Global File Storage & Sharing System - High-Level Design (HLD)

Table of Contents

Problem Statement

Design a scalable, reliable, globally distributed file storage and sharing system like Google Drive or Dropbox.

Core Functionality

- Upload/download files of any size (up to 10GB)
- Share files with other users (permissions management)
- Access files from multiple devices
- Real-time synchronization across devices
- Search files by name and content
- Version history (last 30 days)

Target Scale

- **1 Billion** total users
- **200 Million** daily active users (DAU)
- **10 PB** total storage
- **99.99%** availability
- Global distribution

Requirements

Functional Requirements

1. **File Operations**: Upload, download, delete, update files
2. **Folder Management**: Create folders, organize hierarchy
3. **Sharing**: Share files/folders with users or via public links
4. **Synchronization**: Real-time sync across devices
5. **Search**: Find files by name, content, metadata
6. **Version Control**: Maintain and restore file versions
7. **Permissions**: Owner, editor, viewer access levels

Non-Functional Requirements

1. **Scalability**: Handle billions of users, petabytes of data
2. **Availability**: 99.99% uptime (4 nines)
3. **Consistency**: Strong for metadata, eventual for sync
4. **Performance**:
    - Upload/Download: < 100ms latency (small files)
    - Sync latency: < 5 seconds
5. **Durability**: 99.999999999% (11 nines)
6. **Security**: Encryption at rest and in transit, access control

---

Capacity Estimation

Scale Assumptions

```
Users:
- Total Users: 1 Billion
- Daily Active Users (DAU): 200 Million (20%)
- Average Storage per User: 10 GB

Files:
- Average File Size: 2 MB
- Files per User: 5,000
- Total Files: 5 Trillion files

Read/Write Ratio: 3:1 (more downloads than uploads)
```

Storage Requirements

```
Total Storage = 1B users × 10 GB = 10 PB
With 3x replication = 30 PB
```

Traffic Estimation

```
Upload Traffic:
- Daily uploads: 200M users × 5 files = 1B files/day
- Upload data: 1B × 2 MB = 2 PB/day
- Upload bandwidth: ~23 GB/sec

Download Traffic:
- 3x uploads = ~69 GB/sec

Total Peak Bandwidth: ~100 GB/sec
```

QPS (Queries Per Second)

```
Total Daily Requests:
- Uploads: 1B requests
- Downloads: 3B requests
- Metadata operations: 2B requests
- Total: 6B requests/day

Average QPS: ~70,000
Peak QPS (3x): ~200,000
```

Memory for Caching

```
Metadata Cache:
- 1 KB per file metadata
- Cache 20% hot files = 1T files
- Required cache: ~1 TB
```

Back-of-Envelope Calculations

Latency Numbers Every Programmer Should Know

```
L1 cache reference                      0.5 ns
Branch mispredict                       5   ns
L2 cache reference                      7   ns             14x L1 cache
Mutex lock/unlock                       25  ns
Main memory reference                   100 ns             20x L2, 200x L1
Compress 1K with Snappy               10,000 ns    10 µs
Send 1 KB over 1 Gbps network         10,000 ns    10 µs
Read 4 KB randomly from SSD          150,000 ns   150 µs   ~1GB/sec SSD
Read 1 MB sequentially from memory  250,000 ns   250 µs
Round trip within datacenter         500,000 ns   500 µs
Read 1 MB sequentially from SSD   1,000,000 ns  1000 µs   1 ms
Disk seek                        10,000,000 ns 10000 µs  10 ms
```

```
Read 1 MB sequentially from disk 30,000,000 ns 30000 µs  30 ms
Send packet US→Europe→US        150,000,000 ns   150 ms
```

Powers of Two Table

```
Power   Exact Value         Approx Value    Bytes
-------------------------------------------------------
10      1,024              ~1 thousand      1 KB
16      65,536             ~65 thousand     64 KB
20      1,048,576          ~1 million       1 MB
30      1,073,741,824      ~1 billion       1 GB
40      1,099,511,627,776  ~1 trillion      1 TB
50      ~1 quadrillion                      1 PB
```

Server Capacity Estimation

```
Given:
- Upload bandwidth needed: 23 GB/sec
- Assume each server handles: 1 GB/sec network throughput
- Peak load factor: 3x average

Calculation:
- Average servers = 23 GB/sec ÷ 1 GB/sec = 23 servers
- Peak servers = 23 × 3 = 69 servers
- With redundancy (N+1) = ~70-80 servers

For global deployment:
- 3 regions × 80 servers = 240 servers total
```

Database Sizing

```
Metadata per file: ~1 KB
- file_id: 8 bytes
- owner_id: 8 bytes
- name: 255 bytes
- size: 8 bytes
- timestamps: 24 bytes
- path: 500 bytes
- other fields: ~200 bytes
Total: ~1 KB

For 5 Trillion files:
- Storage needed = 5T × 1 KB = 5 PB metadata
- With indexing (2x) = 10 PB
- Sharded across 1000 DB servers = 10 GB per server
```

Network Bandwidth Analysis

```
Upload Scenario:
- 200M users upload 5 files/day
- Average file size: 2 MB
- Total data: 200M × 5 × 2 MB = 2 PB/day
- Per second: 2 PB / 86,400 sec ≈ 23 GB/sec
- With peaks (3x): ~70 GB/sec

Download Scenario (3:1 read/write ratio):
- Download traffic: 23 × 3 = 69 GB/sec
- With peaks: ~210 GB/sec

Total bandwidth requirement:
- Average: ~100 GB/sec
- Peak: ~300 GB/sec
```

Cost Estimation (Rough)

```
Storage Costs (AWS S3):
- 10 PB × $23/TB/month = $230,000/month
- With replication (3x): ~$700,000/month

Bandwidth Costs:
- 2 PB upload/day × 30 days = 60 PB/month
- 6 PB download/day × 30 days = 180 PB/month
- At $0.09/GB: ~$21M/month for bandwidth

Database:
- 1000 instances × $500/month = $500,000/month

Total Monthly Cost: ~$22M/month
Annual Cost: ~$260M/year

Per User Cost: $260M ÷ 1B users = $0.26/user/year
```

Availability Calculation

Target: 99.99% (Four 9s)

```
Downtime allowed per year:
- 365 days × 24 hours × 60 min × (1 - 0.9999)
- = 52.56 minutes/year
- ≈ 4.38 minutes/month
```

```
Component availability in sequence:
- Load Balancer: 99.99%
- Application: 99.99%
- Database: 99.99%
- Object Storage: 99.99%

Total = 0.9999^4 = 99.96%

To achieve 99.99%:
- Need redundancy (active-active, failover)
- Multiple availability zones
- Health checks and auto-recovery
```

