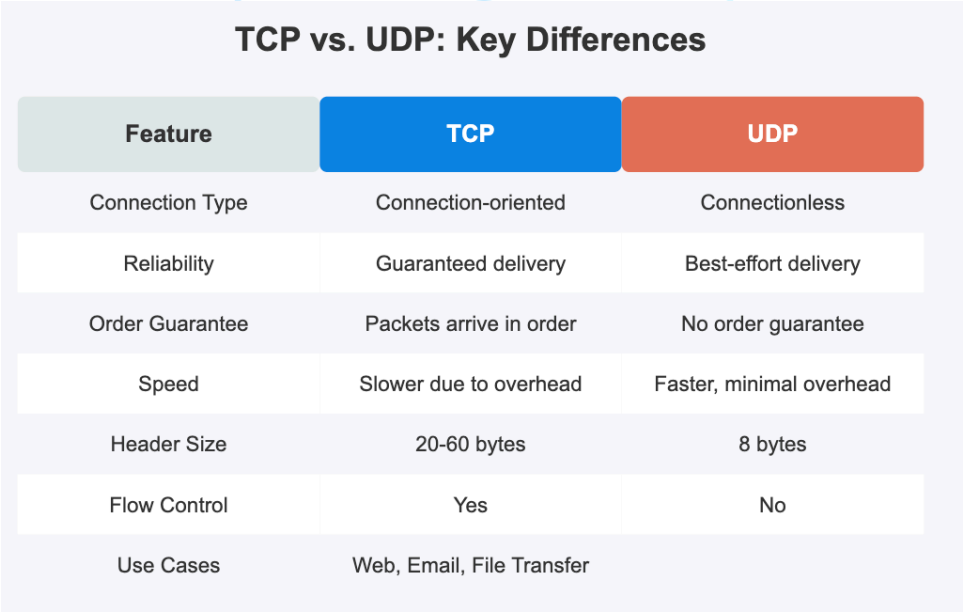


UDP has a much simpler header than TCP with just four fields:

- **Source Port**: Identifies the sending application (optional, can be zero)
- **Destination Port**: Identifies the receiving application
- **Length**: Total length of the datagram (header + data)
- **Checksum**: Basic error detection (optional in IPv4, required in IPv6)

TCP vs. UDP: When to Use Each

TCP vs UDP Comparison



9. API Design Fundamentals: REST vs. GraphQL vs. gRPC

APIs (Application Programming Interfaces) are the building blocks for modern application development. In this article, we explore three popular paradigms—REST, GraphQL, and gRPC—to help you understand their design principles, advantages, trade-offs, and when to use each one.

---

1. Introduction to API Design

APIs allow different software systems to communicate with each other. When designing an API, it's important to consider factors such as performance, scalability, ease of integration, and the needs of both your service and its consumers. Here, we focus on three major API paradigms:

- **REST:** A resource-oriented architecture that uses standard HTTP methods.
- **GraphQL:** A query language that allows clients to request exactly the data they need.
- **gRPC:** A high-performance, open-source RPC framework that uses protocol buffers for message serialization.

---

2. REST APIs

Overview

REST (Representational State Transfer) is an architectural style that leverages HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources. REST APIs are known for their simplicity and scalability.

Key Concepts

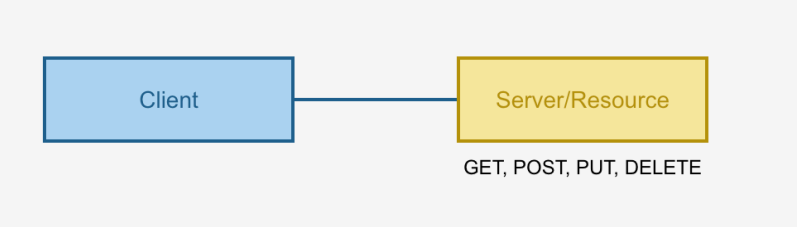
- **Resource-Based:** Every resource (e.g., user, product) is identified by a URI.
- **Stateless:** Each request contains all necessary information, and no client context is stored on the server.
- **Standard Methods:** Utilizes HTTP methods to perform operations.

Example Use Case

Imagine an e-commerce application. A REST API might expose endpoints such as:

- GET /products to retrieve a list of products.
- POST /orders to create a new order.
- PUT /users/{id} to update user details.

REST Flowchart Diagram



---

3. GraphQL

Overview

GraphQL is a query language for APIs developed by Facebook. It enables clients to request exactly the data they need and nothing more, reducing over-fetching and under-fetching of data.

Key Concepts

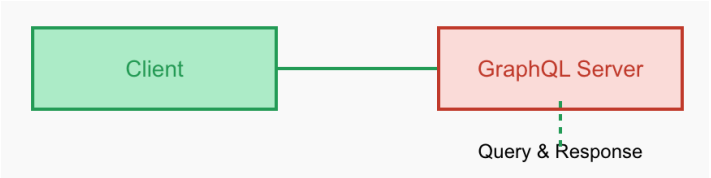
- **Single Endpoint:** Unlike REST, GraphQL APIs typically expose a single endpoint.
- **Flexible Queries:** Clients specify queries that define the structure of the expected response.
- **Type System:** GraphQL uses a strong type system to define the schema, ensuring robust and predictable APIs.

Example Use Case

For a social media platform, a GraphQL API might allow a client to request a user's profile along with their posts and comments in one query:

```
{
  user(id: "123") {
    name
    posts {
      title
      comments {
        text
      }
    }
  }
}
```

GraphQL Query Flow Diagram



---

4. gRPC

Overview

gRPC is a modern, high-performance remote procedure call (RPC) framework that uses HTTP/2 and protocol buffers for serialization. It is designed for low latency and efficient communication, particularly in microservices environments.

Key Concepts

- **RPC Paradigm:** Methods are defined in a service interface and invoked remotely as if they were local calls.
- **Protocol Buffers:** gRPC uses Protobufs for defining the API and serializing structured data.
- **HTTP/2 Benefits:** Supports multiplexed streams, header compression, and bi-directional communication.

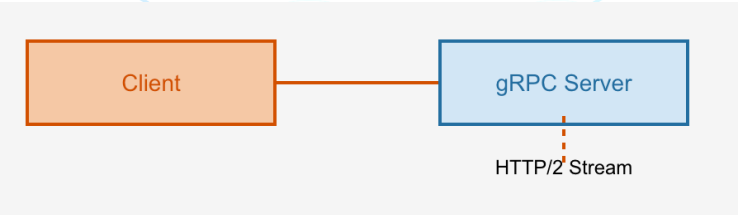
Example Use Case

Consider a real-time data processing system where latency is critical. A gRPC service might define a method like:

```
service DataService {
  rpc GetData (DataRequest) returns (DataResponse);
}
```

The client calls GetData, and the server responds with the structured data, all over a single, efficient HTTP/2 connection.

gRPC Communication Diagram



---

5. Comparison: When to Use Each

REST

- Pros:
  - Easy to implement with standard HTTP.
  - Widely supported by browsers and frameworks.
  - Stateless architecture makes scaling simpler.
- Cons:
  - Over-fetching or under-fetching of data.
  - Versioning challenges when evolving APIs.

GraphQL

- **Pros:**
  - Clients get exactly the data they need.
  - Single endpoint simplifies network traffic.
  - Strong schema enforces type safety.

- **Cons:**
  - Increased complexity in caching.
  - More challenging to optimize for performance with complex queries.

gRPC

- **Pros:**
  - High performance and low latency.
  - Efficient binary serialization with protocol buffers.
  - Built-in support for bi-directional streaming.
- **Cons:**
  - Steeper learning curve, especially for teams new to protocol buffers.
  - Limited browser support without additional tools or proxies.

---

6. Real-World Examples

- **REST Example:**
  A public API like GitHub's REST API allows developers to access user profiles, repositories, and commits using standard HTTP calls.
- **GraphQL Example:**
  Facebook and GitHub use GraphQL to allow clients to retrieve complex, nested data in a single request without multiple round trips.
- **gRPC Example:**
  Google uses gRPC for internal microservices communication, where speed and efficiency are paramount for handling billions of requests.

---

7. Best Practices for API Design

- **Consistency:**
  Ensure your API follows consistent naming conventions and structure.
- **Documentation:**
  Use tools like Swagger (for REST) or GraphiQL (for GraphQL) to document and test your API.
- **Error Handling:**
  Provide clear error messages and use standard error codes.

- **Security:**
  Implement proper authentication and authorization, use HTTPS, and consider rate limiting to protect your services.
- **Performance Optimization:**
  Consider caching strategies, load balancing, and minimizing payload sizes to enhance responsiveness.

---

8. Conclusion

Choosing the right API paradigm depends on your specific use case. REST is great for simplicity and broad compatibility; GraphQL shines when flexible, efficient data retrieval is required; and gRPC excels in high-performance, low-latency scenarios. Understanding these differences will help you design APIs that are both efficient and scalable.

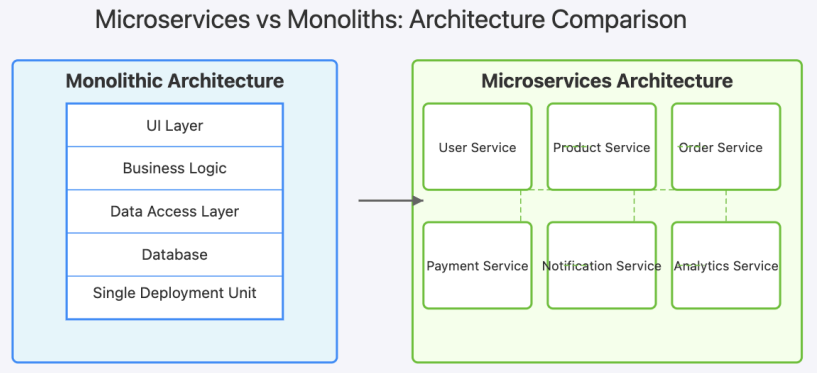
10. Microservices vs. Monoliths: Visual Decision Guide

Introduction

The architectural choice between microservices and monoliths is one of the most fundamental decisions in modern system design. Each approach offers distinct advantages and challenges that significantly impact development workflows, scalability, and maintenance. This guide will explore both architectures in depth, providing visual comparisons and practical decision frameworks.

Microservices vs Monoliths Architecture Comparison

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.



Understanding Monolithic Architecture

A monolithic architecture represents the traditional unified model where an application is built as a single, autonomous unit. All components of the application—user interface, business logic, and data access layer—are interconnected and interdependent within a single codebase.

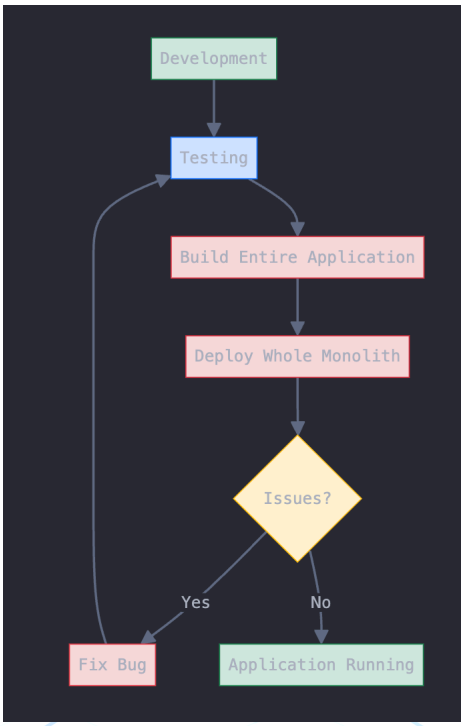
Key Characteristics of Monoliths

1. **Single Codebase**: The entire application exists in one codebase repository.
2. **Unified Deployment**: The whole application is deployed as a single unit.
3. **Shared Database**: All components typically access a single, shared database.
4. **Tight Coupling**: Components are closely interconnected, making isolated changes challenging.
5. **Vertical Scaling**: Primarily scaled by adding more resources to the existing server.

Advantages of Monolithic Architecture

- **Simplicity**: Easier to develop, test, and deploy initially
- **Development Speed**: Faster development cycles for small to medium-sized applications
- **Simplified Debugging**: Easier to trace issues across the application
- **Shared Memory Access**: Components can share memory, reducing latency
- **Consistency**: Enforced consistent coding standards and patterns

Monolithic Development Workflow



Real-World Example: Traditional Banking System

Consider a traditional banking system where the account management, transactions, reporting, and user interface are all part of a single application. Any change to the transaction module requires testing and redeploying the entire application, even if other modules remain untouched.

Understanding Microservices Architecture

Microservices architecture breaks an application into a collection of loosely coupled, independently deployable services. Each service is focused on a specific business function and can be developed, deployed, and scaled independently.

Key Characteristics of Microservices

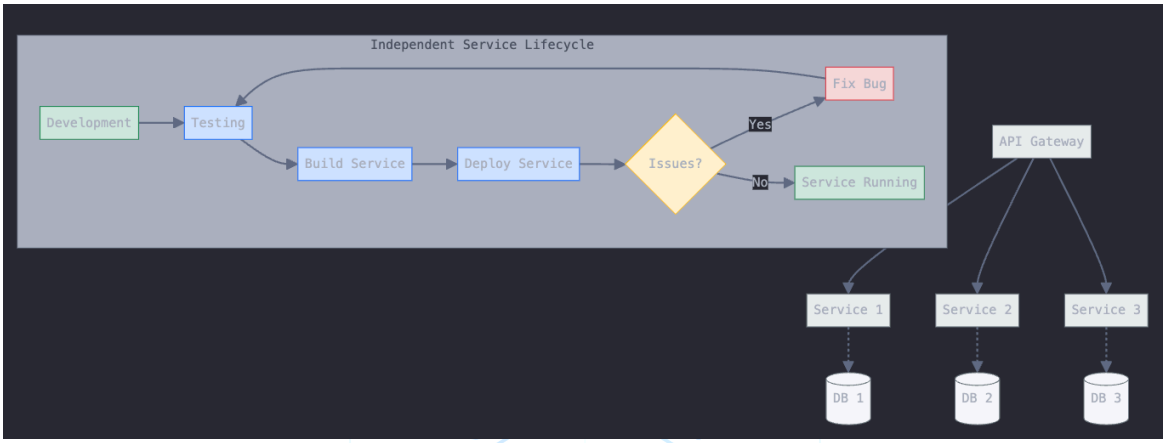
1. **Service Independence**: Each service can be developed, deployed, and scaled independently.
2. **Domain-Focused**: Services are organized around business capabilities.
3. **Decentralized Data Management**: Each service typically manages its own database.
4. **API Communication**: Services interact through well-defined APIs.
5. **Technology Diversity**: Different services can use different technologies and programming languages.
6. **Horizontal Scaling**: Services can be individually scaled based on demand.

Advantages of Microservices Architecture

- **Independent Deployment**: Services can be updated without affecting the entire system
- **Technology Flexibility**: Teams can choose the best technology for each service

- **Resilience**: Failure in one service doesn't necessarily bring down the entire system
- **Scalability**: Individual services can be scaled based on specific needs
- **Team Autonomy**: Different teams can work independently on different services

Microservices Development Workflow



Real-World Example: Netflix

Netflix transitioned from a monolithic to a microservices architecture to support its global scale. They have hundreds of microservices handling different functions:

- User profile service
- Recommendation engine
- Content delivery service
- Billing service
- Viewing history service

Each service has its own database and can be updated independently without affecting other components of the platform.

Comparative Analysis: Strengths and Weaknesses

Let's compare these architectures across several key dimensions:

Comparative Analysis Table

Microservices vs Monoliths: Comparative Analysis

Dimension	Monolithic Architecture	Microservices Architecture
Development Complexity	Lower initially Higher as application grows	Higher initially Lower for large applications
Scalability	Vertical scaling Entire application must scale	Horizontal scaling Individual service scaling
Deployment	Single deployment unit Simpler but more disruptive	Multiple deployment units Complex but less disruptive
Technology Stack	Single technology stack	Multiple technology stacks
Resilience	Single point of failure	Isolated failures Graceful degradation
Team Structure	Single team or functional teams	Small, cross-functional teams
Data Management	Shared database	Decentralized databases Data consistency challenges
Operational Complexity	Lower	Higher

Decision Framework: When to Choose Each Architecture

The choice between microservices and monoliths isn't binary but contextual. Here's a decision framework to guide your architectural choice:

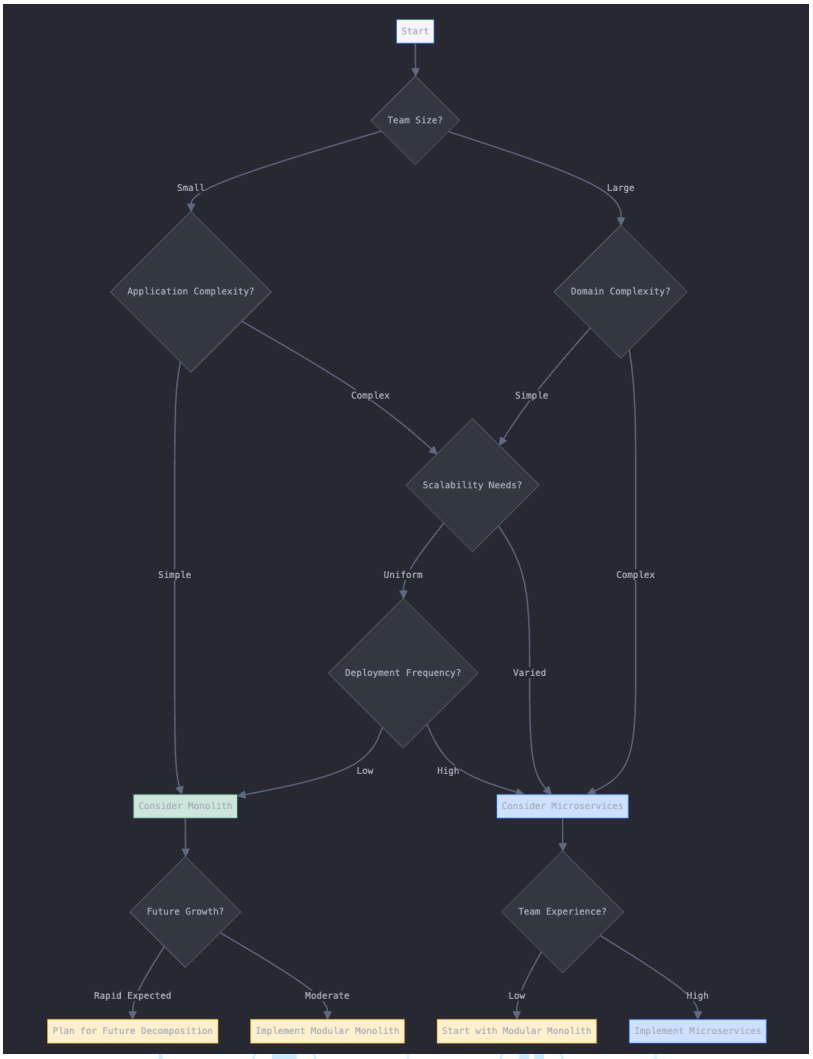
Choose Monolithic Architecture When:

1. **Building a Startup or MVP**: When speed to market is critical and requirements are evolving
2. **Small Team Size**: Limited resources to manage distributed system complexity
3. **Simple Domain**: Application has a straightforward domain with limited complexity
4. **Limited Scalability Needs**: Application doesn't need to scale components independently
5. **Strong Consistency Requirements**: Application requires strong transactional consistency

Choose Microservices Architecture When:

1. **Large, Complex Application**: Application has distinct business domains
2. **Team Size and Structure**: Multiple teams working independently
3. **Scalability Requirements**: Different components need to scale independently
4. **Deployment Frequency**: Need for frequent, independent deployments
5. **Technology Diversity**: Different components benefit from different technologies
6. **Organizational Alignment**: Business units align with service boundaries

Architecture Decision Flowchart



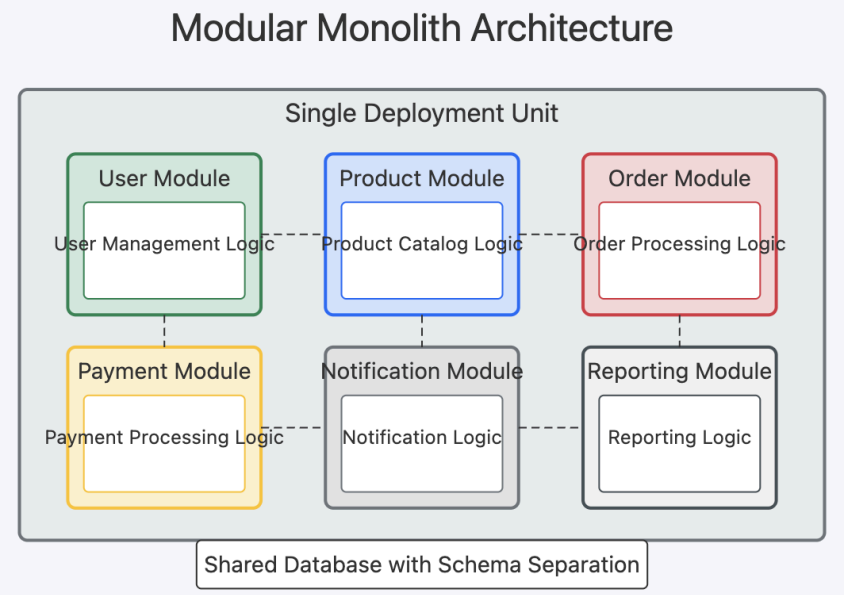
Hybrid Approach: The Modular Monolith

A modular monolith represents a middle ground between monoliths and microservices. It maintains a single deployment unit while enforcing clear boundaries between modules, making future decomposition into microservices easier if needed.

