Global File Storage & Sharing System Design (Like Google Drive)

Table of Contents

Problem Statement

Design a scalable, reliable, and globally distributed file storage and sharing system that allows users to:

- Upload, download, and manage files
- Share files with other users
- Access files from multiple devices
- Sync files across devices in real-time
- Collaborate on documents

Scale: Billions of users, Petabytes of data storage

Requirements

Functional Requirements

1. **File Operations**
    - Upload files (single/multiple, any size)
    - Download files
    - Delete files
    - Update/Replace files
    - Create folders and organize files
2. **Sharing & Permissions**
    - Share files/folders with specific users
    - Share via public links

- Set permissions (view, edit, comment)
- Revoke access

3. **Synchronization**
    - Real-time sync across devices
    - Offline mode support
    - Conflict resolution
4. **Search & Discovery**
    - Search files by name, type, content
    - Filter by date, size, owner
5. **Version Control**
    - Maintain file version history
    - Restore previous versions

Non-Functional Requirements

1. **Scalability**
    - Support billions of users
    - Handle petabytes of data
    - Support millions of concurrent connections
2. **Availability**
    - 99.99% uptime (4 nines)
    - No single point of failure
    - Graceful degradation
3. **Consistency**
    - Eventual consistency for sync
    - Strong consistency for metadata
4. **Performance**
    - Upload/Download: < 100ms latency (for small files)
    - Sync latency: < 5 seconds
    - Search results: < 500ms
5. **Reliability**
    - Data durability: 99.999999999% (11 nines)
    - No data loss
    - Automatic backup and recovery
6. **Security**

- End-to-end encryption option
- Data encryption at rest and in transit
- Access control and authentication
- Audit logging

Capacity Estimation

Assumptions

- **Total Users**: 1 Billion active users
- **Daily Active Users (DAU)**: 200 Million (20%)
- **Average Storage per User**: 10 GB
- **Average File Size**: 2 MB
- **Read:Write Ratio**: 3:1

Storage Estimation

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

Average Files per User = 10 GB / 2 MB = 5,000 files
Total Files = 1B × 5,000 = 5 Trillion files
```

Bandwidth Estimation

```
Upload Traffic:
- Files uploaded per day = 200M × 5 files = 1B files/day
- Data uploaded = 1B × 2 MB = 2 PB/day
- Upload bandwidth = 2 PB / 86,400 sec = ~23 GB/sec

Download Traffic (3x uploads):
- Download bandwidth = 69 GB/sec

Total Bandwidth Required: ~100 GB/sec peak
```

QPS (Queries Per Second)

```
Total Requests per Day:
- Uploads: 200M users × 5 = 1B requests
- Downloads: 3B requests
- Metadata Operations: 2B requests
- Total: 6B requests/day
```

```
Average QPS = 6B / 86,400 = ~70,000 QPS
Peak QPS (3x average) = ~200,000 QPS
```

Memory Estimation (Cache)

```
Metadata Cache:
- Metadata per file: 1 KB (filename, size, owner, timestamps)
- Cache 20% hot files = 1T files × 1 KB = 1 TB
- With overhead: ~2 TB cache needed
```

API Design

RESTful APIs

1. File Management APIs

```
# Upload File
POST /api/v1/files/upload
Headers: Authorization: Bearer <token>
Content-Type: multipart/form-data
Body: {
    file: <binary_data>,
    parent_folder_id: "folder_123",
    metadata: {
        name: "document.pdf",
        description: "Project documentation"
    }
}
Response: {
    file_id: "file_456",
    name: "document.pdf",
    size: 2048576,
    upload_url: "https://upload.server.com/...",
    chunk_size: 5242880,
    created_at: "2024-01-01T10:00:00Z"
}

# Download File
GET /api/v1/files/{file_id}/download
Headers: Authorization: Bearer <token>
Response: 302 Redirect to signed URL or file stream

# Get File Metadata
GET /api/v1/files/{file_id}
Response: {
    file_id: "file_456",
    name: "document.pdf",
```

```
    size: 2048576,
    mime_type: "application/pdf",
    owner_id: "user_123",
    created_at: "2024-01-01T10:00:00Z",
    modified_at: "2024-01-01T12:00:00Z",
    version: 3,
    parent_folder_id: "folder_123",
    shared_with: ["user_789"],
    permissions: ["read", "write"]
}

# Delete File
DELETE /api/v1/files/{file_id}
Response: 204 No Content

# Update File Metadata
PATCH /api/v1/files/{file_id}
Body: {
    name: "new_name.pdf",
    description: "Updated description"
}
```

2. Folder Management APIs

```
# Create Folder
POST /api/v1/folders
Body: {
    name: "Project Documents",
    parent_folder_id: "root"
}

# List Folder Contents
GET /api/v1/folders/{folder_id}/contents
Query Params: ?page=1&limit=50&sort=name&order=asc
Response: {
    items: [
        {type: "folder", id: "folder_789", name: "Subfolder"},
        {type: "file", id: "file_456", name: "document.pdf"}
    ],
    total: 150,
    page: 1,
    has_more: true
}
```

3. Sharing & Permissions APIs

```
# Share File/Folder
POST /api/v1/shares
Body: {
    resource_id: "file_456",
    resource_type: "file",
    share_with: ["user_789", "user_101"],
    permission: "read",
    expiry: "2024-12-31T23:59:59Z"
}

# Create Public Link
POST /api/v1/shares/public
Body: {
    resource_id: "file_456",
    permission: "read",
    expiry: "2024-12-31T23:59:59Z"
}
Response: {
    share_link: "https://drive.example.com/s/abc123xyz",
    short_code: "abc123xyz"
}

# Get Share Information
GET /api/v1/shares/{share_id}

# Revoke Share
DELETE /api/v1/shares/{share_id}
```

4. Sync APIs

```
# Get Changes Since Last Sync
GET /api/v1/sync/changes
Query Params: ?since=<timestamp>&device_id=<id>
Response: {
    changes: [
        {
            type: "create",
            file_id: "file_456",
            path: "/Documents/file.pdf",
            timestamp: "2024-01-01T10:00:00Z"
        },
        {
            type: "delete",
            file_id: "file_789",
            timestamp: "2024-01-01T10:05:00Z"
        }
    ],
    cursor: "next_page_token"
}
```

```
# Register Device
POST /api/v1/devices/register
Body: {
    device_name: "MacBook Pro",
    device_type: "desktop",
    os: "macOS"
}
```

5. Search API

```
# Search Files
GET /api/v1/search
Query Params: ?q=document&type=pdf&modified_after=2024-01-01
Response: {
    results: [
        {
            file_id: "file_456",
            name: "document.pdf",
            path: "/Projects/document.pdf",
            snippet: "...relevant content...",
            score: 0.95
        }
    ],
    total: 42,
    page: 1
}
```

6. Version Control APIs

```
# List File Versions
GET /api/v1/files/{file_id}/versions
Response: {
    versions: [
        {
            version_id: "v3",
            modified_at: "2024-01-01T12:00:00Z",
            modified_by: "user_123",
            size: 2048576
        }
    ]
}

# Restore Version
POST /api/v1/files/{file_id}/versions/{version_id}/restore
```

Database Schema

SQL Database (Metadata - PostgreSQL/MySQL)

```sql
-- Users Table
CREATE TABLE users (
    user_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(255) UNIQUE NOT NULL,
    username VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    storage_quota BIGINT DEFAULT 15000000000, -- 15GB
    storage_used BIGINT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,
    INDEX idx_email (email),
    INDEX idx_username (username)
);

-- Files Table
CREATE TABLE files (
    file_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    file_uuid VARCHAR(36) UNIQUE NOT NULL, -- UUID for external
references
    owner_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    size BIGINT NOT NULL,
    mime_type VARCHAR(100),
    storage_path VARCHAR(1000), -- S3 path
    parent_folder_id BIGINT,
    is_deleted BOOLEAN DEFAULT FALSE,
    deleted_at TIMESTAMP NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    modified_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    checksum VARCHAR(64), -- SHA-256
    version INT DEFAULT 1,
    FOREIGN KEY (owner_id) REFERENCES users(user_id),
    FOREIGN KEY (parent_folder_id) REFERENCES folders(folder_id),
    INDEX idx_owner_parent (owner_id, parent_folder_id),
    INDEX idx_name (name),
    INDEX idx_created (created_at),
    INDEX idx_checksum (checksum) -- For deduplication
);

-- Folders Table
CREATE TABLE folders (
    folder_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    folder_uuid VARCHAR(36) UNIQUE NOT NULL,
    owner_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    parent_folder_id BIGINT,
```

```
    path VARCHAR(2000), -- Full path for quick lookups
    is_deleted BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    modified_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (owner_id) REFERENCES users(user_id),
    FOREIGN KEY (parent_folder_id) REFERENCES folders(folder_id),
    INDEX idx_owner_parent (owner_id, parent_folder_id),
    INDEX idx_path (path(255))
);

-- File Versions Table
CREATE TABLE file_versions (
    version_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    file_id BIGINT NOT NULL,
    version_number INT NOT NULL,
    size BIGINT NOT NULL,
    storage_path VARCHAR(1000),
    checksum VARCHAR(64),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    created_by BIGINT NOT NULL,
    FOREIGN KEY (file_id) REFERENCES files(file_id),
    FOREIGN KEY (created_by) REFERENCES users(user_id),
    UNIQUE KEY unique_file_version (file_id, version_number),
    INDEX idx_file_version (file_id, version_number)
);