High-Level Architecture

```
                         ┌──────────────┐
                         │    Users     │
                         │ (Web/Mobile) │
                         └──────┬───────┘
                                │
                 ┌──────────────┼──────────────┐
                 │              │              │
         ┌───────▼──────┐  ┌────▼─────┐  ┌─────▼──────┐
         │     CDN      │  │   DNS    │  │    CDN     │
         │ (Downloads)  │  │ Route 53 │  │  (Static)  │
         └──────────────┘  └────┬─────┘  └────────────┘
                                │
                 ┌──────────────┴──────────────┐
                 │                             │
         ┌───────▼──────┐              ┌───────▼──────┐
         │Load Balancer │              │Load Balancer │
         │  (Layer 7)   │              │  (Layer 7)   │
         └───────┬──────┘              └───────┬──────┘
                 │                             │
     ┌───────────┴──────────┐          ┌───────┴─────────┐
     │                      │          │                 │
┌────▼─────┐         ┌──────▼────┐ ┌───▼─────┐    ┌──────▼────┐
│   API    │         │  Upload   │ │  Sync   │    │ Download  │
│ Gateway  │         │  Service  │ │ Service │    │  Service  │
└────┬─────┘         └──────┬────┘ └────┬────┘    └──────┬────┘
     │                      │           │                │
┌────┴─────────┬────────┬───┴───────────┴──────┬─────────┴──┐
│              │        │                      │            │
┌▼────┐  ┌─────▼┐  ┌────▼───┐  ┌────────┐  ┌───▼─────┐  ┌────▼─────┐
│Auth │  │Meta  │  │Share   │  │Search  │  │  Notif  │  │Analytics │
│ Svc │  │data  │  │ Svc    │  │Service │  │ Service │  │ Service  │
│     │  │ Svc  │  │        │  │ (ES)   │  │         │  │          │
└─────┘  └──┬───┘  └────────┘  └────────┘  └─────────┘  └──────────┘
```

```
                 │
        ┌────────┴──────────┐
        │                   │
   ┌────▼───┐        ┌──────▼─────┐
   │Primary │        │Read Replicas│
   │  DB    │◄──────►│ (Postgres) │
   │(Write) │        │  (Reads)   │
   └────┬───┘        └────────────┘
        │
   ┌────▼──────────┐
   │Redis Cluster  │
   │   (Cache)     │
   └───────────────┘

   ┌──────────────────────────────────────────┐
   │         Supporting Infrastructure         │
   ├────────────┬──────────────┬──────────────┤
   │   Object   │   Message    │ Async Workers│
   │  Storage   │    Queue     │ - Thumbnails │
   │  (S3/GCS)  │(Kafka/SQS)   │ - Virus Scan │
   │            │              │ - Indexing   │
   └────────────┴──────────────┴──────────────┘

   ┌──────────────────────────────────────────┐
   │     Monitoring & Observability Layer     │
   │  Logs (ELK) | Metrics (Prometheus/Grafana) │
   │     Distributed Tracing (Jaeger)         │
   └──────────────────────────────────────────┘
```

Core Components Deep Dive

1. API Gateway

Purpose: Single entry point for all client requests

Responsibilities:

- Request routing to appropriate services
- Authentication and authorization (JWT tokens)
- Rate limiting (token bucket algorithm)
- Request/response transformation
- SSL termination
- API versioning

Technology Options Analysis:

Technology	Pros	Cons	Use Case

Technology	Pros	Cons	Use Case
Kong	Plugins, extensible, open-source	Complex setup, learning curve	**Recommended** for flexibility
AWS API Gateway	Managed, integrated with AWS	Vendor lock-in, limited control	Quick MVP
Nginx	Lightweight, fast, proven	Manual scaling, less features	High-performance needs
Envoy	Modern, service mesh ready	Newer, smaller community	Microservices architecture

Decision: Kong API Gateway

Why Kong?

- **Plugin ecosystem**: 50+ plugins for auth, rate limiting, logging
- **Scalability**: Proven at 100K+ RPS (Mashape experience)
- **Flexibility**: Can customize with Lua plugins
- **Multi-protocol**: REST, gRPC, WebSocket support
- **Observability**: Built-in metrics and tracing

Trade-offs Accepted:

- ✅ Best features and flexibility
- ✅ Open-source with enterprise option
- ✅ Active community
- ❌ More complex than managed solution
- ❌ Need to manage infrastructure ourselves

Configuration Strategy:

```
Rate Limiting:
├── Free tier: 100 req/min per user
├── Pro tier: 1,000 req/min per user
├── Enterprise: 10,000 req/min per user
└── Burst: 2× sustained rate for 10 seconds

Circuit Breaker:
├── Failure threshold: 50% error rate
├── Open duration: 30 seconds
├── Half-open: Try 1 request
└── Close when: 10 consecutive successes
```

---

2. Upload Service

Purpose: Handle file uploads efficiently at scale

Challenge: How to upload large files reliably?

Problem Analysis:

```
Single-shot upload of 5GB file:
- Network interruption → Complete failure
- 30-minute upload time
- No progress indication
- Cannot resume
- Server memory bottleneck (buffer entire file)
```

Solution Comparison:

Approach	Max File Size	Resume	Speed	Complexity	Choice
Single Upload	100 MB	❌	Slow	Simple	❌
Chunked Sequential	10 GB	✅	Slow	Medium	❌
Chunked Parallel	10 GB+	✅	Fast	Medium	✅ **Recommended**
Streaming	Unlimited	⚠	Medium	High	Special cases