Characteristics of Modular Monoliths:

1. **Single Deployment Unit**: Deployed as a single application
2. **Clear Module Boundaries**: Explicit interfaces between modules
3. **Independent Modules**: Modules with well-defined responsibilities
4. **Shared Database with Schema Separation**: Often uses a shared database but with clear schema separation

Modular Monolith Architecture



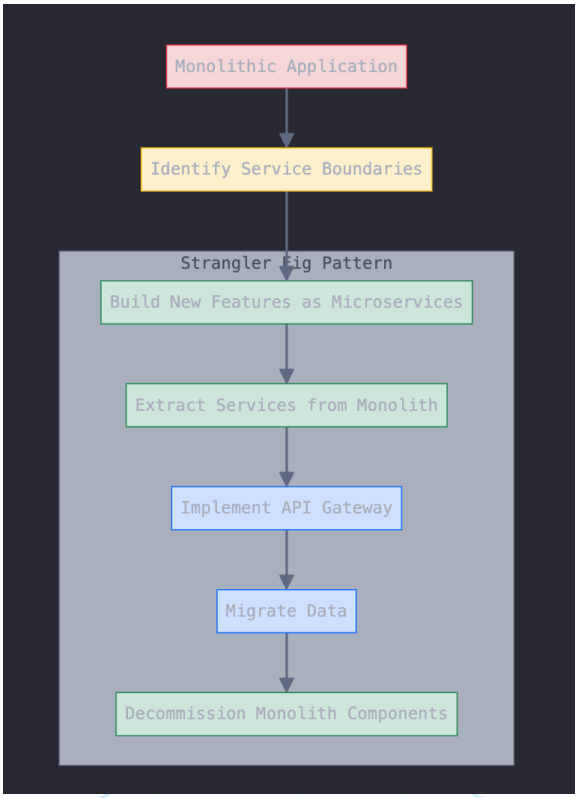
Real-World Example: Shopify

Shopify began as a modular monolith, which allowed them to move quickly while maintaining clear boundaries between different parts of their e-commerce platform. As they grew, they selectively extracted certain modules into microservices when scaling needs required it, while keeping other parts as a modular monolith.

Migration Strategies: Monolith to Microservices

For organizations considering a transition from monolithic to microservices architecture, a strategic approach is essential. Here are proven migration patterns:

Monolith to Microservices Migration Strategy



The Strangler Fig Pattern

Named after a type of vine that gradually overtakes and replaces its host tree, this pattern involves:

1. **Build a facade**: Create an API layer in front of the monolith
2. **Extract services**: Gradually move functionality from the monolith to new microservices
3. **Redirect traffic**: Route requests through the facade to either the monolith or new services
4. **Decommission**: Eventually remove unused monolith components

Domain-Driven Design Approach

Using Domain-Driven Design (DDD) to identify service boundaries:

1. **Identify bounded contexts**: Define clear boundaries in your domain model
2. **Align with business capabilities**: Map services to business functions
3. **Define aggregates**: Identify key entities that should stay together
4. **Establish integration patterns**: Define how services will communicate

Common Challenges and Solutions

Challenges with Monoliths

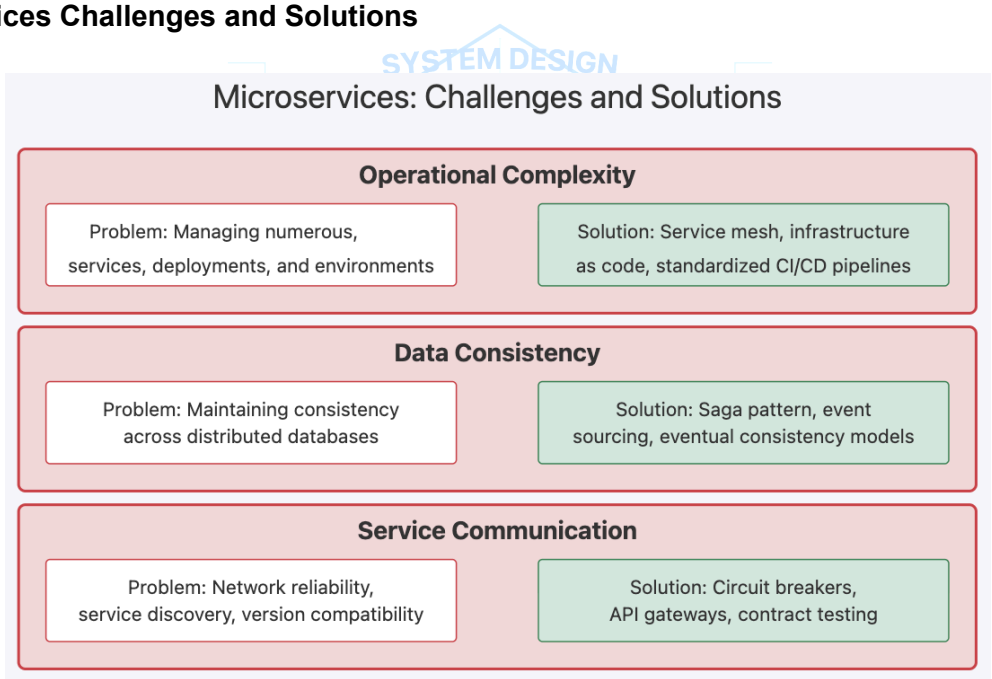
1. **Scaling Issues**: As the application grows, scaling becomes increasingly difficult
    a. **Solution**: Implement caching strategies, optimize database queries, consider vertical scaling
2. **Technology Lock-in**: Difficult to adopt new technologies

    a. **Solution**: Implement modular design with clear interfaces, use adapters to integrate new technologies
3. **Development Bottlenecks**: Teams can block each other when working on the same codebase
    a. **Solution**: Establish clear module boundaries, implement feature toggles, improve CI/CD pipelines

Challenges with Microservices

1. **Operational Complexity**: Managing numerous services can be overwhelming
    a. **Solution**: Implement comprehensive monitoring, standardize deployment processes, use service mesh
2. **Data Consistency**: Maintaining data consistency across services
    a. **Solution**: Implement saga patterns, eventual consistency, domain events
3. **Service Communication**: Managing inter-service dependencies
    a. **Solution**: Implement circuit breakers, retries, bulkheads, and API gateways

Microservices Challenges and Solutions



Industry Case Studies

Amazon's Transition to Microservices

Amazon's journey from a monolithic application to microservices architecture offers valuable insights:

1. **Initial State**: Single monolithic application for all e-commerce operations
2. **Transition Trigger**: Scalability challenges and team growth
3. **Migration Approach**: Gradual decomposition of the monolith into services
4. **Service Philosophy**: "Two-pizza teams" (teams small enough to be fed by two pizzas)
5. **Results**: Significantly increased deployment frequency, improved innovation speed

Spotify's Squad Model

Spotify's organizational and architectural approach aligns teams (squads) with services:

1. **Squad Autonomy**: Teams own specific services end-to-end
2. **Technology Choice**: Freedom for teams to select appropriate technologies
3. **Organizational Structure**: Squads, tribes, chapters, and guilds to balance autonomy and alignment
4. **Interface Contracts**: Clear service boundaries and API contracts

Conclusion: Making the Right Choice

The choice between microservices and monoliths should be guided by:

1. **Current Context**: Team size, domain complexity, and scalability needs
2. **Future Direction**: Anticipated growth and changing requirements
3. **Organizational Structure**: Team organization and communication patterns
4. **Technical Capabilities**: Team experience with distributed systems

Remember that neither architecture is inherently superior. The best choice depends on your specific context, constraints, and goals. Many successful systems start as monoliths and evolve toward a more distributed architecture as needs change.

For organizations just starting out or with smaller teams, consider a modular monolith approach that enables future decomposition if needed. For larger organizations with complex domains and multiple teams, a microservices architecture may provide the flexibility and scalability required.

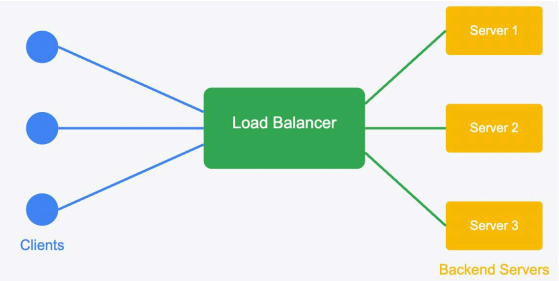
Whatever architecture you choose, focus on establishing clear boundaries, maintaining good documentation, and building robust testing practices—these fundamentals will serve you well regardless of architectural style.

11. Load Balancing 101: How Traffic Gets Distributed

Load balancing is a critical component in modern distributed systems that ensures high availability and reliability by distributing network traffic across multiple servers. Let's explore how it works and why it matters.

Load Balancing Overview

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.



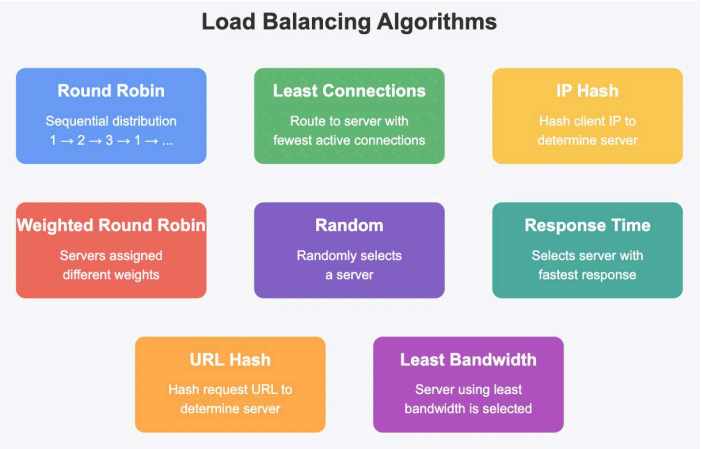
What is Load Balancing?

- Load balancing is the process of distributing network traffic across multiple servers to ensure no single server bears too much demand. By spreading the workload, load balancers help
- Prevent server overloads
- Optimize resource usage
- Reduce response time
- Ensure high availability
- Provide fault tolerance

How Load Balancers Work

Load balancers sit between client devices and backend servers, routing client requests across all available servers capable of fulfilling those requests.

Load Balancing Algorithms



Key Load Balancing Algorithms

1. Round Robin

The simplest method that distributes requests sequentially to each server in the pool. Server 1 gets the first request, Server 2 gets the second, and so on.

Example: A web application with three servers. Request 1 → Server 1, Request 2 → Server 2, Request 3 → Server 3, Request 4 → Server 1, etc.

2. Least Connections

Routes traffic to the server with the fewest active connections, which is particularly useful when requests might require varying processing times.

Example: Server 1 has 15 active connections, Server 2 has 10, and Server 3 has 5. The next request goes to Server 3.

3. IP Hash

Uses the client's IP address to determine which server receives the request. This ensures a client is consistently directed to the same server, which helps maintain session consistency.

Example: User A's IP hash maps to Server 2, so all requests from User A always go to Server 2.

4. Weighted Round Robin

Similar to Round Robin but assigns different weights to servers based on their capacity.

Example: Server 1 (weight: 5), Server 2 (weight: 3), Server 3 (weight: 2). Out of 10 requests, 5 go to Server 1, 3 to Server 2, and 2 to Server 3.

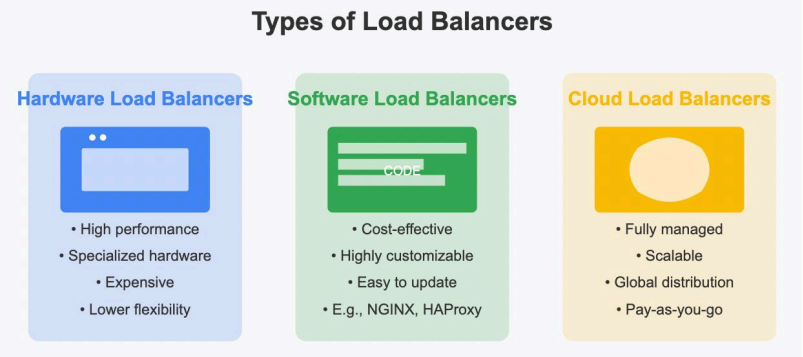
5. Least Response Time

Directs traffic to the server with the quickest response time, which indicates both availability and speed.

Example: Server 1 responds in 15ms, Server 2 in 8ms, and Server 3 in 12ms. The next request goes to Server 2.

Types of Load Balancers

Types of Load Balancers



1. Hardware Load Balancers

Physical devices optimized for load balancing with dedicated processors and ASICs.

Example: F5 BIG-IP, Citrix ADC, Barracuda Load Balancer

- **Pros:**
  - High performance
  - Reliability
  - Security features
- **Cons:**
  - Expensive
  - Limited scalability
  - Hardware maintenance

2. Software Load Balancers

Software applications installed on standard servers that provide load balancing functionality.

Example: NGINX, HAProxy, Apache Traffic Server

- **Pros:**
  - Cost-effective
  - Flexible configuration
  - Easy to update
- **Cons:**
  - May have lower throughput than hardware options
  - Requires separate server maintenance

3. Cloud Load Balancers

Managed services provided by cloud providers.

Example: AWS Elastic Load Balancing, Google Cloud Load Balancing, Azure Load Balancer

Pros:

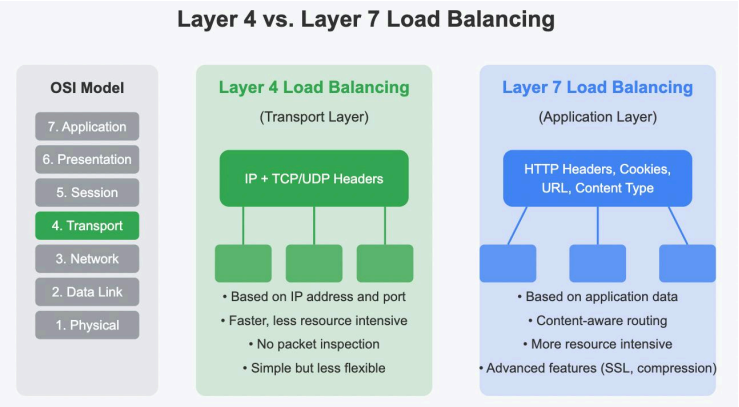
- No maintenance required
- Highly scalable
- Global distribution
- Pay-as-you-go pricing

Cons:

- Less customization
- Vendor lock-in concerns

Layer 4 vs. Layer 7 Load Balancing

Layer 4 vs. Layer 7 Load Balancing



Load balancers can operate at different OSI model layers:

Layer 4 Load Balancing (Transport Layer)

- Routes based on IP address and port
- Doesn't inspect packet content
- Faster and requires fewer resources
- Can't make routing decisions based on content

Example: Distributing traffic based solely on TCP/UDP port numbers. All HTTP traffic (port 80) goes to web servers, while all database traffic (port 3306) goes to database servers.

Layer 7 Load Balancing (Application Layer)

- Routes based on content (URL, HTTP headers, cookies)
- More intelligent routing decisions
- More resource-intensive
- Can implement advanced features like SSL termination

Example: Routing requests to different servers based on the URL path. Requests to /images/* go to image servers, /api/* to API servers, and other URLs to web servers.

Real-World Load Balancing Scenarios

E-commerce Website

- **Layer 7 load balancing** directs catalog browsing to web servers
- Payment processing gets routed to dedicated secure servers
- Product images served from optimized content delivery servers
- During flash sales, **least connections** algorithm ensures even distribution

Video Streaming Service

- **Geolocation-based load balancing** sends users to the nearest data center
- **Content-aware routing** directs video streams to specialized streaming servers
- Authentication requests go to dedicated authentication servers
- **Health checks** constantly monitor server status to prevent routing to failed nodes

Load Balancer Health Checks

Load balancers regularly check the health of backend servers using:

- TCP connection checks: Verify if a TCP connection can be established
- HTTP/HTTPS requests: Send HTTP requests to a specific endpoint
- Custom application checks: Verify application functionality beyond basic connectivity

Session Persistence

Sometimes it's necessary to send a user's requests to the same server:

- Source IP affinity: Routes based on client's IP address
- Cookie-based persistence: Uses HTTP cookies to track which server handled previous requests
- Application-controlled persistence: Application itself manages session state

Benefits of Load Balancing

- High Availability: If one server fails, traffic is automatically distributed to healthy servers
- Scalability: Easily add or remove servers based on demand
- Efficiency: Optimize resource utilization across all servers
- Security: Can act as a buffer between clients and servers, filtering malicious traffic
- Flexibility: Perform maintenance without service disruption

Implementing Load Balancing: A Simple NGINX Example

Here's how to implement a basic round-robin load balancer using NGINX:

```
http {
    upstream backend_servers {
        server backend1.example.com;
        server backend2.example.com;
        server backend3.example.com;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://backend_servers;
        }
    }
}
```

Conclusion

Load balancing is a fundamental component of modern distributed systems architecture. By effectively distributing traffic across multiple servers, organizations can achieve higher reliability, better performance, and greater scalability. Whether implemented through hardware, software, or cloud services, load balancers help ensure that applications remain available and responsive even under heavy load.

