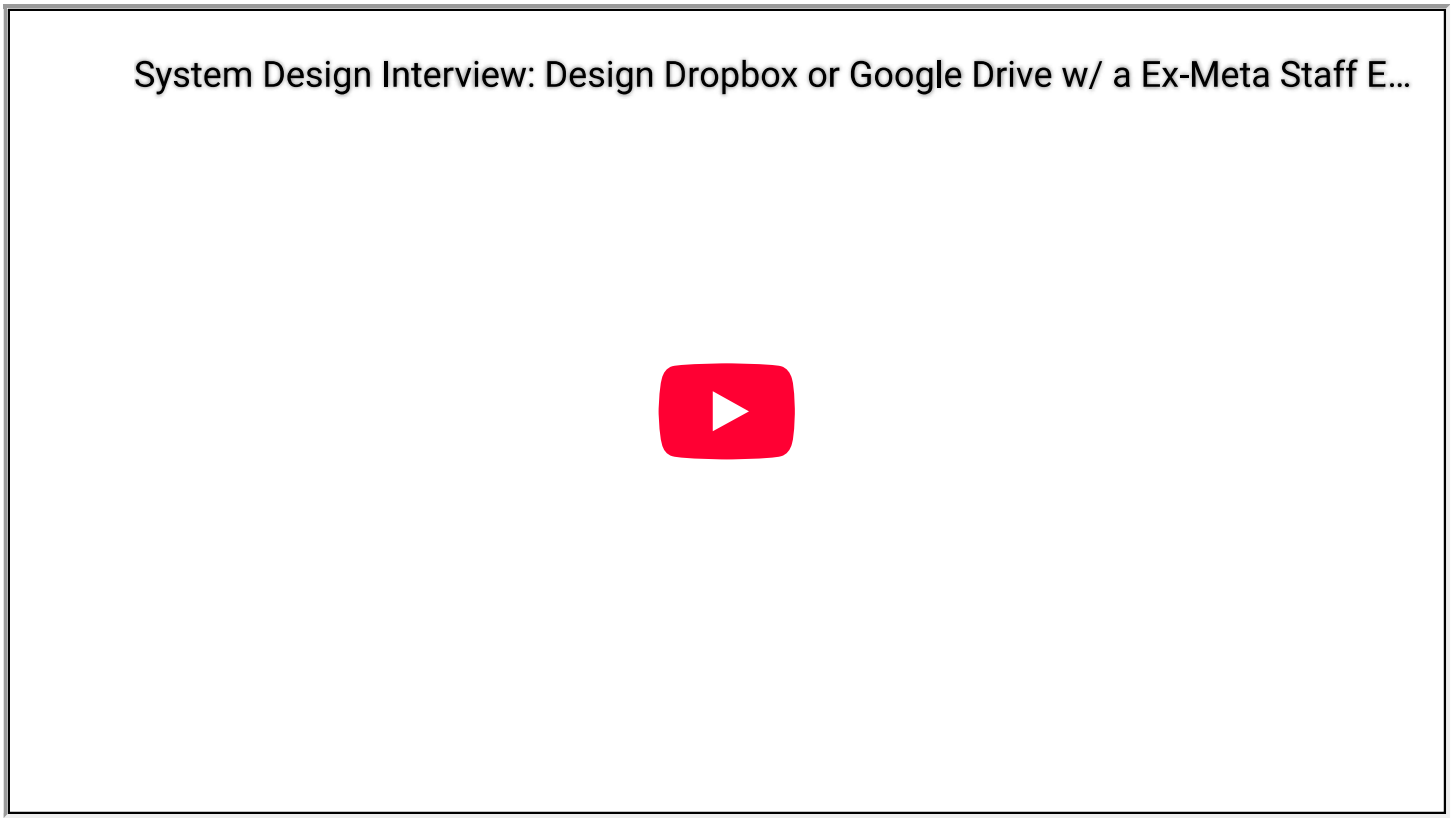
Common Problems

Design a File Storage Service Like Dropbox

Evan King
Ex-Meta Staff Engineer

Practice This Problem

easy 35 min



Understanding the Problem

What is Dropbox

Dropbox is a cloud-based file storage service that allows users to store and share files. It provides a secure and reliable way to store and access files from anywhere, on any device.

