What are BLOBs?

BLOB = Binary Large Object

A BLOB is raw binary data, for example:

Images (JPEG, PNG)
Videos (MP4)
Audio (MP3)
PDFs
ZIP files

In databases, BLOBs are stored as:
BLOB, BYTEA, VARBINARY, etc.

```
CREATE TABLE users (
  id INT,
  profile_pic BLOB
);
```

At first glance, this looks convenient:

"User data + image in one place"

But this breaks down very fast at scale.

Why Storing BLOBs in a Database Is a Bad Idea ?

1. Databases Are Optimized for Structured Data

Databases are built for:

Rows & columns
Indexes
Queries
Joins
Transactions

They are not optimized for large binary payloads.

When you store BLOBs:
Indexes become useless
Pages get bloated
Cache efficiency drops

You're using a Ferrari as a truck.

2. Performance Problems

Let's say:
Profile image = 5 MB
User table row = 1 KB

Now every time you do:
`SELECT * FROM users WHERE id = 123;`

The DB:
Reads the entire row
Pulls the BLOB into memory
Sends it over the network
Even if you don't need the image.

Result:
Higher latency
More memory usage
Slower queries

3. Backup & Restore Become Nightmares

Databases need:

Regular backups
Replication
Point-in-time recovery

With BLOBs:

Backup size explodes
Restore takes hours
Replication lag increases

4. Scaling Becomes Extremely Expensive

To scale a DB:

Vertical scaling (bigger machine)
Sharding (very complex)

Binary data:
Increases storage cost
Increases I/O
Increases replication traffic

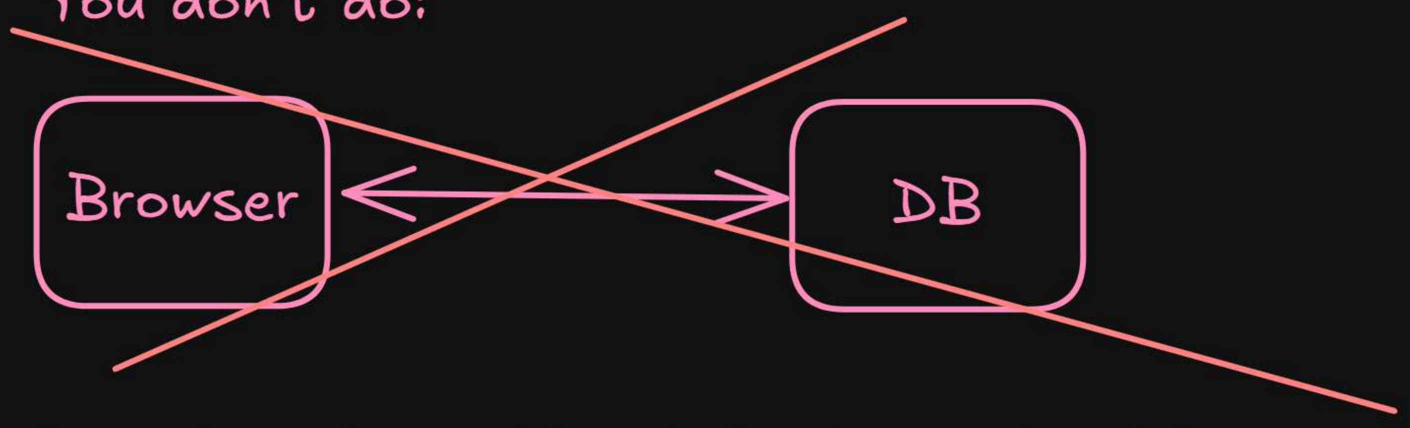
DBs are the expensive storage in your system.
Using them for images/videos is burning money.

5. Databases Are Stateful & Coupled

Databases:

Are tightly coupled to applications
Cannot be easily accessed directly by clients
Don't integrate well with CDNs

You don't do:



That alone disqualifies DBs for media delivery.

The Correct Pattern (Industry Standard)

Image → Object Storage
Metadata → Database

Why We Need Object Storage ?

We want storage that is:

Cheap

Scalable

Durable

Accessible over HTTP

CDN-friendly

Not tied to DB constraints

This is exactly why Object Storage exists.