Decision: Chunked Parallel Upload

Architecture:

```
┌──────────────────────────────────────────────────────────────┐
│ STEP 1: Initiate Upload                                      │
└──────────────────────────────────────────────────────────────┘
Client:
├── File: vacation.mp4 (500 MB)
├── Calculate SHA-256: abc123...def
└── POST /api/v1/files/initiate
    Body: {
      filename: "vacation.mp4",
      size: 524288000,
      checksum: "abc123...def",
      parent_folder_id: "folder_456"
    }

┌──────────────────────────────────────────────────────────────┐
│ STEP 2: Deduplication Check                                  │
└──────────────────────────────────────────────────────────────┘
Upload Service:
├── Query: SELECT file_id FROM files WHERE checksum = 'abc123...def'
├── Result: NULL (file doesn't exist)
└── Decision: Proceed with upload

If file existed:
```

```
├── Create file reference for user
├── No actual upload needed
├── Response: "Upload complete" (instant)
└── Savings: 500 MB bandwidth + storage

┌─────────────────────────────────────────────────────────────────┐
│ STEP 3: Generate Upload Plan                                    │
└─────────────────────────────────────────────────────────────────┘

Upload Service:
├── Calculate chunks: 500 MB ÷ 5 MB = 100 chunks
├── Generate upload_id: "upload_789"
├── Create presigned S3 URLs (one per chunk):
│   ├── Chunk 1: PUT s3://bucket/uploads/upload_789/chunk_1?sig=...
│   ├── Chunk 2: PUT s3://bucket/uploads/upload_789/chunk_2?sig=...
│   └── ... (100 URLs total)
├── Store in Redis:
│   ├── Key: upload:789:metadata
│   ├── Value: {status: "in_progress", chunks: 100, completed: 0}
│   └── TTL: 24 hours
└── Response: {
      upload_id: "upload_789",
      chunk_size: 5242880,
      chunks: 100,
      urls: [...]
    }

┌─────────────────────────────────────────────────────────────────┐
│ STEP 4: Upload Chunks (Parallel)                                │
└─────────────────────────────────────────────────────────────────┘

Client:
├── Split file into 100 chunks
├── Upload 10 chunks concurrently (parallel)
├── For each chunk:
│   ├── PUT to presigned URL
│   ├── Verify ETag matches
│   ├── Notify server: POST /api/v1/files/upload_789/chunks/1
│   └── Update progress bar
└── Continue until all 100 chunks uploaded

Why 10 concurrent chunks?
├── Too few (1-5): Slow, underutilizes bandwidth
├── Optimal (8-12): Saturates bandwidth, manageable
└── Too many (20+): Overhead, diminishing returns

┌─────────────────────────────────────────────────────────────────┐
│ STEP 5: Track Progress                                          │
└─────────────────────────────────────────────────────────────────┘

Upload Service:
├── Each chunk completion:
│   ├── HINCRBY upload:789:metadata completed 1
│   └── HSET upload:789:chunks chunk_1 "completed"
├── WebSocket: Push progress to client
│   └── {uploaded: 45, total: 100, percent: 45}
```

```
└── Client shows: "Uploading... 45%"

Failure Handling:
├── Chunk 42 fails (network timeout)
├── Client retries chunk 42 (3 attempts)
├── Success on retry
└── Continue with remaining chunks

Resume from Failure:
├── Client crashes at 60% complete
├── Client restarts
├── GET /api/v1/files/upload_789/status
├── Response: {completed_chunks: [1-60], pending: [61-100]}
└── Resume uploading chunks 61-100 only

┌─────────────────────────────────────────────────────────────┐
│ STEP 6: Assemble File                                       │
└─────────────────────────────────────────────────────────────┘
Client:
└── POST /api/v1/files/upload_789/complete
    Body: {checksum: "abc123...def"}

Upload Service:
├── Verify all 100 chunks received
├── S3 Multipart Complete API:
│   └── Combines all chunks into single file
├── Calculate final checksum
├── Verify matches client checksum
└── Atomic operation (all-or-nothing)

If checksum mismatch:
├── File corrupted during upload
├── Delete all chunks
├── Response: 400 Bad Request
└── Client retries entire upload

┌─────────────────────────────────────────────────────────────┐
│ STEP 7: Finalize & Trigger Jobs                             │
└─────────────────────────────────────────────────────────────┘
Upload Service:
├── Move file to permanent location:
│   └── s3://bucket/users/user_123/files/file_789
├── Update metadata database:
│   ├── INSERT INTO files (...)
│   └── UPDATE users SET storage_used += 500MB
├── Delete temporary chunks & Redis state
└── Publish to message queue:
    ├── Event: file_uploaded
    ├── Payload: {file_id, user_id, size, mime_type}
    └── Consumers:
        ├── Thumbnail generator (if image/video)
        ├── Virus scanner (all files)
        ├── Content indexer (if document)
```

```
        └── Sync notifier (notify other devices)

Total Latency Breakdown:
├── Deduplication check: 5ms
├── Generate URLs: 10ms
├── Upload chunks (parallel): 30s (depends on bandwidth)
├── Assemble file: 2s
├── Update metadata: 50ms
└── Trigger jobs: 10ms
```

Key Optimizations:

1. **Direct to S3 Upload**

```
Without presigned URLs:
  Client → Upload Service → S3
  - Upload Service buffers data
  - Memory bottleneck
  - Double bandwidth usage

With presigned URLs:
  Client → S3 (direct)
  - Zero server load
  - Single network hop
  - Upload Service only coordinates
```

2. **Chunk Size Selection: Why 5MB?**

```
Too Small (1 MB):
├── 500 chunks for 500 MB file
├── High HTTP overhead
├── More S3 API calls ($$$)
└── ❌ Not optimal

Optimal (5 MB):
├── 100 chunks for 500 MB file
├── Balanced overhead
├── Good for resume
└── ✅ **Recommended**

Too Large (50 MB):
├── Only 10 chunks
├── Less granular progress
├── Longer retry on failure
└── ❌ Not optimal
```

3. **Deduplication Savings**

```
Scenario: 1000 users upload same 100MB video

Without deduplication:
└── Store: 1000 × 100 MB = 100 GB

With deduplication:
├── Store: 100 MB (single copy)
├── 1000 references to same file
└── Savings: 99.9 GB (99.9%)

Real-world impact:
- Popular files (memes, movies, documents): 20-30% dedup rate
- Backups/system files: 50-70% dedup rate
- Average: 30% storage savings
```

---

3. Download Service

Purpose: Efficient file downloads

Key Features:

- **CDN Integration**: Serve files from edge locations
- **Signed URLs**: Time-limited download links (15 min)
- **Range Requests**: Support partial downloads (resume)
- **Permission Validation**: Check access before generating URL
- **Streaming**: Support for large files

---

4. Metadata Service

Purpose: Manage file and folder metadata at global scale

Challenge: How to store 5 trillion file metadata records?

Problem Analysis:

```
Single PostgreSQL instance limits:
├── Max connections: ~10,000
├── Max storage: ~16 TB (practical limit)
├── QPS capacity: ~10,000 reads, ~1,000 writes
└── Our need: 70,000 QPS, 10 PB metadata

Conclusion: Single database cannot handle the scale
```

Database Technology Comparison:

Database	Consistency	Queries	Scale	Ops	Choice
PostgreSQL	Strong (ACID)	Complex (JOINs)	Shard required	Mature	✅ **Primary**
MySQL	Strong (ACID)	Complex	Shard required	Very mature	Alternative
MongoDB	Eventual	Simple	Auto-shard	Easy	❌ Eventual consistency issue
Cassandra	Eventual	Limited	Excellent	Complex	❌ No JOINs
DynamoDB	Eventual	Limited	Excellent	Managed	❌ No JOINs for folders

Decision: PostgreSQL with Sharding

Why PostgreSQL over alternatives?

1. **Strong Consistency Needed**:

```
Scenario: User deletes file while another device reads it

With Strong Consistency (PostgreSQL):
├── Device A: DELETE file
├── Transaction commits
├── Device B: SELECT file
└── File not found (correct)

With Eventual Consistency (NoSQL):
├── Device A: DELETE file
├── Device B: SELECT file (different replica)
├── File still exists (stale read)
└── Conflict! Device B might re-upload deleted file
```

2. **Complex Queries Required**:

```
Query: "Get all files in folder and subfolders shared with user X"

SELECT f.*
FROM files f
JOIN folders fol ON f.parent_folder_id = fol.folder_id
JOIN shares s ON (s.resource_id = f.file_id OR s.resource_id =
fol.folder_id)
WHERE s.shared_with_user_id = 'user_X'
  AND fol.path LIKE '/Documents/%'
ORDER BY f.modified_at DESC;

This is trivial in SQL, very complex in NoSQL
```

3. **ACID Transactions Required**:

```
Operation: Move file to different folder

BEGIN TRANSACTION;
  UPDATE files SET parent_folder_id = 'new_folder' WHERE file_id =
'file_123';
  UPDATE users SET storage_used = storage_used WHERE user_id =
'user_123';
  INSERT INTO sync_events (...);
COMMIT;

If any fails, all rollback (atomic)
NoSQL databases struggle with multi-document transactions
```

Sharding Strategy: Why shard by user_id?