Functional Requirements

Core Requirements

1. Users should be able to upload a file from any device
2. Users should be able to download a file from any device
3. Users should be able to share a file with other users and view the files shared with them

4. Users can automatically sync files across devices

Below the line (out of scope):

- Users should be able to edit files
- Users should be able to view files without downloading them

> It's worth noting that there are related system design problems around designing Blob Storage itself. This is out of scope for this problem, but you may consider doing some research on your own to understand how Blob Storage works and how it's designed.

Non-Functional Requirements

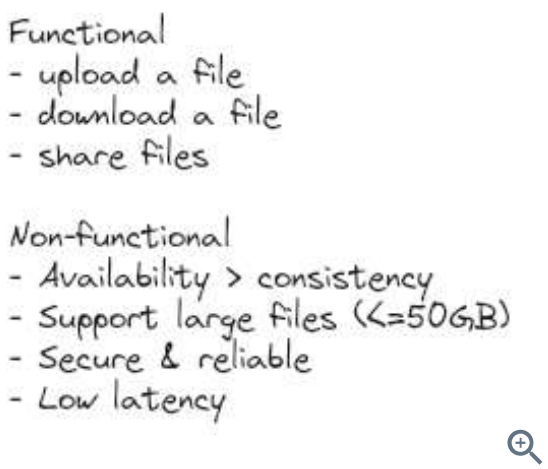
Core Requirements

1. The system should be highly available (prioritizing availability over consistency).
2. The system should support files as large as 50GB.
3. The system should be secure and reliable. We should be able to recover files if they are lost or corrupted.
4. The system should make upload, download, and sync times as fast as possible (low latency).

Below the line (out of scope):

- The system should have a storage limit per user
- The system should support file versioning
- The system should scan files for viruses and malware

Here's how it might look on your whiteboard:



Many candidates struggle with the CAP theorem trade-off for this question. Remember, you prioritize consistency over availability only if every read must receive the most recent write; otherwise, the system will break. For example, with a stock trading app, if a user buys a share of APPL in Germany and then another user immediately tries to buy a share of APPL in the US, you need to be sure that the first transaction has been replicated to the US before you can proceed. However, for a file storage system like Dropbox, it's okay if a user in Germany uploads a file and a user in the US can't see it for a few seconds.

The Set Up

Planning the Approach

Before you move on to designing the system, it's important to start by taking a moment to plan your strategy. Fortunately, for these **product design** style questions, the plan should be straightforward: build your design up sequentially, going one by one through your functional requirements. This will help you stay focused and ensure you don't get lost in the weeds as you go. Once you've satisfied the functional requirements, you'll rely on your non-functional requirements to guide you through the deep dives.

Defining the Core Entities

I like to start with a broad overview of the primary entities. At this stage, it is not necessary to know every specific column or detail. We will focus on these intricacies later when we have a clearer grasp of the system (during the high-level design). Initially, establishing these key entities will guide our thought process and lay a solid foundation as we progress towards defining the API.

For Dropbox, the primary entities are incredibly straightforward:

1. **File**: This is the raw data that users will be uploading, downloading, and sharing.
2. **FileMetadata**: This is the metadata associated with the file. It will include information like the file's name, size, mime type, and the user who uploaded it.
3. **User**: The user of our system.

In the actual interview, this can be as simple as a short list like this. Just make sure you talk through the entities with your interviewer to ensure you are on the same page.

As you move onto the design, your objective is simple: create a system that meets all functional and non-functional requirements. To do this, I recommend you start by satisfying the functional requirements and then layer in the non-functional requirements afterward. This will help you stay focused and ensure you don't get lost in the weeds as you go.

API or System Interface

The API is the primary interface that users will interact with. It's important to define the API early on, as it will guide your high-level design. We just need to define an endpoint for each of our functional requirements.

Starting with uploading a file, we might have an endpoint like this:

```
POST /files
Request:
{
  File,
  FileMetadata
}
```

To download a file, our endpoint can be:

```
GET /files/{fileId} -> File & FileMetadata
```

Be aware that your APIs may change or evolve as you progress. In this case, our upload and download APIs actually evolve significantly as we weigh the trade-offs of various approaches in our high-level design (more on this later). You can proactively communicate this to your interviewer by saying, "I am going to outline some simple APIs, but may come back and improve them as we delve deeper into the design."