What Is Object Storage?

Object Storage stores data as objects, where each object contains:

Data (binary content)
Metadata (key–value info)
Object ID (key)
Objects live inside buckets and are accessed using HTTP APIs.

Example:

bucket: user-uploads
key: profile-pics/aniket.png

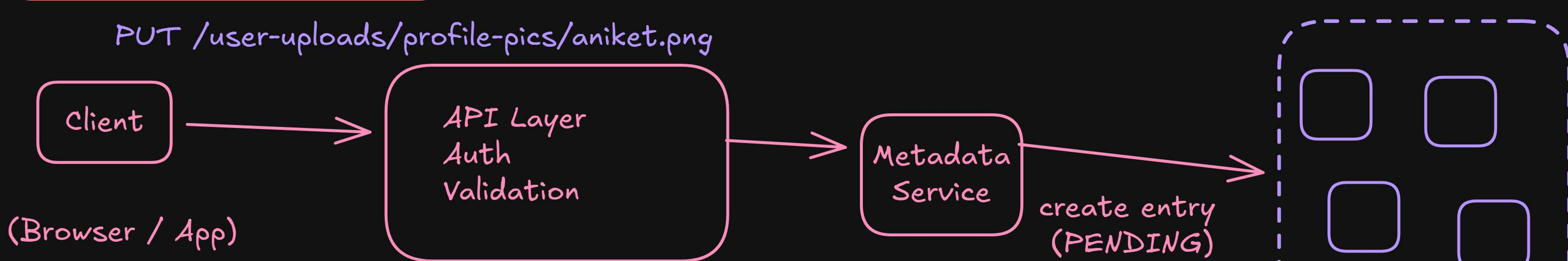
Examples of Object Storage

Amazon S3

Google Cloud Storage

Azure Blob Storage

Upload Flow (PUT Object)



Step 1: Request Comes In

API Layer checks:

Auth (IAM / token)
Bucket exists?
Permission?
Size limits?

Step 2: Metadata Entry Created

Metadata DB:

key	size	status	locations
img1	5MB	PENDING	NULL

Why PENDING?

Upload not finished yet

Step 3: Object Is Chunked

Object (20MB) | C1 | C2 | C3 | C4 |

Chunks allow:

Parallel upload
Retry on failure
Large object support

Step 4: Chunks Distributed to Nodes

Step 5: Durability Applied (Replication)

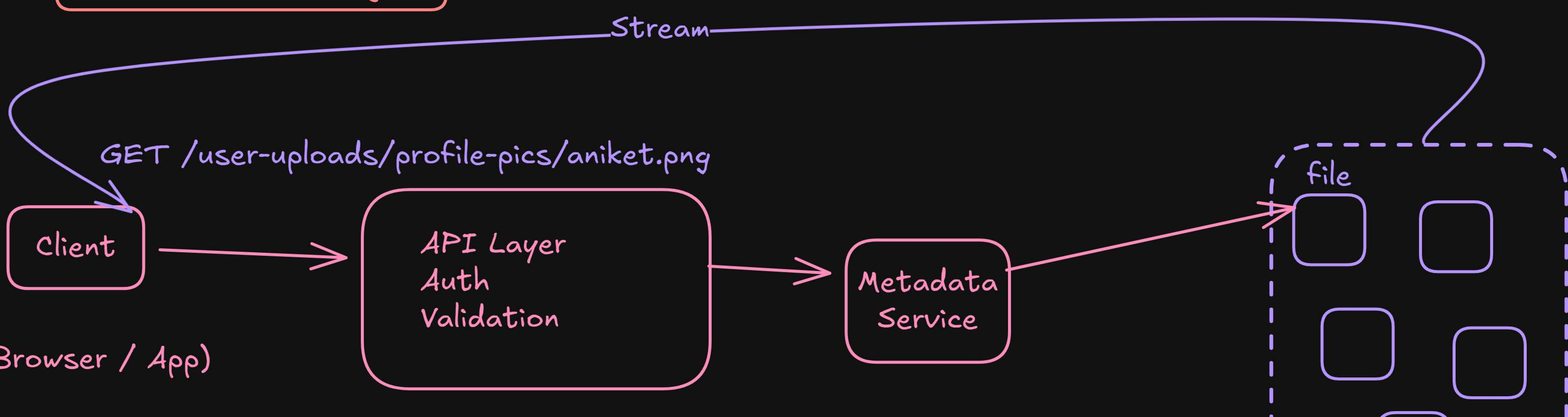
Step 6: Metadata Updated (COMMITTED)

Metadata DB:

key	size	status	locations
img1	5MB	READY	NULL

Only now client gets 200 OK.

