Design: Scalable File Upload / Download System with Load Balancer

Goals & Requirements

Functional

- Users can upload and download files (images, videos, docs).
- Support large files (up to many GB).
- Support resumable/multipart uploads.
- Provide fast downloads (low latency).
- Support metadata (owner, content-type, size, versions).
- Allow public and private (authenticated) file access.

Non-functional

- High availability, low latency for downloads, high throughput for uploads.
- Scalability (millions of objects, thousands req/s).
- Durability (no single-point data loss).
- Cost-effective storage tiering.
- Secure (auth, encryption).
- Observable and testable.

Assumptions

- Cloud object storage (S3) or on-prem object store (MinIO) available.
- CDN can be used for delivery.
- System should be stateless where possible so it can scale behind LBs.

High-level Architecture (components)

```
Client
  ├─> CDN (cache static files, edge)
  └─> Load Balancer (L7: Nginx/HAProxy/ALB)
       ├─> API Gateway / Auth Service
       ├─> Upload Service (stateless)
       ├─> Metadata Service (SQL/NoSQL)
       ├─> Processing Workers (transcode, thumbnail)
       ├─> Job Queue (Kafka/RabbitMQ/SQS)
       └─> Object Storage (S3 / MinIO / GFS)
```

Notes

- CDN sits in front for downloads (edge caching). For uploads, clients may upload directly to S3 via **pre-signed URLs** to bypass server bandwidth, with progress reported to backend.
- Load Balancer routes API traffic to multiple instances of Upload/Download services. Health checks and autoscaling applied.
- Metadata DB holds file records, ownership, ACLs, versions, storage location, processing status.
- Optional Redis caches metadata and access tokens.
- Workers handle background processing (thumbnails, virus scan, format conversion), consume events from message queue.
- Monitoring & logging: Prometheus, Grafana, ELK/Fluentd.

Data Model (example)

Table: Files

- file_id (UUID / PK)
- owner_id (user id)
- filename
- content_type
- size_bytes
- storage_key (S3 key)

- storage_class (STANDARD / INFREQUENT / GLACIER)
- versions (if versioning)
- checksum (SHA256)
- upload_status (pending/complete/failed)
- created_at, updated_at
- acl (public/private, link-expiry)
- thumbnail_keys (map)
- tags (json)

Table: UploadSessions (for multipart/resumable)

- session_id
- file_id
- owner_id
- parts_uploaded (list or count)
- expected_parts
- expires_at
- created_at

Cache keys (Redis)

- `file_meta:<file_id>` => serialized metadata
- `upload_session:<session_id>` => session info, expiry

API Design (REST example)

1. **Initiate upload (server-managed or presigned)**

- `POST /v1/uploads/initiate`

- request: { filename, content_type, size, storage_class, public:boolean }
- response (server-managed): { upload_id, presigned_urls: [..] }
- response (single-part): { upload_id, presigned_url }

2. **Upload a file part (direct to S3)**

- Client PUTs to `presigned_url` returned. Server not involved in data path.

3. **Complete upload**

- `POST /v1/uploads/{upload_id}/complete`
  - server validates parts, stores final metadata, sets `upload_status=complete`.

4. **Get download URL**

- `GET /v1/files/{file_id}/download?expiry=300`
  - returns presigned URL (or CDN URL). If public => direct CDN URL.

5. **Get metadata**

- `GET /v1/files/{file_id}`

6. **List files**

- `GET /v1/users/{user_id}/files?page=..`

7. **Delete file**

- `DELETE /v1/files/{file_id}`

8. **Generate thumbnail / transforms**

- `POST /v1/files/{file_id}/transform {size, format}`
  - returns URL when processed (async via queue).

Auth

- Use JWT/OAuth at API gateway. Only presigned URLs are time-limited and scoped.

Upload Patterns

1) Direct-to-object-storage (recommended)

- Client requests presigned URL(s) from Upload Service.
- Client uploads parts directly to object storage (S3) using multipart upload.
- Upon completion, client calls server to finalize — server verifies checksums and writes metadata.