To share a file, we might have an endpoint like this:

```
POST /files/{fileId}/share
Request:
{
  User[] // The users to share the file with
}
```

Lastly, we need a way for clients to query for changes to files on the remote server. This way we know which files need to be synced to the local device.

```
GET /files/{fileId}/changes -> FileMetadata[]
```

> With each of these requests, the user information will be passed in the headers (either via session token or JWT). This is a common pattern for APIs and is a good way to ensure that the user is authenticated and authorized to perform the action while preserving security. You should avoid passing user information in the request body, as this can be easily manipulated by the client.

High-Level Design

1) Users should be able to upload a file from any device

The main requirement for a system like Dropbox is to allow users to upload files. When it comes to storing a file, we need to consider two things:

1. Where do we store the file contents (the raw bytes)?
2. Where do we store the file metadata?

For the metadata, we can use a **NoSQL database** like DynamoDB. **DynamoDB** is a fully managed NoSQL database hosted by AWS. Our metadata is loosely structured, with few relations and the main query pattern being to fetch files by user. This makes DynamoDB a solid choice, but don't get too caught up in making the right choice here in your interview. The reality is a SQL database like PostgreSQL would work just as well for this use case. Learn more about how to choose the right database (and why it may not matter), **here**.

Our schema will be a simple document and can start with something like this:

```
{
  "id": "123",
  "name": "file.txt",
  "size": 1000,
  "mimeType": "text/plain",
```

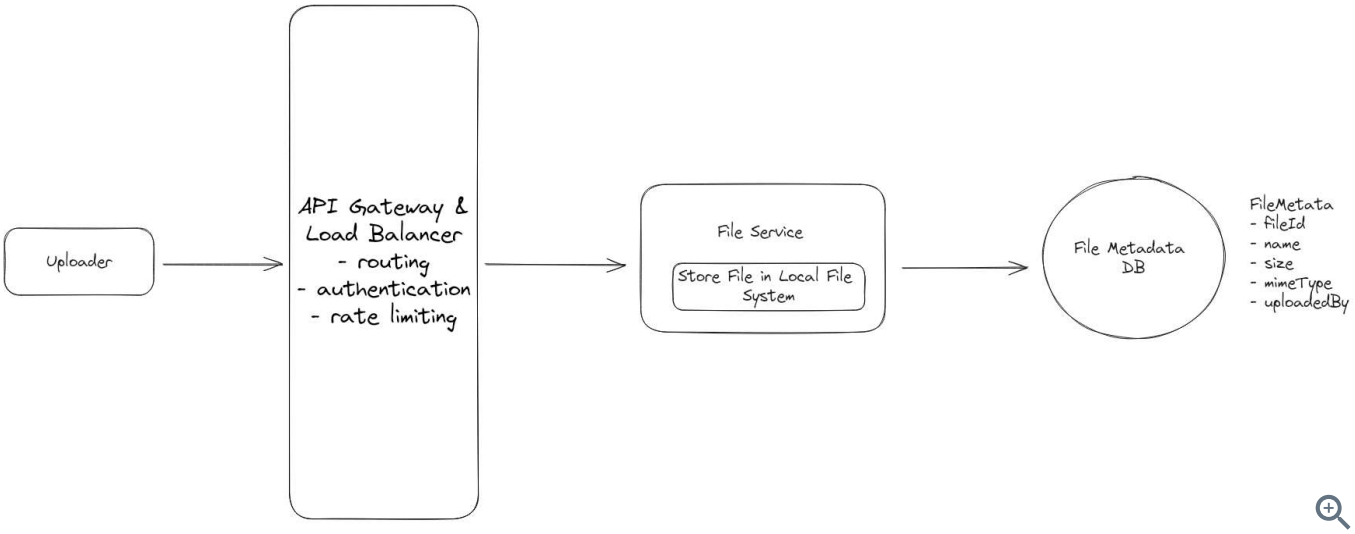
```
  "uploadedBy": "user1"
}
```

As for how we store the file itself, we have a few options. Let's take a look at the trade-offs of each.

Bad Solution: Upload File to a Single Server

Approach

The simplest approach we can take is to upload files directly to our backend server (we can call it the File Service) and store them there. Our `POST /files` endpoint will accept the file and metadata, and then store the file on the server's local file system while saving the metadata in our database. This is a reasonable approach for a small application, but it won't scale well and is not reliable.