-- Shares Table
CREATE TABLE shares (
    share_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    resource_id BIGINT NOT NULL,
    resource_type ENUM('file', 'folder') NOT NULL,
    owner_id BIGINT NOT NULL,
    shared_with_user_id BIGINT, -- NULL for public shares
    permission ENUM('read', 'write', 'comment') DEFAULT 'read',
    share_token VARCHAR(64) UNIQUE, -- For public links
    is_public BOOLEAN DEFAULT FALSE,
    expires_at TIMESTAMP NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (owner_id) REFERENCES users(user_id),
    FOREIGN KEY (shared_with_user_id) REFERENCES users(user_id),
    INDEX idx_resource (resource_type, resource_id),
    INDEX idx_shared_with (shared_with_user_id),
    INDEX idx_share_token (share_token)
);

-- Devices Table (for sync)
CREATE TABLE devices (
    device_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    device_uuid VARCHAR(36) UNIQUE NOT NULL,
    device_name VARCHAR(100),
    device_type ENUM('desktop', 'mobile', 'web'),
```

```
    os VARCHAR(50),
    last_sync_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    INDEX idx_user_device (user_id, device_uuid)
);

-- Sync Events Table (for tracking changes)
CREATE TABLE sync_events (
    event_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    resource_id BIGINT NOT NULL,
    resource_type ENUM('file', 'folder') NOT NULL,
    event_type ENUM('create', 'update', 'delete', 'move') NOT NULL,
    event_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    device_id BIGINT,
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    INDEX idx_user_timestamp (user_id, event_timestamp),
    INDEX idx_resource (resource_type, resource_id)
);
```

NoSQL Database (Document Metadata - MongoDB/DynamoDB)

```
// File Chunks Collection (for large file uploads)
{
    _id: ObjectId,
    file_id: "file_456",
    chunk_number: 1,
    chunk_size: 5242880,
    storage_path: "s3://bucket/chunks/file_456_chunk_1",
    checksum: "abc123...",
    uploaded_at: ISODate("2024-01-01T10:00:00Z"),
    status: "uploaded" // pending, uploaded, verified
}

// Activity Log Collection
{
    _id: ObjectId,
    user_id: "user_123",
    action: "file_upload",
    resource_id: "file_456",
    resource_type: "file",
    timestamp: ISODate("2024-01-01T10:00:00Z"),
    ip_address: "192.168.1.1",
    device_id: "device_789",
    metadata: {
        file_name: "document.pdf",
        file_size: 2048576
    }
}
```

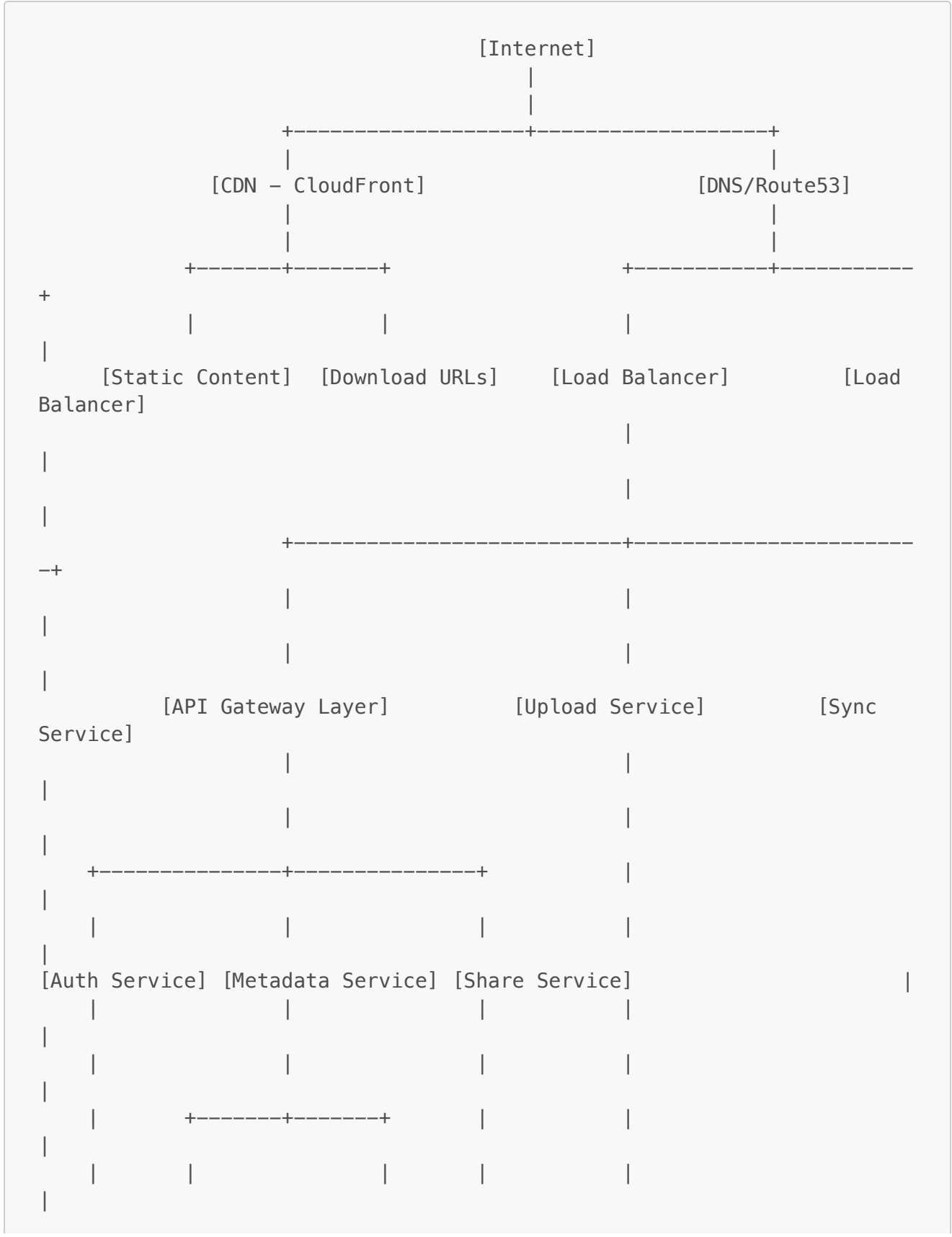
```
// Search Index Document (Elasticsearch)
{
    file_id: "file_456",
    name: "document.pdf",
    content: "extracted text content...",
    owner: "user_123",
    tags: ["project", "documentation"],
    created_at: "2024-01-01T10:00:00Z",
    modified_at: "2024-01-01T12:00:00Z",
    path: "/Projects/document.pdf",
    mime_type: "application/pdf",
    size: 2048576
}
```

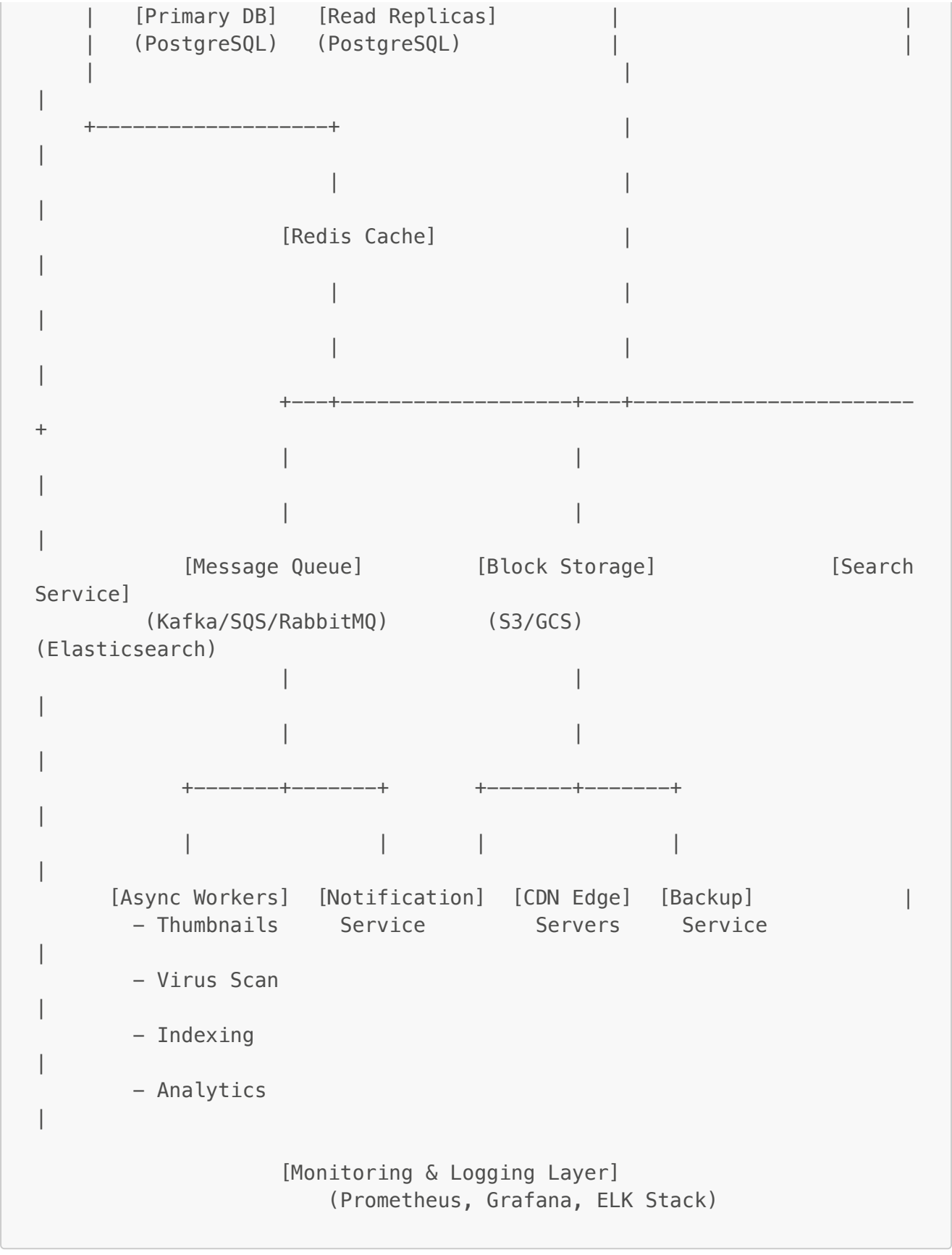
Cache Schema (Redis)