Read Flow (GET Object)



Step 1: Lookup Metadata

key → [Node A, Node B, Node C, Node D]

Step 2: Fetch Chunks (Parallel)

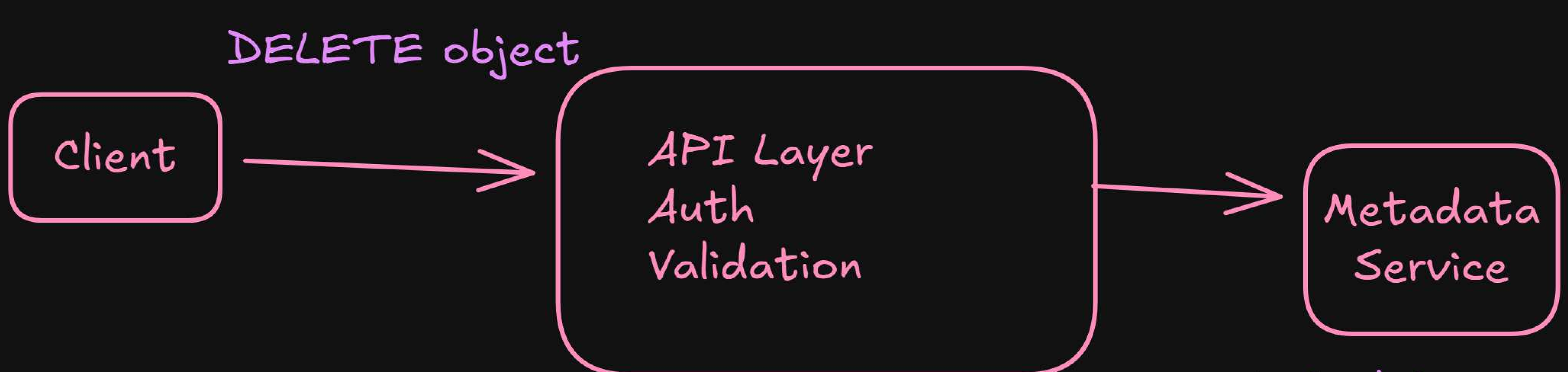
Node A -> C1
Node B -> C2
Node C -> C3
Node D -> C4

Step 3: Reassemble Object

C1 + C2 + C3 + C4 = Full Object

Step 4: Stream to Client

Delete Flow



Metadata:
status = DELETED

Actual data deletion:

Happens asynchronously

Garbage collector removes chunks later

Metadata Is the Brain

```
Metadata Service
├── Object key
├── Size
├── Version
├── ACL
├── Checksums
└── Data locations
```

If metadata is down:
Storage is down (even if data exists)

Versioning (Overwrite Case)

PUT img.png (v1)
PUT img.png (v2)
Metadata:

```
img.png → v2 (active)
          v1 (older)
```

Delete:

DELETE img.png
→ delete marker added

Multipart Upload (Large Files)

```
Initiate Upload
   |
Upload Part 1
Upload Part 2
Upload Part 3
   |
Complete Upload
```

```
Part1 --> Node A
Part2 --> Node B
Part3 --> Node C
```

Benefits:

Resume

Parallelism

Reliability

Core Characteristics

1. Immutability

Objects cannot be modified

You overwrite or delete + re-upload

2. Flat Namespace

No real folders
/photos/2025/img.png is just a string key

Folders are:

UI illusion
Prefix-based filtering

3. Globally Unique Object Key

Bucket + object key

Example:

s3://user-uploads/profile-pics/aniket.png

Durability (Why Your Data Never Gets Lost)

Object storage achieves insane durability using:

-> Replication

Multiple full copies

Simple but costly

-> Erasure Coding (Preferred)

Data split into k + m blocks

Lose some blocks → still recover

This is how services like Amazon S3 achieve
99.999999999% durability (11 nines).