12. Database Basics: SQL vs. NoSQL Decision Tree

Introduction

Selecting the right database for your application is a critical architectural decision that can significantly impact performance, scalability, and development efficiency. This guide presents a systematic approach to choosing between SQL and NoSQL databases based on your specific requirements.

Understanding SQL and NoSQL Databases

SQL Databases

SQL (Structured Query Language) databases organize data in tables with predefined schemas. They follow the ACID (Atomicity, Consistency, Isolation, Durability) properties and establish relationships between tables through foreign keys.

Key characteristics:

- Structured data with rigid schemas
- Strong consistency and transaction support
- Vertical scaling primarily (scale-up)
- Mature ecosystem with standardized query language
- Powerful for complex queries and relationships

Popular examples: PostgreSQL, MySQL, Oracle, SQL Server, MariaDB

NoSQL Databases

NoSQL ("Not Only SQL") databases emerged to address limitations of relational databases, particularly for handling large volumes of unstructured or semi-structured data, and for horizontal scaling.

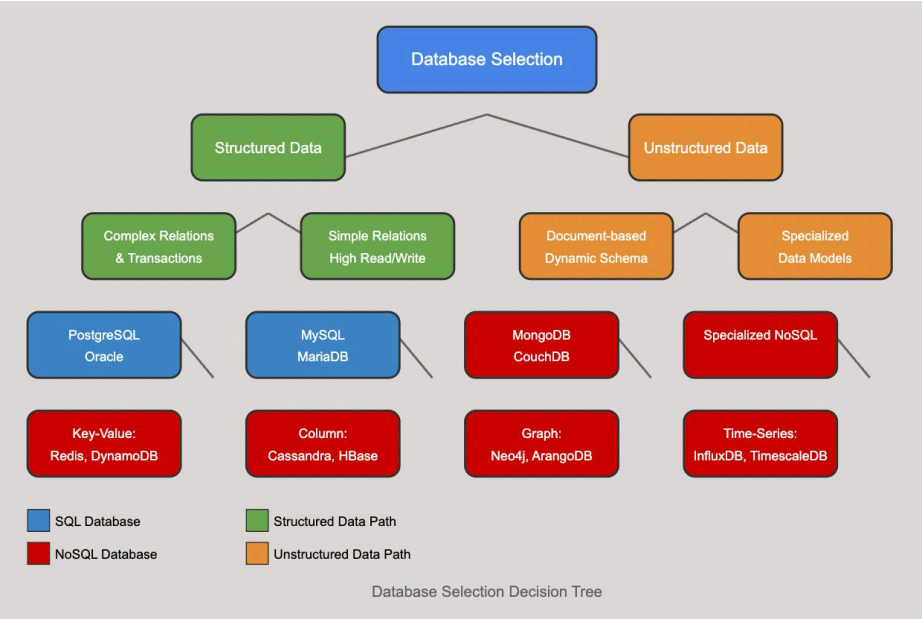
Key characteristics:

- Flexible schemas
- Designed for horizontal scaling (scale-out)
- Often prioritize availability and partition tolerance over consistency
- Specialized for specific data models
- Typically offer simpler query capabilities

Main types of NoSQL databases:

- Document stores (MongoDB, CouchDB)
- Key-value stores (Redis, DynamoDB)
- Column-family stores (Cassandra, HBase)
- Graph databases (Neo4j, ArangoDB)

SQL vs NoSQL Decision Tree



Decision Tree for Database Selection

Let's explore the key decision factors to help determine which database type is most suitable for your needs:

1. Data Structure Requirements

If your data is highly structured:

- Well-defined schema that rarely changes
- Clear relationships between different data entities
- Complex transactions across multiple data entities
- Need for strong data integrity constraints
- → Consider SQL databases

If your data is semi-structured or unstructured:

- Evolving schema requirements (schema-less)
- Document-oriented data (JSON, XML, etc.)
- Varying attributes across similar items
- → Consider NoSQL databases

2. Scalability Needs

If you need vertical scalability:

- Predictable, moderate growth
- More powerful single-server solution is adequate
- → Consider SQL databases

If you need horizontal scalability:

- Massive data volumes
- Unpredictable or explosive growth patterns
- Distributed data architecture requirements
- → Consider NoSQL databases (particularly key-value or column-family)

SQL vs NoSQL Comparison

3. Query Complexity

If you need complex queries and transactions:

- Multiple table joins
- Complex aggregations and reporting
- ACID transaction requirements
- → Consider SQL databases

If your queries are simpler:

- Key-based lookups
- High read/write throughput
- Document-centric operations
- → Consider NoSQL databases

4. Consistency Requirements

If you need strong consistency:

- Financial applications
- Inventory management
- Systems where data accuracy is critical
- → Consider SQL databases

If eventual consistency is acceptable:

- Social media applications
- Content management systems
- Analytics applications
- → Consider NoSQL databases

5. Development Speed and Flexibility

If you need structured, predictable development:

- Well-defined data models before development
- Strong data validation requirements
- → Consider SQL databases

If you need agile development:

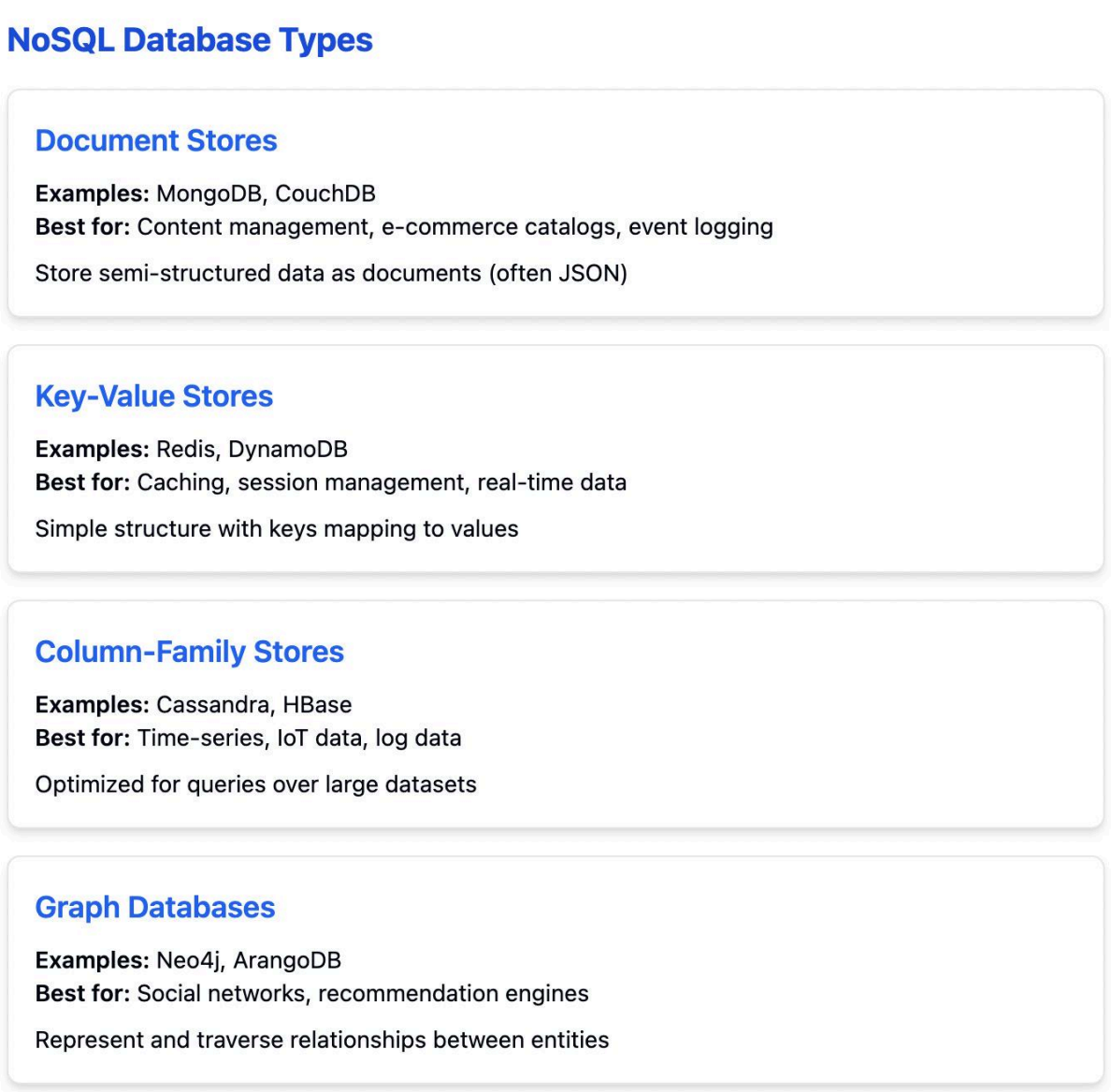
- Rapidly evolving data requirements

- Frequent iterations
- Microservices architecture
- → Consider NoSQL databases

NoSQL Database Type Selection

Once you've determined NoSQL is the right choice, you need to select the appropriate type:

NoSQL Database Types



Document Stores

Best for applications with complex but flexible data structures that don't require many relationships. Examples include content management systems and e-commerce product catalogs.

Key-Value Stores

Ideal for simple data models requiring high performance and scalability. Perfect for caching, session management, and real-time applications.

Column-Family Stores

Designed for handling massive amounts of data distributed across many servers. Great for time-series data, IoT, and systems that need to write data quickly.

Graph Databases

Specialized for data with complex relationships and interconnections. Ideal for social networks, recommendation engines, and fraud detection systems.

Real-World Selection Examples

Example 1: E-commerce Platform

Requirements:

- Product catalog with varying attributes
- Customer data and order history
- Inventory management
- Order processing and payment transactions

Decision Path:

- Product catalog: Varying attributes suggest NoSQL (document store)
- Customer/order data: Relational structure suggests SQL
- Inventory & transactions: ACID requirements suggest SQL

Potential Solution: Polyglot persistence - Use PostgreSQL for transactional data (orders, inventory) and MongoDB for product catalog

Example 2: IoT Analytics Platform

Requirements:

- Billions of time-stamped sensor readings
- Real-time data ingestion
- Historical trend analysis
- Device metadata storage

Decision Path:

- Sensor data: Time-series, high volume suggests NoSQL (column-family)
- Device metadata: Structured data suggests SQL or document store
- Real-time needs: Suggest specialized time-series database

Potential Solution: TimescaleDB or InfluxDB for time-series data, PostgreSQL for metadata

Example 3: Social Media Application

Requirements:

- User profiles and authentication
- Friend/follower relationships
- Content posting and sharing
- Feed generation and recommendations

Decision Path:

- User profiles: Structured data suggests SQL
- Relationships: Complex network suggests graph database
- Content: Semi-structured suggests document store
- Recommendations: Graph traversal suggests graph database

Potential Solution: Neo4j for social graph, MongoDB for content, PostgreSQL for user data

Performance Considerations

When comparing SQL and NoSQL databases, performance characteristics vary significantly:

Operation	SQL	Document Store	Key-Value	Column-Family	Graph
Single-record read	Good	Excellent	Excellent	Good	Good
Single-record write	Good	Excellent	Excellent	Excellent	Good
Multi-record transactions	Excellent	Limited	Poor	Poor	Variable
Relationship traversal	Good (with joins)	Poor	Poor	Poor	Excellent
Range queries	Excellent	Good	Limited	Good	Limited
Aggregations	Excellent	Good	Poor	Limited	Limited
Schema flexibility	Poor	Excellent	Excellent	Good	Good

Conclusion

The choice between SQL and NoSQL is not binary but depends on your specific application requirements. Many modern applications utilize polyglot persistence, combining different database types to leverage their respective strengths. Consider your data structure, scalability needs, consistency requirements, and query patterns when making this important architectural decision.

Remember that database migration is often costly, so invest time in proper analysis before implementation. Start with a clear understanding of your data model and access patterns, then select the database technology that best aligns with those requirements.

13. Caching Strategies Explained

Introduction

Imagine you're doing your math homework. When you need to solve 7 × 8, do you calculate it every time or remember the answer is 56? Remembering the answer is faster than recalculating it—this is exactly what caching does in computers!

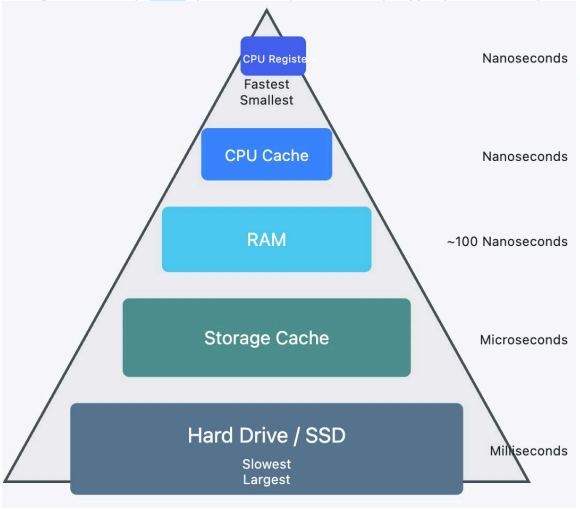
Caching is like having a special notebook where you write down answers you've already figured out, so you don't have to solve the same problem again and again.

Thanks for reading System Design Interview Roadmap! Subscribe for free to receive new posts and support my work.

What is Caching?

Caching is keeping frequently used information in a place that's quick and easy to access. It's like keeping your favorite snacks in your desk drawer instead of having to walk to the kitchen every time you're hungry.

Memory Hierarchy Visualization



Why Do We Need Caching?

Imagine going to the library every time you need to look up a simple fact like "What's the capital of France?" That would take forever! Instead, you remember that Paris is the capital of France.

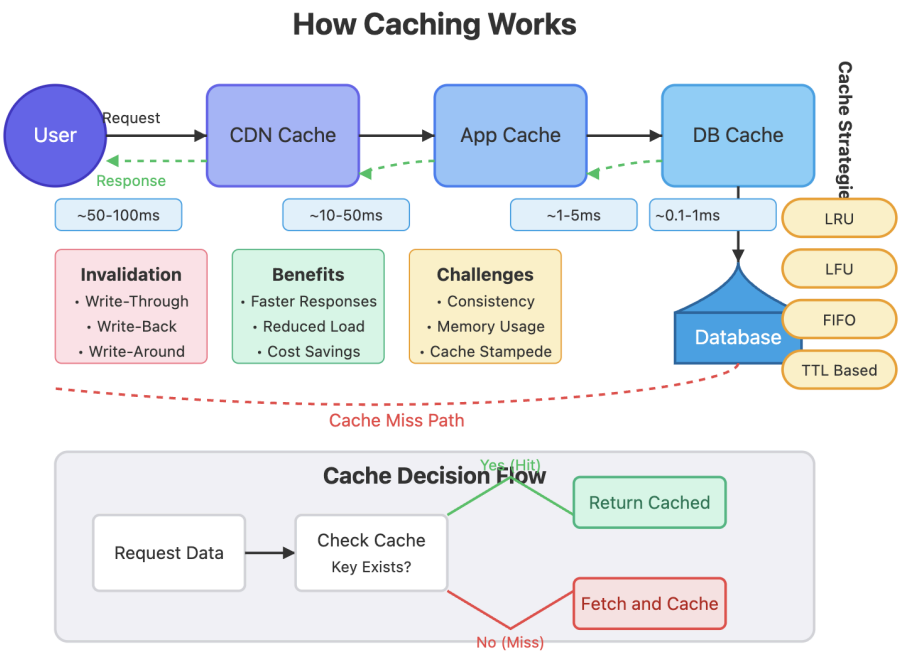
In computing:

- Getting data from the main storage (like a hard drive) is slow
- Getting data from memory (RAM) is faster
- Getting data from cache (special, super-fast memory) is even faster!

Computers use caching because there's a huge speed difference between different types of storage. The closer the data is to the processor, the faster it can be accessed.

Caching Strategies

Let's explore different ways we can manage our special "quick answer notebook."



1. LRU (Least Recently Used)

Imagine your notebook can only hold 5 math facts. When you need to add a new one but your notebook is full, which one do you remove? LRU says to remove the one you haven't used in the longest time.

Example: You have math facts for 2×2, 3×3, 4×4, 5×5, and 6×6 in your cache. If you haven't used 3×3 for ages and now need to add 7×7, you'd forget the 3×3 fact to make room.

2. LFU (Least Frequently Used)

Instead of removing the oldest fact, LFU removes the fact you've used the least number of times.

Example: In your notebook, you've used 2×2 twenty times, 3×3 only twice, and others multiple times. When adding a new fact, you'd remove 3×3 because you've used it the least frequently.

3. FIFO (First In, First Out)

This is like a line at a water fountain. The first person to join the line is the first to leave. In caching, the oldest item gets removed first, regardless of how often it's used.

Example: If you added facts to your notebook in this order: 2×2, 3×3, 4×4, 5×5, 6×6, and now need to add 7×7, you'd remove 2×2 because it was the first one added.

4. Time-Based Expiration

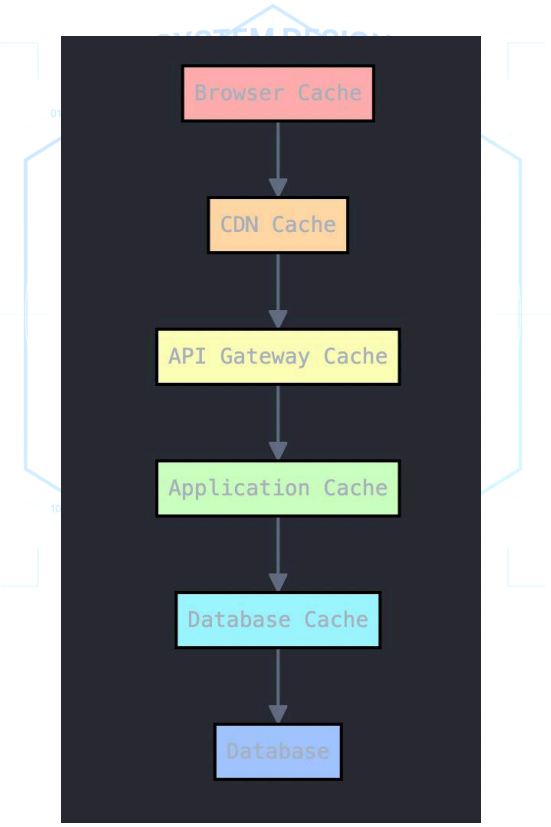
Some facts might become outdated. For instance, the current temperature outside will change throughout the day. With time-based expiration, you set an "expiration date" for each cached item.

Example: You cache the weather forecast, but set it to expire after 3 hours since weather predictions change.

Caching Layers

Caching can happen at different levels:

Caching Layers Diagram



1. Browser Cache

When you visit websites, your browser keeps copies of images and other files. Next time you visit, it loads faster because some files are already saved on your computer.

Example: When you visit YouTube, your browser saves the YouTube logo. Next time you visit, it doesn't need to download it again.