```
// User Session Cache
Key: session:{session_id}
Value: {
    user_id: "user_123",
    email: "user@example.com",
    expires_at: timestamp
}
TTL: 7 days

// File Metadata Cache
Key: file:metadata:{file_id}
Value: {
    file_id, name, size, owner_id, created_at, modified_at
}
TTL: 1 hour

// User Quota Cache
Key: user:quota:{user_id}
Value: {
    storage_used: 5000000000,
    storage_quota: 15000000000
}
TTL: 5 minutes

// Recent Files Cache
Key: user:recent:{user_id}
Value: [file_id_1, file_id_2, ..., file_id_20]
TTL: 1 hour

// Sync Cursor Cache
Key: sync:cursor:{user_id}:{device_id}
Value: {
    last_sync_timestamp: timestamp,
```

```
    last_event_id: event_id
}
TTL: 24 hours
```

High-Level Architecture





Detailed Component Design

1. API Gateway

Responsibilities:

- Request routing
- Rate limiting
- Authentication/Authorization
- Request validation
- API versioning
- SSL termination

Technology: Kong, AWS API Gateway, Nginx

Key Features:

- **Rate Limiting**: Token bucket algorithm per user/IP
  - Free tier: 100 requests/minute
  - Premium: 1000 requests/minute
- **Authentication**: JWT-based with refresh tokens
- **Circuit Breaker**: Fail fast when services are down
- **Request Logging**: All requests logged for audit

---

2. Upload Service

Responsibilities:

- Handle file uploads
- Chunk large files
- Generate upload URLs
- Coordinate multipart uploads

Design:

Chunked Upload Strategy:

```
1. Client requests upload initiation
2. Server responds with upload_id and chunk_size (5MB)
3. Client splits file into chunks
4. Client uploads chunks in parallel
5. Server assembles chunks after all uploaded
6. Server verifies checksum
7. Server stores file in S3 and updates metadata
```

Deduplication:

- Calculate SHA-256 hash on client
- Check if hash exists in database
- If exists, create reference instead of uploading
- Saves storage and bandwidth (copy-on-write)

Flow Diagram:

```
Client                Upload Service              Storage Service
  |                          |                           |
  |--Upload Request--------->|                           |
  |<--Upload ID + URLs------|                           |
  |                          |                           |
  |--Chunk 1---------------->|                           |
  |                          |--Store Chunk 1----------->|
  |<--ACK Chunk 1-----------|<--Success-----------------|
  |                          |                           |
  |--Chunk N---------------->|                           |
  |                          |--Store Chunk N----------->|
  |<--ACK Chunk N-----------|<--Success-----------------|
  |                          |                           |
  |--Complete Upload------->|                           |
  |                          |--Assemble File----------->|
  |                          |--Verify Checksum--------->|
  |<--Upload Success--------|<--Success-----------------|
```

Key Optimizations:

- **Parallel Uploads**: Upload multiple chunks simultaneously
- **Resumable Uploads**: Track uploaded chunks, resume from failure
- **Pre-signed URLs**: Direct upload to S3, bypass server
- **Compression**: Optional client-side compression
- **Progress Tracking**: WebSocket updates for real-time progress

---

3. Download Service

Responsibilities:

- Handle file downloads
- Stream large files
- Generate signed download URLs
- Support range requests (partial downloads)

Design:

CDN Integration:

- Static files served from CDN edge locations
- Reduced latency for global users
- Cache popular files
- Invalidate cache on file updates

Streaming:

```
1. Client requests download
2. Server validates permissions
```

```
3. Server generates signed URL (valid 15 minutes)
4. Client downloads directly from S3/CDN
5. Support HTTP Range header for resumable downloads
```

---

4. Metadata Service

Responsibilities:

- Store file/folder metadata
- Handle CRUD operations on metadata
- Maintain folder hierarchy
- Track file versions

Database Sharding Strategy:

- **Shard by user_id**: All user's data on same shard
- **Consistent Hashing**: Distribute load evenly
- **Virtual Nodes**: Better distribution, easier rebalancing

Caching Strategy:

- **Cache-Aside Pattern**:
    1. Check cache for metadata
    2. If miss, query database
    3. Store in cache for future requests
- **TTL**: 1 hour for file metadata, 5 minutes for quota
- **Cache Invalidation**: On updates, delete cache key

---

5. Synchronization Service

Responsibilities:

- Real-time sync across devices
- Detect and resolve conflicts
- Maintain sync state per device
- Push notifications for changes

Design:

Change Detection:

- Event-driven architecture using message queue
- Each file operation publishes event
- Sync service consumes events and notifies devices

Long Polling / WebSocket:

```
Client connects --> Server holds connection -->
Event occurs --> Server pushes event -->
Client processes --> Client reconnects
```

Conflict Resolution Strategy:

```
1. Last-Write-Wins (LWW):
   - Use timestamp + device_id as tiebreaker
   - Simpler but may lose data

2. Version Vector:
   - Track version per device
   - Detect conflicts when vectors diverge
   - Create conflict copy, let user resolve

3. Operational Transform:
   - For collaborative editing
   - Transform operations to maintain consistency
```

Implementation:

```
// Sync Algorithm (Simplified)
function syncChanges(user_id, device_id, last_sync_timestamp) {
    // Get all events since last sync
    const events = getEventsFromQueue(user_id, last_sync_timestamp);

    // Filter events not from current device
    const relevantEvents = events.filter(e => e.device_id !==
device_id);

    // Group by file/folder
    const changes = groupEventsByResource(relevantEvents);

    // Detect conflicts
    const conflicts = detectConflicts(changes, device_id);

    return {
        changes: changes,
        conflicts: conflicts,
        new_sync_timestamp: Date.now()
    };
}
```

---

6. Sharing Service

Responsibilities:

- Manage file/folder sharing
- Generate public links
- Enforce permissions
- Track access logs

Permission Model:

```
Permission Levels:
1. Owner: Full control (read, write, delete, share)
2. Editor: Read, write, comment
3. Commenter: Read, comment
4. Viewer: Read only

Inheritance:
- Folder permissions cascade to children
- Explicit permissions override inherited
```

Public Link Security:

- Generate cryptographically secure random tokens
- Store hash of token (bcrypt)
- Support expiration time
- Support password protection
- Track access count

---

7. Search Service

Responsibilities:

- Index file content and metadata
- Provide fast search results
- Support filters and facets
- Rank results by relevance

Technology: Elasticsearch / Apache Solr

Indexing Strategy:

```
1. Async Indexing:
   - File upload triggers indexing job
   - Worker extracts text from documents
   - Worker indexes in Elasticsearch

2. Index Structure:
   {
```

```
        "file_id": "file_456",
        "name": "document.pdf",
        "content": "extracted text...",
        "owner": "user_123",
        "created_at": "2024-01-01",
        "tags": ["project", "doc"],
        "path": "/Projects/doc.pdf",
        "size": 2048576
    }

3. Text Extraction:
   - PDF: Apache Tika
   - Images: OCR (Tesseract)
   - Office Docs: Apache POI
```

Search Query:

```
GET /files/_search
{
    "query": {
        "bool": {
            "must": [
                {"match": {"content": "project documentation"}},
                {"term": {"owner": "user_123"}}
            ],
            "filter": [
                {"range": {"created_at": {"gte": "2024-01-01"}}},
                {"term": {"mime_type": "application/pdf"}}
            ]
        }
    },
    "sort": [
        {"_score": "desc"},
        {"created_at": "desc"}
    ]
}
```

---

8. Storage Layer (S3/GCS)

Responsibilities:

- Store actual file data
- Ensure data durability
- Enable versioning
- Support lifecycle policies

Design Decisions:

Storage Classes:

- **Hot Storage**: Frequently accessed (S3 Standard)
- **Warm Storage**: Infrequently accessed (S3 IA)
- **Cold Storage**: Archive (S3 Glacier)
- **Auto-tiering**: Move based on access patterns

Bucket Organization:

```
Bucket Structure:
/user_data/
    /{user_id_prefix}/   # First 4 chars of user_id for sharding
        /{user_id}/
            /files/
                /{file_id}
            /versions/
                /{file_id}/
                    /v1
                    /v2
            /thumbnails/
                /{file_id}_thumb.jpg
```

Replication:

- **Cross-Region Replication**: For disaster recovery
- **Multi-Region**: Serve global users faster
- **3x Replication**: Standard across regions

Lifecycle Policies:

```
Rules:
1. Move to IA after 30 days of no access
2. Move to Glacier after 90 days
3. Delete file versions older than 1 year (except latest)
4. Delete deleted files permanently after 30 days (trash)
```

---

9. Notification Service

Responsibilities:

- Send real-time notifications
- Email notifications
- Push notifications (mobile)
- In-app notifications

Events to Notify:

- File shared with user
- Comment on file
- File upload complete
- Storage quota exceeded
- Security alerts

Technology:

- WebSocket / Server-Sent Events (SSE)
- Firebase Cloud Messaging (FCM) for mobile
- SendGrid / Amazon SES for email

---

10. Async Workers

Responsibilities:

- Process background jobs
- Generate thumbnails
- Scan for viruses
- Index documents
- Send emails
- Generate analytics

Job Queue:

- **Technology**: Apache Kafka, RabbitMQ, AWS SQS
- **Priority Queues**: Critical jobs (security) processed first
- **Retry Logic**: Exponential backoff
- **Dead Letter Queue**: Failed jobs after max retries

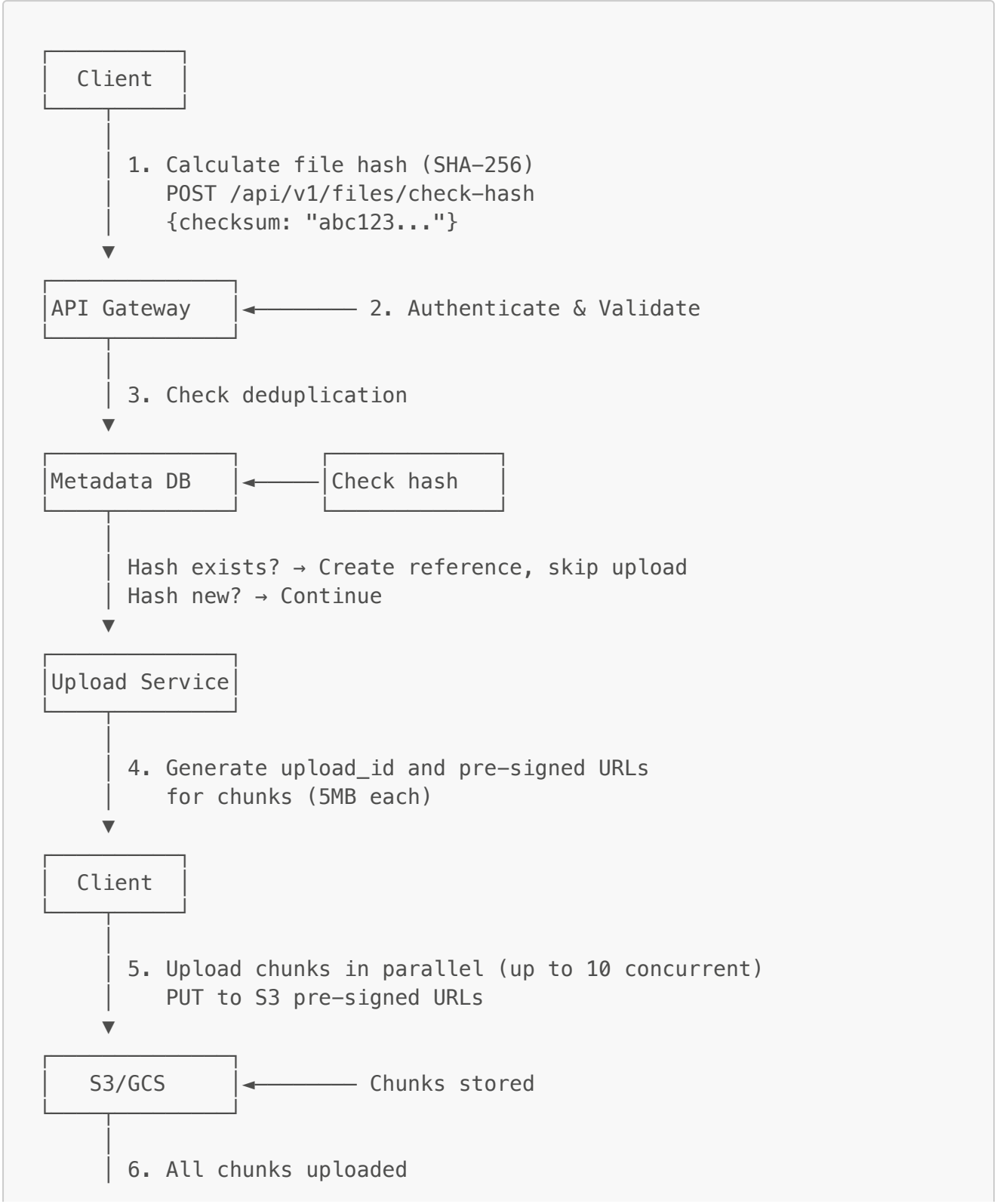
Worker Types:

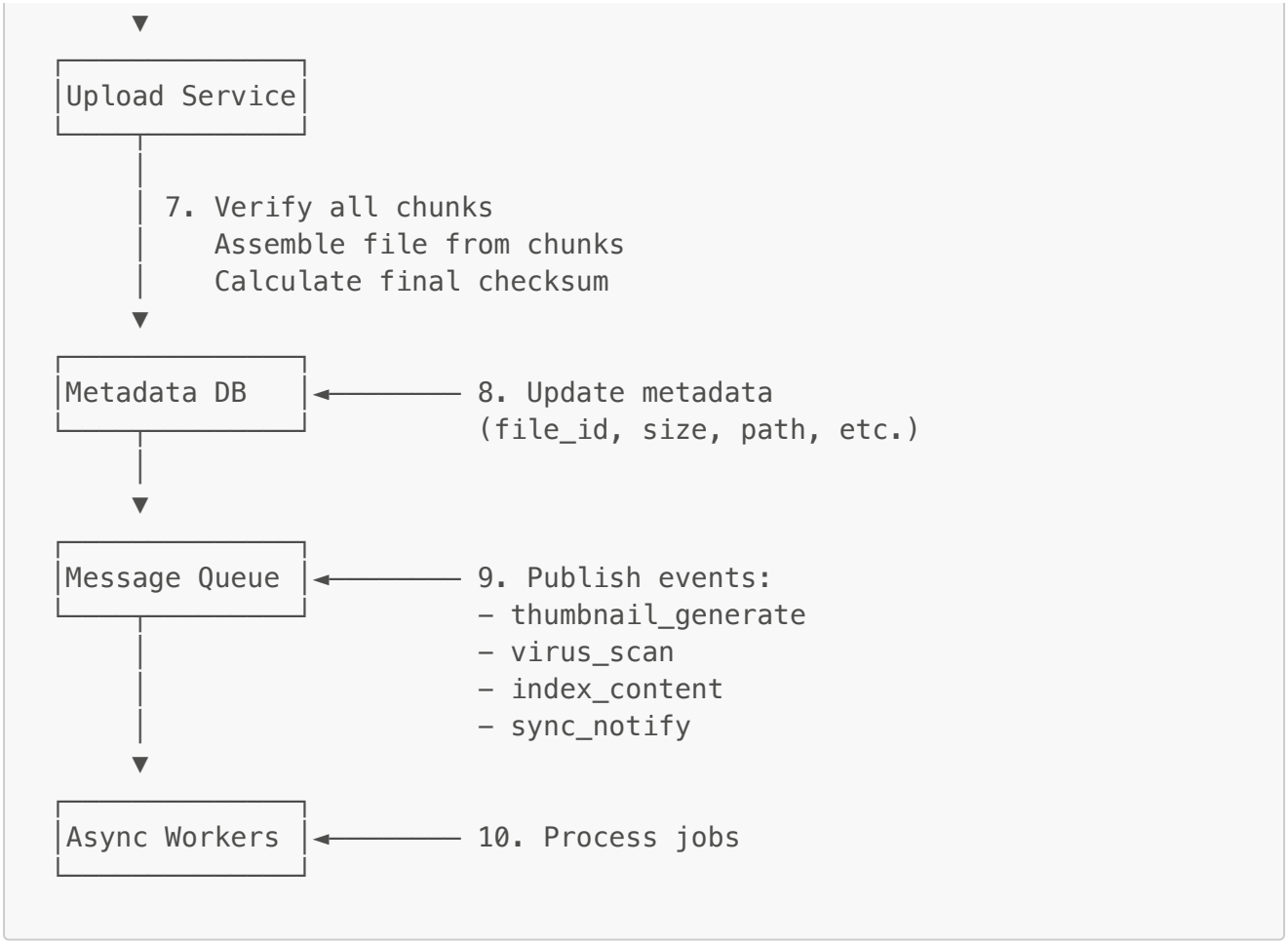
1. **Thumbnail Generator:**
    - For images and videos
    - Multiple sizes (small, medium, large)
    - Store in separate location
2. **Virus Scanner:**
    - Scan uploaded files with ClamAV
    - Quarantine suspicious files
    - Notify user
3. **Content Indexer:**
    - Extract text from documents
    - Index in Elasticsearch
    - Update search index

4. **Analytics Processor:**
    - Track usage metrics
    - Generate insights
    - Store in time-series DB

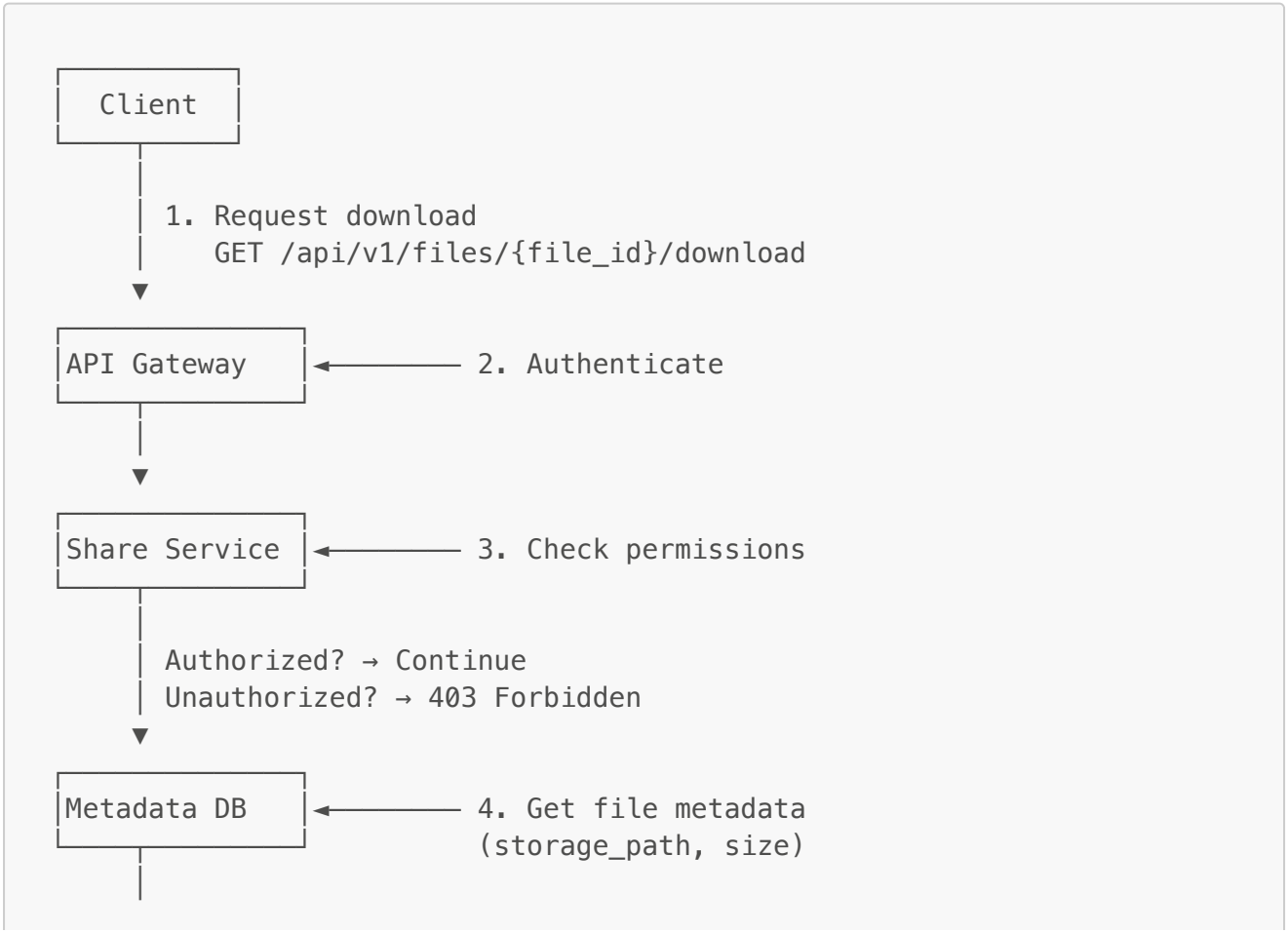
Data Flow

Upload Flow (Detailed)



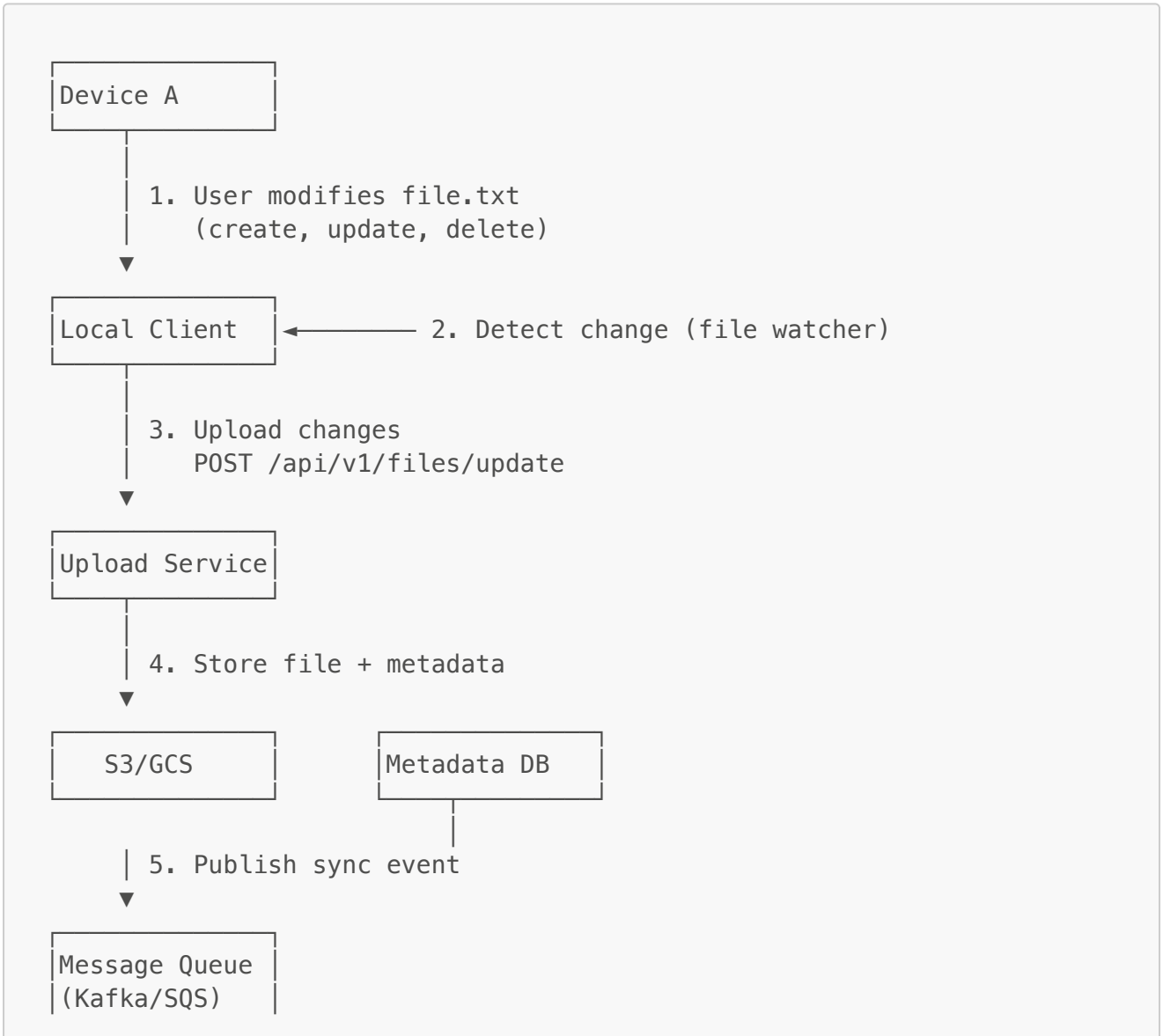


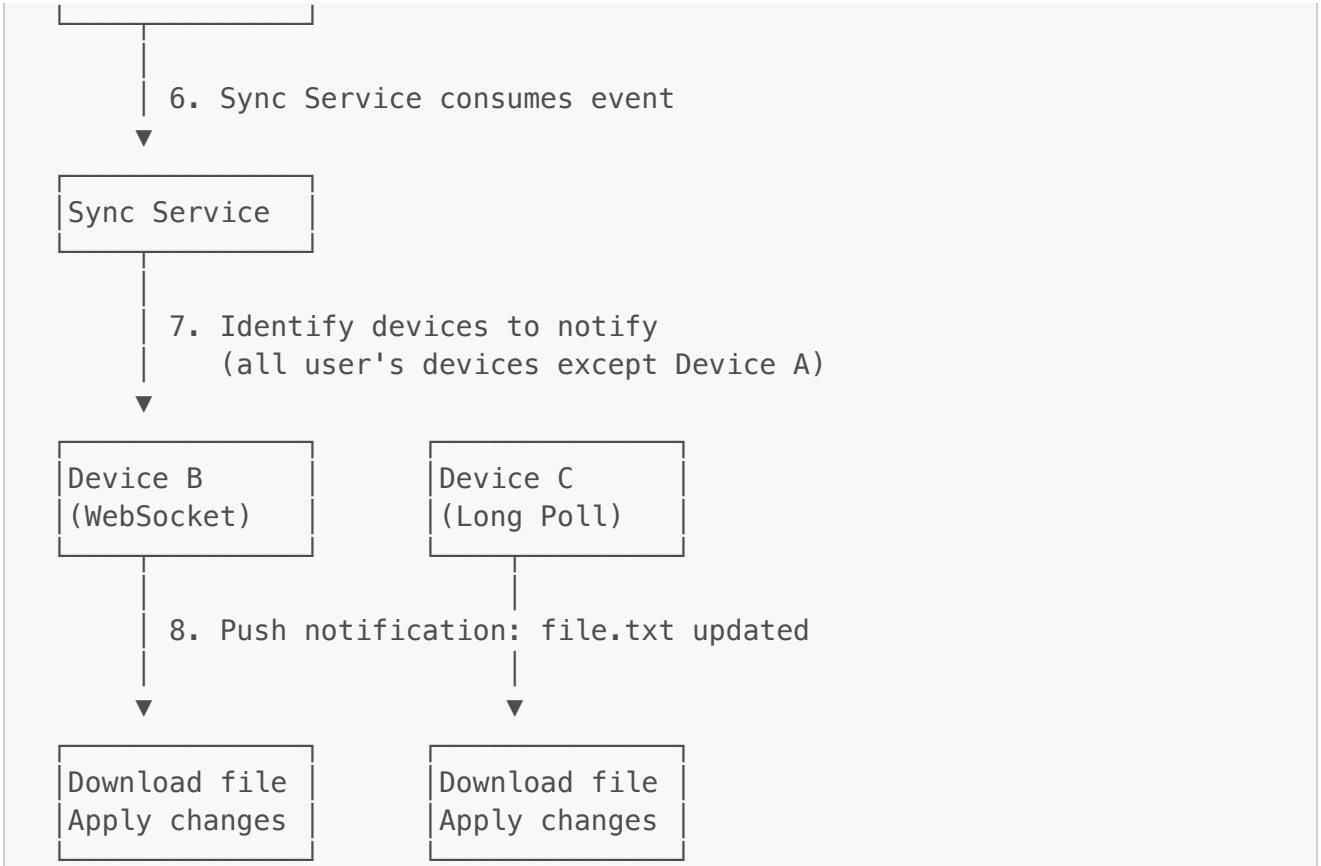
Download Flow



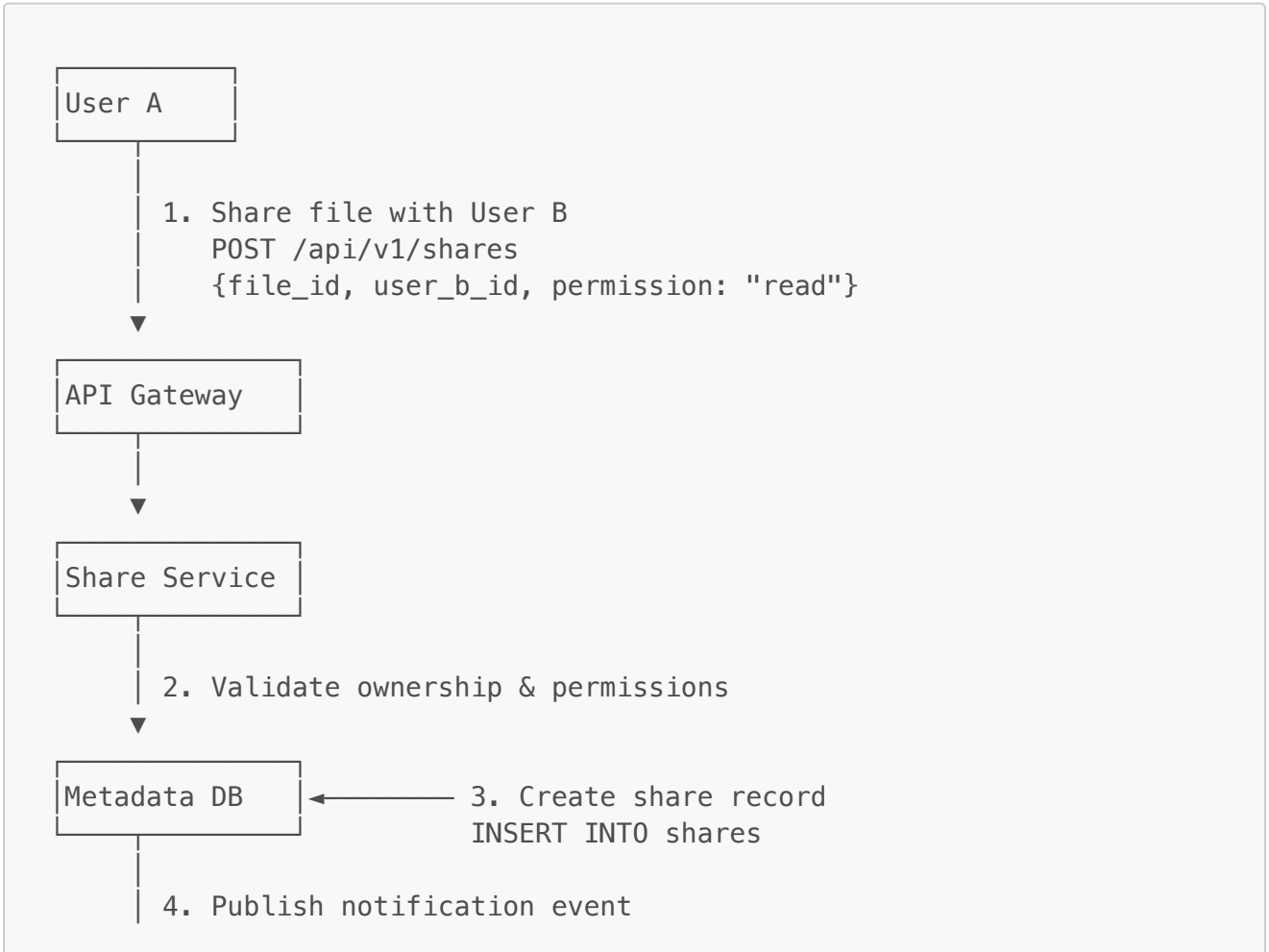

Download Svc
5. Generate signed URL (15 min expiry)
302 Redirect or direct URL
Client
6. Download from S3/CDN
Support Range requests
CDN/S3
File streamed to client

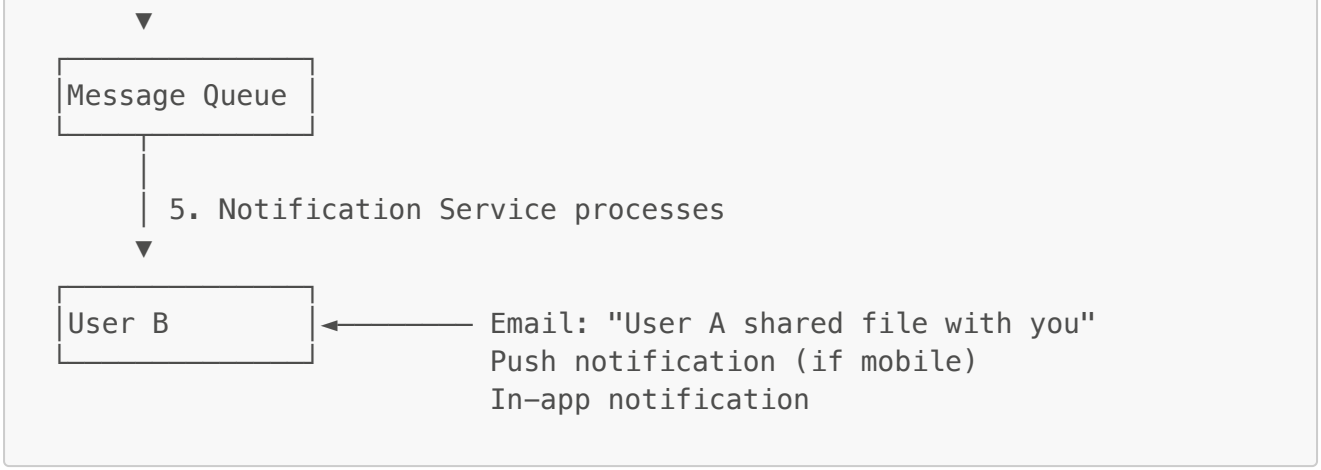
Sync Flow


Device A
1. User modifies file.txt
(create, update, delete)
Local Client
2. Detect change (file watcher)
3. Upload changes
POST /api/v1/files/update
Upload Service
4. Store file + metadata
S3/GCS
Metadata DB
5. Publish sync event
Message Queue
(Kafka/SQS)



Sharing Flow





Design Decisions & Trade-offs

1. **Metadata Storage: SQL vs NoSQL**

Decision: Use SQL (PostgreSQL) for primary metadata storage

Rationale:

- Strong consistency for critical operations (file ownership, permissions)
- ACID transactions for atomicity (e.g., move file + update parent)
- Complex queries with JOINs (folder hierarchy, shared files)
- Mature ecosystem and tooling

Trade-offs:

- ✅ Strong consistency guarantees
- ✅ Relational integrity (foreign keys)
- ✅ Complex query support
- ❌ Harder to scale horizontally (requires sharding)
- ❌ Fixed schema (migrations needed)

Alternatives Considered:

- **NoSQL (DynamoDB/MongoDB)**: Better horizontal scalability but eventual consistency
- **Hybrid**: SQL for metadata, NoSQL for logs/analytics (chosen for async data)

---

2. **File Storage: Block Storage vs Object Storage**

Decision: Use Object Storage (S3/GCS)

Rationale:

- Designed for massive scale (petabytes)
- Built-in redundancy and durability (11 9's)
- Cost-effective for large files
- Global CDN integration

- Versioning support

Trade-offs:

- ✅ Infinite scalability
- ✅ High durability (11 9's)
- ✅ Low cost per GB