Alternatives Considered:

```
Option 1: Shard by file_id (Hash Sharding)
Pros:
├── Even distribution
└── No hot shards

Cons:
├── User's files scattered across shards
├── Cannot query "all files for user" on single shard
├── Cross-shard JOIN for folder hierarchy
└── ❌ Rejected

Option 2: Shard by geography
Pros:
├── Low latency (data near users)
└── Regulatory compliance (data residency)

Cons:
├── Users travel → need cross-region queries
├── Uneven distribution (US > Antarctica)
└── ⚠ Consider for global deployment

Option 3: Shard by user_id (Range Sharding)
Pros:
├── All user data on same shard ✓
├── Single-shard queries for user operations ✓
├── Folder hierarchy queries efficient ✓
└── ✅ **Selected**

Cons:
├── Celebrity users create hot shards
```

```
├── Need rebalancing as users grow
└── Mitigation: Virtual nodes + monitoring
```

Implementation: Consistent Hashing with Virtual Nodes

```
Hash Ring with 1000 virtual nodes:
┌─────────────────────────────────────────────┐
│         Hash Ring (0 to 2^32)               │
│                                             │
│  VN_001 (Shard A)                           │
│      VN_334 (Shard B)                       │
│          VN_667 (Shard C)                   │
│              VN_002 (Shard A)               │
│    ...                                      │
│              VN_999 (Shard C)               │
└─────────────────────────────────────────────┘

User assignment:
├── hash(user_12345) → 445329847
├── Find next virtual node: VN_334
├── VN_334 → Shard B
└── Store all user_12345 data in Shard B

Benefits:
├── Adding shard: Only ~1/N data moves
├── Removing shard: Evenly redistributed
└── Hot users: Can assign dedicated shard
```

Read Replica Strategy:

```
┌──────────────────────────────────────────────┐
│            Shard A (user_id 0-333M)          │
│                                              │
│  ┌──────────┐                                │
│  │ Primary  │ (Writes only)                  │
│  │ US-EAST  │                                │
│  └────┬─────┘                                │
│       │ Async replication                    │
│       ├──────────┬──────────┐                │
│       ▼          ▼          ▼                │
│  ┌─────────┐┌─────────┐┌─────────┐           │
│  │Replica1 ││Replica2 ││Replica3 │ (Reads)  │
│  │US-WEST  ││EU-WEST  ││AP-SE    │          │
│  └─────────┘└─────────┘└─────────┘          │
└──────────────────────────────────────────────┘

Read distribution:
├── 80% reads go to replicas (distributed)
```

```
├── 20% writes go to primary
└── Replication lag: <100ms (acceptable)

Failover on primary failure:
├── Promote Replica1 to primary (<30s)
├── Reconfigure replication topology
└── Resume operations
```

Caching Strategy: Multi-Layer

```
L1: Application Cache (In-Memory)
├── Store: Recently accessed file metadata
├── Size: 1 GB per API gateway instance
├── TTL: 5 minutes
├── Hit rate: 40%
└── Latency: 0.1ms

L2: Redis Cluster
├── Store: Hot file metadata, user quotas
├── Size: 100 GB total
├── TTL: 1 hour (metadata), 5 min (quotas)
├── Hit rate: 90% (of L1 misses)
└── Latency: 1ms

L3: Database
├── Store: All metadata (source of truth)
├── Size: 10 PB (5T files × 1KB × 2 for indexes)
├── Hit rate: 10% (cache misses)
└── Latency: 10-50ms

Combined Hit Rate = 40% + (60% × 90%) = 94%
Average Latency = (0.4 × 0.1ms) + (0.54 × 1ms) + (0.06 × 30ms) = 2.4ms
```

Cache Invalidation Strategy:

```
Problem: File metadata updated, caches become stale

Solutions Compared:

Option 1: Write-Through Cache
└── Update cache + database simultaneously
    ├── Pros: Always consistent
    └── Cons: Complex, all writes slower

Option 2: Cache Invalidation on Write
└── Delete cache key on update
    ├── Pros: Simple, cache always fresh on next read
    └── Cons: Next read slower (cache miss)
```

```
    ✅ **Selected**

Option 3: TTL-based Expiration
└── Let cache expire naturally
    ├── Pros: Zero overhead
    └── Cons: Stale data for TTL duration
    ⚠ Use with invalidation

Implementation:
├── On file update:
│   ├── UPDATE database
│   ├── DEL cache key in Redis
│   └── Publish event to invalidate L1 cache
├── Next read:
│   ├── Cache miss
│   ├── Query database
│   └── Populate cache
```

Quota Enforcement:

```
Challenge: Prevent storage quota exceeded

Naive approach (doesn't work):
GET user quota → Check limit → Upload file → Update quota
Problem: Race condition if concurrent uploads

Solution: Atomic check-and-reserve
├── Use database transaction:
│   BEGIN;
│     SELECT storage_used, storage_quota FROM users WHERE user_id = X
FOR UPDATE;
│     -- Row locked, no other transaction can modify
│     IF storage_used + file_size <= storage_quota THEN
│       UPDATE users SET storage_used = storage_used + file_size;
│       -- Reserve space
│       COMMIT;
│     ELSE
│       ROLLBACK;
│       RETURN 'Quota exceeded';
│     END IF;
│   COMMIT;

Alternative: Pessimistic locking ensures atomicity
```

5. Synchronization Service

Purpose: Keep files in sync across devices in real-time

Challenge: How to sync files across billions of devices in real-time?

Problem Statement:

```
Scenario: User has 4 devices
- MacBook (editing document)
- iPhone (wants to view)
- iPad (offline)
- Work PC (wants to edit)

Requirements:
├── MacBook saves → iPhone sees update in <5 seconds
├── iPad offline → Syncs when online
├── Work PC edits same file → Conflict resolution
└── Scale: 200M users × 3 devices = 600M active connections
```

Synchronization Approach Comparison:

Approach	Latency	Scale	Offline	Conflicts	Bandwidth	Choice
Polling (5s)	5s avg	✓	✓	Manual	High	❌ Outdated
Long Polling	1-5s	✓✓	✓	Manual	Medium	⚠ Fallback
WebSocket	<1s	✓✓✓	✓	Auto	Low	✅ **Primary**
Server-Sent Events	<1s	✓✓	✓	Auto	Low	⚠ Alternative

Decision: WebSocket with Long Polling fallback

Why WebSocket?