2. CDN Cache

Content Delivery Networks (CDNs) are like having copies of a popular book in libraries all around the world, so people can borrow it from a nearby library instead of ordering it from far away.

Example: Netflix stores copies of popular movies in servers around the world, so when you watch "Frozen," it comes from a server near you instead of from across the country

3. Application Cache

This is like a teacher keeping frequently used teaching materials on their desk instead of going to the supply closet every time.

Example: A social media app might cache your friend list so it doesn't have to look it up every time you open the app.

4. Database Cache

Databases store tons of information, but looking through all of it takes time. A database cache keeps copies of frequently requested data for quick access.

Example: An online store might cache its top 20 bestselling products so it doesn't have to search through thousands of products every time someone visits the homepage

Cache Invalidation: The Hard Problem

The trickiest part of caching is knowing when your cached information is outdated and needs to be refreshed.

Example: Imagine your friend changes their phone number. If you keep using the old number from your "cache" (memory), you'll never reach them!

There are three main approaches:

- **Write-through cache**: Update both the cache and the main storage at the same time.
    - Like updating both your personal address book and the school directory immediately when a friend moves.
- **Write-back cache**: Update the cache first, then update the main storage later.
    - Like jotting down a friend's new phone number on a sticky note (cache) and updating your phone contacts (main storage) when you have time.
- **Write-around cache**: Update only the main storage, bypassing the cache.
    - Like updating only the school directory when a friend moves, but not your personal address book. The next time you need their address, you'll check the directory.

Real-World Examples

Example 1: School Supplies

Imagine organizing your school supplies:

- Pencil case (cache): Things you need immediately
- Desk (RAM): Things you need frequently
- Backpack (storage): Things you might need during the day
- Home (distant storage): Other supplies you rarely need

You keep pencils in your pencil case because you use them constantly. If you had to reach into your backpack for a pencil every time you needed to write something, homework would take forever!

Example 2: Web Browsing

When you visit a website like a news site:

- Your browser checks if it has a cached version of the site
- If yes (cache hit), it shows you the saved version (super fast!)
- If no (cache miss), it downloads the site from the internet (slower)
- It then saves a copy for next time

Conclusion

Caching is all about balancing speed and freshness. It's like the difference between:

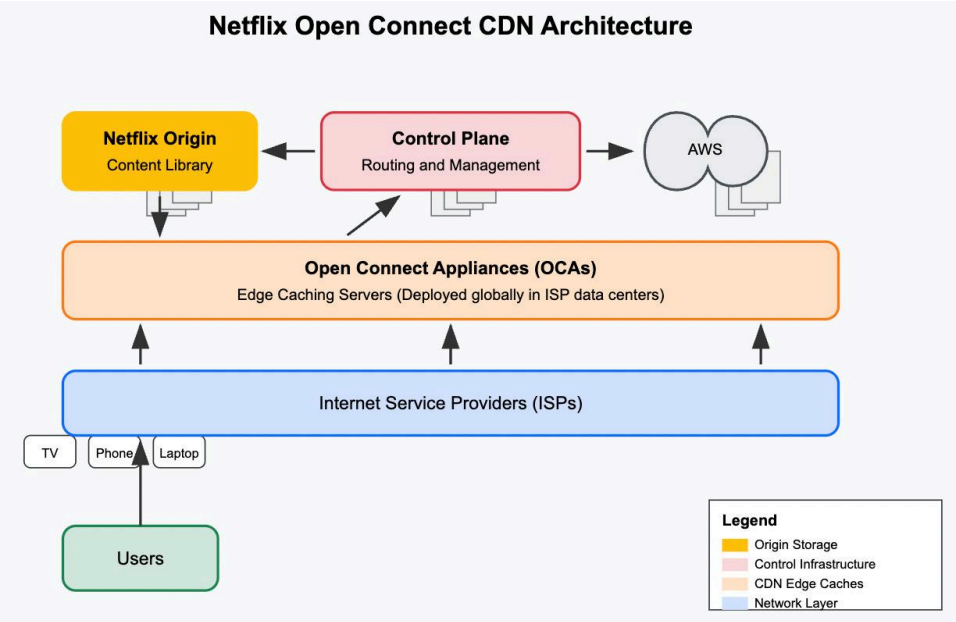
- Recalculating 7 × 8 each time (slow but always correct)
- Remembering that 7 × 8 = 56 (fast and still correct since math facts don't change)
- Remembering yesterday's weather (fast but might be outdated)

Understanding caching helps us build faster applications and systems. The next time a website or app loads quickly, remember it might be using some of these caching strategies to give you a better experience!

14. Content Delivery Networks: How Netflix Delivers Movies

Content Delivery Networks (CDNs) form the backbone of modern internet content distribution, enabling companies like Netflix to deliver high-quality video to millions of concurrent users worldwide. Let's dive into how CDNs work, with Netflix as our primary case study.

Netflix CDN Architecture Diagram



The CDN Challenge: Delivering Petabytes of Video Content

Netflix streams over 220 million hours of video per day across 190+ countries. At peak times, Netflix traffic can account for up to 15% of global internet traffic. How do they manage this tremendous load?

The answer lies in their custom-built CDN called **Open Connect**.

What is a CDN?

A Content Delivery Network is a distributed network of servers that delivers content to users based on their geographic location. Instead of serving content from a central location, CDNs cache content on edge servers positioned closer to end users, minimizing latency and reducing bandwidth costs.

Why Traditional CDNs Weren't Enough for Netflix

Before 2011, Netflix relied on third-party CDNs like Akamai, Limelight, and Level 3. However, as their subscriber base grew exponentially, Netflix faced several limitations:

1. **Cost efficiency**: Third-party CDNs charged per gigabyte delivered, making the model financially unsustainable as Netflix's traffic grew
2. **Quality control**: Limited control over content delivery quality
3. **Scale**: Unprecedented video streaming requirements demanded specialized infrastructure
4. **Consistency**: Varying performance across different regions and ISPs

This led to Netflix developing their proprietary CDN solution: Open Connect.

Netflix Open Connect: Architecture Deep Dive

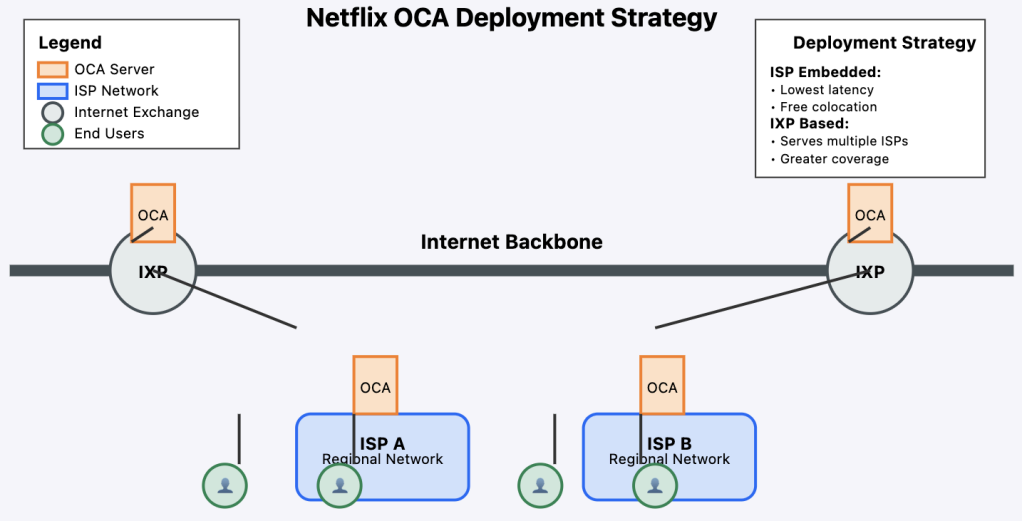
Open Connect is Netflix's purpose-built CDN, optimized specifically for video delivery. Let's examine its key components:

1. Open Connect Appliances (OCAs)

The backbone of Open Connect consists of custom-designed server appliances deployed worldwide:

- **Hardware specification**: Custom-designed servers with high storage density (currently up to 500TB per server)
- **Deployment model**: Two strategic placement strategies:
- **ISP-embedded OCAs**: Placed directly within Internet Service Provider networks
- **Internet Exchange Points (IXPs)**: Positioned at major internet exchange locations

Netflix OCA Deployment Strategy



2. Content Storage & Distribution

Netflix's content storage strategy is highly optimized:

- **Tiered storage**: Content is classified based on popularity
- **Content popularity prediction**: Machine learning algorithms predict which titles will be popular in specific regions

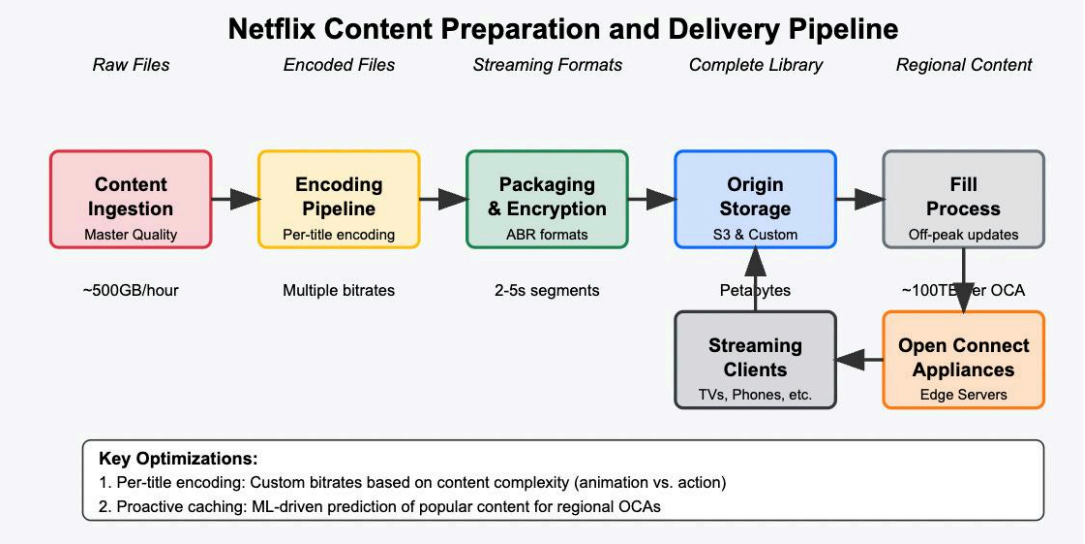
- **Proactive caching**: Popular content is pushed to OCAs during off-peak hours (fill process)
- **Storage efficiency**: Each OCA doesn't need to store the entire Netflix library, just regionally relevant content

3. Content Preparation Pipeline

Before reaching the CDN, content undergoes an extensive preparation process:

- **Content ingestion**: Original master copies are uploaded to Netflix's cloud infrastructure
- **Encoding**: Using a technique called "per-title encoding," Netflix creates multiple quality variants of each title
- **Chunking**: Content is split into small segments (typically 2-5 seconds) for efficient streaming
- **Adaptive Bitrate (ABR) packaging**: Different quality renditions are prepared for adaptive streaming

Netflix Content Flow Diagram



4. Intelligent Client

Netflix's client applications play a crucial role in content delivery:

- **Adaptive Bitrate Streaming**: Clients continuously measure network conditions and adjust video quality accordingly
- **Buffer management**: Dynamic adjustment of buffer size based on network stability
- **Device-specific optimization**: Customized playback settings for different devices (TVs vs. mobile)
- **Content pre-fetching**: Predictive loading of the next episodes or recommended content

This client-side intelligence works in tandem with the CDN to ensure smooth playback across varying network conditions.

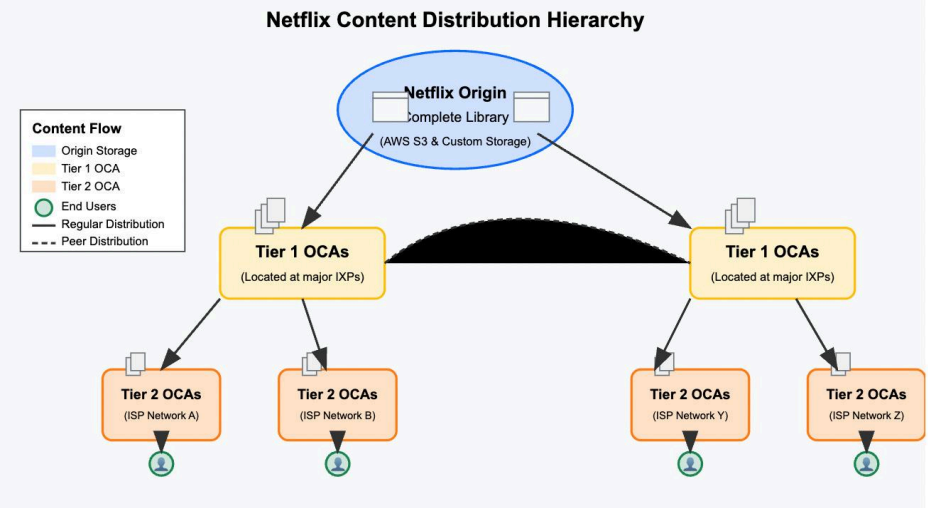
Open Connect: Technical Implementation Details

Let's explore how Netflix actually implements this massive content delivery system.

1. Content Distribution Strategy

Netflix employs a hierarchical distribution approach:

Netflix Content Distribution Hierarchy



- **Tier 1 OCAs**: Located at major internet exchange points, these servers receive content directly from the origin
- **Tier 2 OCAs**: Located within ISP networks, these receive content from Tier 1 OCAs
- **Fill process**: Content is distributed during off-peak hours (typically 2 AM - 2 PM local time)
- **Peer-to-peer distribution**: OCAs within the same tier can exchange content with each other

This hierarchical approach minimizes the need to pull content from the origin, which would be expensive and generate high latency.

2. Routing and Load Balancing

Netflix employs a sophisticated request routing system:

- **Dynamic DNS routing**: Custom DNS resolution directs users to the optimal OCA
- **Health checking**: Continuous monitoring of OCA health and availability
- **Load balancing factors**:
  - Server capacity and current load
  - Network conditions
  - Content availability
  - Geographic proximity to user

3. Adaptive Streaming Protocol

Netflix utilizes HTTP-based adaptive streaming:
Client Request:
GET /titles/12345/video/manifest.m3u8 HTTP/1.1
Host: cdn.netflix.com
Range: bytes=0-1023

Server Response:
HTTP/1.1 200 OK
Content-Type: application/vnd.apple.mpegurl
Content-Length: 1024

#EXTM3U
#EXT-X-VERSION:4
#EXT-X-STREAM-INF:BANDWIDTH=2500000,RESOLUTION=1280x720
chunklist_2500.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=5000000,RESOLUTION=1920x1080
chunklist_5000.m3u8
...

This approach allows for:

- **Mid-stream quality switching**: Adapting to changing network conditions
- **Stateless connections**: Using standard HTTP for firewall-friendly delivery
- **Cache-friendly operation**: Leveraging standard web caching mechanisms

Key Technical Innovations

Netflix's CDN includes several groundbreaking innovations:

1. FreeBSD-Based Custom OS

Netflix developed a custom FreeBSD distribution for their OCAs, optimized for:

- Video content delivery
- High throughput
- Minimal resource usage
- Advanced traffic management

2. Open Connect Telemetry and Monitoring

Netflix's telemetry system collects over 70 billion events daily:

- **Real-time health monitoring**: Tracking server status, network connectivity, and hardware health
- **Performance metrics**: Measuring throughput, latency, cache efficiency
- **Client experience metrics**: Buffering events, startup times, quality shifts
- **Machine learning for anomaly detection**: Identifying issues before they impact users

3. TLS Optimization

Netflix developed custom TLS termination approaches:

- **Session resumption**: Reducing handshake overhead
- **Custom cipher suite selection**: Balancing security with performance
- **Optimized certificate handling**: Reducing verification overhead

4. Predictive Caching Algorithm

Netflix leverages viewing data and machine learning to predict content popularity:

- **Regional popularity modeling**: Based on historical viewing patterns
- **Content launch predictions**: Estimating demand for new releases
- **Seasonal trend analysis**: Adjusting for time-based viewing patterns (holidays, weekends)
- **Metadata-based correlations**: Finding relationships between content types

Business Benefits of Netflix's Custom CDN

The Open Connect CDN provides Netflix with several advantages:

- **Cost efficiency**: Significant reduction in third-party CDN costs
- **Quality control**: Direct control over the entire content delivery pipeline
- **ISP relationships**: Improved partnerships with ISPs (reduced transit costs)
- **Scalability**: Ability to expand efficiently into new markets
- **Competitive advantage**: Technological differentiation from competitors

Challenges and Solutions

Netflix's CDN faces several challenges:

1. **Regional content restrictions**: Geo-fencing enforcement through IP-based controls
2. **DRM management**: Integration with multiple DRM systems (Widevine, PlayReady, FairPlay)
3. **IPv6 transition**: Dual-stack support for both IPv4 and IPv6
4. **Flash crowds**: Managing sudden spikes for popular content releases
5. **Traffic shaping**: Working with ISPs to avoid congestion during peak usage

Lessons for Other Systems

Netflix's CDN teaches us several valuable lessons about large-scale content delivery:

1. **Purpose-built solutions**: While general-purpose CDNs work for many applications, specialized workloads may justify custom solutions

2. **Proactive vs. reactive caching**: Pushing content to the edge before demand arises
3. **Multiple tiers of caching**: Creating hierarchical relationships between caches
4. **Intelligent clients**: Offloading decision-making to client applications
5. **ISP partnerships**: Working with network providers rather than against them

CDN Architecture Patterns Applied Beyond Video

The patterns used by Netflix have broader applications:

- **Software updates**: Similar approaches work for distributing software updates (like Windows Update or game patches)
- **IoT firmware delivery**: Smart device manufacturers use tiered distribution for firmware updates
- **Large dataset distribution**: Academic and scientific large datasets use similar hierarchical approaches

Future Trends in Content Delivery

Where are CDNs headed next?

1. **Edge computing**: Moving beyond content delivery to code execution at the edge
2. **AI-driven cache optimization**: More sophisticated prediction algorithms
3. **5G integration**: Direct peering with mobile network operators
4. **Protocol innovations**: HTTP/3 and QUIC adoption for improved performance
5. **Multi-CDN strategies**: Using multiple CDNs for resilience and optimization

Conclusion

Netflix's Open Connect CDN demonstrates how company-specific CDN solutions can be tailored to unique content delivery needs. While a custom CDN isn't right for every organization, the principles of hierarchical distribution, intelligent client applications, and proactive caching can apply broadly.

By placing content closer to users, optimizing for specific workloads, and partnering with network providers, Netflix has created one of the world's most efficient video delivery systems—capable of delivering millions of concurrent streams while maintaining high quality and reliability.

Understanding these patterns helps us design better content delivery solutions for any application that needs to move large volumes of data to distributed users efficiently.

15. Message Queues Explained with Café Analogies

Picture the morning rush at your favourite café. Customers stream in, baristas work frantically, and somehow everyone (eventually) gets their coffee. This organized chaos perfectly illustrates how message queues function in modern software systems.

What Is a Message Queue?