Security

Encryption

At rest (AES-256)
In transit (TLS)

Access Control

Bucket policies
Object ACLs
IAM roles

Pre-Signed URLs

Temporary access

Used for uploads/downloads without exposing credentials

Performance Characteristics

Not for:

Low-latency random writes
Frequent updates
Databases

Good for:

Large objects
Streaming
Read-heavy workloads
CDN integration

Common Use Cases

Media hosting (video/audio/images)

Backups & snapshots

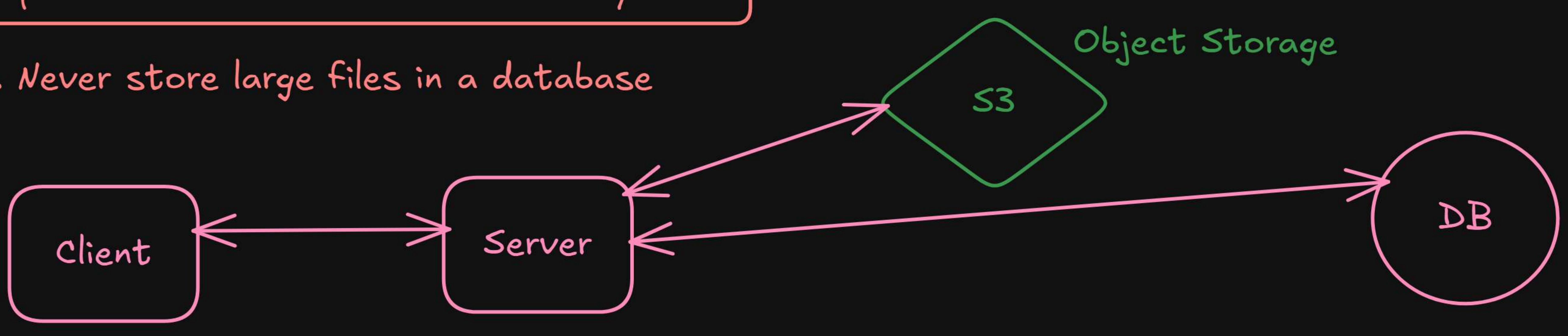
Logs & analytics

Data lakes

ML training datasets

Tips for Interviews and real-world systems

1. Never store large files in a database



Instead, we split responsibilities:

Object Storage

Stores:
Images
Videos
PDFs
Large binary files

Example:
Post image
Video file
Backup file

Database

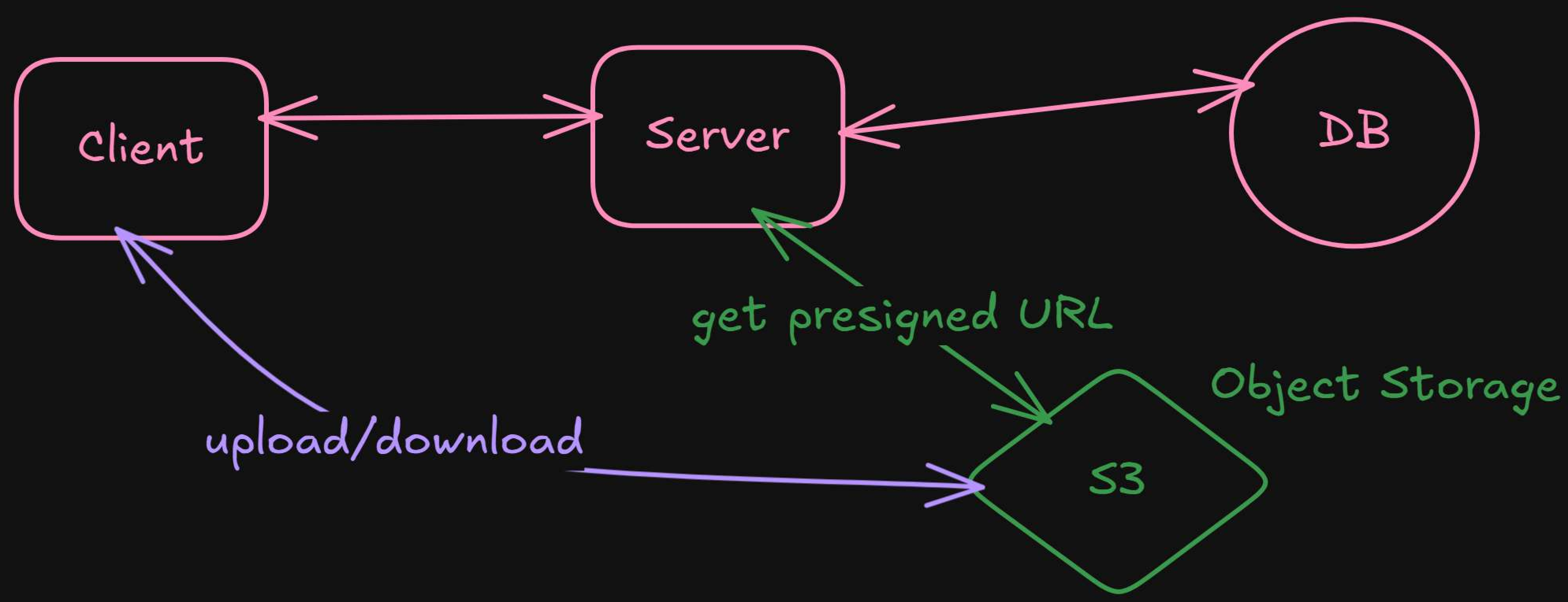
Stores metadata only:

Post
- id
- creatorId
- text
- linkToObjectStore

That linkToObjectStore is usually:

S3 object key
Or a URL

2. Direct upload to Object Storage using pre-signed URLs



1. Client asks server:
"I want to upload a file"

2. Server:
Authenticates user
Generates a pre-signed URL from object storage

3. Server sends URL back to client

4. Client uploads directly to object storage

This is how:

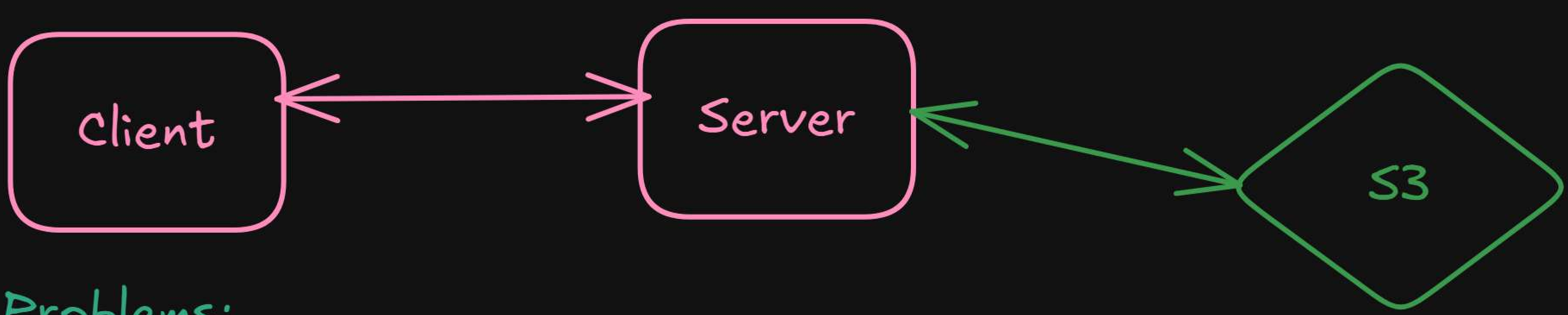
Instagram
Netflix uploads
Google Drive
work.

In real systems

With Amazon S3:

Server creates a pre-signed URL
Browser uploads directly to S3
Server never touches the file bytes

Bad approach:



Problems:

Server bandwidth bottleneck
High memory usage
Poor scalability

3. Large files are uploaded in chunks (multipart upload)

Why chunked upload is required

Large files (like 5–10 GB):

Cannot be uploaded reliably in one go
Network failures are common

So object storage:

Accepts file in multiple parts
Stores parts independently
Combines them at the end