1. **Real-Time Bidirectional Communication**

```
Polling (old way):
  Every 5 seconds:
    Client → Server: "Any changes?"
    Server → Client: "No" (99% of the time)

  Waste: 99% of requests return "no changes"
  Latency: Average 2.5 seconds to detect change

WebSocket (new way):
  One persistent connection
  Server pushes when change occurs

  Efficiency: 100× less network traffic
  Latency: Milliseconds to detect change
```

2. **Scalability Analysis**

```
Polling approach:
├── 200M users × 12 polls/min = 2.4B requests/min = 40M RPS
├── Each poll: Network roundtrip, server processing
└── Cost: Extremely high server load

WebSocket approach:
├── 200M persistent connections
├── Each connection idle most of the time (epoll)
├── One server handles 100K connections (C10K problem solved)
└── Need: 2,000 servers for connections
    (Much better than 400,000 servers for polling)
```

3. **Battery Efficiency (Mobile)**

```
Polling on mobile:
├── Wake up every 5 seconds
├── Establish TCP connection
├── Send HTTP request
└── Battery drain: Significant

WebSocket:
├── Single persistent connection
├── Server pushes data
├── No repeated wake-ups
└── Battery drain: Minimal

For Mobile: Use native push notifications (APNs/FCM)
```

Architecture in Detail:

```
┌─────────────────────────────────────────────────────────────────┐
│ SYNC SCENARIO: User edits file on MacBook                       │
└─────────────────────────────────────────────────────────────────┘

Step 1: Detect Local Change (MacBook)
┌─────────────────────────────────────────┐
│ File System Watcher (inotify/FSEvents)  │
│  - Monitors: ~/Drive directory          │
│  - Detects: file.txt modified           │
│  - Debounce: Wait 1s for more changes   │
└─────────────────────────────────────────┘
         ↓
Step 2: Calculate Delta
┌─────────────────────────────────────────┐
│ Sync Client                             │
```

```
  - Get file's last sync version
  - Calculate diff (rsync algorithm)
  - Compression: gzip the diff
  - Result: 500 KB → 50 KB (90% smaller)

         ↓
Step 3: Upload Change

 POST /api/v1/sync/upload
 Body: {
   file_id: "file_123",
   device_id: "device_mac",
   version: 42,
   delta: <compressed_binary>,
   checksum: "xyz789"
 }

         ↓
Step 4: Store & Version

 Upload Service
  - Store delta in S3
  - Update metadata:
    UPDATE files
    SET version = 43,
        modified_at = NOW()
    WHERE file_id = 'file_123'

         ↓
Step 5: Publish Sync Event

 Message Queue (Kafka)
 Topic: file_changes
 Event: {
   type: "file_modified",
   file_id: "file_123",
   user_id: "user_456",
   version: 43,
   device_id: "device_mac",
   timestamp: 1640995200
 }

         ↓
Step 6: Sync Service Processes Event

 Sync Service Consumer
  - Read from Kafka
  - Query: Get all devices for user_456
  - Filter: Exclude device_mac (originator)
  - Result: [device_iphone, device_ipad,
             device_workpc]

         ↓
```

```
Step 7: Notify Devices

┌─────────────────────────────────────────┐
│ Device iPhone (WebSocket - Active)      │
│  - Connection alive                     │
│  - Push: {file_id, version, delta_url}  │
│  - Latency: 50ms                        │
└─────────────────────────────────────────┘

┌─────────────────────────────────────────┐
│ Device iPad (Offline)                   │
│  - No active connection                 │
│  - Store in pending_sync queue          │
│  - Deliver when device comes online     │
└─────────────────────────────────────────┘

┌─────────────────────────────────────────┐
│ Device WorkPC (WebSocket - Active)      │
│  - Currently editing same file!         │
│  - Conflict detected!                   │
│  - Handle conflict resolution           │
└─────────────────────────────────────────┘
         ↓
Step 8: Apply Changes
┌─────────────────────────────────────────┐
│ iPhone Sync Client                      │
│  - Download delta from URL              │
│  - Apply patch to local file            │
│  - Verify checksum                      │
│  - Update local version to 43           │
│  - Notify user: "file.txt updated"      │
└─────────────────────────────────────────┘
```

Conflict Resolution: The Hard Problem

```
CONFLICT SCENARIO:
Time 10:00:00 - Both devices start with file version 42
Time 10:00:30 - MacBook saves "Hello World" (version 43)
Time 10:00:31 - WorkPC saves "Goodbye World" (version 43)

Both think they're at version 43!

┌─────────────────────────────────────────────────────────────┐
│ Conflict Detection Algorithm                                │
└─────────────────────────────────────────────────────────────┘

Server receives MacBook update:
├── Current server version: 42
├── MacBook claims base version: 42 ✓
├── No conflict
└── Accept, increment to version 43
```

```
Server receives WorkPC update (1 second later):
├── Current server version: 43 (MacBook just updated)
├── WorkPC claims base version: 42 ✗
├── CONFLICT DETECTED!
└── Need resolution strategy
```

Conflict Resolution Strategies Compared:

```
Strategy 1: Last-Write-Wins (LWW)
├── Compare timestamps
├── MacBook: 10:00:30
├── WorkPC: 10:00:31 → Winner
├── WorkPC version becomes version 44
├── MacBook version discarded
└── Problem: Data loss! (MacBook's edits lost)

Strategy 2: Merge Attempts (Operational Transform)
├── Analyze both edits
├── Try to merge intelligently
├── "Hello World" + "Goodbye World" = ???
└── Problem: Very complex, doesn't always work

Strategy 3: Last-Write-Wins + Conflict Copy ✅
├── WorkPC version becomes version 44 (latest timestamp)
├── Save MacBook version as conflict copy:
│   "file (conflicted copy from MacBook 10:00:30).txt"
├── Notify user on both devices
└── User manually resolves
    Pros: No data loss, simple, always works
    Cons: Manual intervention needed
```

Implementation of LWW + Conflict Copy:

```
┌──────────────────────────────────────────────────────────────┐
│ Conflict Resolution Flow                                     │
└──────────────────────────────────────────────────────────────┘

Server detects conflict:
├── Base version: 42
├── Current version: 43 (MacBook)
├── Incoming version: 43 (WorkPC claims 42 base)
└── Timestamp comparison:
    ├── MacBook: 10:00:30
    └── WorkPC: 10:00:31 (winner)

Resolution:
├── Accept WorkPC as version 44
├── Create conflict file for MacBook:
```

```
│   ├── Name: "file (MacBook 10:00:30 conflicted).txt"
│   ├── Content: MacBook's version
│   └── Metadata: marked as conflict
├── INSERT INTO files (conflict file)
├── Publish sync events:
│   ├── MacBook: Download conflict file
│   └── WorkPC: Download winning version
└── Notify both users:
    "Conflicted copy created"

User Experience:
MacBook sees:
├── file.txt (WorkPC version)
├── file (MacBook 10:00:30 conflicted).txt (their version)
└── Notification: "Conflict detected, review changes"

WorkPC sees:
├── file.txt (their version, now official)
├── file (MacBook 10:00:30 conflicted).txt (other version)
└── Notification: "Another device edited simultaneously"
```