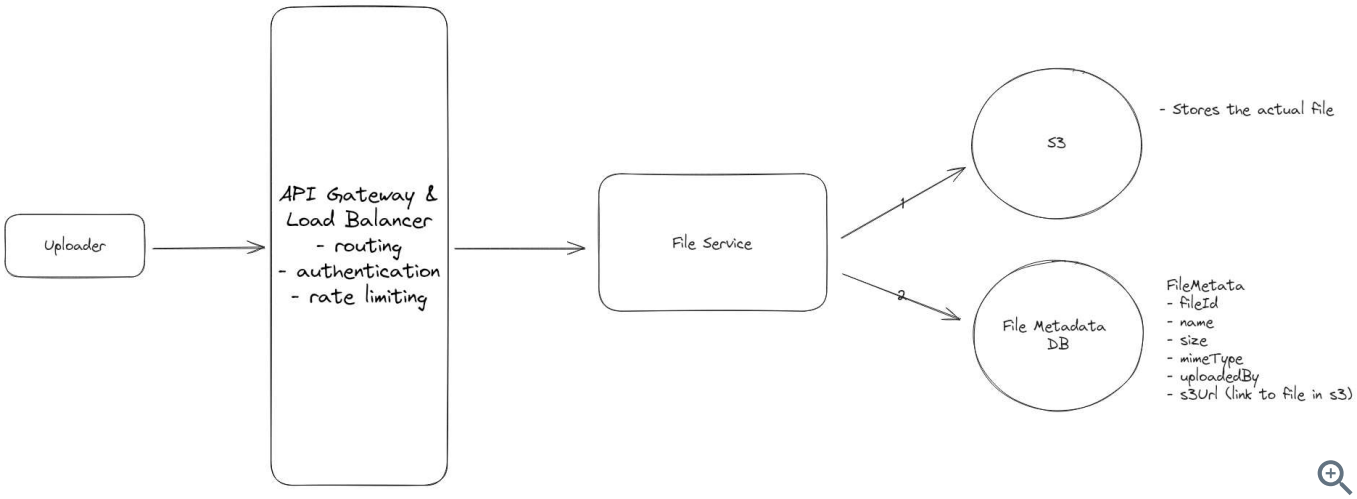
Challenges

This simple approach has a number of issues. As the number of files grows, we will need to add more and more storage to our server and/or horizontally scale by adding more servers. Second, it's not reliable. If our server goes down, we lose access to all of our files. We need a more reliable solution that can handle server failures and scale with ease. Fortunately, this is a solved problem. We can use **Blob Storage** to solve both of these issues.

Good Solution: Store File in Blob Storage

Approach

A better approach is to store the file in a Blob Storage service like Amazon S3 or Google Cloud Storage. When a user uploads a file to our backend, we can send the file directly to Blob Storage and store the metadata in our database. We can store a (virtually) unlimited number of files in Blob Storage as it will handle the scaling for us. It's also more reliable. If our server goes down, we don't lose access to our files. We can also take advantage of Blob Storage features like lifecycle policies to automatically delete old files and versioning to keep track of file changes if needed (though this is out of scope for this problem).



Challenges

One challenge with this approach is that it's more complex. We need to integrate with the Blob Storage service and handle the case where the file is uploaded but the metadata is not saved. We also need to handle the case where the metadata is saved but the file is not uploaded. We can solve these issues by using a transactional approach where we only save the metadata if the file is successfully uploaded and vice versa.

Second, this approach (as depicted above) requires that we technically upload a file twice -- once to our backend and once to Blob Storage. This is redundant. We can solve this issue by allowing the user to upload the file directly to the Blob Storage service.

Great Solution: Upload File Directly to Blob Storage

Approach

The best approach is to allow the user to upload the file directly to Blob Storage from the client. This is faster and cheaper than uploading the file to our backend first. We can use presigned URLs to generate a URL that the user can use to upload the file directly to the Blob Storage service. Once the file is uploaded, the Blob Storage service will send a notification to our backend so we can save the metadata.

