CDN (Content Delivery Network) Guide for System Design Interviews

Table of Contents

Introduction

A **Content Delivery Network (CDN)** is a geographically distributed network of servers that delivers content to users based on their geographic location, the origin of the content, and the content delivery server.

Why Use a CDN?

Key Benefits:

- **Reduced Latency**: Content served from nearest edge location
- **High Availability**: Redundancy across multiple servers
- **Reduced Bandwidth Costs**: Offload traffic from origin servers
- **Improved Security**: DDoS protection, WAF, SSL/TLS
- **Scalability**: Handle traffic spikes easily
- **Better User Experience**: Faster page loads

Common Use Cases

```
1. Static Content Delivery
   - Images, CSS, JavaScript
   - Videos, audio files
```

```
   - Downloadable files (PDFs, ZIPs)

2. Dynamic Content Acceleration
   - API responses
   - Personalized content
   - Real-time data

3. Video Streaming
   - Live streaming
   - Video on Demand (VOD)
   - Adaptive bitrate streaming

4. Software Distribution
   - Application updates
   - Game downloads
   - OS patches

5. Website Acceleration
   - Entire website delivery
   - Mobile app content
   - E-commerce platforms
```

CDN Fundamentals

Basic Concepts

1. Origin Server

```
- Source of original content
- Central server where content is stored
- CDN fetches content from origin
- Can be your application server, S3, etc.

Example:
Origin: https://origin.example.com
CDN: https://cdn.example.com
```

2. Edge Server (PoP - Point of Presence)

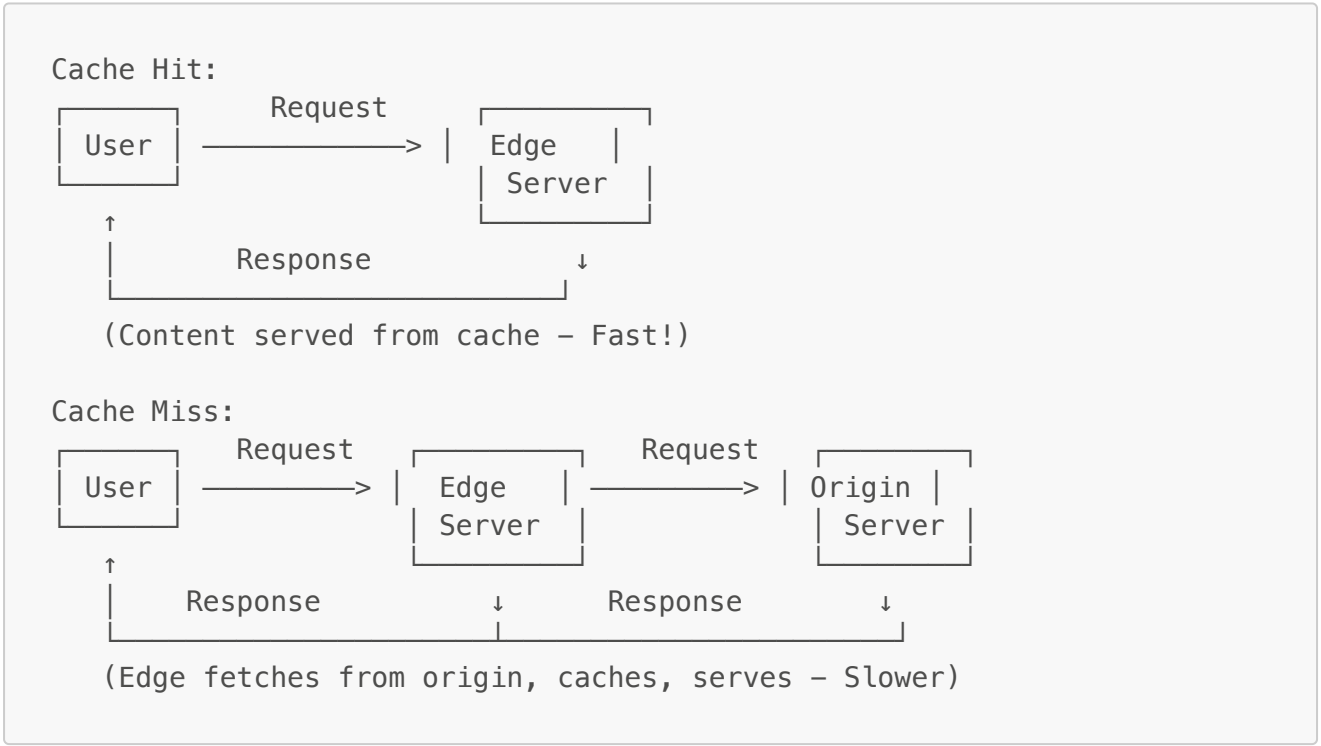
```
- Geographically distributed cache servers
- Store cached copies of content
- Serve content to nearby users
- Reduce latency and origin load

Global Distribution:
- North America: 50+ locations
```

```
- Europe: 40+ locations
- Asia Pacific: 60+ locations
- South America: 15+ locations
- Africa: 10+ locations
```

3. Edge Location

```
- Physical data center with edge servers
- Multiple edge servers per location
- Connected via high-speed networks
- Strategic placement near users

Example Cities:
- USA: New York, Los Angeles, Chicago, Miami
- Europe: London, Paris, Frankfurt, Amsterdam
- Asia: Tokyo, Singapore, Mumbai, Seoul
```

4. Cache Hit vs Cache Miss



CDN Metrics

```
Cache Hit Ratio (CHR):
CHR = (Cache Hits / Total Requests) × 100%

Good CHR: 85–95%
Excellent CHR: 95%+
```