Advantages

- Server CPU/network spared for data path.
- Scales well.

Implementation details

- For large files, use S3 multipart API: create multipart upload => get presigned URLs for each part => upload parts => complete multipart upload.

2) Upload via application servers

- Client uploads to an app server behind LB.
- App server streams to object storage.

Use when

- Need server-side virus scanning before storing.
- Need centralized transformations inline.

Disadvantages

- Higher network & CPU cost; app server becomes bottleneck.

3) Resumable uploads

- Use session IDs, track uploaded parts, expiry.
- Support client-side retries on part failures.

Download Flow

1. Client requests download URL.
2. If file public & cached, return CDN URL (fastest).
3. If private, generate presigned S3 URL (short expiry) OR edge-authenticated CDN signed URL.
4. Client downloads from CDN or S3.

If transformations are needed (thumbnails, resized images)

- Return pre-generated transformed versions (stored as separate objects).
- If transformation not present, enqueue processing and return placeholder or generate on-the-fly with caching.

Load Balancer Configuration Examples (conceptual)

Nginx upstream (round robin)

```
upstream upload_backend {
    server 10.0.1.11:8080;
    server 10.0.1.12:8080;
}

server {
    listen 443 ssl;
    server_name api.example.com;
    location /v1/uploads/ {
        proxy_pass http://upload_backend;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Health checks

- LB performs `GET /health` on backend. Backends must return 200 and readiness.

Sticky sessions

- Avoid sticky sessions. Make app stateless; store session in Redis if necessary.

Caching & CDN

- Use CDN (CloudFront/Cloudflare/Fastly) for downloads of static files.
- CDN uses origin at object storage or origin server (if origin needs auth).
- For dynamic content, keep TTL low and use Signed URLs for private content.
- Cache metadata in Redis for hot objects to reduce DB hits.

Security

- Authenticate requests with OAuth2/JWT at API gateway.
- Use HTTPS/TLS everywhere.
- Use presigned URLs with short expiries for uploads/downloads.
- Validate MIME types and file extensions server-side.
- Virus scanning via sandboxed workers before marking file as "trusted".
- Encrypt objects at rest (S3 SSE) and in transit (HTTPS).
- Rate-limit uploads per user/IP to prevent abuse.

Durability & Consistency

- Use multi-AZ replication in object storage (S3 replicated).
- Keep strong metadata consistency in DB (SQL recommended for metadata). For high write scale, use partitioning or NoSQL with versioned writes.
- After multipart complete, ensure metadata write and object finalization are atomic — use transactions or create record after successful storage confirmation.

Background Processing

- Use message queue for tasks: thumbnail generation, transcoding, virus scanning, metadata indexing.
- Workers autoscale independently.

Monitoring & Observability

- Metrics: upload rate, download rate, 4xx/5xx rates, latency, presigned URL generation latency, multipart failures, queue length.
- Logs: access logs at LB (Nginx), app logs, object store events (S3 events).
- Traces: instrument APIs with distributed tracing (OpenTelemetry).
- Alerts: high error rate, high latency, queue backlog > threshold.

Failure Modes & Mitigations

- **LB failure**: run active-active LBs with health checks, use DNS failover or anycast.
- **Worker/back-end failure**: autoscale and use dead-letter queues for failing jobs.
- **Object store I/O issues**: replicate to another region, failover to backup.
- **Partial uploads**: expire and cleanup incomplete multipart uploads after TTL.
- **Hotspot (one object very popular)**: cache at CDN, in-memory caches, scale read capacity.
- **Data corruption**: store checksums, validate on upload and download.

Cost & Optimization

- Store infrequently accessed files in cheaper tiers (GLACIER) and move via lifecycle policies.
- Use presigned URLs and direct uploads to avoid traffic egress through app servers.
- Deduplicate: compute hash (SHA256) and avoid duplicate objects; reference-count metadata.

- Use CDN caching aggressively for static assets.

Deployment & Scaling Strategies

- Horizontal scale API servers behind LB with autoscaling rules (CPU/requests/queue length).
- Separate services: upload-initiate service, metadata service, worker pool, gateway, to scale independently.
- Use blue-green or canary deploy behind LB to avoid downtime.

Interview Questions (30+) with Detailed Answers

Below are focused interview questions that a candidate may be asked about this system, with model answers you can deliver in an interview.

Basics & Architecture

Q1. Give a high-level design for a file upload/download service.
A1. (Answer in 2–3 mins): Describe client → LB → API Gateway → Upload Service → Object Storage + Metadata DB + Processing Workers + CDN. Explain using presigned URLs for direct uploads, CDN for downloads, stateless app servers behind LB, asynchronous processing via queue. Mention security (presigned URLs, auth), durability (replication), and monitoring.