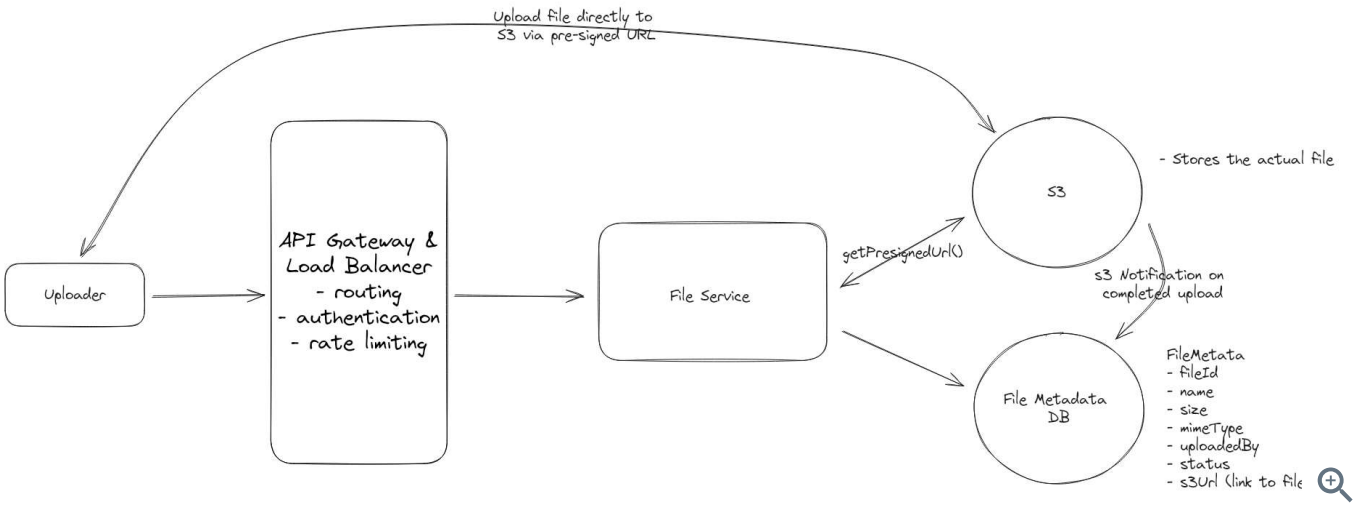
Presigned URLs are URLs that give the user permission to upload a file to a specific location in the Blob Storage service. We can generate a presigned URL and send it to the user when they want to upload a file. So whereas our initial API for upload was a POST to `/files`, it will now be a three step process:

1. Request a pre-signed URL from our backend (which itself gets the URL from the Blob Storage service like S3) and save the file metadata in our database with a status of "uploading."

```
POST /files/presigned-url -> PresignedUrl
Request:
{
  FileMetadata
}
```

1. Use the presigned URL to upload the file to Blob Storage directly from the client. This is via a PUT request directly to the presigned URL where the file is the body of the request.

2. Once the file is uploaded, the Blob Storage service will send a notification to our backend using S3 Notifications. Our backend will then update the file metadata in our database with a status of "uploaded".



What are Presigned URLs?

Presigned URLs are a feature provided by cloud storage services, such as Amazon S3, that allow temporary access to private resources. These URLs are generated with a specific expiration time, after which they become invalid, offering a secure way to share files without altering permissions. When a

presigned URL is created, it includes authentication information as part of the query string, enabling controlled access to otherwise private objects.

This makes them ideal for use cases like temporary file sharing, uploading objects to a bucket without giving users full API access, or providing limited-time access to resources. Presigned URLs can be generated programmatically using the cloud provider's SDK, allowing developers to integrate this functionality into applications seamlessly. This method enhances security by ensuring that sensitive

Show More

2) Users should be able to download a file from any device

The next step is making sure users can download their saved files. Just like with uploads, there are a few different ways to approach this.

Bad Solution: Download through File Server

Approach

The most common solution candidates come up with is to download the file once from Blob Storage to our backend server and then once more from our backend to the client.

Challenges

Of course, the solution is suboptimal as we end up downloading the file twice, which is both slow and expensive. We can solve this issue by allowing the user to download the file directly from the Blob Storage service, just like we did with the upload.