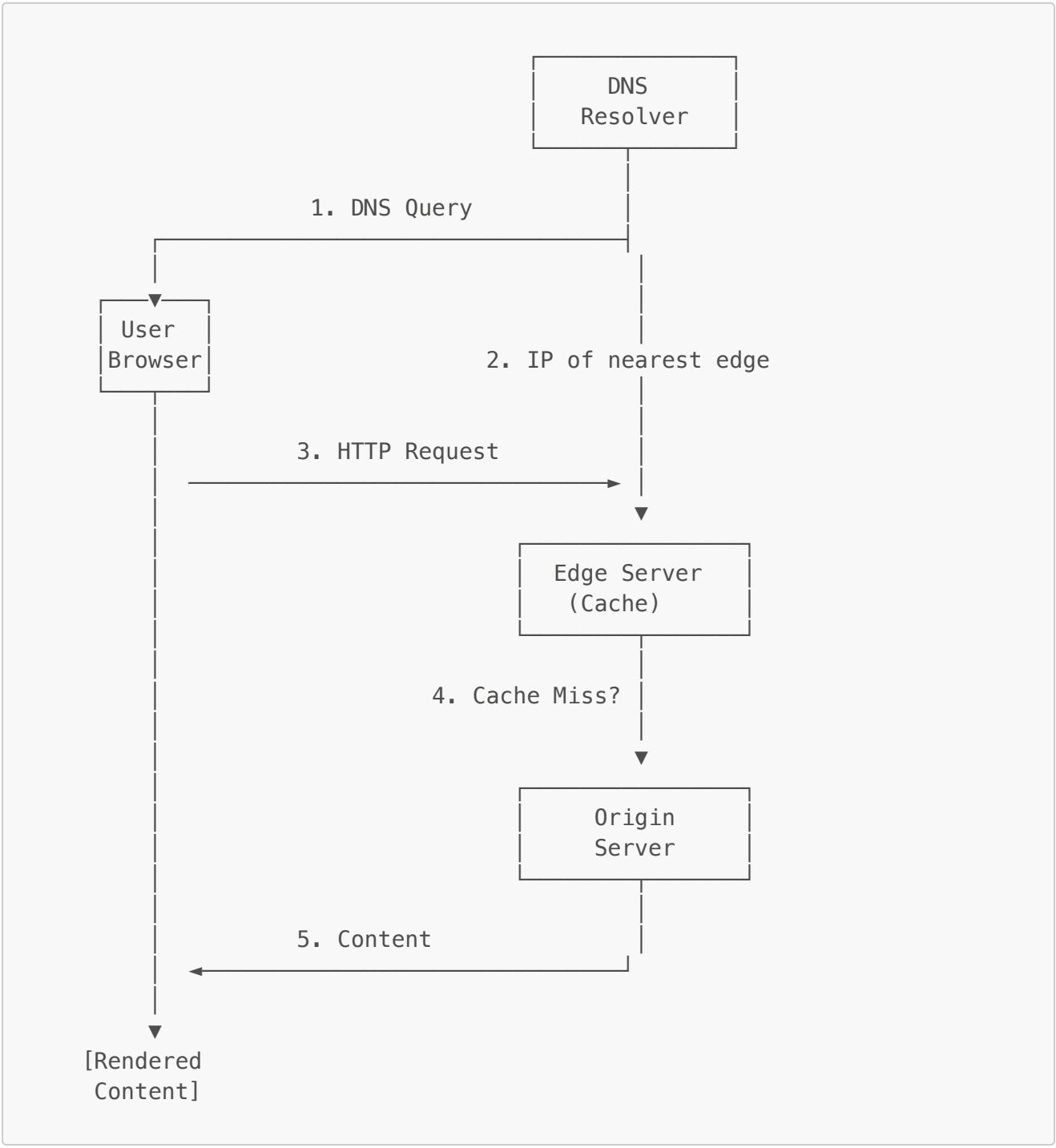
```
Time to First Byte (TTFB):
- Cache Hit: 10-50ms
- Cache Miss: 100-500ms

Bandwidth Savings:
Savings = (Origin Bandwidth - Actual Bandwidth) / Origin Bandwidth

Offload Rate:
Percentage of requests served by CDN vs origin
Target: 80-95% offload rate
```

How CDNs Work

Request Flow

```
Step-by-Step Process:

1. User Request
   User types: https://cdn.example.com/image.jpg

2. DNS Resolution
   - DNS returns IP of nearest edge server
   - Based on user's geographic location
   - Anycast or GeoDNS routing

3. Edge Server Check
   - Check if content exists in cache
   - Verify if cached content is still valid (not expired)

4a. Cache Hit Path
   - Content found and valid
   - Serve directly to user
   - Update access metrics
   - Total time: 10-50ms

4b. Cache Miss Path
   - Content not in cache or expired
   - Edge server requests from origin
   - Origin validates and sends content
   - Edge server caches content
   - Edge server serves to user
   - Total time: 100-500ms

5. Content Delivery
   - Content streamed to user
   - Connection maintained for additional resources
   - HTTP/2 or HTTP/3 for efficiency
```

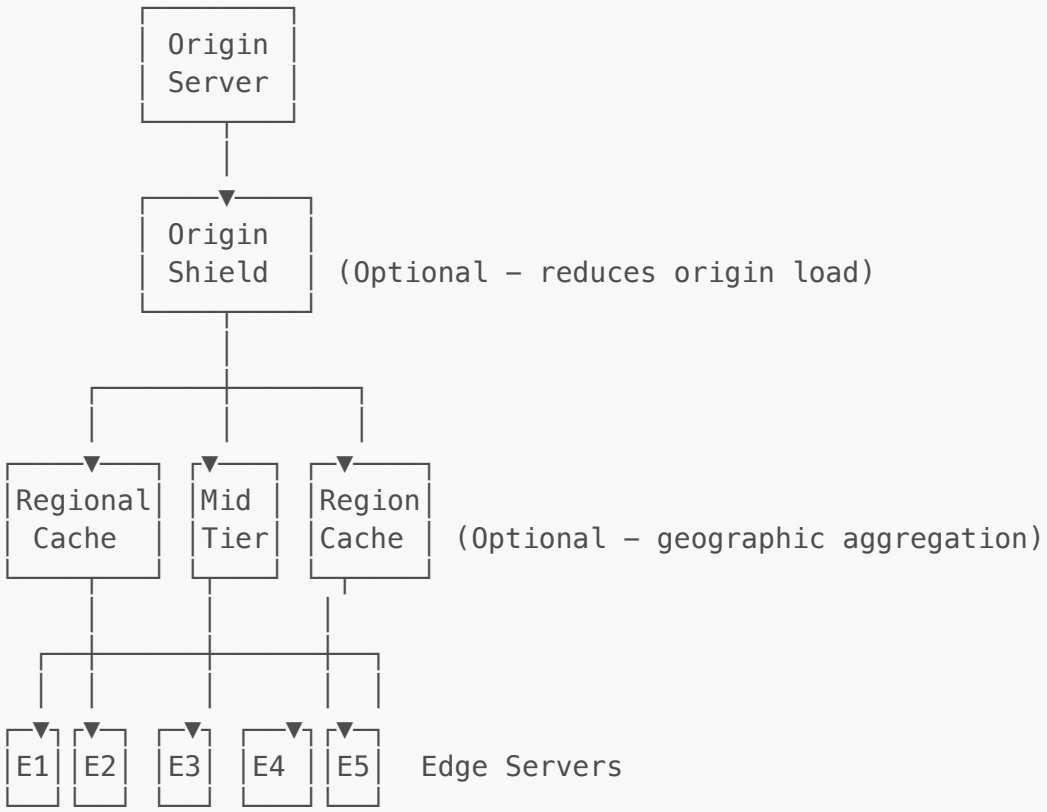
Detailed Flow Diagram



CDN Architecture Components

1. Edge Network

```
Components:
- Edge Servers: Front-line cache servers
- Mid-tier Caches: Regional caching layer (optional)
- Origin Shield: Protects origin from edge requests

Three-Tier Architecture:
```