WebSocket Connection Management:

```
Challenge: Maintain 600M WebSocket connections

Solution: Connection Pooling + Load Distribution

Architecture:
┌─────────────────────────────────────────────┐
│ Sync Service Cluster                        │
│                                             │
│  ┌────────┐  ┌────────┐  ┌────────┐        │
│  │Sync Srv│  │Sync Srv│  │Sync Srv│ ...    │
│  │   #1   │  │   #2   │  │   #3   │        │
│  │        │  │        │  │        │        │
│  │100K WS │  │100K WS │  │100K WS │        │
│  │conns   │  │conns   │  │conns   │        │
│  └───┬────┘  └───┬────┘  └───┬────┘         │
│      │           │           │              │
│      └───────────┴───────────┘              │
│                  │                          │
│         Subscribe to Kafka                  │
│       Topic: file_changes.user_456          │
└─────────────────────────────────────────────┘

Distribution Strategy:
├── Each server: 100,000 connections
├── Total servers: 600M ÷ 100K = 6,000 servers
├── Load balancer: Consistent hashing by user_id
└── Benefit: All user's devices on same server
    (Easier to broadcast changes)
```

```
Connection stickiness:
├── user_456 → Always routes to Sync Server #42
├── All user_456's devices on same server
└── Broadcast to 3-5 devices instead of searching 6000 servers

Cost optimization:
├── WebSocket keep-alive: 30 seconds
├── Idle connections use ~4 KB memory
├── 100K connections = 400 MB memory
└── Acceptable on modern servers (128 GB RAM)
```

Fallback to Long Polling:

```
Why need fallback?

WebSocket failures:
├── Corporate firewalls block WS
├── Proxy servers don't support WS
├── Old browsers (IE 9)
└── Mobile apps occasionally

Long Polling mechanism:
┌─────────────────────────────────────────┐
│ 1. Client: GET /sync/poll?cursor=123  │
│    Server: Hold connection (60s)        │
├─────────────────────────────────────────┤
│ 2. If change occurs → Return immediately│
│    If 60s timeout → Return "no changes"│
├─────────────────────────────────────────┤
│ 3. Client immediately reconnects        │
│    (Keeps connection quasi-persistent) │
└─────────────────────────────────────────┘

Comparison:
├── WebSocket: 1 connection, push messages
├── Long Poll: Reconnect every 60s, pull messages
└── WebSocket preferred, Long Poll acceptable fallback
```

Sync Protocol Design:

```
Delta Sync vs Full Sync:

Small change (edited 1 line in 10 MB document):

Full Sync:
├── Download entire 10 MB file
├── Bandwidth: 10 MB
```

```
└── Time: 10 seconds

Delta Sync:
├── Download only changed bytes
├── Bandwidth: 5 KB (0.05%)
├── Time: 50ms (200× faster)
└── ✅ **Use for files < 100 MB**

Algorithm: rsync-like binary diff
├── Divide file into blocks (4 KB each)
├── Calculate rolling checksum per block
├── Compare checksums
├── Transfer only changed blocks
└── Reconstruct file from blocks + deltas

When to use Full Sync:
├── New file (no previous version)
├── File completely rewritten
├── Delta larger than original (rare)
└── First sync for device
```

Sync State Management:

```
Per-Device State Tracking:

Redis storage:
Key: sync:state:user_456:device_iphone
Value: {
  last_sync_timestamp: 1640995200,
  last_event_id: "event_789",
  pending_downloads: ["file_123", "file_456"],
  pending_uploads: [],
  sync_status: "synced"
}

Sync cursor:
├── Tracks last successfully processed event
├── On reconnect: Send all events after cursor
├── Ensures no events missed
└── Even if device offline for days
```

Offline Support:

```
iPad goes offline for 6 hours:

During offline:
├── Local changes queued (SQLite queue)
├── 10 files edited
```

```
├── 5 files created
└── Queue stored persistently

When online:
┌─────────────────────────────────────────┐
│ 1. Reconnect to Sync Service            │
│    GET /sync/changes?cursor=last_event  │
└─────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────┐
│ 2. Download Remote Changes               │
│    - 15 files changed by other devices  │
│    - Download deltas                     │
│    - Apply to local files                │
└─────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────┐
│ 3. Upload Local Changes                  │
│    - 10 edited files → Check conflicts  │
│    - 5 new files → Upload                │
│    - 2 conflicts detected → Create copies │
└─────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────┐
│ 4. Resolve Conflicts                     │
│    - file.txt has conflict              │
│    - Create: file (iPad conflicted).txt │
│    - Notify user                         │
└─────────────────────────────────────────┘

Conflict Detection:
├── Remote version: 44 (edited at 10:05)
├── Local version: 43 (edited at 10:03 while offline)
├── Both diverged from version 42
└── CONFLICT! (cannot auto-merge)
```

Connection Recovery:

```
Challenge: WebSocket connections drop frequently

Causes:
├── Network switches (WiFi → Cellular)
├── Network congestion
├── Server restarts
├── Load balancer timeout
└── Laptop sleep/wake

Recovery Strategy:

Detection:
├── Client: Heartbeat every 30s
```

```
├── Server: Heartbeat every 30s
├── No heartbeat for 60s → Connection dead
└── Reconnect immediately

Reconnection:
├── Exponential backoff: 1s, 2s, 4s, 8s, 16s, 30s (max)
├── Include cursor in reconnect request
├── Server sends missed events
└── Resume normal operation

State preservation:
├── Server: Keep connection state for 5 minutes after disconnect
├── Client: Keep pending changes in queue
└── No data loss even with frequent reconnects
```

Bandwidth Optimization:

```
Smart Sync Features:

1. Selective Sync:
   User marks folders:
   ├── "Always available offline" → Full sync
   ├── "Available on demand" → Metadata only
   └── "Online only" → No local copy

   Bandwidth savings: 50-70% for typical user

2. Sync Scheduling:
   Mobile devices:
   ├── WiFi: Sync immediately
   ├── Cellular: Sync metadata only
   ├── Low battery: Pause sync
   └── Charging + WiFi: Full sync

   Battery & data plan friendly

3. Compression:
   ├── Text files: gzip (70% savings)
   ├── Images: Already compressed (skip)
   ├── Videos: Already compressed (skip)
   └── Applied automatically per file type

4. Throttling:
   ├── Detect rapid changes (save every second)
   ├── Debounce: Wait 5 seconds for more changes
   ├── Batch upload: Send one delta with all changes
   └── Reduces sync events by 80%
```

---

6. Sharing Service

Purpose: Manage file/folder sharing

Features:

- **User Sharing**: Share with specific users
- **Public Links**: Generate shareable URLs
- **Permission Levels**: Owner, Editor, Commenter, Viewer
- **Expiration**: Time-limited shares
- **Access Logs**: Track who accessed what

Security:

- Cryptographically secure tokens
- Store hashed tokens (bcrypt)
- Password protection for public links
- Audit trail for all access

---

7. Search Service

Purpose: Fast file search across content and metadata

Technology: Elasticsearch

Search Capabilities:

- Full-text search in document content
- Metadata search (name, type, owner, date)
- Filters and facets
- Relevance ranking

Indexing Strategy:

- Asynchronous indexing via background workers
- Text extraction from PDFs, Office docs
- OCR for images (if enabled)
- Real-time index updates

---

8. Storage Layer (Object Storage)

Purpose: Store actual file data

Technology: AWS S3 / Google Cloud Storage

Features:

- **Versioning**: Keep multiple versions of files
- **Lifecycle Policies**: Auto-tier to cold storage

- **Cross-Region Replication**: For disaster recovery
- **Encryption**: At rest (AES-256)

Storage Classes:

- **Hot**: Frequently accessed (S3 Standard)
- **Warm**: Infrequent access (S3 IA)
- **Cold**: Archive (S3 Glacier)

---

9. Message Queue

Purpose: Asynchronous communication between services

Technology: Apache Kafka, RabbitMQ, AWS SQS

Event Types:

- File upload complete → Trigger virus scan, thumbnail generation
- File modified → Trigger sync notification
- File shared → Send notification
- File accessed → Update analytics

Benefits:

- Decouple services
- Handle traffic spikes
- Guarantee delivery
- Replay capability

---

10. Async Workers

Purpose: Process background jobs

Worker Types:

1. **Thumbnail Generator**: Create image/video previews
2. **Virus Scanner**: Scan files with ClamAV
3. **Content Indexer**: Extract text, update search index
4. **Analytics Processor**: Track usage metrics
5. **Notification Sender**: Email/push notifications

Processing:

- Pull jobs from message queue
- Process independently
- Retry with exponential backoff
- Dead letter queue for failed jobs

---

11. Notification Service

Purpose: Real-time notifications to users

Channels:

- **WebSocket/SSE**: Real-time web notifications
- **Push Notifications**: Mobile (FCM/APNS)
- **Email**: Via SendGrid/SES
- **In-App**: Notification center

Events:

- File shared with you
- Comment on your file
- Upload complete
- Storage quota warning

---

12. Cache Layer (Redis)

Purpose: Reduce database load, improve performance

Cached Data:

- User sessions (7 day TTL)
- File metadata (1 hour TTL)
- User quotas (5 min TTL)
- Recent files list (1 hour TTL)
- Sync cursors (24 hour TTL)

Strategy:

- Cache-aside pattern
- Write-through for critical data
- Cache invalidation on updates

Data Flow

1. File Upload Flow

```
┌──────────┐
│  Client  │ 1. Calculate file hash (SHA-256)
└──────────┘    POST /api/v1/files/check-hash
     │
     ▼
┌──────────┐
│   API    │ 2. Authenticate & validate
│ Gateway  │
```

```
       │
       ▼
┌──────────┐
│ Metadata │  3. Check if hash exists (deduplication)
│ Service  │     - If exists: Create reference, done
└──────────┘     - If new: Continue
       │
       ▼
┌──────────┐
│  Upload  │  4. Generate upload_id & pre-signed URLs
│ Service  │     Return to client
└──────────┘
       │
       ▼
┌──────────┐
│  Client  │  5. Upload chunks in parallel (5MB each)
└──────────┘     Direct to S3 using pre-signed URLs
       │
       ▼
┌──────────┐
│   S3     │  6. Store chunks
└──────────┘
       │
       ▼
┌──────────┐
│  Upload  │  7. Client signals completion
│ Service  │     - Verify all chunks received
└──────────┘     - Assemble file
       │          - Verify checksum
       ▼
┌──────────┐
│ Metadata │  8. Create file metadata record
│ Service  │     - Update user quota
└──────────┘     - Store file path
       │
       ▼
┌──────────┐
│ Message  │  9. Publish events:
│  Queue   │     - thumbnail_generate
└──────────┘     - virus_scan
       │          - index_content
       │          - sync_notify
       ▼
┌──────────┐
│  Async   │  10. Workers process jobs
│ Workers  │
└──────────┘
```

2. File Download Flow

```
Client → API Gateway → Auth Service → Share Service
   ↓                                        ↓
   └──────────────────► (Check Permissions)
                                ↓
                          Authorized?
                                ↓
                         Metadata Service
                                ↓
                     (Get file location in S3)
                                ↓
                         Download Service
                                ↓
                   (Generate signed URL - 15 min)
                                ↓
                             Client
                                ↓
                     (Download directly from S3/CDN)
```

3. Real-Time Sync Flow

```
Device A: User edits file
     ↓
Local File System Watcher detects change
     ↓
Upload Service: Store new version
     ↓
Metadata Service: Update metadata
     ↓
Message Queue: Publish sync event
     ↓
Sync Service: Consume event
     ↓
Identify affected devices (Device B, C, D)
     ↓
Push notification via WebSocket/Long Polling
     ↓
Device B, C, D: Receive update
     ↓
Download new version and apply
```

4. File Sharing Flow

```
User A shares file with User B
     ↓
API Gateway → Auth Service
     ↓
Share Service:
```

```
  - Validate User A owns file
  - Create share record in DB
  - Generate share token
     ↓
Message Queue: Publish notification event
     ↓
Notification Service:
  - Send email to User B
  - Send push notification
  - Create in-app notification
     ↓
User B receives notification
  - Click link → Download Service
  - Validate share token
  - Generate signed URL
  - Download file
```

Database Design

Metadata Database (PostgreSQL)

Core Entities:

1. **Users**: user_id, email, storage_quota, storage_used
2. **Files**: file_id, owner_id, name, size, checksum, parent_folder_id, version
3. **Folders**: folder_id, owner_id, name, parent_folder_id, path
4. **File_Versions**: version_id, file_id, version_number, storage_path
5. **Shares**: share_id, resource_id, owner_id, shared_with_user_id, permission
6. **Devices**: device_id, user_id, device_name, last_sync_at
7. **Sync_Events**: event_id, user_id, resource_id, event_type, timestamp

Key Indexes:

- owner_id + parent_folder_id (files/folders)
- checksum (for deduplication)
- share_token (for public links)
- user_id + timestamp (for sync events)

Sharding Strategy:

- Shard by user_id
- Consistent hashing
- Virtual nodes for better distribution

Document Store (MongoDB/DynamoDB)

Collections:

4. **File_Chunks**: Track chunks during upload
5. **Activity_Logs**: User actions audit trail
6. **Analytics_Events**: Usage metrics

Search Index (Elasticsearch)

Document Structure:

- file_id, name, content (extracted text)
- owner, tags, created_at, modified_at
- path, mime_type, size

Cache (Redis)

Key Patterns:

- `session:{session_id}` → User session
- `file:metadata:{file_id}` → File metadata
- `user:quota:{user_id}` → Storage usage
- `user:recent:{user_id}` → Recent files list
- `sync:cursor:{user_id}:{device_id}` → Sync state