Good Solution: Download from Blob Storage

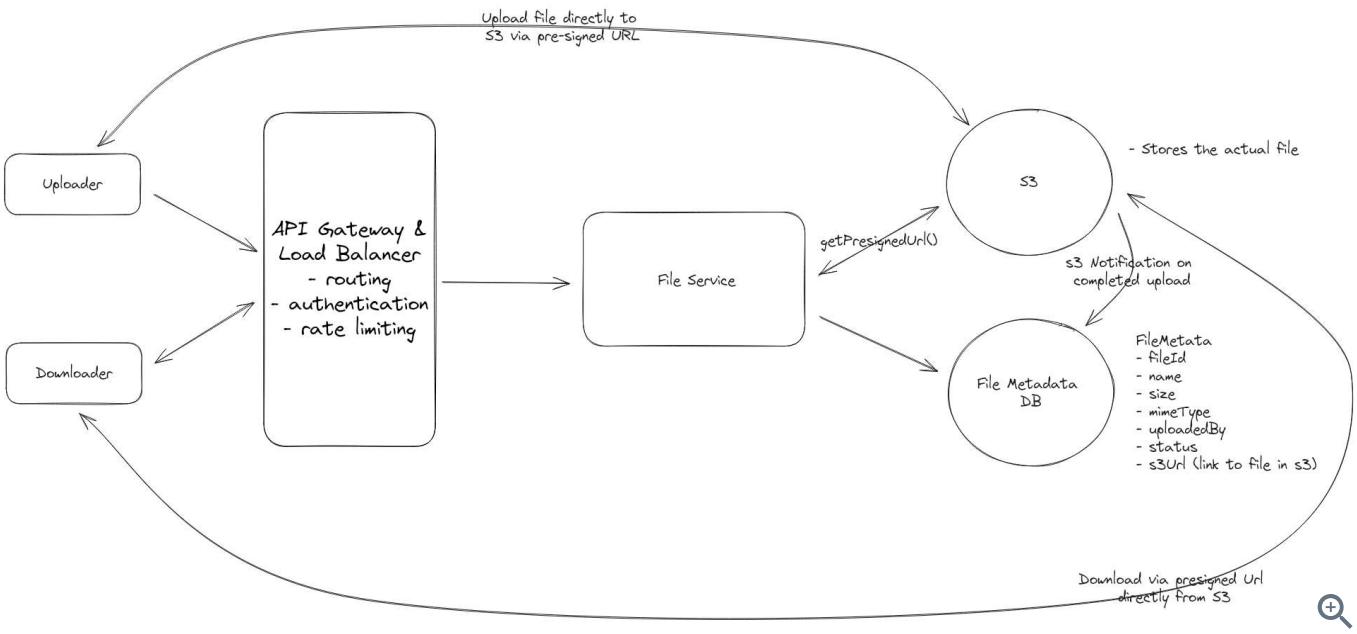
Approach

A better approach is to allow the user to download the file directly from Blob Storage. We can use presigned URLs to generate a URL that the user can use to download the file directly from Blob Storage. Like with uploading, the presigned url will give the user permission to download the file from a specific location in the Blob Storage service for a limited time.

1. Request a presigned download URL from our backend

```
GET /files/{fileId}/presigned-url -> PresignedUrl
```

1. Use the presigned URL to download the file from the Blob Storage service directly to the client.



Challenges

While nearly optimal, the main limitation is that this can still be slow for a large, geographically distributed user base. Your Blob Storage is located in a single region, so users far away from that region will have slower download times. We can solve this issue by using a **content delivery network (CDN)** to cache the file closer to the user.