2. Caching Layers

```
L1 Cache (Memory):
- RAM-based cache
- Fastest access (< 1ms)
- Limited capacity (GB range)
- Most frequently accessed content

L2 Cache (SSD):
- SSD-based cache
- Fast access (1-10ms)
- Medium capacity (TB range)
- Recently accessed content

L3 Cache (HDD):
- Disk-based cache
- Slower access (10-50ms)
- Large capacity (PB range)
- Less frequently accessed content

Cold Storage:
- Archive storage
- Very large capacity
- Rare access
- Cost-optimized
```

3. Routing Intelligence

```
1. DNS-based Routing (GeoDNS)
   - Maps user location to nearest edge
   - Returns IP of optimal edge server
   - Simple but less granular

2. Anycast Routing
   - Same IP announced from multiple locations
   - Network routes to nearest announcement
   - Automatic failover
   - Used by major CDNs

3. Dynamic Routing
   - Real-time performance metrics
   - Server health monitoring
   - Load-based decisions
   - Network congestion aware

Selection Criteria:
- Geographic proximity
- Server load
- Network latency
- Server health
- Content availability
```

4. Health Monitoring

```
Health Checks:
- HTTP/HTTPS probes
- TCP connection tests
- Custom application checks
- Frequency: Every 5-30 seconds

Metrics Tracked:
- Response time
- Error rates
- Cache hit ratio
- Bandwidth usage
- CPU/Memory utilization
- Disk I/O

Alerting:
- Failed health checks
- Performance degradation
- Capacity thresholds
- Security incidents

Example Health Check:
```

```
GET /health
Host: edge.example.com

Response:
{
  "status": "healthy",
  "latency_ms": 15,
  "cache_hit_ratio": 94.2,
  "cpu_usage": 45.3,
  "active_connections": 15234
}
```

Caching Strategies

1. Cache-Control Headers

```
Static Assets (1 year):
Cache-Control: public, max-age=31536000, immutable

Semi-static Content (1 hour):
Cache-Control: public, max-age=3600

Private User Data:
Cache-Control: private, max-age=300

Dynamic Content (revalidate):
Cache-Control: public, max-age=60, must-revalidate

Never Cache:
Cache-Control: no-store, no-cache
```

2. ETags and Validation

```
Initial Request:
GET /api/data
Host: example.com

Response:
HTTP/1.1 200 OK
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Cache-Control: public, max-age=3600
Content-Type: application/json

{data}

Subsequent Request (Validation):
GET /api/data
```

```
Host: example.com
If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"

Response (Not Modified):
HTTP/1.1 304 Not Modified
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
(no body - client uses cached version)
```

3. Tiered Caching Strategy

```
Content Type         | Edge Cache | Origin Shield | Origin
---------------------|------------|---------------|--------
Static Images        | 30 days    | 30 days       | ∞
CSS/JS               | 1 year     | 1 year        | ∞
HTML Pages           | 5 minutes  | 1 hour        | ∞
API Responses        | 1 minute   | 5 minutes     | Real-time
User-specific        | None       | None          | Real-time
Videos               | 7 days     | 30 days       | ∞
```

4. Cache Key Design

```
Default Cache Key:
URL only: https://cdn.example.com/image.jpg

Advanced Cache Key (including query params):
https://cdn.example.com/image.jpg?width=800&format=webp

Normalized Cache Key:
- Sort query parameters
- Lowercase keys
- Remove tracking parameters

Example:
Original: /image.jpg?utm_source=email&width=800&format=webp
Normalized: /image.jpg?format=webp&width=800

Custom Cache Key Components:
- URL path
- Query parameters (selected)
- Request headers (Accept, Accept-Language)
- Cookies (selected)
- Device type
- Geographic location
```

5. Stale-While-Revalidate

```
Cache-Control: max-age=3600, stale-while-revalidate=86400

Behavior:
1. Content fresh (< 1 hour): Serve from cache
2. Content stale (1 hour - 25 hours):
   - Serve stale content immediately
   - Fetch fresh content in background
   - Update cache asynchronously
3. Content too stale (> 25 hours): Block and fetch fresh

Benefits:
- Always fast responses
- Automatic cache refresh
- Zero-downtime updates
```

Content Types & Optimization

1. Static Content

```
Images:
- JPEG/PNG/GIF: Traditional formats
- WebP: Modern, better compression
- AVIF: Next-gen, best compression
- Optimization: Resize, compress, lazy load

Example Configuration:
images/
  - Cache: 1 year
  - Compression: Brotli/Gzip
  - Formats: Auto-negotiate (WebP/AVIF)
  - Resize: On-demand image resizing
  - Lazy loading: Native browser support

CSS/JavaScript:
- Minification: Remove whitespace, comments
- Bundling: Combine multiple files
- Code splitting: Load only needed code
- Tree shaking: Remove unused code
- Compression: Brotli (better) or Gzip

Example:
Before: styles.css (150 KB)
After:  styles.min.css.br (35 KB) - 77% reduction
```

2. Dynamic Content

```
API Responses:
- Short TTL (30-60 seconds)
- Vary by query parameters
- User-specific: Don't cache or use private cache
- Compression: Always enable

Personalization:
- Edge-side includes (ESI)
- Cookie-based variations
- A/B testing variants
- Geographic variations

Example ESI:
<html>
  <body>
    <!-- Cached common content -->
    <header>Site Header</header>

    <!-- Dynamic user-specific content -->
    <esi:include src="/api/user-profile" />

    <!-- Cached common content -->
    <footer>Site Footer</footer>
  </body>
</html>
```

3. Video Streaming

```
Adaptive Bitrate Streaming (ABR):

Manifest File (.m3u8):
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=800000,RESOLUTION=640x360
360p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1400000,RESOLUTION=842x480
480p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2800000,RESOLUTION=1280x720
720p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=5000000,RESOLUTION=1920x1080
1080p.m3u8

Segment Files:
- 2-10 second chunks
- Multiple quality levels
- Client adapts based on bandwidth
- Seamless quality switching

CDN Optimization:
- Segment caching: 7-30 days
```

```
- Manifest caching: 5-60 seconds
- Range request support
- Prefetching next segments
```