A message queue is a communication mechanism that allows different parts of a software system to exchange messages asynchronously. Like a café's order system, it creates a buffer between the components that produce information (producers) and those that consume it (consumers).

System Design Interview Roadmap is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

Basic Message Queue Flow

The Café Analogy

Imagine a busy café with these key components:

1. **Customers (Producers)** - They place orders at varying rates and times
2. **Order Counter (Message Queue)** - Where orders wait to be processed
3. **Baristas (Consumers)** - They prepare drinks when they're available

In this scenario:

- Orders (messages) are placed regardless of how busy the baristas are
- The counter (queue) holds orders in the sequence they were received
- Baristas (consumers) process orders at their own pace without rushing

Core Benefits of Message Queues

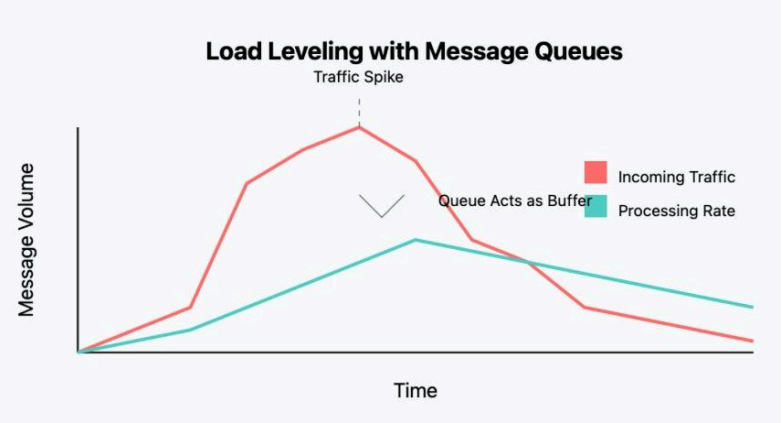
1. Decoupling

Just as customers don't need to wait for baristas to be free before ordering, producers can send messages without worrying about consumer availability. This independence creates robust systems where components can evolve separately.

2. Load Leveling

During morning rush hour, the café counter accumulates orders while baristas work steadily. Similarly, message queues absorb traffic spikes, preventing system overload during peak times.

Load Leveling Visualization



3. Reliability

If a barista takes a break, orders don't disappear—they wait on the counter. Similarly, if a consumer service fails, messages remain safely in the queue until processing can resume.

4. Scalability

During rush hour, the café might add more baristas. Likewise, multiple consumer instances can process messages from the same queue, distributing workload dynamically.

Message Queue Patterns

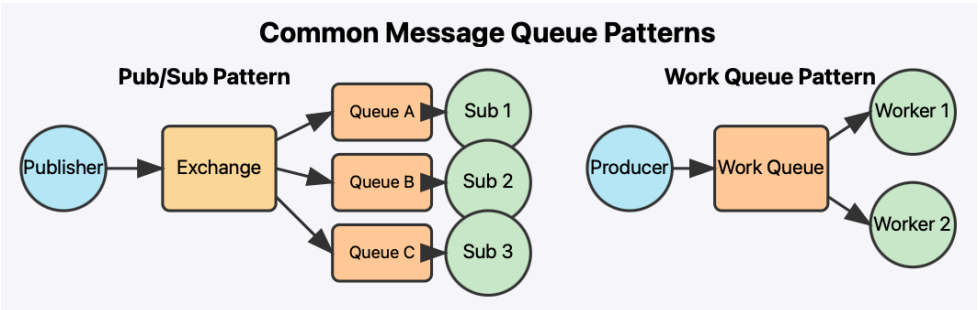
Pub/Sub (Publication/Subscription)

Like a café's intercom system where multiple baristas hear "Order up!" announcements, pub/sub allows one message to reach multiple consumers.

Work Queues

Similar to specialized baristas (one for espresso, another for pastries), work queues distribute tasks among different worker services based on their specialties.

Message Queue Patterns



Real-World Applications

E-commerce Order Processing

When you click "Buy Now," your order enters a queue. Even if payment processing is slow or inventory systems are busy, your order waits safely in line without causing the website to freeze.

IoT Data Collection

Smart devices (like weather sensors) generate data constantly. Message queues collect these readings, ensuring no data is lost even when processing systems are overwhelmed.

Microservice Communication

In large applications with dozens of separate services, message queues enable reliable communication between components that operate at different speeds or availability levels.

Popular Message Queue Technologies

- **RabbitMQ** - Feature-rich, general-purpose message broker supporting multiple protocols
- **Apache Kafka** - High-throughput distributed streaming platform, ideal for event streams
- **Amazon SQS** - Fully managed queue service with guaranteed delivery
- **Redis Streams** - In-memory data structure providing queue-like functionality with high performance

Implementing Basic Message Queuing

Think of implementing a message queue like setting up the café's order system:

1. **Design your messages** - Like order tickets with clear information
2. **Configure durability settings** - Will messages persist if the system restarts?
3. **Set up retry policies** - How to handle failed processing attempts
4. **Implement poison message handling** - What happens with problematic orders

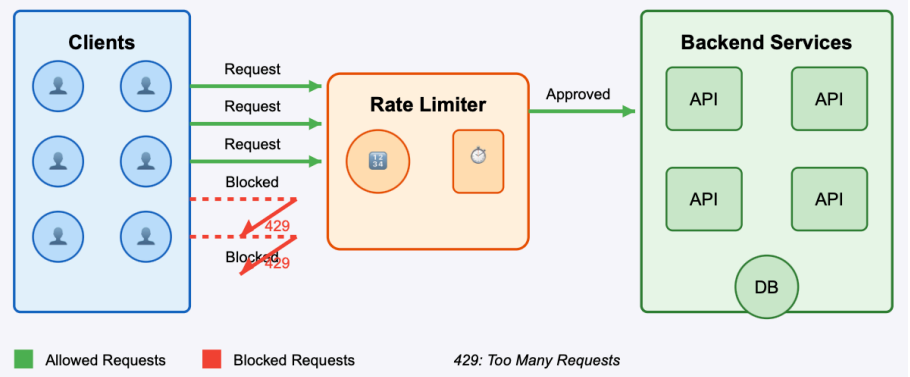
Conclusion

Message queues transform chaotic interactions into organized flows, just like a well-run café turns the morning coffee rush into a smooth, efficient process. By buffering communication between components, they create systems that are more reliable, scalable, and resilient.

Whether you're building a small application or an enterprise system handling millions of operations, understanding queue-based architectures will help you design solutions that gracefully handle real-world conditions.

16. Rate Limiting: Protecting Your System from Overload

When I was leading the infrastructure team at a rapidly growing fintech company, we experienced what I call "the perfect storm." During a major product launch, our APIs suddenly received 20x normal traffic. Some was legitimate user interest, but much was from aggressive bots and scrapers. Within minutes, database connections were exhausted, response times skyrocketed, and the entire system became unresponsive. That day taught me a vital lesson about system resilience that I never forgot.



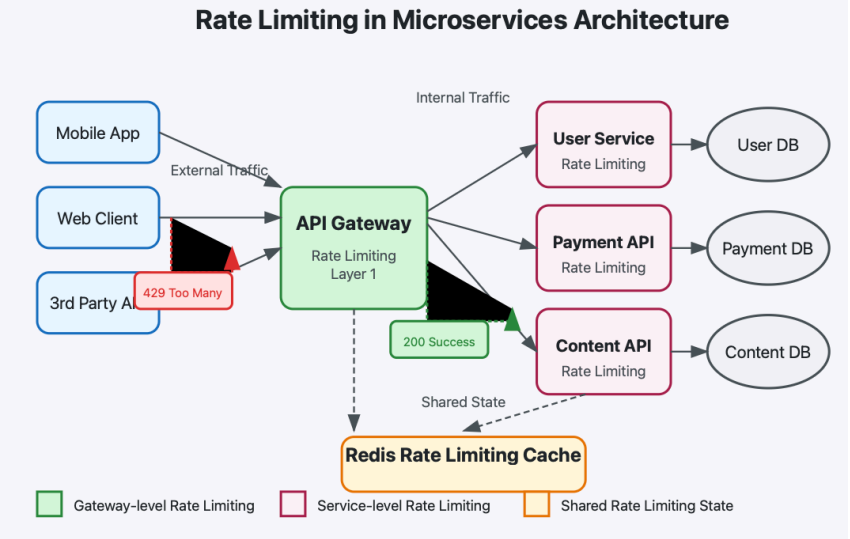
Why Rate Limiting Matters

Rate limiting is like having a bouncer at your API's door – it determines who gets in and at what pace. In today's high-traffic digital landscape, your system can easily become overwhelmed by request floods – whether from legitimate traffic spikes, internal bugs, or malicious attacks. Rate limiting serves as your first line of defense, ensuring system stability and reliability even under extreme conditions.

Without it, your system remains vulnerable to:

- Denial of service attacks (DoS/DDoS)
- Traffic spikes that exceed capacity
- Aggressive clients consuming disproportionate resources
- Cascading failures as overloaded services affect others
- Unexpected billing spikes from excessive API usage

The beauty of rate limiting is its dual nature: it's both defensive (protecting systems) and fair (ensuring equitable resource distribution among all users).



Core Rate Limiting Algorithms

Understanding different rate limiting approaches helps you select the right one for your specific needs:

1. Token Bucket Algorithm

Imagine each user has a bucket that holds tokens. Each request consumes one token. Tokens regenerate at a fixed rate over time. When a user's bucket is empty, their requests are rejected until more tokens become available.

Properties:

- Allows for bursts of traffic (up to bucket size)
- Simple to implement and understand
- Memory efficient

Can handle variable-cost operations by consuming different token amounts

- A typical implementation requires tracking:
- The bucket capacity (maximum tokens)
- The refill rate (tokens per time unit)
- The last refill timestamp
- Current token count

2. Leaky Bucket Algorithm

This approach processes requests at a constant rate. If requests arrive faster than can be processed, they're queued. When the queue fills, additional requests are rejected.

Properties:

- Smooths out traffic (no bursts)
- Provides predictable resource usage
- Can introduce latency due to queuing

- Ensures smooth, predictable load on backend systems
- More complex implementation

Best for: Systems where protecting backend processing capacity is the primary concern

3. Fixed Window Counter

In this approach, you track request counts in fixed time windows (e.g., 60-second periods). When a window's counter reaches its limit, additional requests are rejected until the next window starts.

Properties:

- Simple to implement
- Less memory overhead
- Has an "edge problem" - traffic can double at window boundaries
- Easy to understand conceptually

Best for: Simple applications with predictable, evenly distributed traffic

4. Sliding Window Log

This algorithm tracks timestamps of all requests in a time window. When a new request arrives, old timestamps outside the window are discarded, and the remaining count determines if the new request is allowed.

Properties:

- More precise than fixed windows
- Higher memory usage (stores all timestamps)
- No edge problem
- Highly precise limiting

Best for: Applications requiring precise control where memory isn't a constraint

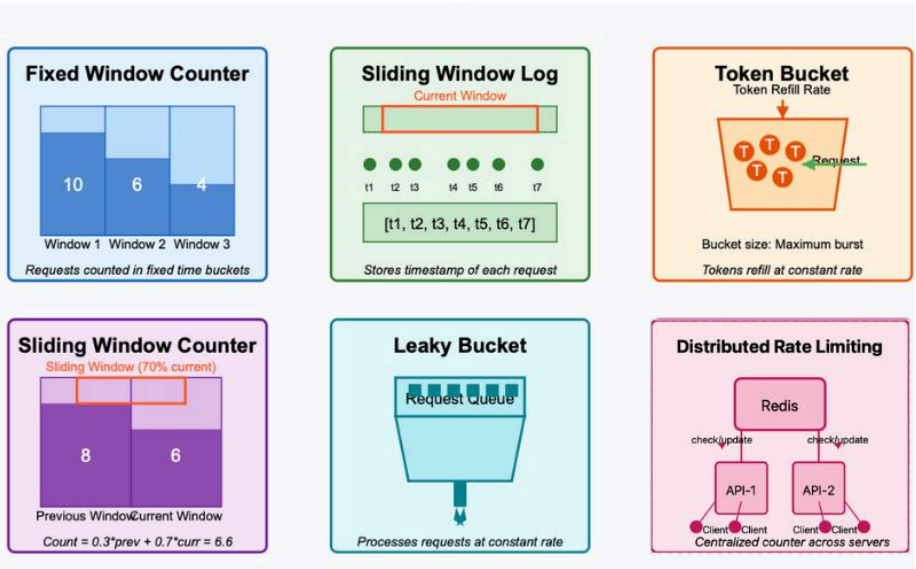
5. Sliding Window Counter

A hybrid approach that approximates request distribution by weighing the current and previous fixed windows.

Properties:

- Good balance of precision and memory efficiency
- Minimal edge problem
- More complex to implement
- Approximates rather than perfectly measuring traffic

Best for: Most production systems with moderate to high traffic



Implementation Strategies - Local vs. Distributed Rate Limiting

In single-server applications, local rate limiting is straightforward. However, in distributed systems with multiple instances, challenges arise:

Local Rate Limiting:

- Each instance maintains its own limits
- Simple implementation
- Limited effectiveness in distributed environments

Distributed Rate Limiting:

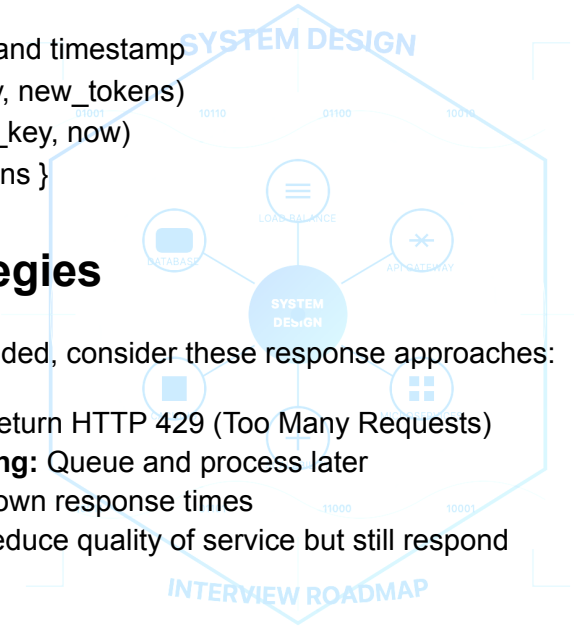
- Requires a shared state (e.g., Redis, Memcached)
- Consistent across all service instances
- More complex, but more effective

Redis-Based Implementation

Redis is particularly well-suited for distributed rate limiting due to its atomic operations and time-based data expiration. Here's a simplified Redis implementation of the token bucket algorithm:

```
--lua
-- KEYS[1]: the redis key to use
-- ARGV[1]: the maximum bucket size
-- ARGV[2]: the refill rate (tokens per second)
-- ARGV[3]: the requested token count
-- ARGV[4]: the current timestamp

local tokens_key = KEYS[1]
local timestamp_key = KEYS[1] .. ":ts"
local max_tokens = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])  -- tokens/second
local requested = tonumber(ARGV[3])
```

```
local now = tonumber(ARGV[4])

-- Get the current token count and last refill timestamp
local last_tokens = tonumber(redis.call("get", tokens_key)) or max_tokens
local last_refreshed = tonumber(redis.call("get", timestamp_key)) or 0

-- Calculate tokens to add based on time elapsed
local elapsed = math.max(0, now - last_refreshed)
local new_tokens = math.min(max_tokens, last_tokens + (elapsed * refill_rate))

-- Determine if the request can be fulfilled
local allowed = 0
if new_tokens >= requested then
  new_tokens = new_tokens - requested
  allowed = 1
End

-- Update the token count and timestamp
redis.call("set", tokens_key, new_tokens)
redis.call("set", timestamp_key, now)
return { allowed, new_tokens }
```



Response Strategies

When rate limits are exceeded, consider these response approaches:

1. **Hard Rejection:** Return HTTP 429 (Too Many Requests)
2. **Delayed Processing:** Queue and process later
3. **Throttling:** Slow down response times
4. **Tiered Service:** Reduce quality of service but still respond

Always include useful headers:

X-RateLimit-Limit: 100

X-RateLimit-Remaining: 0

X-RateLimit-Reset: 1609459200

Retry-After: 60

Advanced Considerations

Multi-Dimensional Rate Limiting

Simple rate limiting applies the same limits to all users and endpoints. A more sophisticated approach uses multiple dimensions:

- Per-user limits (authenticated users)
- Per-IP limits (unauthenticated traffic)
- Per-endpoint limits (protecting critical paths)
- Global limits (overall system protection)

Adaptive Rate Limiting

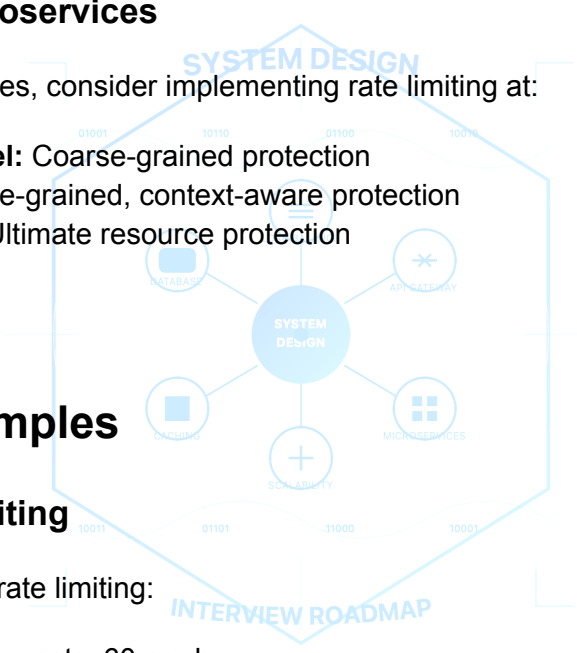
Rather than static limits, consider adjusting thresholds based on:

- Server load
- Time of day
- User behavior patterns
- Business priorities

Rate Limiting in Microservices

In microservice architectures, consider implementing rate limiting at:

1. **API Gateway Level:** Coarse-grained protection
2. **Service Level:** Fine-grained, context-aware protection
3. **Database Level:** Ultimate resource protection



Real-World Examples

GitHub API Rate Limiting

GitHub implements tiered rate limiting:

- Unauthenticated requests: 60 per hour
- Authenticated requests: 5,000 per hour
- Conditional requests against modified resources don't count toward limits

Cloudflare Rate Limiting

- Cloudflare provides customizable rules based on:
- Request counts
- Response codes
- URI paths
- Client characteristics

Redis Rate Limiter

Redis's own rate-limiting module (redis-cell) implements a sophisticated algorithm called Generic Cell Rate Algorithm (GCRA) that efficiently manages rate limiting with minimal memory overhead.

Bringing It All Together

Looking at our token bucket diagram, you can see how this algorithm allows for bursts of traffic while maintaining long-term limits. When a client makes a request, we check if there's a token available. If yes, the request proceeds and consumes a token; if not, it's rejected. The bucket refills at a constant rate, allowing clients that briefly exceed limits to still proceed if they have tokens accumulated.

The comparison diagram shows why different algorithms suit different scenarios. Fixed window is simplest but has boundary problems. Leaky buckets provide the most consistent processing rate. The sliding window approaches strike a balance between precision and efficiency.