Q2. Why use presigned URLs instead of routing uploads through your servers?
A2. Presigned URLs let clients PUT directly to object storage, saving server network bandwidth and CPU. This reduces cost, improves throughput, and simplifies scaling because app servers are not in the data path. Use servers for auth and metadata updates only.

Q3. How would you support resumable uploads?
A3. Use multipart upload (S3-style) or chunked uploads with upload sessions. Server returns an upload_id and presigned URLs for each part. Client uploads parts and calls complete. Server tracks parts; incomplete sessions expire and are garbage-collected. Support client-side retry and part checksums.

Q4. Which DB would you choose for metadata and why?
A4. Use relational DB (Postgres) if you need strong consistency, joins, and transactional updates (e.g., versions, ownership). For extremely high write scale, use a scalable NoSQL (Cassandra) but ensure you can handle consistency needs; hybrid approach (polyglot) is common.

Load Balancer Specific

Q5. Where in the architecture do you place the load balancer(s)?
A5. Place L7 load balancers (Nginx/HAProxy/ALB) between clients and API servers. For global traffic, use DNS-based/global LB to route to regional LBs. For redundancy, deploy LBs in active-active across AZs with health checks.

Q6. Which LB algorithm would you use? Round robin, least_conn, consistent hashing?
A6. For stateless APIs, round robin or least_conn are fine. For session affinity (if unavoidable) use IP-hash or cookies. For caches (e.g., routing to cache nodes), use consistent hashing to minimize rehashing when nodes change.

Q7. How do you avoid the LB becoming a single point of failure?
A7. Use multiple LB instances behind DNS or anycast; active-active with a managed service (ELB) or use floating IPs with keepalived for failover. Ensure health checks and automatic failover.

Upload/Download Flow & Optimizations

Q8. Explain step-by-step how a large file upload would work.
A8.

1. Client calls `POST /uploads/initiate` with metadata.
2. Server allocates `upload_id`, creates multipart upload at S3, returns presigned URLs for parts.
3. Client PUTs parts directly to S3 using those URLs. Each part uploaded can be retried.
4. Client calls `POST /uploads/{upload_id}/complete`.
5. Server validates ETags / checksums, finalizes metadata, marks file `upload_status = complete`.
6. Optionally enqueue tasks for virus scanning, transcoding.
7. CDN caches the file on subsequent downloads.

Q9. How to serve files with low latency globally?
A9. Use CDN to cache objects at edge nodes. For private content, use CDN signed URLs (CloudFront signed URLs). Keep TTL for static files high and use versioned keys to invalidate.

Q10. How do you ensure upload integrity?
A10. Compute checksums client-side (MD5/SHA256) and validate after upload. For multipart, compute per-part checksum and verify during `complete multipart upload`. Store checksum in metadata for later verification.

Consistency, Durability & Failure Handling

Q11. What happens if metadata write succeeds but storing file fails (or vice versa)?
A11. Use an idempotent finalize flow: finalize only after confirming object store returns success. If metadata write happens first, mark file as `pending` and have a reconciliation job that checks for files in inconsistent states and retries or cleans up. Consider two-phase commit-like patterns or store event logs and process asynchronously.

Q12. How to handle partially uploaded sessions?
A12. Use TTL for upload sessions; periodically run a cleanup job to abort incomplete multipart uploads, free storage, and mark session expired.

Q13. How to ensure durability (no data loss)?
A13. Use object storage with multi-AZ replication, versioning, and lifecycle policies. Periodic backups of metadata DB. Use checksums and integrity checks.

Security & Privacy

Q14. How to provide private file access?
A14. Use signed URLs with short expiry generated by the backend after auth/authorization checks. Alternatively, use signed cookies or CDN signed URLs for edge distribution.

Q15. How to prevent malicious uploads?
A15. Validate file types and sizes, scan files for malware using sandboxed workers, enforce rate limiting, do content-disposition checks, and sanitize filenames.

Q16. Should uploaded files be encrypted?
A16. Yes — encrypt at rest (SSE-S3, KMS) and enforce TLS for transit. For extra security, use client-side encryption and server-side key management.

Scalability & Performance

Q17. Where are the scalability bottlenecks and how to mitigate them?
A17.

- App servers: scale horizontally behind LB.
- Metadata DB: shard/partition or use read replicas and caching (Redis).
- Object store: scale via cloud provider or distributed object store, store in multiple regions.
- Network: use CDN and direct-to-storage uploads to avoid server bandwidth bottlenecks.

Q18. How would you handle "hot" objects (one file very popular)?