4. Compression

```
Compression Algorithms:

Brotli (Recommended):
- Best compression ratio
- 20-30% better than Gzip
- Supported by modern browsers
- CPU intensive

Gzip:
- Universal support
- Good compression ratio
- Lower CPU usage
- Fallback option

File Size Comparison (HTML):
Original:   100 KB
Gzip:       25 KB  (75% reduction)
Brotli:     20 KB  (80% reduction)

When to Compress:
✅ Text files (HTML, CSS, JS, JSON, XML)
✅ SVG images
✅ Fonts (WOFF, TTF)
❌ Already compressed (JPEG, PNG, WebP, MP4)
❌ Very small files (< 1KB)
```

CDN Security Features

1. DDoS Protection

```
Layers of Protection:

1. Network Layer (L3/L4)
   - SYN flood protection
   - UDP amplification mitigation
   - IP reputation filtering
   - Rate limiting by IP

2. Application Layer (L7)
   - HTTP flood protection
   - Slowloris attack mitigation
```

```
   - Request rate limiting
   - Challenge pages (CAPTCHA)

Example Rate Limiting:
- Limit: 100 requests/second per IP
- Burst: 200 requests
- Action: Challenge, block, or throttle

Mitigation Strategies:
- Traffic scrubbing centers
- Anycast distribution
- Automatic scaling
- Always-on protection
```

2. Web Application Firewall (WAF)

```
Protection Against:
- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Remote Code Execution
- Path Traversal
- Command Injection

Rule Sets:
1. OWASP Core Rule Set
2. Custom application rules
3. Known CVE protections
4. Rate limiting rules
5. Bot management

Example WAF Rule:
Rule: Block SQL Injection
Pattern: /(union|select|insert|update|delete|drop).*from/i
Action: Block
Log: Yes

Response:
HTTP/1.1 403 Forbidden
X-WAF-Block-Reason: SQL Injection Attempt
```

3. SSL/TLS Termination

```
Benefits:
- Encrypted data in transit
- Certificate management
- Protocol optimization (TLS 1.3)
- Reduced origin load
```

```
Configuration:
- Automatic certificate provisioning
- Let's Encrypt integration
- Custom certificate upload
- Wildcard certificates support

SSL Offloading:
Client —(HTTPS)—> CDN Edge —(HTTP/HTTPS)—> Origin

Or Full SSL:
Client —(HTTPS)—> CDN Edge —(HTTPS)—> Origin

TLS 1.3 Benefits:
- Faster handshake (1-RTT)
- Better security
- 0-RTT resumption
- Forward secrecy
```

4. Token Authentication

```
Signed URLs (Time-limited):
https://cdn.example.com/video.mp4?
  expires=1642118400&
  signature=abc123def456

Generation (Server-side):
secret = "your-secret-key"
expires = current_time + 3600  # 1 hour
data = url + expires
signature = HMAC-SHA256(data, secret)

Validation (CDN):
1. Check if current_time < expires
2. Recompute signature
3. Compare signatures
4. Allow/Deny access

Secure Tokens:
- IP-based restrictions
- User agent validation
- Geographic restrictions
- One-time use tokens
```

5. Bot Management

```
Detection Methods:
- Behavioral analysis
```

```
- Browser fingerprinting
- Challenge-response tests
- Machine learning models
- Known bot signatures

Actions:
1. Allow: Good bots (Googlebot, Bingbot)
2. Challenge: Suspicious traffic (CAPTCHA)
3. Block: Known bad bots
4. Monitor: Collect data
5. Rate limit: Slow down suspicious IPs

Good Bots:
- Search engine crawlers
- Monitoring services
- Feed readers
- Verified partners

Bad Bots:
- Scrapers
- Credential stuffers
- Spam bots
- Click fraud bots
```

Performance Optimization

1. Connection Optimization

```
HTTP/2 Features:
- Multiplexing: Multiple requests over one connection
- Header compression: HPACK algorithm
- Server push: Proactive resource delivery
- Binary protocol: More efficient parsing

HTTP/3 (QUIC) Features:
- UDP-based: Faster connection establishment
- 0-RTT: Resume connections instantly
- Built-in encryption: TLS 1.3
- Better packet loss handling

TCP Optimization:
- TCP Fast Open (TFO)
- BBR congestion control
- Increased initial window size
- TCP keepalive tuning

Connection Pooling:
- Persistent connections
- Connection reuse
```

```
- Optimal pool sizing
- Health monitoring
```

2. Image Optimization

```
Techniques:

1. Format Selection
   - JPEG: Photos, complex images
   - PNG: Graphics, transparency
   - WebP: Modern browsers, better compression
   - AVIF: Next-gen, best compression
   - SVG: Logos, icons, simple graphics

2. Responsive Images
   <img
     src="image-800w.jpg"
     srcset="
       image-400w.jpg 400w,
       image-800w.jpg 800w,
       image-1200w.jpg 1200w
     "
     sizes="(max-width: 600px) 400px, 800px"
     alt="Description"
   />

3. Lazy Loading
   <img
     src="placeholder.jpg"
     data-src="actual-image.jpg"
     loading="lazy"
     alt="Description"
   />

4. Image CDN Features
   - On-the-fly resizing
   - Format conversion
   - Quality optimization
   - Watermarking
   - Smart cropping

URL Parameters:
/image.jpg?w=800&h=600&fit=crop&format=webp&quality=85
```

3. Minification & Bundling

```
JavaScript:
Before (main.js - 150KB):
```

```
function calculateTotal(items) {
  let total = 0;
  for (let i = 0; i < items.length; i++) {
    total += items[i].price * items[i].quantity;
  }
  return total;
}

After (main.min.js - 45KB):
function calculateTotal(t){let e=0;for(let l=0;l<t.length;l++)
e+=t[l].price*t[l].quantity;return e}

CSS:
Before (styles.css - 80KB):
.button {
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
}

After (styles.min.css - 25KB):
.button{background-color:#007bff;color:#fff;padding:10px 20px;
border:none;border-radius:4px}

Bundling Strategy:
- Critical CSS: Inline in <head>
- Main bundle: Core functionality
- Vendor bundle: Third-party libraries
- Route-based chunks: Lazy load per route
- Common chunk: Shared code
```

4. Prefetching & Preloading

```
<!-- DNS Prefetch: Resolve domain early -->
<link rel="dns-prefetch" href="//cdn.example.com">

<!-- Preconnect: Establish early connection -->
<link rel="preconnect" href="https://cdn.example.com">

<!-- Prefetch: Load resource for next page -->
<link rel="prefetch" href="/next-page.html">

<!-- Preload: High-priority current page resource -->
<link rel="preload" href="/critical.css" as="style">
<link rel="preload" href="/hero-image.jpg" as="image">

<!-- Module Preload: JavaScript modules -->
<link rel="modulepreload" href="/app.js">
```

```
Strategy:
1. Preload critical resources (above-the-fold)
2. Prefetch likely next pages
3. Preconnect to critical domains
4. DNS prefetch for all domains
```