In the microservices architecture diagram, rate limiting becomes a multi-layered defense. The API gateway provides coarse-grained protection, catching obvious abuse before it reaches internal services. Individual microservices then implement their own rate limits, tailored to their specific resources and constraints. The shared Redis cache ensures consistent enforcement across all service instances.

Implementation Best Practices

For effective implementation, remember these best practices:

1. **Set appropriate limits**: Too strict and you'll frustrate legitimate users; too loose and you won't protect your system effectively.
2. **Provide clear feedback**: Rate limit responses should include clear headers indicating limits, remaining requests, and reset times.
3. **Consider tiered limiting**: Different client types, authenticated vs. anonymous, or premium vs. free users might deserve different limits.
4. **Monitor and adjust**: Rate limits aren't set-and-forget. Analyze traffic patterns and adjust accordingly.

Conclusion

Rate limiting is not just a defensive mechanism—it's a cornerstone of reliable system design. The right implementation protects your infrastructure, ensures fair resource allocation, and ultimately improves user experience by maintaining performance under varying load conditions.

As you design your next system, remember that rate limiting is like an insurance policy—you hope you'll never need it, but you'll be incredibly grateful it's there when you do.

In our next newsletter, we'll explore another critical aspect of system design: data partitioning strategies. Until then, happy coding!

17. Proxies vs. API Gateways: Understanding the Differences

Understanding the Critical Gateway Components

Welcome to the next instalment in our System Design Interview Roadmap series! Today, we're diving deep into proxies and API gateways—two critical components that often cause confusion despite their fundamental role in modern distributed systems.

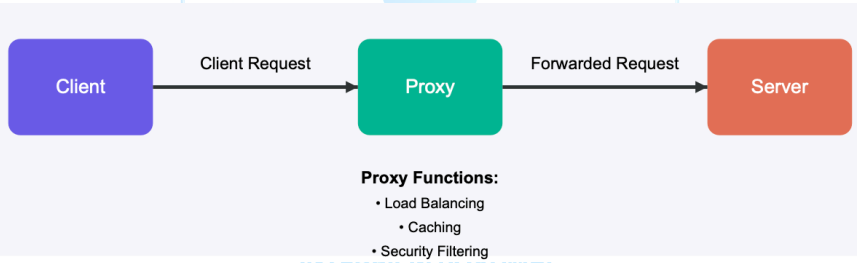
When designing systems that handle millions of requests per second, these components become essential building blocks rather than theoretical concepts. Let's unpack how they differ and when to use each one.

System Design Interview Roadmap is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

What Are Proxies?

At their core, proxies are intermediaries sitting between clients and servers. Think of them as trusted middlemen in network communication that intercept traffic before it reaches its final destination.

Basic Proxy Architecture



Key Types of Proxies

1. **Forward Proxies**: Sit closer to clients and represent clients to servers
    a. Hide client identity from servers
    b. Enable content filtering and access control
    c. Example: Corporate proxies controlling employee internet access
2. **Reverse Proxies**: Sit closer to servers and represent servers to clients
    a. Hide server details from clients
    b. Distribute traffic among backend servers
    c. Provide SSL termination and compression
    d. Example: Nginx sitting in front of application servers

Core Proxy Functionality

Proxies excel at **protocol-level operations**:

- Load balancing (distributing traffic across servers)
- Caching frequently requested content

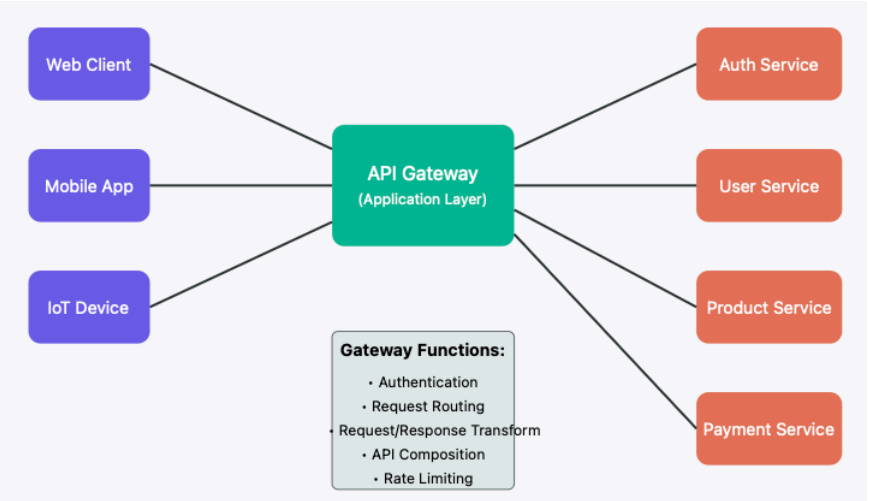
- SSL termination (handling encryption/decryption)
- Basic content filtering
- Connection pooling

A well-configured proxy allows you to handle massive scale by distributing requests efficiently across your infrastructure without changing your application code.

What Are API Gateways?

API gateways evolved from proxies but operate at a higher abstraction level. While proxies handle general network traffic, API gateways understand and manipulate API requests specifically.

API Gateway Architecture



Key API Gateway Features

API gateways operate at the **application layer** with these distinguishing capabilities:

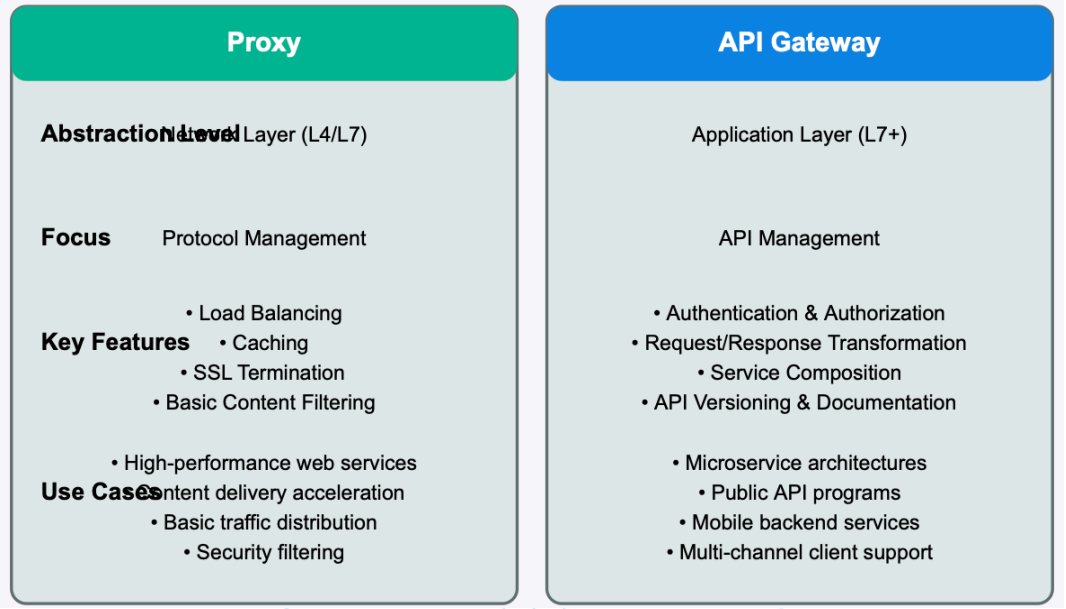
1. **API-aware processing**:
    a. Understanding API semantics (methods, paths, parameters)
    b. Request/response transformation
    c. Protocol translation (e.g., converting SOAP to REST)
2. **Advanced traffic management**:
    a. Sophisticated routing based on content
    b. API versioning
    c. A/B testing capabilities
    d. Canary deployments
3. **Developer-focused features**:
    a. API documentation
    b. Developer portals
    c. API usage analytics
    d. SDK generation
4. **Business capabilities**:
    a. Monetization features
    b. Usage quotas

c. Rate limiting tied to business rules
d. Granular access control

When handling millions of requests in microservice environments, API gateways become critical in maintaining system coherence while allowing individual services to evolve independently.

Key Differences: A Comparative View

Proxy vs API Gateway Comparison



The Layer of Operation

The fundamental difference lies in their operational layer:

- **Proxies** operate primarily at the network/protocol layer (though L7 proxies can understand HTTP)
- **API Gateways** operate at the application layer with API semantic understanding

This distinction becomes crucial when deciding which component to use in your architecture.

Real-World Implementations

Popular Proxy Solutions

1. **NGINX**: Powers ~40% of the world's busiest websites
    - Strengths: Exceptional performance, low memory footprint
    - Use case: Netflix uses NGINX for traffic routing and load balancing at massive scale
2. **HAProxy**: High-performance TCP/HTTP load balancer
    - Strengths: Session persistence, advanced health checking
    - Use case: GitHub uses HAProxy for their load balancing needs
3. **Envoy**: Modern L7 proxy designed for cloud-native applications

- Strengths: Dynamic configuration, observability
- Use case: Lyft built and uses Envoy as their service mesh proxy

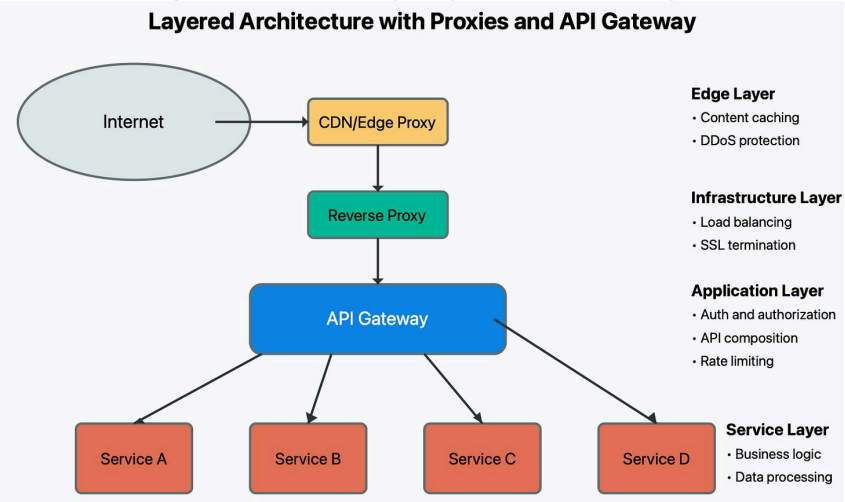
Popular API Gateway Solutions

1. **Kong**: Open-source API Gateway built on NGINX
   - Strengths: Plugin architecture, high performance
   - Use case: Expedia uses Kong to manage thousands of APIs
2. **Amazon API Gateway**: Fully managed AWS service
   - Strengths: Serverless integration, AWS ecosystem
   - Use case: Airbnb uses it to handle millions of API calls from their mobile apps

3. **Apigee**: Google Cloud's API management platform
   - Strengths: Advanced analytics, developer portal
   - Use case: Walgreens uses Apigee to expose pharmacy services to partners

Integration Patterns: How They Work Together

In sophisticated architectures handling millions of requests per second, proxies and API gateways often work together in complementary roles:



Typical Request Flow in Layered Architecture

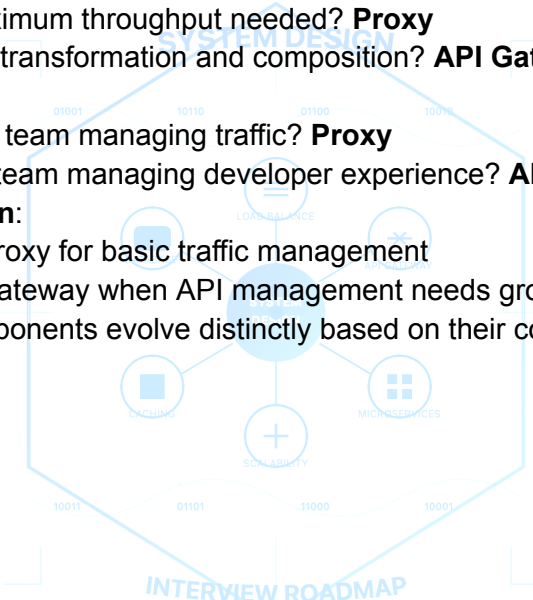
1. **Edge Layer** (CDN/Edge Proxy)
   - Handles global traffic distribution
   - Provides DDoS protection
   - Caches static content
2. **Infrastructure Layer** (Reverse Proxy)
   - Manages traffic across data centers
   - Handles SSL termination
   - Provides basic load balancing
3. **Application Layer** (API Gateway)

    - Manages API-specific concerns
    - Routes to appropriate microservices
    - Handles authentication and transformations
4. **Service Layer** (Microservices)
    - Implements business logic
    - Processes data
    - Manages domain-specific operations

Making the Right Choice

Consider these factors when deciding what to deploy:

1. **Traffic Characteristics**:
    a. Mostly undifferentiated HTTP traffic? **Proxy**
    b. Complex API ecosystem with multiple clients? **API Gateway**
2. **Performance Requirements**:
    a. Absolute maximum throughput needed? **Proxy**
    b. Need for API transformation and composition? **API Gateway**
3. **Team Structure**:
    a. Infrastructure team managing traffic? **Proxy**
    b. API platform team managing developer experience? **API Gateway**
4. **Evolutionary Pattern**:
    a. Start with a proxy for basic traffic management
    b. Add an API gateway when API management needs grow
    c. Let both components evolve distinctly based on their concerns



Conclusion

Understanding the distinction between proxies and API gateways is crucial for designing systems that can handle millions of requests per second. While proxies excel at protocol-level operations with maximum efficiency, API gateways provide the application-level intelligence needed for sophisticated API management.

In practice, modern architectures often employ both—proxies handling the raw networking efficiency and API gateways managing the business of APIs. This layered approach allows each component to focus on what it does best, creating systems that are both robust and flexible.

18. Consistency Models in Distributed Systems: Balancing Truth in a Divided World

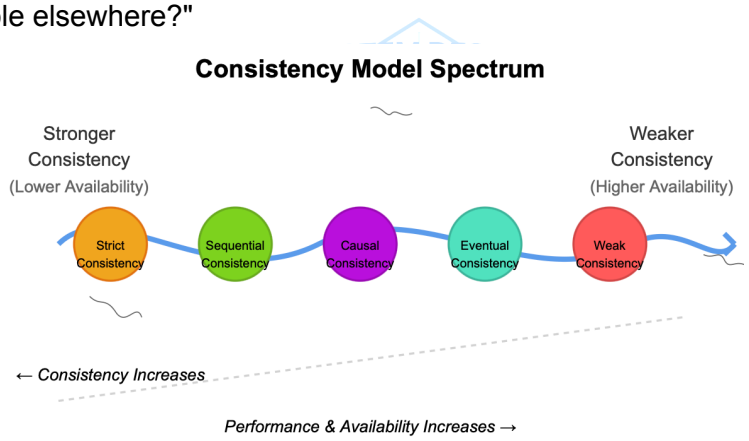
Welcome back, fellow system architects! Last week we explored load balancing strategies for high-throughput APIs. Today, we're diving into what might be the most intellectually challenging aspect of distributed systems: consistency models.

As our systems scale to handle millions of requests per second, we inevitably face a fundamental dilemma: how do we ensure all nodes in our distributed system share the same view of reality?

Let's explore consistency models through practical examples and visualizations that make these abstract concepts concrete.

Understanding Consistency in Distributed Systems

Consistency in distributed systems refers to how and when updates made to data become visible to different parts of the system. It's about answering a seemingly simple question: "If I write data here, when and how will it be readable elsewhere?"

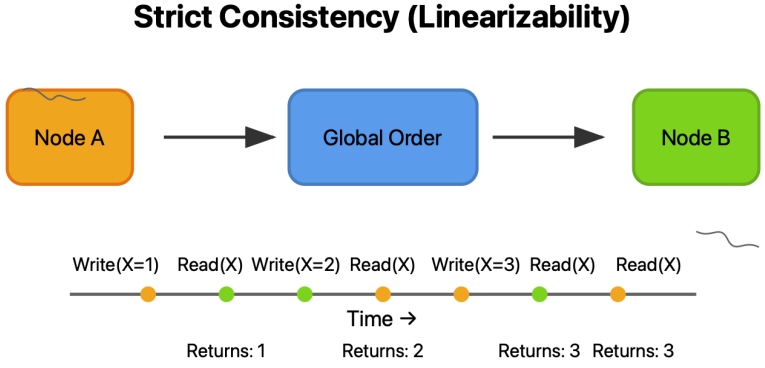


The spectrum above illustrates the trade-offs we make between consistency and availability. As we move from left to right, we gain performance and availability but sacrifice consistency. Let's explore each model:

Strong Consistency Models

Strict Consistency (Linearizability)

This is the gold standard, where any read operation returns the most recent write, regardless of which node performs the operation. It's as if all operations happen atomically on a single node.



Real-world use case: Financial transactions where every penny must be accounted for. Google's Spanner database uses TrueTime API to achieve this.

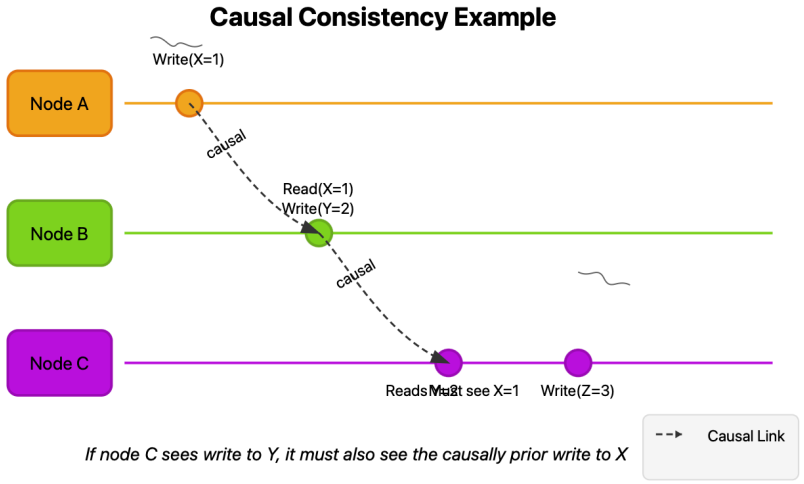
Trade-offs: Achieving strict consistency typically requires:

- Performance penalties due to synchronization
- Limited availability during network partitions (CAP theorem in action)

Sequential Consistency

Sequential consistency guarantees that all operations appear to execute in some sequential order, and operations from each node appear in the order they were issued. However, unlike strict consistency, there's no real-time constraint between operations from different nodes.