A18. Cache at CDN and edge; use in-memory cache for metadata; replicate to multiple origins; use rate limiting if abuse suspected.

Operational / Monitoring

Q19. What metrics would you expose?
A19. Upload/download RPS, latency (p95/p99), error rate (4xx/5xx), number of active upload sessions, queue lengths, worker error rates, storage cost per day, bandwidth egress, multipart failures.

Q20. How to test this system before production?
A20. Load test with k6 / Locust for uploads/downloads, chaos test region failures, run failure injection (network partition), simulate partial uploads, check correctness with checksum validations.

Storage, Lifecycle & Cost

Q21. How do you manage storage costs?
A21. Use lifecycle policies to move cold data to cheaper storage tiers (infrequent/archival). Deduplicate by hashing content. Use compression/format conversion where applicable. Chargeback per user if multi-tenant.

Q22. When to use object storage vs file system?
A22. For massive, distributed, HTTP-accessible objects, use object storage (S3) due to scalability and metadata. Use file system (NFS) if you need POSIX semantics and low-latency filesystem operations.

Advanced Features

Q23. How to support versioning and rollback?
A23. Keep versioned keys in object store or enable object-store versioning. On update, write new key and mark previous as older version. Use metadata to reference current version. Enable lifecycle rules for cleanup.

Q24. How to implement deduplication?
A24. Compute file hash on upload initiation. If hash exists, link to existing storage object and increment reference count. Watch out for hash collision (very rare for SHA256).

Q25. How do you perform server-side image transformations at scale?
A25. Use event-driven workers or serverless functions triggered by upload events to generate thumbnails and store transformed outputs. For on-demand transforms, use a transform service + cache results at CDN.

Troubleshooting / Edge Cases

Q26. If users report downloads failing intermittently, where do you inspect?
A26. Check CDN logs, origin availability, LB health, app logs, S3 access logs, edge errors, and presigned URL expiry mismatches. Also monitor network egress quotas and firewall rules.

Q27. How to reconcile mismatched metadata and storage?
A27. Run periodic reconciliation job that lists object-store keys and cross-references DB metadata. For missing objects, mark metadata invalid and notify owners or attempt recovery from backups.

Q28. What to do if CDN cache is out-of-date after file update?
A28. Use cache invalidation (purge), short TTLs for frequently updated files, or versioned object keys to avoid invalidation.

System Design Questions (Hard / Whiteboard style)

Q29. Design a system to support 10k concurrent uploads/sec.
A29. Summarize scaling: front with global LB, regional upload endpoints, direct-to-object-storage via presigned URLs, multipart uploads, auto-scaling metadata services and DB sharding, SQS/Kafka for background processing, aggressive CDN for downloads, use rate-limiting & quotas, monitor and autoscale worker pools.

Q30. How to guarantee exactly-once processing for background tasks (e.g., virus scan) that may fail and retry?
A30. Design idempotent worker operations by using unique job IDs, track processed job IDs in a dedupe store (Redis with TTL or DB). Use at-least-once delivery with idempotency, or use transactional outbox patterns.

Q31. Explain how you would implement signed URLs for CDN.
A31. Use edge provider signed URL mechanism (CloudFront signed URLs) where backend signs URL with private key and sets expiry and headers; CDN validates signature before serving content. Alternatively, use signed cookies for multiple objects.

Behavioral / Tradeoff Questions

Q32. If asked to choose SQL vs NoSQL for metadata, what do you ask?
A32. Ask: read/write QPS, transaction needs (multi-record atomicity), query patterns (joins, complex queries), expected scaling growth, and whether strong consistency is required. Choose SQL for relational transactional needs; NoSQL for scale and flexible schema but design for consistency.

Q33. What tradeoffs do presigned URLs introduce?
A33. Pros: bandwidth offload, scalability. Cons: complexity for auth/expiry, potential for leaked URLs, limited server-side validation before storing (unless we enforce post-upload validation). Use short expiry and server-side validation on complete.

Implementation / Ops Questions

Q34. How do you roll out a backward-incompatible metadata schema change?
A34. Use DB migration with backward compatibility: add new fields defaulted, update services to write both old/new, use feature flags for read path, gradually migrate clients, then drop old fields once safe.

Q35. How to implement observability for upload latency?
A35. Instrument request lifecycle: client-to-LB time, LB-to-backend time, presigned URL generation time, multipart upload time, finalization time. Expose histograms for p50/p95/p99 and trace a request using distributed tracing.

Extra practical / code-level questions

Q36. Provide Nginx snippet to proxy /health and upstream.
A36.