CDN Providers Comparison

Major CDN Providers

Provider	PoPs	Strength	Pricing	Use Case
Cloudflare	300+	Security DDoS	Free tier Low cost	All-in-one Security
Akamai	4,000+	Scale Global	Premium Custom	Enterprise Large scale
CloudFront	450+	AWS Integrate	Pay-as-go Low cost	AWS users Integration
Fastly	60+	Real-time VCL	Mid-high Custom	Dynamic content
Cloudinary	Global	Images Video Transform	Free tier Usage-based	Media management

Feature Comparison

Feature	Cloudflare	Akamai	CloudFront	Fastly
Free Tier	✅	❌	AWS Free	❌
DDoS Protection	✅ Always	✅	✅ Shield	✅
WAF	✅	✅	✅	✅
Image Optimization	✅	✅	❌	✅
Video Streaming	✅	✅	✅	✅
Edge Computing	Workers	EdgeWorkers	Lambda@Edge	Compute@Edge
Real-time Purge	< 1s	< 5s	< 5min	< 5s
HTTP/3 Support	✅	✅	✅	✅
Custom SSL	✅	✅	✅	✅
API	✅	✅	✅	✅

Cost Comparison (Example)

```
Traffic: 10TB/month, 100M requests

Cloudflare Pro:
- Base: $20/month
- Data transfer: Unlimited
- Requests: Unlimited
Total: $20/month

AWS CloudFront (us-east-1):
- First 10TB: $0.085/GB
- Requests: $0.0075/10,000
Total: ~$940/month

Akamai:
- Custom pricing
- Volume discounts
- Typical: $1,000-3,000/month

Fastly:
- Data transfer: $0.12/GB
- Requests: $0.0075/10,000
Total: ~$1,295/month
```

Push vs Pull CDN

Pull CDN (Origin Pull)

```
How it Works:
1. CDN doesn't have content initially
2. First user request triggers fetch from origin
3. CDN caches content
4. Subsequent requests served from cache

Flow:
User → CDN (cache miss) → Origin → CDN → User
User → CDN (cache hit) → User

Advantages:
✅ Automatic caching
✅ No manual uploads
✅ Always sync with origin
✅ Simple setup
✅ Dynamic content friendly

Disadvantages:
❌ First request is slow (cache miss)
```

```
❌ Origin must always be accessible
❌ Potential thundering herd problem

Best For:
- Frequently changing content
- Large content libraries
- Unpredictable access patterns
- Dynamic websites
```

Push CDN

```
How it Works:
1. You manually upload content to CDN
2. Content distributed to all edge servers
3. All requests served from cache
4. Update requires new upload

Flow:
You → CDN → Distributed to edges
User → CDN (always cache hit) → User

Advantages:
✅ No origin needed
✅ Always fast (no cold starts)
✅ Predictable performance
✅ Lower origin costs

Disadvantages:
❌ Manual content management
❌ Storage costs
❌ Stale content risk
❌ Complex updates

Best For:
- Static content
- Infrequently changing files
- Predictable access patterns
- Software distribution
- Media libraries
```

Hybrid Approach

```
Common Strategy:
- Static assets: Push (CSS, JS, images)
- Dynamic content: Pull (HTML, API responses)
- Videos: Push (large files, stable)
- User uploads: Pull (unpredictable)
```

```
Example Configuration:
/static/*     → Push CDN
/api/*        → Pull CDN (short TTL)
/media/*      → Push CDN
/user-content/* → Pull CDN
```

Cache Invalidation

Invalidation Methods

1. Time-based Expiration (TTL)

```
Cache-Control: max-age=3600

Automatic expiration after 1 hour
No manual intervention needed

Pros: Simple, automatic
Cons: May serve stale content
```

2. Purge/Invalidation

```
Single URL Purge:
POST /purge
{
  "url": "https://cdn.example.com/style.css"
}

Wildcard Purge:
POST /purge
{
  "pattern": "https://cdn.example.com/images/*"
}

Tag-based Purge:
POST /purge
{
  "tags": ["product-123", "category-electronics"]
}

Response:
{
  "status": "success",
  "purged_urls": 1523,
  "estimated_time": "5 seconds"
}
```

3. Versioned URLs

```
Strategy: Change URL when content changes

Version in Query String:
/style.css?v=1.2.3
/style.css?v=1.2.4  (new version)

Version in Path:
/v1.2.3/style.css
/v1.2.4/style.css  (new version)

Content Hash in Filename:
/style-a3f2b1.css
/style-d8e5c9.css  (new version)

Pros:
✅ No manual purge needed
✅ Instant updates
✅ No stale content
✅ Cache forever (immutable)

Cons:
❌ URL management complexity
❌ HTML updates required
```

4. Cache Tags

```
Tagging Content:
Cache-Tag: product-123, category-electronics, homepage

Purge by Tag:
POST /purge
{
  "tags": ["product-123"]
}

Benefits:
- Purge related content together
- Fine-grained invalidation
- Logical grouping
- Easy to manage

Example Use Case:
Product Update:
- Tag all product images: product-123
- Tag product page: product-123, category-electronics
```

```
- Tag category page: category-electronics
- Update product → purge tag "product-123"
```

Geographic Distribution

1. Global Edge Network

```
Major Regions:

North America:
- USA: 50+ locations (NYC, LA, Chicago, Dallas, Miami)
- Canada: 5+ locations (Toronto, Montreal, Vancouver)
- Mexico: 2+ locations (Mexico City, Guadalajara)

Europe:
- Western: London, Paris, Frankfurt, Amsterdam, Madrid
- Eastern: Warsaw, Prague, Bucharest
- Nordic: Stockholm, Helsinki, Copenhagen

Asia Pacific:
- East Asia: Tokyo, Seoul, Hong Kong, Taipei
- Southeast: Singapore, Bangkok, Jakarta, Manila
- South Asia: Mumbai, Bangalore, New Delhi
- Oceania: Sydney, Melbourne, Auckland

South America:
- Brazil: São Paulo, Rio de Janeiro
- Argentina: Buenos Aires
- Chile: Santiago

Africa:
- South Africa: Johannesburg, Cape Town
- Nigeria: Lagos
- Kenya: Nairobi

Middle East:
- UAE: Dubai
- Saudi Arabia: Riyadh
- Israel: Tel Aviv
```

2. Geographic Routing

```
Routing Strategies:

1. Latency-Based
   - Route to lowest latency edge
   - Real-time latency measurement
```

```
   - Dynamic adjustments

2. Geographic Proximity
   - Route to nearest location
   - Based on IP geolocation
   - Simple but effective

3. Load-Based
   - Consider server load
   - Avoid overloaded edges
   - Better distribution

4. Cost-Based
   - Factor in data transfer costs
   - Optimize for budget
   - Regional pricing differences

Example Decision:
User in Sydney:
1. Check latency to nearby edges
   - Sydney: 5ms
   - Melbourne: 12ms
   - Singapore: 45ms
2. Check server health
   - Sydney: 85% capacity
   - Melbourne: 40% capacity
3. Select Melbourne (good latency, lower load)
```