Causal Consistency



Weaker Consistency Models

Causal Consistency

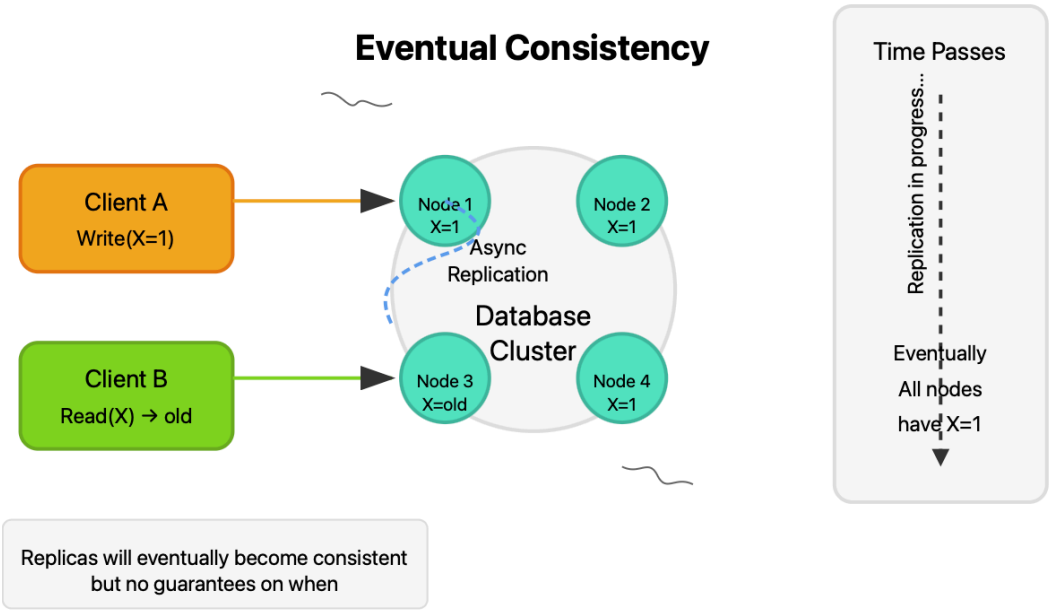
Causal consistency ensures that operations that are causally related are seen by all nodes in the same order. Operations with no causal relationship (concurrent operations) may be seen in different orders on different nodes.

The diagram above illustrates a scenario where:

1. Node A writes X=1
2. Node B reads X=1 and then writes Y=2 (causally dependent on reading X)
3. Node C reads Y=2, which means it must also see X=1 to maintain causal consistency

Real-world use case: Social media platforms where comment threads need to appear in causal order (replies appearing after the original post).

Eventual Consistency



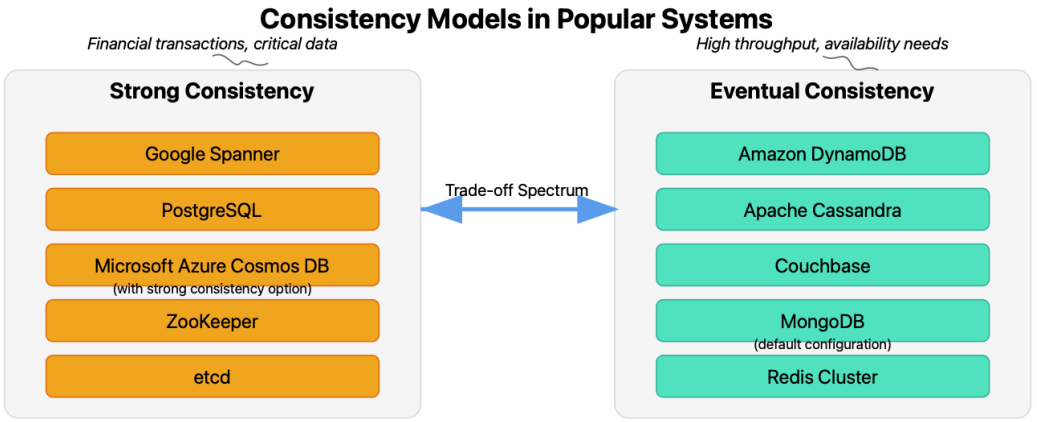
Eventual consistency guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. There are no guarantees about when this will happen, just that it eventually will.

Real-world use case: Amazon DynamoDB, Cassandra, and most NoSQL databases favor this model. It's perfect for:

- Product catalogs where slight inconsistencies are acceptable
- Social media non-critical data like follower counts
- Content delivery networks (CDNs)

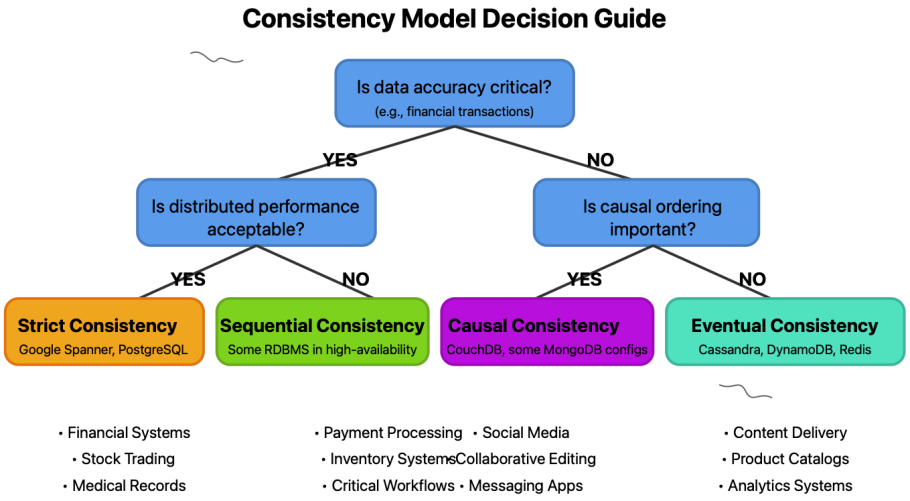
Real-World Implementations

Real-World Implementations



Selecting the Right Consistency Model

The right consistency model depends on your specific application requirements. Let's explore some common scenarios:



Implementation Strategies

Implementing different consistency models often requires specific approaches:

For Strong Consistency

- Use consensus algorithms like Paxos or Raft
- Implement two-phase commit (2PC) for distributed transactions
- Consider the performance impact of synchronous operations

For Eventual Consistency

- Use vector clocks or logical timestamps to track updates
- Implement conflict resolution strategies (Last-Writer-Wins, CRDTs)
- Design your application to handle temporary inconsistencies gracefully

Common Challenges and Solutions

1. **Network Partitions**: Network partitions are inevitable in distributed systems. The CAP theorem tells us we must choose between consistency and availability when partitions occur.
    - Solution: Choose consistency models appropriate for your use case, and consider using different models for different data types.
2. **Data Conflicts**: In eventual consistency models, conflicts are inevitable.
    - Solution: Implement Conflict-free Replicated Data Types (CRDTs) or custom conflict resolution mechanisms.
3. **Stale Reads**: Clients may read stale data in weaker consistency models.
    - Solution: Implement read-your-writes consistency by directing reads to nodes that processed the writes.

Conclusion

Choosing the right consistency model is a critical architectural decision that affects your system's scalability, availability, and correctness. The best approach is often to use different consistency models for different parts of your system based on their specific requirements.

Remember:

- Strong consistency is necessary for financial and critical data
- Causal consistency works well for social and collaborative applications
- Eventual consistency provides high availability and performance for less critical data

In the next newsletter, we'll explore how to implement these consistency models in microservice architectures and how to design systems that maintain consistency across service boundaries.

Until then, happy designing!

P.S. Did you struggle with implementing any particular consistency model in your systems? Share your experiences in our community forum, and I'll address common challenges in a future deep-dive article.

19. Data Serialization Formats: JSON, Protobuf, Avro

Today, we're diving into data serialization formats—a critical component of any distributed system handling millions of requests per second.

Why Serialization Matters

When building high-throughput systems, how you represent data as it flows between services can make or break your architecture. Choosing the right serialization format impacts:

System Design Interview Roadmap is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

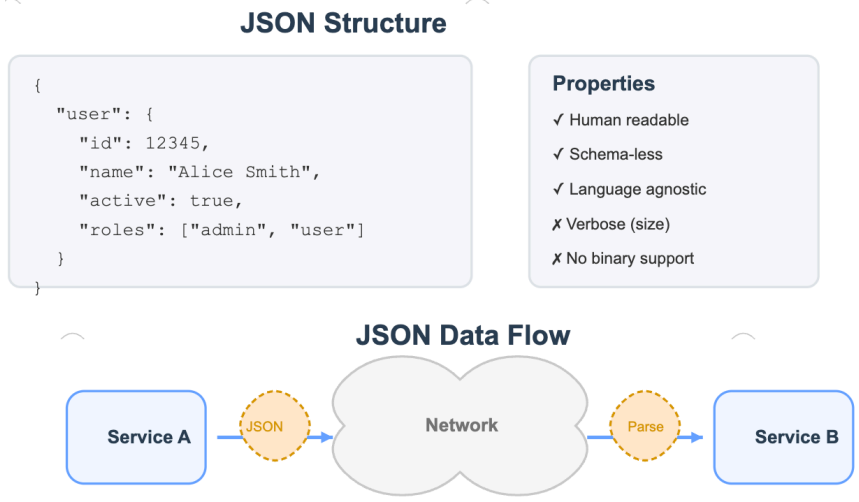
- Network bandwidth utilization
- Processing overhead
- Storage efficiency
- Developer productivity
- Cross-language compatibility

Let's examine three popular formats that dominate modern system design.

JSON: The Universal Language

JavaScript Object Notation (JSON) has become the lingua franca of web APIs and configuration files. Its human-readable format makes it developer-friendly while still being machine-parsable.

JSON Structure and Flow



JSON Characteristics

Strengths:

- **Simplicity**: Easy to understand and debug
- **Ubiquity**: Supported in virtually every programming language
- **Schema Flexibility**: Add or remove fields without breaking clients

Limitations:

- **Verbosity**: Field names repeated in every instance
- **Type Handling**: Limited primitive types
- **Performance**: Text parsing is computationally expensive
- **Size**: Larger payload size impacts network efficiency

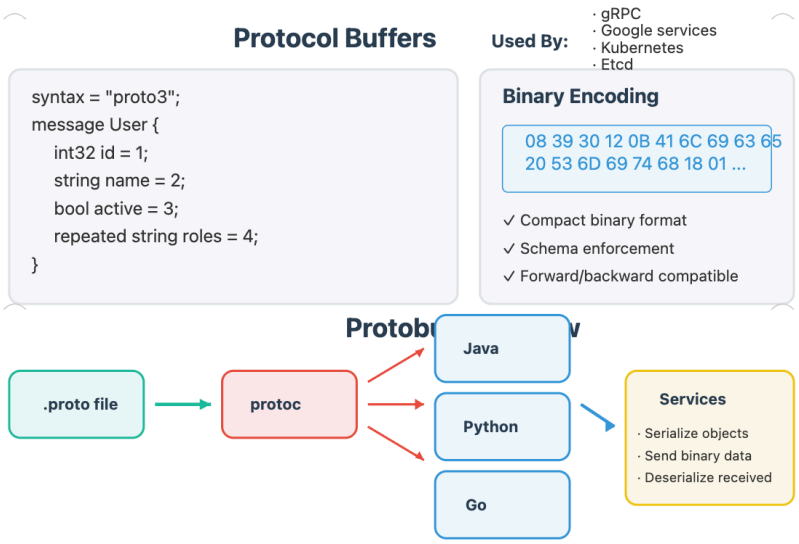
Real-World Use Cases

- **RESTful APIs**: Most public APIs use JSON
- **Configuration Files**: Docker, npm, and countless others
- **Document Databases**: MongoDB and CouchDB store JSON documents

Protocol Buffers (Protobuf): Google's High-Performance Format

When Google needed something faster and more compact than XML or JSON, they created Protocol Buffers (Protobuf). Now open-source, Protobuf has become the backbone of high-performance RPC systems.

Protocol Buffers Structure and Flow



Protobuf Characteristics

Strengths:

- **Performance**: 3-10x smaller than JSON, up to 100x faster to parse
- **Strictly Typed**: Built-in schema validation
- **Code Generation**: Automatically generates client libraries
- **Schema Evolution**: Field additions/removals maintain compatibility
- **Cross-Language**: Seamless interoperability across languages

Limitations:

- **Development Overhead**: Requires schema compilation step
- **Human Readability**: Binary format requires tools to inspect
- **Ecosystem**: Less widespread than JSON (but growing rapidly)

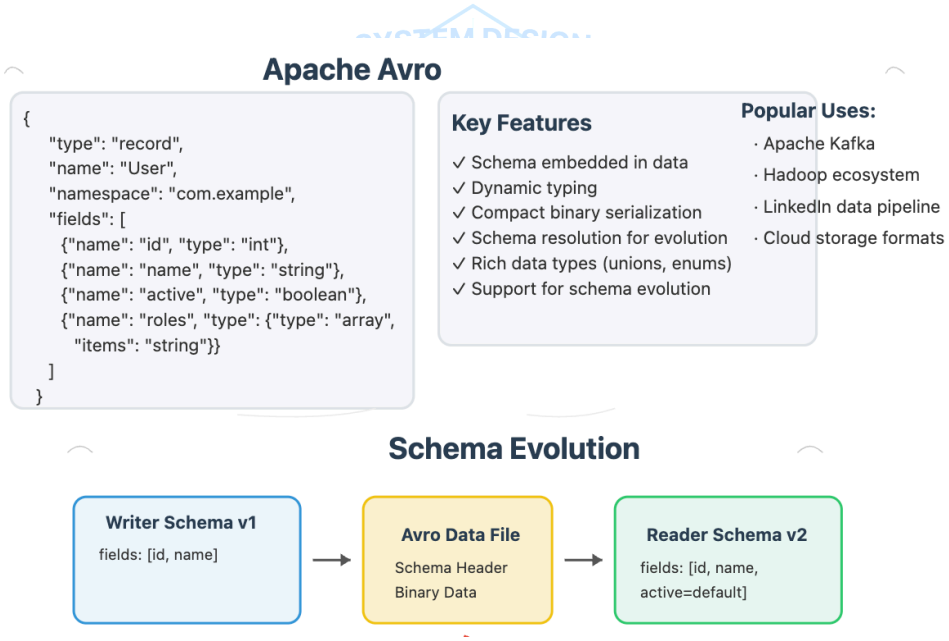
Real-World Use Cases

- **gRPC**: Powers high-performance microservices communication
- **Google's Internal Systems**: Powers most of Google's infrastructure
- **Kubernetes**: Used for internal communication
- **ETCD**: High-performance distributed key-value store

Apache Avro: Hadoop's Compact Binary Format

Developed for Apache Hadoop, Avro combines schema-based serialization with dynamic typing, making it particularly well-suited for data-intensive applications.

Apache Avro Structure and Flow



Avro Characteristics

Strengths:

- **Schema Evolution**: Exceptional handling of schema changes
- **Self-Describing**: Schema embedded in data files
- **Rich Data Types**: Complex nested structures
- **Speed & Size**: Comparable to Protobuf for performance
- **Dynamic Typing**: No code generation step required
- **Splittable**: Important for big data processing

Limitations:

- **JSON Schema Syntax**: Steeper learning curve
- **Schema Registry**: Often requires external schema management

- **Community Size**: Smaller than JSON or Protobuf communities

Real-World Use Cases

- **Apache Kafka**: Default serialization format for data pipelines
- **Hadoop Ecosystem**: Used throughout the Hadoop stack
- **LinkedIn Data Pipeline**: Powers LinkedIn's data infrastructure
- **Parquet Files**: Used with Avro for columnar storage

Comparative Analysis

Serialization Format Comparison

Format Comparison

Feature	JSON	Protobuf	Avro
Human Readability	High	Low	Medium
Message Size	Large	Small	Small
Schema Required	Optional	Required	Required
Parsing Speed	Slow	Fast	Fast
Schema Evolution	Informal	Good	Excellent
Best For	Web APIs	Microservices	Big Data

Making the Right Choice: Decision Framework

Selecting the right serialization format depends on your specific requirements. Here's a simple decision framework:

Choose JSON when:

- Human readability is critical
- You need wide platform support
- You're building public-facing APIs
- Schema flexibility outweighs performance concerns
- Debugging and inspection needs are high

Choose Protocol Buffers when:

- Performance is critical (especially for microservices)
- Strict typing is required
- You have multiple programming languages communicating
- You need robust backward compatibility
- Code generation is beneficial

Choose Avro when:

- Data evolution is frequent
- Working with big data frameworks
- Self-describing data is needed
- You require rich, complex data structures
- Storage efficiency is important

Real-World Performance Benchmarks

For a system handling 10 million requests per second, these performance differences matter:

- **JSON**: Approximately 1.5-2KB per message
    - 15-20 GB/s bandwidth consumption
    - High CPU usage for parsing
- **Protobuf/Avro**: Approximately 200-300 bytes per message
    - 2-3 GB/s bandwidth consumption
    - Minimal CPU usage for serialization/deserialization

At Netflix, switching from JSON to Protocol Buffers for their API gateway reduced bandwidth by 80% and parsing CPU cycles by 60%, allowing them to handle 5x more requests on the same hardware.

Mixing Formats: The Hybrid Approach

Many large-scale systems use multiple formats in different parts of their architecture:

1. **Public APIs**: JSON for developer accessibility
2. **Internal Services**: Protocol Buffers for performance
3. **Data Pipelines**: Avro for schema evolution and storage

This approach leverages the strengths of each format while minimizing their weaknesses.

Conclusion

Data serialization is a critical component of high-performance system design. While JSON remains the most accessible choice for many applications, Protocol Buffers and Avro offer significant performance advantages for systems at scale.

In our next newsletter, we'll explore how to implement efficient caching strategies to further reduce serialization overhead in your distributed systems.

Happy engineering!

This article is part of our "System Design Interview Roadmap" series.

System Design Interview Roadmap is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

20. Web Sockets vs. Long Polling vs. Server-Sent Events - Real-Time Communication Patterns

In today's interconnected world, real-time communication has become essential for many applications. Whether you're building a chat application, a live dashboard, or a collaborative document editor, selecting the right communication pattern can significantly impact your system's performance, scalability, and user experience.

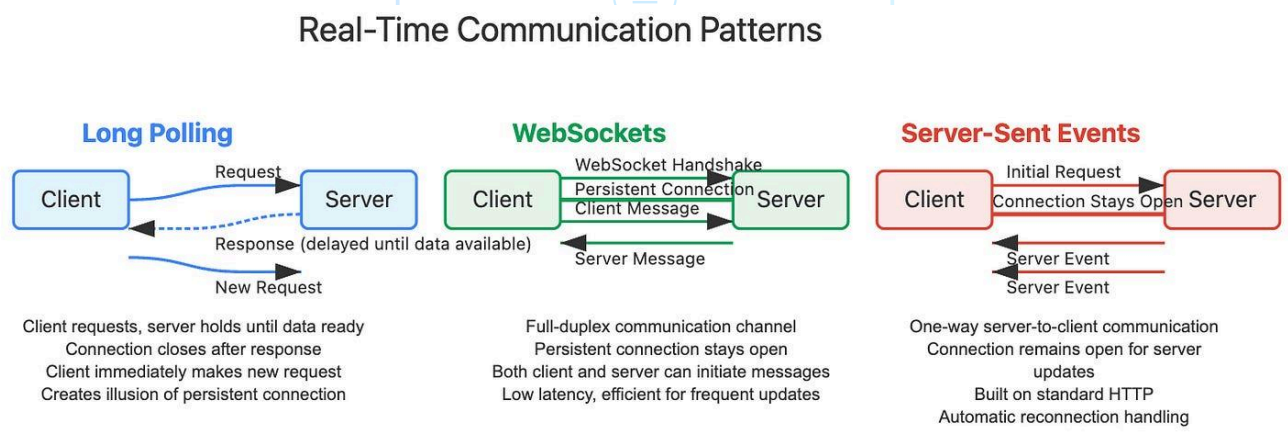
Today, we'll dive into three key approaches for establishing real-time connections between clients and servers: WebSockets, Long Polling, and Server-Sent Events (SSE). Let's understand how each works, their strengths and limitations, and when to use them.