- ✅ Built-in replication
- ❌ Higher latency than block storage
- ❌ Eventual consistency for some operations

Alternatives Considered:

- **Block Storage (EBS)**: Better for databases, but doesn't scale for file storage
- **HDFS**: Good for big data, but operational complexity

---

3. **Consistency Model: Strong vs Eventual**

Decision: Hybrid approach

Rationale:

- **Strong Consistency** for:
  - File metadata (ownership, permissions)
  - User quota enforcement
  - Critical transactions
- **Eventual Consistency** for:
  - File synchronization across devices
  - Search indices
  - Analytics and logs

Trade-offs:

- ✅ Best of both worlds
- ✅ Strong guarantees where needed
- ✅ High availability for non-critical data
- ❌ Complex to implement
- ❌ Requires careful design

---

4. **Upload Strategy: Chunked vs Streaming**

Decision: Chunked multipart upload

Rationale:

- Support large files (>5GB)
- Parallel upload for speed
- Resumable uploads (network failures)

- Better progress tracking
- Client can calculate chunk checksums

Trade-offs:

- ✅ Faster for large files (parallel)
- ✅ Resumable after failures
- ✅ Better client experience
- ❌ More complex implementation
- ❌ Requires chunk assembly

---

5. **Deduplication: Client-side vs Server-side**

Decision: Client-side hash, server-side verification

Rationale:

- Reduce bandwidth (don't upload duplicates)
- Save storage costs
- Fast for users (instant "upload")
- Security: server verifies before trusting

Trade-offs:

- ✅ Bandwidth savings (up to 30%)
- ✅ Storage savings (single copy)
- ✅ Faster uploads for duplicates
- ❌ Privacy concerns (hash reveals content)
- ❌ Complexity in reference counting

---

6. **Sync: Long Polling vs WebSocket vs Server-Sent Events**

Decision: WebSocket for web/desktop, Push notifications for mobile

Rationale:

- Real-time bidirectional communication
- Low latency (milliseconds)
- Single connection for multiple updates
- Battery efficient on mobile with FCM/APNS

Trade-offs:

- ✅ Real-time updates
- ✅ Low latency
- ✅ Efficient for high-frequency changes
- ❌ More complex than polling
- ❌ Requires connection management

- ❌ Firewall/proxy issues

Alternatives:

- **Long Polling**: Simpler, but higher latency and server load
- **SSE**: Good for server→client, but not bidirectional

---

7. **Sharding Strategy**

Decision: Shard by user_id

Rationale:

- User's data stays together (query efficiency)
- Natural data locality
- Easy to implement
- Predictable shard key

Trade-offs:

- ✅ Simple queries (no cross-shard joins)
- ✅ Data locality
- ✅ Easy to implement
- ❌ Hot shards if power users exist
- ❌ Harder to rebalance

Alternatives:

- **Geographic Sharding**: Better for global users, but complex
- **Hash Sharding**: Better distribution, but loses locality

---

8. **Caching Strategy**

Decision: Multi-layer cache (Redis + CDN)

Layers:

1. **CDN Cache**: Static files, download URLs (edge locations)
2. **Redis Cache**: Metadata, session, quota (application layer)
3. **Database Cache**: Query results (PostgreSQL buffer)

Trade-offs:

- ✅ Reduced database load (80%+ cache hit rate)
- ✅ Lower latency (sub-millisecond)
- ✅ Better scalability
- ❌ Cache invalidation complexity
- ❌ Additional infrastructure cost

---

9. **Conflict Resolution**

Decision: Last-Write-Wins with conflict copies

Rationale:

- Simple to implement
- Works for 99% of cases
- User has final say
- Preserves all data

Flow:

```
1. Detect conflict (file modified on 2 devices)
2. Keep version with latest timestamp (LWW)
3. Create "file (conflicted copy from Device A).txt"
4. Notify user to review
```

Alternatives:

- **Operational Transform**: Complex, for real-time collaboration
- **Version Vectors**: More accurate, but complex
- **Manual Resolution**: Better accuracy, worse UX

---

10. **Database Replication**

Decision: Primary-Replica with read replicas

Setup:

- 1 Primary (writes)
- 3 Read Replicas (reads)
- Async replication
- Failover to replica on primary failure

Trade-offs:

- ✅ Read scalability (3x capacity)
- ✅ High availability
- ✅ Geographic distribution
- ❌ Replication lag (eventual consistency for reads)
- ❌ Failover complexity

Advanced Features

1. **Collaborative Editing**

Requirements:

- Multiple users edit same document simultaneously
- See real-time changes
- No conflicts

Design:

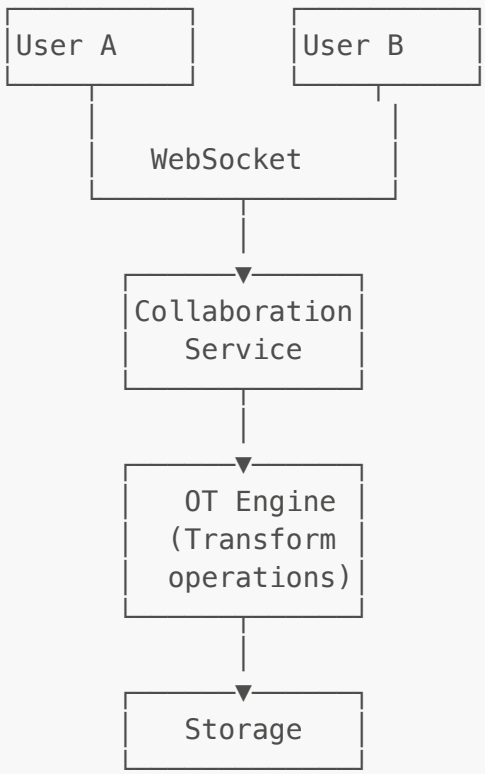
Operational Transformation (OT):