3. Regional Caching

```
Tiered Regional Strategy:

Global Cache Hierarchy:

         [Origin - US East]
                |
     ┌──────────┼──────────┐
     |          |          |
[Regional  [Regional  [Regional
 NA]        EU]        APAC]
     |          |          |
  ┌──┼──┐    ┌──┼──┐    ┌──┼──┐
  |  |  |    |  |  |    |  |  |
  E  E  E    E  E  E    E  E  E  (Edge Servers)

Benefits:
- Reduced origin load
- Better cache hit ratio
- Lower latency
- Regional compliance
```

```
Regional Policies:
- EU: GDPR compliance, data residency
- China: Great Firewall considerations
- Russia: Data localization laws
- Brazil: Local data requirements
```

Load Balancing

1. Load Balancing Algorithms

```
1. Round Robin
   - Distribute requests evenly
   - Simple, no state needed
   - May not consider server load

   Example:
   Request 1 → Server A
   Request 2 → Server B
   Request 3 → Server C
   Request 4 → Server A (repeat)

2. Least Connections
   - Route to server with fewest active connections
   - Better for long-lived connections
   - Requires state tracking

   Example:
   Server A: 50 connections
   Server B: 35 connections ← Route here
   Server C: 60 connections

3. Weighted Round Robin
   - Servers have different capacities
   - Weight based on capacity
   - More powerful servers get more traffic

   Example:
   Server A: Weight 3 (3 requests)
   Server B: Weight 2 (2 requests)
   Server C: Weight 1 (1 request)

4. IP Hash
   - Hash client IP to server
   - Same client → same server
   - Maintains session affinity

   Hash(ClientIP) % ServerCount = Server
```

```
5. Least Response Time
   - Route to fastest responding server
   - Dynamic performance-based
   - Best user experience

   Example:
   Server A: 50ms average
   Server B: 30ms average ← Route here
   Server C: 45ms average
```

2. Health Checks

```
Types of Health Checks:

1. Passive Health Checks
   - Monitor real traffic
   - Detect failures from actual requests
   - No overhead

   Criteria:
   - 5xx errors > threshold
   - Response time > threshold
   - Connection failures

2. Active Health Checks
   - Periodic probes
   - Synthetic monitoring
   - Proactive detection

   Example:
   GET /health HTTP/1.1
   Host: edge.example.com
   User-Agent: LoadBalancer/1.0

   Expected Response:
   HTTP/1.1 200 OK
   Content-Type: application/json

   {
     "status": "healthy",
     "version": "1.2.3",
     "uptime": 3600,
     "load": 0.45
   }

Configuration:
- Interval: 5-30 seconds
- Timeout: 2-5 seconds
- Healthy threshold: 2 consecutive successes
- Unhealthy threshold: 3 consecutive failures
```

3. Failover Strategies

```
1. Automatic Failover
   - Detect failure
   - Route to healthy server
   - Transparent to users

   Process:
   1. Health check fails
   2. Mark server unhealthy
   3. Remove from rotation
   4. Redistribute traffic
   5. Monitor recovery
   6. Re-add when healthy

2. Geographic Failover
   - Primary region fails
   - Route to secondary region
   - May increase latency

   Example:
   Primary: US East
   Secondary: US West
   Tertiary: Europe

3. Graceful Degradation
   - Serve cached content
   - Reduce feature set
   - Maintain core functionality

   Fallback Strategy:
   1. Try primary edge
   2. Try regional cache
   3. Serve stale content
   4. Show error page
```

Common Interview Scenarios

1. Video Streaming Platform (Netflix-like)

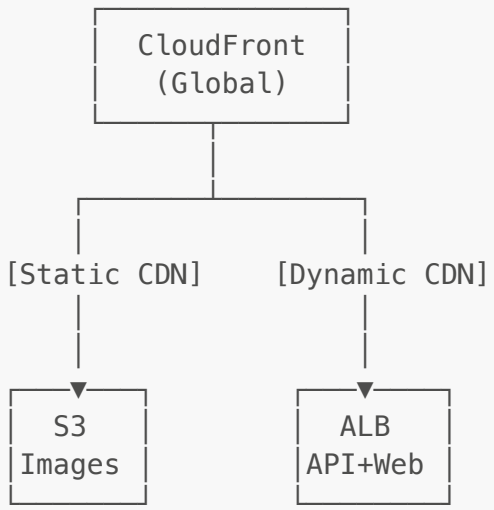
```
Requirements:
- 100M+ users globally
- 4K video streaming
- Adaptive bitrate
- Low latency (< 100ms start)
- High availability (99.99%)
```

```
CDN Design:

Origin Servers:
- Multi-region (US, EU, APAC)
- Store master files
- Transcode videos
- Generate manifests

Edge Strategy:
- 200+ PoPs globally
- 50TB+ cache per PoP
- L1: SSD (hot content)
- L2: HDD (warm content)
- L3: Cold storage (archives)

Content Delivery:
1. User requests video
2. DNS routes to nearest edge
3. Edge checks cache
4. If miss, fetch from regional cache
5. If still miss, fetch from origin
6. Stream to user with ABR

Caching Policy:
- Popular videos: 30 days
- Manifest files: 60 seconds
- Thumbnails: 7 days
- Recommendations: 5 minutes

Optimization:
- Pre-populate popular content
- Prefetch next segments
- Use HTTP/3 for better performance
- Implement smart CDN routing
- Content-aware encoding

Cost Estimation:
- 10PB/month data transfer
- $0.02/GB average cost
- Total: $200,000/month
- With CDN: 95% cache hit ratio
- Actual origin: $10,000/month
```

2. E-commerce Platform (Amazon-like)

```
Requirements:
- Millions of product images
- Dynamic pricing
- Personalized recommendations
- Flash sales support
```

```
- Global presence

CDN Design:

Static Content:
- Product images: 1 year cache
- CSS/JS: Immutable, versioned URLs
- Category images: 30 days cache
- Icons/logos: 1 year cache

Dynamic Content:
- Product pages: 5 minutes cache
- Search results: 1 minute cache
- Cart: No cache (user-specific)
- Checkout: No cache (secure)

Architecture:
```




```
Image Optimization:
- WebP/AVIF format
- Responsive images
- Lazy loading
- On-the-fly resizing

Flash Sale Handling:
- Pre-warm cache
- Rate limiting
- Queue system
- Stale-while-revalidate

Security:
- WAF rules
- DDoS protection
- Bot management
- Signed URLs for downloads
```