```
upstream api_backend {
  server 10.0.1.11:8080;
  server 10.0.1.12:8080;
  health_check;
}

server {
  listen 80;
  location /health {
    proxy_pass http://api_backend/health;
  }

  location /v1/ {
    proxy_pass http://api_backend;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

Q37. How would you test multipart upload correctness?
A37. Unit tests for part ETag validations, integration tests with S3 multipart APIs, failure injection for part uploads, end-to-end upload+complete tests and checks that checksum matches.

Interview wrap-up questions

Q38. What are the most important trade-offs you made in your design?
A38. Offloading data path to object store (presigned URLs) trades server control for scalability; we use background workers for validation to restore security. Using CDN favors read-latency at cost of cache invalidation complexity. Using relational metadata DB gives strong consistency but requires sharding for huge scale.

Q39. If the company must avoid any third-party cloud provider (no S3), what changes?
A39. Use self-hosted object store (MinIO) with multi-node replication and erasure coding, ensure robust backup/restore, build regional replication, and ensure network egress and bandwidth planning for scale.

Q40. How would you evolve this system for live streaming?
A40. For live streaming, integrate ingest servers (RTMP/WebRTC) → chunk-based HLS/DASH generation → CDN for distribution with low-latency edge caching and manifest updating. Add per-segment storage and retention rules.

Quick checklist to present in interview (short)

- Show a clear high-level diagram.
- State requirements (functional, non-functional).
- Describe data flow for upload and download.
- Explain presigned URL flow + multipart upload.
- Cover metadata model and transactionality.
- Explain LB placement, algorithm choice, and redundancy.
- Talk about CDN, caching, and cache invalidation/versioning.
- Discuss security: auth, signed URLs, virus scan.
- Discuss scalability, partitioning, autoscaling rules.
- Discuss monitoring, failure modes, and cost optimization.

Practice prompts (use these to rehearse)

1. Draw the architecture and explain how a 2 GB file is uploaded from mobile with a flaky network.
2. Explain data race/consistency when two clients try to upload same filename simultaneously.
3. Explain how you would migrate existing files from on-prem storage to cloud object storage with zero downtime.
4. Walk through the sequence when a worker fails while processing a newly uploaded file (transcoding).
5. How to provide audit trail for file downloads (who downloaded what and when)?