```
User A types "Hello" at position 0
User B types "World" at position 0

Without OT: Conflict
With OT: Transform operations to maintain intent
Result: "HelloWorld" or "WorldHello" (depending on order)
```

Technology:

- WebSocket for real-time communication
- OT algorithm (Google Docs approach)
- Conflict-free Replicated Data Type (CRDT) alternative

Architecture:



2. **Smart Suggestions & AI Features**

Features:

- Auto-categorization (tag files)
- Smart search (semantic search)
- Duplicate detection
- Storage optimization suggestions

Implementation:

```
1. ML Model Training:
   - Train on file metadata
   - Learn user patterns

2. Inference:
   - Run on new uploads
   - Generate suggestions

3. Storage:
   - Store in metadata DB
   - Cache in Redis
```

---

3. **Offline Mode**

Requirements:

- Access files without internet
- Sync when online
- Conflict resolution

Design:

Local Storage:

```
Client maintains local database:
- SQLite for metadata
- Local filesystem for files
- Sync queue for pending operations
```

Sync Algorithm:

```
1. Track operations offline (create, update, delete)
2. Store in local queue
3. When online:
   a. Upload local changes
```

```
   b. Download remote changes
   c. Resolve conflicts (LWW + conflict copies)
   d. Update local state
```

---

4. **File Compression**

Strategy:

- Client-side compression (optional)
- Lossless algorithms (gzip, brotli)
- User choice (compress vs speed)

Benefits:

- Reduce bandwidth (30-70% savings)
- Faster uploads
- Storage savings

Trade-offs:

- CPU overhead (compression/decompression)
- Not effective for pre-compressed (images, videos)

---

5. **Smart Sync (Selective Sync)**

Feature:

- User selects which folders to sync
- Save local disk space
- On-demand download

Implementation:

```
User marks folders:
- "Always available offline"
- "Online only"
- "Available offline until space needed"

Client:
- Syncs only selected folders
- Shows placeholders for others
- Downloads on access
```

---

6. **File Recovery (Trash & Versioning)**

Trash:

- Deleted files moved to trash
- 30-day retention
- Restore or permanent delete
- Trash counts toward quota

Versioning:

- Keep last 100 versions
- Or versions from last 30 days
- Automatic for Office docs
- Manual snapshots

Storage Optimization:

- Delta storage (store only differences)
- Compress old versions
- Tier to cold storage

---

7. **Bandwidth Optimization**

Techniques:

1. **Delta Sync:**
    - Only sync changed bytes (rsync-like)
    - Save bandwidth for large file updates
2. **Adaptive Quality:**
    - Lower quality for slow connections
    - Progressive download (thumbnails first)
3. **Compression:**
    - Compress data in transit
    - Brotli for text, H.264 for video
4. **Prefetching:**
    - Predict next file access
    - Preload in background

---

8. **Security Enhancements**

Features:

1. **End-to-End Encryption (E2EE):**
    - User controls encryption keys
    - Server can't read files

- Trade-off: No server-side search/indexing

2. **Zero-Knowledge Architecture:**
    - Server knows nothing about content
    - Higher security, limited features
3. **Audit Logs:**
    - Track all access (who, when, what)
    - Compliance (GDPR, HIPAA)
    - Retention policies
4. **Two-Factor Authentication (2FA):**
    - TOTP (Google Authenticator)
    - SMS backup
    - Biometric (mobile)

---

9. **Admin Features**

Team/Organization Management:

- Team shared folders
- Admin policies (retention, sharing)
- Usage analytics
- Quota management
- User provisioning (SSO integration)

Compliance:

- Legal holds (prevent deletion)
- eDiscovery (search all content)
- Data residency (store in specific regions)
- Audit trails

---

10. **Mobile-Specific Features**

Camera Upload:

- Auto-upload photos/videos
- Background sync
- Wi-Fi only option

Offline Files:

- Mark files available offline
- Smart cache management
- Free up space automatically

Battery & Data Optimization:

- Throttle sync on low battery
- Pause on cellular data
- Compress uploads

Monitoring & Operations

1. **Key Metrics to Monitor**

Application Metrics