3. Social Media Platform (Instagram-like)

```
Requirements:
- Billions of images/videos
- Real-time uploads
- Stories (24-hour content)
- Live streaming
- Global users

CDN Design:

Upload Flow:
User → Upload API → Origin → CDN
1. Upload to origin
2. Process (resize, compress)
3. Push to CDN
4. Return CDN URLs

Download Flow:
User → CDN → (Cache miss) → Origin

Content Types:

Profile Pictures:
- Cache: 30 days
- Formats: JPEG, WebP
- Sizes: 50x50, 150x150, 400x400

Feed Images:
- Cache: 7 days
- Formats: JPEG, WebP, AVIF
- Lazy load
- Progressive JPEG

Stories:
- Cache: 24 hours
- Auto-delete after expiry
- High compression
- Low latency

Videos:
- ABR streaming
- Cache: 30 days
- Segment-based (10s chunks)
- Prefetch next segments

Optimization:
- Image CDN with transforms
- Edge-side resizing
- Smart compression
- Geographic distribution
```

```
Architecture:

┌────────┐    Upload     ┌──────────┐
│ User │ ──────────> │ Origin │
└────────┘               └──────────┘
   ↑                          │
   │    View                  │ Push
   │                          ┘
   │    ┌──────────────────┐
   │    │
   │  ┌─▼──────────┐
   └──│    CDN     │
      │  (Global)  │
      └────────────┘

Cost Optimization:
- Aggressive caching
- Smart purging
- Tiered storage
- Regional optimization
```

4. News/Media Site

```
Requirements:
- Breaking news
- High traffic spikes
- Global audience
- Rich media content
- SEO optimization

CDN Design:

Content Strategy:

Articles:
- HTML: 2 minutes cache
- Update on publish
- Stale-while-revalidate
- Edge-side includes (ESI)

Images:
- Cache: 30 days
- Responsive images
- Lazy loading
- WebP/AVIF

Videos:
- Cache: 7 days
- ABR streaming
- Autoplay muted
```

```
- Poster images

Breaking News Handling:
1. Publish article
2. Purge cache immediately
3. Pre-warm popular edges
4. Monitor traffic spike
5. Auto-scale if needed

Traffic Spike Mitigation:
- Cache everything possible
- Use stale content during origin issues
- Queue system for comments
- CDN-level rate limiting

SEO Optimization:
- Separate cache for bots
- Server-side rendering
- Fast TTFB
- Preload critical resources

Performance Targets:
- TTFB: < 200ms
- LCP: < 2.5s
- CLS: < 0.1
- FID: < 100ms
```

Design Decisions & Trade-offs

1. Cache TTL Selection

```
Trade-offs:

Long TTL (Hours/Days):
Pros:
✅ High cache hit ratio
✅ Lower origin load
✅ Better performance
✅ Lower costs

Cons:
❌ Stale content risk
❌ Slower updates
❌ Requires purging
❌ Storage costs

Short TTL (Minutes):
Pros:
✅ Fresh content
```

```
✅ No purging needed
✅ Dynamic-friendly
✅ Easy to manage

Cons:
❌ Lower cache hit ratio
❌ Higher origin load
❌ More expensive
❌ Slower performance

Decision Matrix:

Content Type         | Recommended TTL | Reasoning
---------------------|-----------------|-----------------
Static assets        | 1 year          | Never change
API responses        | 1-5 minutes     | Frequently update
HTML pages           | 5-15 minutes    | Semi-dynamic
Product images       | 30 days         | Rarely change
User avatars         | 7 days          | Occasional updates
Videos               | 30 days         | Large, stable
News articles        | 2-5 minutes     | Time-sensitive
```

2. Push vs Pull

```
When to Use Push CDN:

✅ Software downloads
✅ Game assets
✅ Large video libraries
✅ Infrequently changing content
✅ Predictable demand

When to Use Pull CDN:

✅ Dynamic websites
✅ User-generated content
✅ Frequently updating content
✅ Unpredictable patterns
✅ Large content library

Hybrid Examples:

E-commerce:
- Product images: Push
- Product data: Pull
- User reviews: Pull
- Static assets: Push

Media Site:
- Historical videos: Push
```

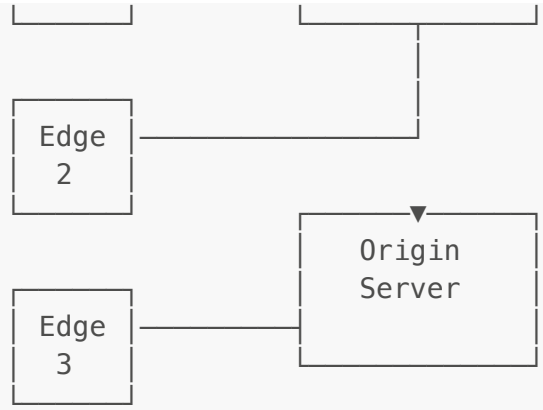
```
- Recent videos: Pull
- Articles: Pull
- Site assets: Push
```

3. Geographic Distribution

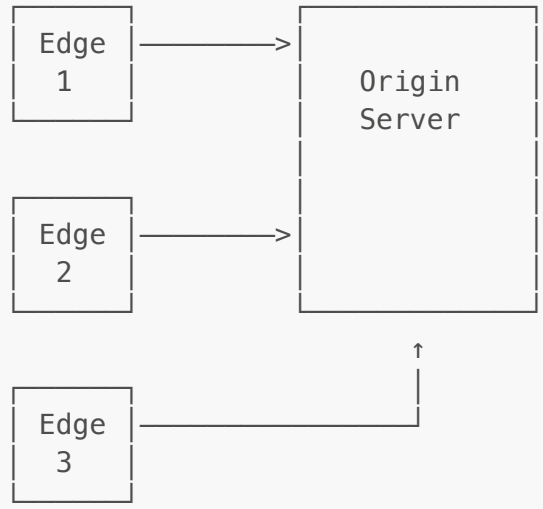
```
Considerations:

More PoPs:
Pros:
✅ Lower latency
✅ Better user experience
✅ Higher availability
✅ Better DDoS mitigation

Cons:
❌ Higher costs
❌ Complex management
❌ Lower cache hit ratio per PoP
❌ Sync overhead

Fewer PoPs:
Pros:
✅ Lower costs
✅ Higher cache efficiency
✅ Simpler management
✅ Better cache hit ratio

Cons:
❌ Higher latency for some users
❌ Potential bottlenecks
❌ Less redundancy
❌ Worse DDoS protection

Decision Factors:
- User distribution
- Latency requirements
- Budget constraints
- Compliance needs
- Traffic patterns
```