The Real-Time Communication Landscape

To grasp these patterns, we first need to understand the fundamental challenge: HTTP was originally designed for one-way, request-response communication. A client makes a request, the server responds, and the connection closes. This model doesn't naturally support servers pushing data to clients without being asked first.

Let's visualize how each pattern addresses this challenge:

Real-Time Communication Patterns



Comparison at a Glance

Feature	Long Polling	WebSockets	Server-Sent Events
Communication	Request-Response	Bidirectional	Server → Client
Connection	Multiple HTTP	Persistent TCP	Single HTTP
Best For	Simple updates	Interactive apps	Notifications

WebSockets: The Full-Duplex Powerhouse

WebSockets establish a persistent, bidirectional communication channel between client and server over a single TCP connection. Unlike traditional HTTP, which follows a request-response pattern, WebSockets allow both client and server to send messages independently at any time.

How WebSockets Work:

1. **Handshake**: The connection begins with a standard HTTP request containing special headers that request an upgrade to the WebSocket protocol.
2. **Upgrade**: If the server supports WebSockets, it responds with an HTTP 101 (Switching Protocols) status code.
3. **Data Transfer**: Once established, both client and server can send data frames to each other without overhead of HTTP headers.

Code Example:

```
// Client-side WebSocket
const socket = new WebSocket('ws://example.com/socket');

socket.onopen = () => {
  console.log('Connection established');
  socket.send('Hello server!');
};

socket.onmessage = (event) => {
  console.log('Message from server:', event.data);
};

// Server-side (Node.js with ws library)
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    console.log('Received:', message);
    ws.send('Message received!');
  });
});
```

Long Polling: The Legacy Approach

Long polling simulates real-time communication within the constraints of the HTTP protocol. It's essentially an adaptation of the traditional request-response model.

How Long Polling Works:

1. **Client Request**: The client makes an HTTP request to the server.
2. **Delayed Response**: Instead of responding immediately, the server holds the request open until new data is available.
3. **Response & Reconnect**: Once data becomes available, the server responds, and the client immediately makes a new request to maintain the "connection."

Code Example:

```
// Client-side Long Polling
function longPoll() {
  fetch('/api/updates')
```

```
    .then(response => response.json())
    .then(data => {
      console.log('Received data:', data);
      // Process the data
      // Then immediately reconnect
      longPoll();
    })
    .catch(error => {
      console.error('Error:', error);
      // Wait before reconnecting after error
      setTimeout(longPoll, 5000);
    });
}

longPoll();

// Server-side (Express.js)
const pendingRequests = [];

app.get('/api/updates', (req, res) => {
  // Store the response object for later use
  pendingRequests.push(res);

  // Set timeout to prevent connection from hanging indefinitely
  req.on('close', () => {
    const index = pendingRequests.indexOf(res);
    if (index !== -1) pendingRequests.splice(index, 1);
  });
});

// When new data is available
function sendUpdate(data) {
  pendingRequests.forEach(res => {
    res.json(data);
  });
  pendingRequests.length = 0;
}
```

Server-Sent Events: The One-Way Street

Server-Sent Events (SSE) provide a one-way channel from server to client over a single, long-lived HTTP connection. They're perfect when you need server-to-client updates without needing to send data back.

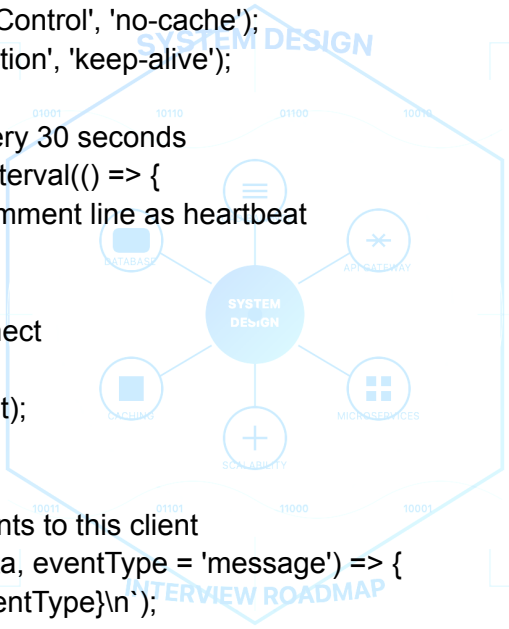
How SSE Works:

1. **Connection**: The client establishes a persistent connection to an endpoint on the server.
2. **Event Stream**: The server keeps the connection open and sends events in a special text format when new data is available.

3. **Automatic Reconnection**: If the connection breaks, browsers automatically attempt to reconnect.

Code Example:

```
// Client-side SSE
const eventSource = new EventSource('/api/events');

eventSource.onmessage = (event) => {
  console.log('New event:', event.data);
};

eventSource.addEventListener('custom-event', (event) => {
  console.log('Custom event:', event.data);
});

// Server-side (Express.js)
app.get('/api/events', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Send a heartbeat every 30 seconds
  const heartbeat = setInterval(() => {
    res.write(':\n\n'); // Comment line as heartbeat
  }, 30000);

  // Handle client disconnect
  req.on('close', () => {
    clearInterval(heartbeat);
  });

  // Function to send events to this client
  const sendEvent = (data, eventType = 'message') => {
    res.write(`event: ${eventType}\n`);
    res.write(`data: ${JSON.stringify(data)}\n\n`);
  };

  // Store the sendEvent function somewhere to use when new data arrives
  clients.push(sendEvent);
});
```



Real-World Applications

When to Use WebSockets:

- **Chat Applications**: Slack, WhatsApp, and Discord use WebSockets for instant message delivery.
- **Collaborative Editors**: Google Docs leverages WebSockets to synchronize changes between multiple users in real-time.

- **Live Gaming**: Online multiplayer games require the low-latency, bidirectional communication that WebSockets provide.

When to Use Long Polling:

- **Legacy System Integration**: When working with older systems that don't support newer protocols.
- **Simple Notifications**: For applications where updates are infrequent and bidirectional communication isn't necessary.
- **When WebSocket Support Is Limited**: In environments where WebSockets might be blocked by proxies or firewalls.

When to Use SSE:

- **News Feeds**: Twitter's streaming API uses SSE to push new tweets to timelines.
- **Stock Tickers**: Financial applications use SSE to stream market data updates.
- **Status Updates**: GitHub uses SSE to provide build and deployment status updates.

Conclusion

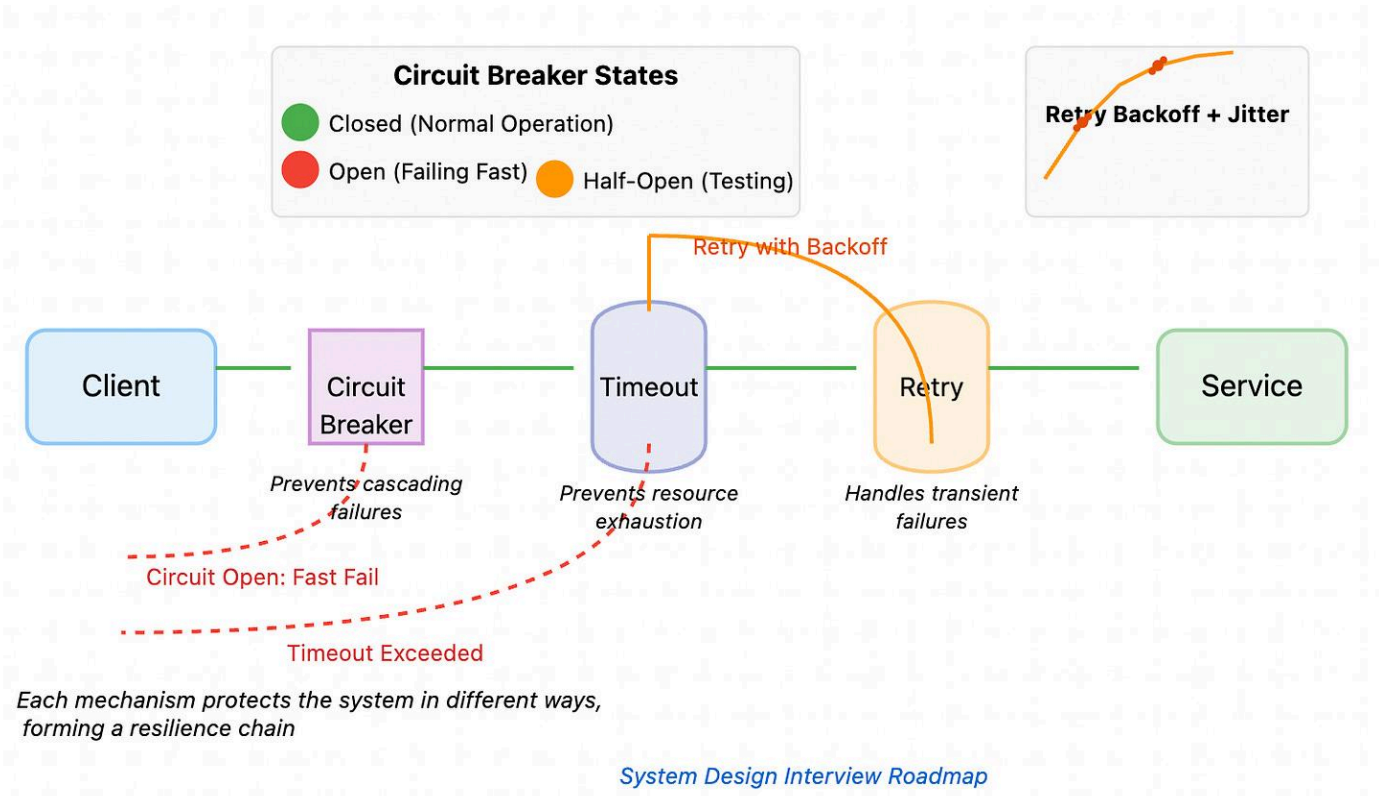
Each of these communication patterns has its place in modern web architecture. WebSockets offer the most power and flexibility with true bidirectional communication but come with added complexity. Long Polling provides a fallback solution that works everywhere but is less efficient. Server-Sent Events strike a balance with excellent browser support and efficiency for one-way communications.

When designing your next real-time system, consider your specific requirements around message direction, frequency, payload size, and browser compatibility before selecting the appropriate pattern. Remember that many production systems use a combination of these approaches for optimal performance across different scenarios.

21. Designing for Failure: Mastering Timeouts, Retries, and Circuit Breakers

Last month, a payment processor I consulted with experienced a 4-hour outage that cost them $2.3M in revenue. The root cause? A single overloaded service without proper timeout handling triggered cascading failures across their system. It's a story I've seen repeated countless times—sophisticated systems brought to their knees not by complex bugs, but by missing or misconfigured failure handling mechanisms.

Today, we'll explore three critical patterns that form the foundation of resilient distributed systems: timeouts, retries, and circuit breakers. These aren't just interview topics; they're the difference between a system that gracefully weathers storms and one that collapses under pressure.



Timeouts: Your First Line of Defense

A timeout is essentially a promise to yourself: "I will not wait forever." While conceptually simple, timeouts are surprisingly nuanced in practice.

What they are: Timeouts set boundaries on how long an operation can take before we consider it failed. They're like the chess clock that prevents a player from stalling indefinitely.

Why they matter: Without timeouts, your system can accumulate blocked threads or connections waiting for responses that may never come. This silently consumes resources until your service collapses—often without clear error signals.

How they work: A properly implemented timeout triggers two actions: terminating the ongoing operation and signaling failure to the caller. The challenge lies in ensuring both actions complete reliably.

When to use them: Implement timeouts on *every* remote call—no exceptions. But how do you choose the right duration? Start with your service's SLO. If you promise 99.9% of requests complete within 300ms, your timeout should exceed this (perhaps 500ms) while leaving room for retries. Remember: a timeout that's too short causes unnecessary retries; too long risks resource exhaustion.

The non-obvious insight: **Timeouts should get stricter as you move toward the user**. At Netflix, frontend services might have 1-second timeouts, while internal microservices use tighter 200-300ms timeouts, creating a "deadline funnel" that prevents resources being wasted on requests that will ultimately time out anyway.

Retries: Second Chances, Carefully Managed

When a request fails, retrying seems obvious—but doing it correctly is surprisingly complex.

What they are: Retries are deliberate reattempts of failed operations, based on the assumption that many failures are transient.

Why they matter: Well-crafted retry policies can increase system availability by smoothing over temporary network blips, server overloads, or deployment-related disruptions.

How they work: Effective retry systems incorporate:

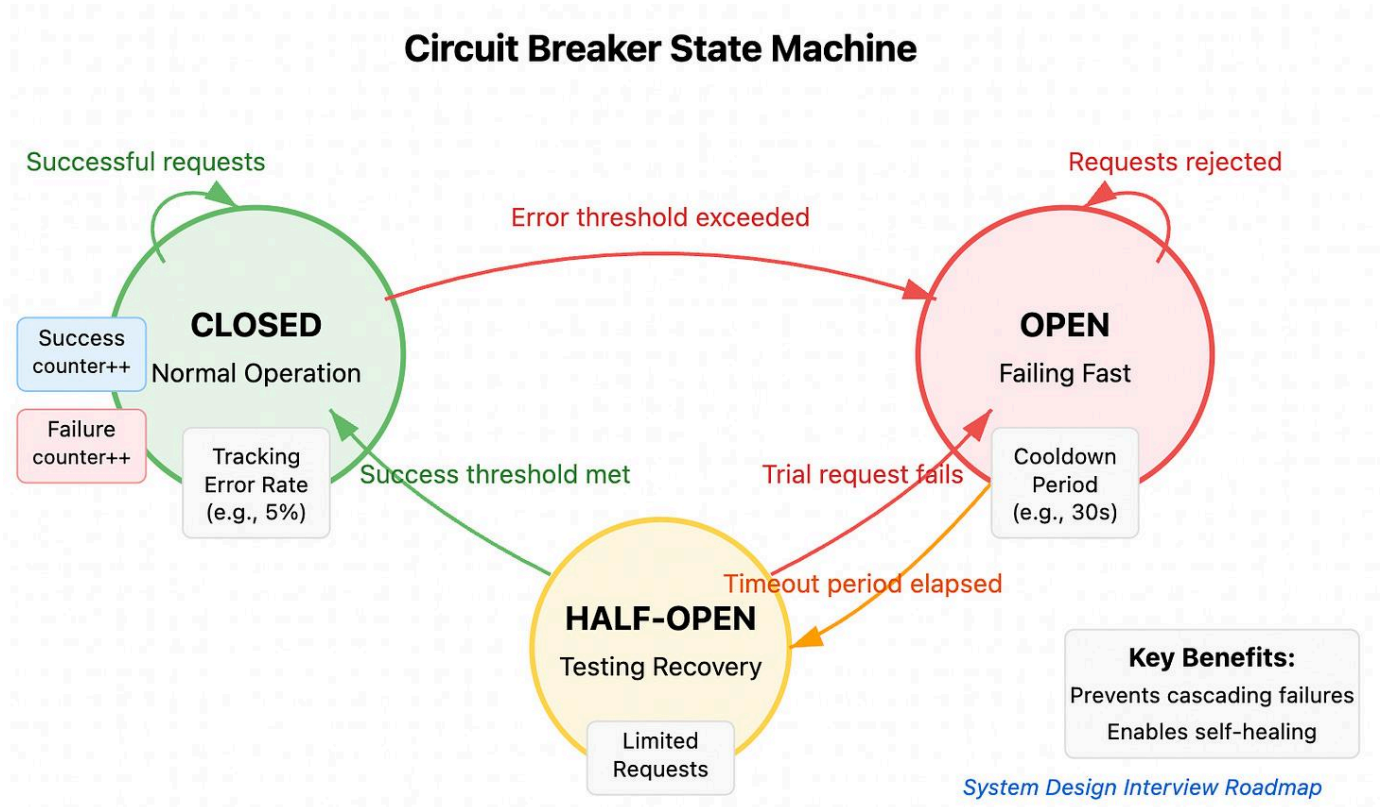
- **Backoff algorithms** that increase delay between attempts (preventing retry storms)
- **Jitter** (randomized delay variation) to prevent synchronized retries
- **Idempotency guarantees** to ensure operations can be safely repeated

When to use them: Retries work best for idempotent operations (GET, PUT) with transient failures. For non-idempotent operations (POST), implement idempotency tokens or be extremely careful.

The non-obvious insight: **Not all errors deserve retries**. Authentication failures, validation errors, or "resource not found" responses won't resolve with retries and only waste resources. At Stripe, they classify errors into "retriable" and "non-retriable" categories and track these metrics separately—allowing precise tuning of retry policies.

Circuit Breakers: Protecting Systems Under Stress

While timeouts and retries handle individual requests, circuit breakers manage system-wide behaviour during periods of stress.



What they are: Circuit breakers monitor failure rates and temporarily block requests when thresholds are exceeded—just like electrical circuit breakers prevent overload by cutting power.

Why they matter: They prevent your system from making calls to services that are likely to fail, preserving resources and enabling faster recovery.

How they work: Circuit breakers maintain three states:

1. **Closed:** Requests flow normally, failures are counted
2. **Open:** When failure threshold is exceeded, requests are immediately rejected
3. **Half-open:** After a cooldown period, a limited number of requests are allowed through to test if the dependency has recovered

When to use them: Implement circuit breakers in front of any dependency that could slow down or fail, especially those with varying performance characteristics.

The non-obvious insight: **Circuit breakers should be tuned differently based on dependency criticality**. For non-critical paths (e.g., recommendation services), use aggressive thresholds that trip quickly. For critical paths (authentication), use conservative settings that prefer degraded service over outright rejection.

Putting It All Together: The Resilience Chain

The real magic happens when these patterns work together. Here's a practical implementation in Go:

```go
func resilientCall(ctx context.Context, request Request) (Response, error) {
    // Circuit breaker check
    if !circuitBreaker.AllowRequest() {
        return fallbackResponse(), ErrCircuitOpen
    }

    var lastErr error
    // Retry loop with exponential backoff
    for attempt := 0; attempt < maxRetries; attempt++ {
        // Create context with timeout
        timeoutCtx, cancel := context.WithTimeout(ctx, calculateTimeout(attempt))
        defer cancel()

        resp, err := doRequest(timeoutCtx, request)
        if err == nil {
            circuitBreaker.RecordSuccess()
            return resp, nil
        }

        lastErr = err
        if !isRetriableError(err) {
            break
        }

        circuitBreaker.RecordFailure()
        // Exponential backoff with jitter
        sleepTime := baseDelay * math.Pow(2, float64(attempt)) + jitter()
        select {
        case <-time.After(sleepTime):
            continue
        case <-ctx.Done():
            return fallbackResponse(), ctx.Err()
        }
    }

    return fallbackResponse(), lastErr
}
```

Key Takeaway

These patterns aren't just about preventing failures—they're about controlling failure modes. In production systems, the question isn't if components will fail, but how your system responds when they do.

Next time you design a service, take a page from Amazon's playbook: design your failure handling first, then build your service around it. Your future self—likely debugging production at 3 AM—will thank you.