Design Decisions & Trade-offs

1. Metadata: SQL vs NoSQL

Decision: PostgreSQL (SQL)

Reasoning:

- Strong consistency needed for ownership, permissions
- Complex queries (folder hierarchy, shared files)
- ACID transactions required
- Mature ecosystem

Trade-offs:

- ✅ Strong consistency
- ✅ Relational integrity
- ✅ Complex queries
- ❌ Harder horizontal scaling (needs sharding)

---

2. Storage: Object Storage (S3/GCS)

Decision: Object Storage

Reasoning:

- Designed for petabyte scale
- Built-in 11 9's durability
- Cost-effective
- Versioning support
- Global CDN integration

Trade-offs:

- ✅ Infinite scalability
- ✅ High durability
- ✅ Low cost per GB
- ❌ Higher latency than block storage

---

3. Consistency Model: Hybrid

Decision: Strong for metadata, Eventual for sync

Reasoning:

- Metadata (ownership, permissions) must be consistent
- Sync can tolerate eventual consistency
- Best balance of consistency and availability

Trade-offs:

- ✅ Guarantees where needed
- ✅ High availability
- ❌ More complex implementation

---

4. Upload: Chunked Multipart

Decision: 5MB chunks, parallel upload

Reasoning:

- Support large files (>5GB)
- Faster (parallel upload)
- Resumable after failures
- Better progress tracking

Trade-offs:

- ✅ Fast for large files
- ✅ Resumable
- ✅ Better UX
- ❌ More complex

---

5. Deduplication: Client-side hash

Decision: Client calculates hash, server verifies

Reasoning:

- Save bandwidth (don't upload duplicates)
- Save storage (single copy)
- Fast for users

Trade-offs:

- ✅ 30% bandwidth savings
- ✅ Storage savings
- ❌ Privacy concerns (hash reveals content)
- ❌ Reference counting complexity

---

6. Sync: WebSocket + Push Notifications

Decision: WebSocket for web/desktop, FCM/APNS for mobile

Reasoning:

- Real-time updates (low latency)
- Single persistent connection
- Battery efficient on mobile

Trade-offs:

- ✅ Real-time (milliseconds)
- ✅ Efficient
- ❌ Connection management complexity
- ❌ Firewall issues

---

7. Sharding: By user_id

Decision: Shard metadata by user_id

Reasoning:

- User's data co-located
- No cross-shard queries for user operations
- Simple to implement

Trade-offs:

- ✅ Query efficiency
- ✅ Data locality
- ❌ Hot shards (power users)
- ❌ Rebalancing complexity

---

8. Caching: Multi-layer (CDN + Redis + DB)

Decision: 3-tier caching

Reasoning:

- CDN for downloads (edge caching)
- Redis for metadata (application caching)
- DB buffer for query results

Trade-offs:

- ✅ 80%+ cache hit rate
- ✅ Reduced DB load
- ✅ Lower latency
- ❌ Cache invalidation complexity
- ❌ Additional cost

---

9. Conflict Resolution: Last-Write-Wins

Decision: LWW with conflict copies

Reasoning:

- Simple to implement
- Works for 99% of cases
- Preserves all data

Flow:

1. Detect conflict
2. Latest timestamp wins
3. Create conflict copy
4. User reviews

Trade-offs:

- ✅ Simple
- ✅ No data loss
- ❌ User intervention needed

---

10. Database Replication: Primary-Replica

Decision: 1 Primary + 3 Read Replicas

Reasoning:

- Scale reads (most operations)
- High availability

- Geographic distribution

Trade-offs:

- ✅ Read scalability (3x)
- ✅ HA
- ❌ Replication lag
- ❌ Failover complexity

Scalability Strategy

Horizontal Scaling

Application Layer:

- Stateless services behind load balancers
- Auto-scaling based on CPU/memory
- Can add/remove instances dynamically

Database Layer:

- Shard by user_id
- Add more shards as needed
- Read replicas for read scaling
- Eventually migrate to distributed SQL (CockroachDB, YugabyteDB)

Cache Layer:

- Redis cluster with consistent hashing
- Add nodes to handle more load
- Partition hot keys

Storage Layer:

- Object storage scales automatically
- No manual intervention needed

Vertical Scaling (When Needed)

- Upgrade database instance types
- More memory for cache
- Faster disks for database

Geographic Distribution

- Multi-region deployment
- Route users to nearest region
- Cross-region replication for DR

Reliability & Availability

High Availability Strategies

1. Redundancy:

- Multiple availability zones
- Load balancers in active-active
- Database primary-replica setup
- 3x replication for object storage

2. Failover:

- Automatic failover for database
- Health checks for all services
- Circuit breakers to prevent cascade failures

3. Graceful Degradation:

- Serve stale cache if DB down
- Disable features if dependencies fail
- Show user-friendly error messages

Disaster Recovery

Backup Strategy:

- Database: Continuous WAL archiving, daily full backups
- Object Storage: Cross-region replication, versioning
- Metadata: Daily exports to S3

Recovery:

- RPO (Recovery Point Objective): < 1 hour
- RTO (Recovery Time Objective): < 4 hours

Security Considerations

1. Authentication & Authorization

- JWT tokens with short expiry (15 min)
- Refresh tokens for session management
- OAuth 2.0 for third-party integrations
- Multi-factor authentication (TOTP, SMS)

2. Encryption

- **At Rest**: AES-256 encryption in S3
- **In Transit**: TLS 1.3 for all connections
- **End-to-End** (optional): Client-side encryption

3. Access Control

- Role-based permissions (Owner, Editor, Viewer)
- Least privilege principle
- Regular permission audits

4. Network Security

- VPC with private subnets
- Security groups and NACLs
- DDoS protection (CloudFlare, AWS Shield)
- Rate limiting at API Gateway

5. Data Protection

- Regular backups
- Data lifecycle policies
- Compliance (GDPR, HIPAA)
- Audit logs for all access

Summary & Key Points

Architecture Highlights

1. **Microservices**: Separate concerns (upload, download, metadata, sync, search)
2. **Horizontal Scalability**: Shard databases, object storage, read replicas
3. **High Availability**: Multi-region, replication, failover
4. **Strong Security**: Encryption, access control, audit logging
5. **Optimized Performance**: Caching, CDN, chunking, async processing

Key Design Principles

- **Scalability First**: Built to handle billions of users
- **Reliability**: Multiple layers of redundancy
- **Performance**: Caching and CDN for low latency
- **Security**: Defense in depth
- **Cost Efficiency**: Lifecycle policies, deduplication

Technology Choices

- **Metadata**: PostgreSQL (sharded)
- **Storage**: AWS S3 / GCS
- **Cache**: Redis Cluster
- **Message Queue**: Kafka / SQS
- **Search**: Elasticsearch
- **CDN**: CloudFront / Cloudflare

This design can handle billions of users and petabytes of data while maintaining 99.99% availability and strong security! 🚀