4. Origin Shield

```
With Origin Shield:

┌────────┐          ┌─────────────┐
│  Edge  │─────────>│    Origin   │
│   1    │          │    Shield   │
```




```
Benefits:
✅ Reduced origin load (1 request vs N)
✅ Better cache consolidation
✅ Lower costs
✅ Fewer origin requests

Without Origin Shield:
```




```
Trade-off:
- Extra hop (latency)
- Additional cost
- vs. Lower origin load
- Better efficiency
```

Interview Tips & Best Practices

Interview Framework

```
1. Clarify Requirements (5 minutes)
   Questions to Ask:
   - What type of content? (static/dynamic)
   - Expected scale? (users, traffic, geography)
```

```
   - Performance requirements? (latency, availability)
   - Security needs? (DDoS, WAF, authentication)
   - Budget constraints?
   - Compliance requirements?

2. High-Level Design (10 minutes)
   Components:
   - Origin servers
   - CDN provider selection
   - Edge locations
   - Caching strategy
   - Security measures

3. Deep Dive (15 minutes)
   - Cache invalidation strategy
   - Handling traffic spikes
   - Monitoring and alerting
   - Cost optimization
   - Failover strategy

4. Trade-offs Discussion (5 minutes)
   - Push vs Pull
   - TTL selection
   - Geographic distribution
   - Cost vs Performance
```

Common Interview Questions

```
Q: How would you handle a flash sale with 1M concurrent users?

A: Multi-layered approach:
1. Pre-warm cache for product pages
2. Queue system for checkout
3. Rate limiting per user
4. Separate CDN for static assets
5. Database read replicas
6. Auto-scaling backend
7. Circuit breakers
8. Monitoring and alerts

Q: How do you ensure content freshness with aggressive caching?

A: Multiple strategies:
1. Appropriate TTL per content type
2. Cache invalidation API
3. Versioned URLs (immutable content)
4. Stale-while-revalidate
5. Cache tags for related content
6. Real-time purge for critical updates
7. Monitoring stale content metrics
```

```
Q: How would you reduce CDN costs?

A: Cost optimization techniques:
1. Increase cache hit ratio
2. Optimize content (compression, formats)
3. Smart TTL configuration
4. Origin shield to reduce origin requests
5. Regional routing to lower-cost regions
6. Remove unused content
7. Negotiate volume discounts
8. Use tiered storage
```

Key Talking Points

```
1. Scalability
   - Horizontal scaling with more PoPs
   - Auto-scaling origin servers
   - Load balancing across edges
   - Database optimization

2. Performance
   - Edge caching reduces latency
   - HTTP/3 for better connection
   - Image optimization
   - Compression (Brotli/Gzip)
   - Prefetching/Preloading

3. Reliability
   - Multiple PoPs for redundancy
   - Health checks and failover
   - Origin shield
   - Graceful degradation
   - Monitoring and alerts

4. Security
   - DDoS protection at edge
   - WAF rules
   - SSL/TLS termination
   - Token authentication
   - Bot management
   - Rate limiting

5. Cost
   - High cache hit ratio
   - Origin shield
   - Content optimization
   - Regional routing
   - Tiered storage
```

Common Mistakes to Avoid

```
❌ Don't:
1. Ignore cache invalidation strategy
2. Over-cache dynamic content
3. Under-cache static content
4. Forget about security (DDoS, WAF)
5. Ignore geographic distribution
6. Neglect monitoring and alerts
7. Overlook cost optimization
8. Skip error handling
9. Forget about origin protection
10. Ignore compliance requirements

✅ Do:
1. Design cache strategy for each content type
2. Implement proper invalidation
3. Consider security from the start
4. Plan for traffic spikes
5. Monitor cache metrics
6. Optimize costs
7. Use appropriate TTLs
8. Implement health checks
9. Have failover strategy
10. Consider user experience
```

Quick Reference Checklist

CDN Design Checklist

- [ ] Identify content types and access patterns
- [ ] Select appropriate CDN provider
- [ ] Design caching strategy with TTLs
- [ ] Plan cache invalidation approach
- [ ] Configure geographic distribution
- [ ] Implement security measures (DDoS, WAF, SSL)
- [ ] Set up health monitoring and alerts
- [ ] Configure load balancing
- [ ] Implement failover strategy
- [ ] Optimize for cost
- [ ] Plan for traffic spikes
- [ ] Set up logging and analytics
- [ ] Configure origin shield (if needed)
- [ ] Implement proper error handling
- [ ] Plan for compliance requirements
- [ ] Test performance and latency

- [ ] Document configuration
- [ ] Set up automated deployments

Additional Resources

Key Concepts to Master

```
1. Caching Fundamentals
   - HTTP caching headers
   - Cache invalidation strategies
   - Cache key design
   - TTL selection

2. Network Protocols
   - HTTP/1.1, HTTP/2, HTTP/3
   - TLS/SSL
   - TCP vs UDP (QUIC)
   - DNS and routing

3. Performance Metrics
   - Cache hit ratio
   - TTFB (Time to First Byte)
   - Latency percentiles (p50, p95, p99)
   - Bandwidth utilization

4. Security
   - DDoS mitigation
   - WAF rules
   - Token authentication
   - Bot management

5. Cost Optimization
   - Cache efficiency
   - Origin protection
   - Geographic routing
   - Content optimization
```

Learning Resources

```
Books:
- "High Performance Browser Networking" by Ilya Grigorik
- "Web Performance in Action" by Jeremy Wagner
- "Designing Data-Intensive Applications" by Martin Kleppmann

Online:
- Cloudflare Learning Center
- AWS CloudFront Documentation
- Akamai Developer Resources
```

```
- CDN Planet (comparisons and reviews)

Tools:
- WebPageTest (performance testing)
- Chrome DevTools (network analysis)
- curl/HTTPie (API testing)
- Grafana (monitoring)
```

Conclusion

CDN design is crucial for modern web applications. During system design interviews, remember to:

1. **Understand requirements** - Ask about scale, geography, content types
2. **Choose appropriate strategy** - Push vs Pull, TTLs, invalidation
3. **Consider security** - DDoS, WAF, authentication
4. **Optimize performance** - Caching, compression, HTTP/3
5. **Plan for costs** - Cache efficiency, origin protection
6. **Design for reliability** - Health checks, failover, monitoring
7. **Think globally** - Geographic distribution, compliance

Key principles:

- Cache aggressively but intelligently
- Protect your origin
- Monitor everything
- Plan for failures
- Optimize costs
- Prioritize user experience

Remember, there's no one-size-fits-all solution. The best CDN design depends on your specific requirements, constraints, and trade-offs you're willing to make.

Good luck with your interviews!