```
- Upload Success Rate: Target > 99.5%
- Download Success Rate: Target > 99.9%
- API Latency: P95 < 200ms, P99 < 500ms
- Sync Latency: P95 < 5 seconds
- Search Latency: P95 < 300ms
- Error Rate: < 0.1%
- Concurrent Users: Track peak times
```

Infrastructure Metrics

```
- CPU Utilization: Target 60-70% (room for spikes)
- Memory Usage: Alert > 80%
- Disk I/O: Track IOPS, latency
- Network: Bandwidth usage, packet loss
- Database:
  - Query latency (P95, P99)
  - Connection pool usage
  - Replication lag (< 1 second)
- Cache:
  - Hit rate (target > 80%)
  - Eviction rate
  - Memory usage
```

Business Metrics

```
- Daily Active Users (DAU)
- Storage per user (average, P95, P99)
- Files uploaded per day
- Sharing activity
- Collaboration sessions
- Revenue metrics (if freemium)
```

2. Logging Strategy

Log Levels:

```
ERROR: Service failures, critical errors
WARN: Degraded performance, retries
INFO: Significant events (uploads, shares)
DEBUG: Detailed troubleshooting (dev only)
```

Structured Logging (JSON):

```
{
    "timestamp": "2024-01-01T10:00:00Z",
    "level": "INFO",
    "service": "upload-service",
    "trace_id": "abc123",
    "user_id": "user_123",
    "action": "file_upload",
    "file_id": "file_456",
    "duration_ms": 1250,
    "status": "success"
}
```

Centralized Logging:

- **Technology**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Alternative**: Splunk, Datadog, CloudWatch Logs
- **Retention**: 30 days hot, 1 year warm, 7 years cold

---

3. Alerting Strategy

Alert Levels:

1. **Critical (Page Immediately):**
    - Service down (health check fails)
    - Database primary failure
    - Error rate > 1%
    - Data loss detected
2. **High (Page during business hours):**
    - Latency spike (P95 > 1 second)
    - Disk space > 90%
    - Replication lag > 10 seconds

3. **Medium (Email/Slack):**
    - Cache hit rate < 60%
    - Upload success rate < 99%
    - Unusual traffic patterns
4. **Low (Daily digest):**
    - Usage trends
    - Cost anomalies

Alert Routing:

```
Critical → PagerDuty → On-call engineer
High → PagerDuty (business hours)
Medium → Slack channel
Low → Email digest
```

---

4. **Distributed Tracing**

Technology: Jaeger, Zipkin, AWS X-Ray

Purpose:

- Track request flow across services
- Identify bottlenecks
- Debug failures

Example Trace:

```
Upload Request [500ms total]
├─ API Gateway [5ms]
├─ Auth Service [10ms]
├─ Upload Service [400ms]
│  ├─ Generate URLs [5ms]
│  ├─ Upload to S3 [380ms] ← Bottleneck
│  └─ Update Metadata [15ms]
└─ Return Response [85ms]
```

---

5. **Health Checks**

Endpoint: `GET /health`

Response:

```
{
    "status": "healthy",
    "timestamp": "2024-01-01T10:00:00Z",
    "version": "1.2.3",
    "checks": {
        "database": {
            "status": "healthy",
            "latency_ms": 5
        },
        "cache": {
            "status": "healthy",
            "hit_rate": 0.85
        },
        "storage": {
            "status": "healthy"
        },
        "message_queue": {
            "status": "degraded",
            "lag": 150
        }
    }
}
```

Types:

- **Liveness**: Is service running?
- **Readiness**: Can service handle traffic?
- **Deep Health**: Check dependencies

---

6. **Disaster Recovery**

RPO (Recovery Point Objective): < 1 hour (data loss tolerance)
RTO (Recovery Time Objective): < 4 hours (downtime tolerance)

Backup Strategy:

```
1. Database:
   - Continuous WAL archiving
   - Daily full backups
   - Hourly incremental
   - Multi-region replication

2. Object Storage:
   - Versioning enabled
   - Cross-region replication
   - Lifecycle policies

3. Metadata:
```

```
   - Daily exports to S3
   - Keep 30 days
```

DR Runbook:

```
1. Detect failure (monitoring alerts)
2. Assess impact (scope of failure)
3. Activate DR plan
4. Failover to secondary region
5. Verify data integrity
6. Update DNS (route traffic)
7. Monitor recovery
8. Post-mortem analysis
```

---

7. **Capacity Planning**

Monitoring:

- Track growth rates (daily, weekly, monthly)
- Predict future needs (6-12 months)
- Plan infrastructure expansions

Key Questions:

- When will we hit 80% capacity?
- What's the cost of next tier?
- Can we optimize before scaling?

Scaling Triggers:

```
Storage: > 70% used → Add capacity
Database: QPS > 80% capacity → Add read replica
Cache: Hit rate < 60% → Increase size
API: P95 latency > 500ms → Add servers
```

---

8. **Cost Optimization**

Strategies:

1. **Storage:**
    - Lifecycle policies (move to cold storage)
    - Compression (reduce size)
    - Deduplication (single copy)
    - Delete old versions

2. **Compute:**
    - Auto-scaling (scale down off-peak)
    - Reserved instances (predictable load)
    - Spot instances (async workers)
3. **Bandwidth:**
    - CDN caching (reduce origin load)
    - Compression (reduce transfer)
    - Smart routing (cheapest path)
4. **Database:**
    - Query optimization (reduce load)
    - Connection pooling (reuse connections)
    - Read replicas (distribute load)

Cost Monitoring:

- Tag resources by team/project
- Track cost per user
- Set budgets and alerts
- Regular cost reviews

---

9. **Security Operations**

Continuous Monitoring:

- Failed login attempts (brute force detection)
- Unusual access patterns (compromised accounts)
- Data exfiltration (large downloads)
- Permission changes (audit trail)

Incident Response:

```
1. Detect: Automated alerts, user reports
2. Contain: Disable account, revoke tokens
3. Investigate: Analyze logs, identify scope
4. Remediate: Patch vulnerability, reset passwords
5. Recover: Restore affected data
6. Post-Mortem: Document and improve
```

Security Audits:

- Quarterly penetration testing
- Annual security certifications (SOC 2, ISO 27001)
- Compliance audits (GDPR, HIPAA)

10. **Performance Testing**

Load Testing:

- Simulate peak traffic (3x average)
- Identify bottlenecks
- Validate scaling plans

Stress Testing:

- Push beyond limits
- Find breaking points
- Test graceful degradation

Chaos Engineering:

- Randomly kill services
- Test fault tolerance
- Validate failover mechanisms

Tools:

- JMeter, Locust (load testing)
- Chaos Monkey (chaos engineering)

Summary & Interview Tips

Key Talking Points for Interviews

1. **Start with Requirements Clarification:**
    - Ask about scale (users, storage)
    - Functional vs non-functional requirements
    - Read/write ratio
    - Consistency requirements
2. **Discuss Trade-offs:**
    - SQL vs NoSQL for metadata
    - Strong vs eventual consistency
    - Chunked vs streaming uploads
    - Client-side vs server-side processing
3. **Highlight Scalability:**
    - Database sharding strategies
    - Caching at multiple layers
    - CDN for global distribution
    - Async processing for heavy tasks

4. **Address Reliability:**
    - Replication and redundancy
    - Failure detection and recovery
    - Data durability guarantees
    - Backup and disaster recovery
5. **Security Considerations:**
    - Authentication and authorization
    - Encryption (at rest and in transit)
    - Access control and permissions
    - Audit logging
6. **Mention Real-World Challenges:**
    - Conflict resolution in sync
    - Deduplication strategies
    - Large file handling
    - Mobile optimization

Common Follow-up Questions

Q: How do you handle files larger than 5GB?
A: Multipart upload with chunking (5MB chunks), parallel upload, resumable uploads, and progress tracking.

Q: How do you prevent data loss?
A: 3x replication across regions, versioning, continuous backups, WAL archiving, and cross-region replication.

Q: How do you scale the metadata database?
A: Shard by user_id, read replicas for read scalability, caching layer (Redis), and eventual migration to distributed SQL (CockroachDB).

Q: How do you handle concurrent edits?
A: Operational Transform for collaborative editing, or Last-Write-Wins with conflict copies for simple cases.

Q: How do you optimize costs?
A: Deduplication, compression, lifecycle policies (tiering to cold storage), and auto-scaling.

Q: How do you ensure global performance?
A: Multi-region deployment, CDN for downloads, geographic routing, edge caching, and regional read replicas.

Conclusion

This system design covers all major aspects of building a global file storage and sharing system like Google Drive. Key takeaways:

1. **Microservices Architecture**: Separate concerns (upload, download, metadata, sync, search)
2. **Horizontal Scalability**: Shard databases, use object storage, add read replicas
3. **High Availability**: Multi-region, replication, failover mechanisms
4. **Strong Security**: Encryption, access control, audit logging
5. **Optimized Performance**: Caching, CDN, compression, chunking
6. **Reliability**: Versioning, backups, monitoring, alerts

The design balances complexity with practicality, making trade-offs based on requirements and scale. It can handle billions of users and petabytes of data while maintaining performance and reliability.

Good luck with your system design interview! 🚀