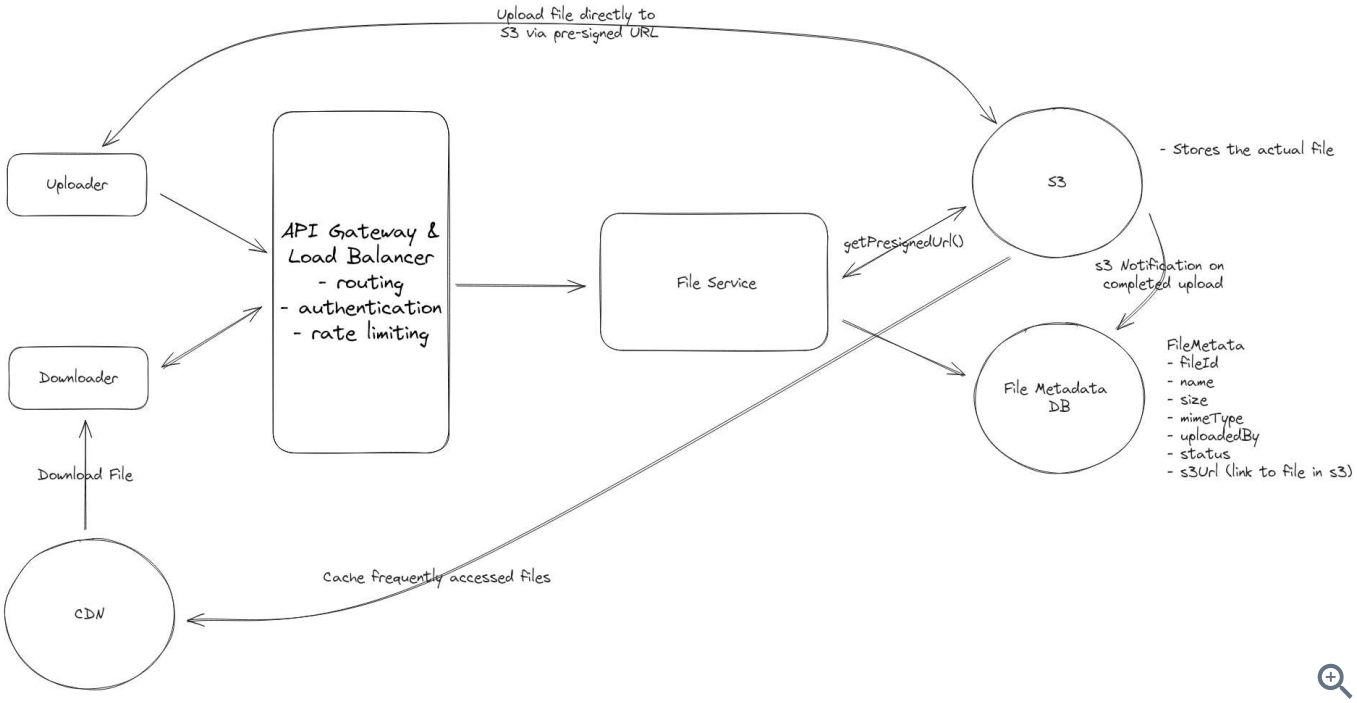
Great Solution: Download from CDN

Approach

The best approach is to use a content delivery network (CDN) to cache the file closer to the user. A CDN is a network of servers distributed across the globe that cache files and serve them to users from the server closest to them. This reduces latency and speeds up download times.

When a user requests a file, we can use the CDN to serve the file from the server closest to the user. This is much faster than serving the file from our backend or the Blob Storage service.

For security, just like with our S3 presigned URLs, we can generate a URL that the user can use to download the file from the CDN. This URL will give the user permission to download the file from a specific location in the CDN for a limited time. More on this in our **deep dives on security**.



Challenges

CDNs are relatively expensive. To address this, it is common to be strategic about what files are cached and for how long. We can use a cache control header to specify how long the file should be cached in the CDN. We can also use a cache invalidation mechanism to remove files from the CDN when they are updated or deleted. This way, only files that are frequently accessed are cached and we don't waste money caching files that are rarely accessed.

3) Users should be able to share a file with other users

To round out the functional requirements, we need to support sharing files with other users. We will implement this similarly to Google Drive, where you just need to enter the email address of the user you want to share the file with. We can assume users are already authenticated.

The main consideration here in an interview is how you can make this process fast and efficient. Let's break it down.

Bad Solution: Add a Sharelist to Metadata

Approach

A good start is to simply add a list of users who have access to the file directly to the file metadata. When a user shares a file, we can add the user to the list. When a user downloads a file, we can check if they are in the list. This is a simple and effective approach.

```
{
  "id": "123",
  "name": "file.txt",
  "size": 1000,
  "mimeType": "text/plain",
  "uploadedBy": "user1",
  "sharelist": ["user2", "user3"]
}
```

Challenges

When a user opens our site they expect to see a list of all of their files and files that have been shared with them. Getting the list of their files is easy, especially since we will have `uploadedBy` as the primary key. But getting the list of files shared with them is slow with this approach. We would need to scan the `sharelist` of every file to see if the user is in it.

Good Solution: Caching to speed up fetching the Sharelist

Approach

A better approach is, in addition to the `sharelist` in the file metadata, to cache a list that maps the inverse relationship. This would be a mapping from any given user to the list of files shared with them. This way, when a user opens our site, we can quickly get the list of files shared with them by looking up their userId in our `sharedFiles` cache.

Our cache entry would be a simple key-value pair like this:

```
user1:["fileId1", "fileId2"]
```

Challenges

We need to keep the `sharedFiles` list in sync with the `sharelist` in the file metadata. The best way to overcome this would be to just keep this user to file list mapping in the same database and update both the `sharelist` and `sharedFiles` list in a transaction.

Great Solution: Create a separate table for shares

Approach

Another approach is to fully normalize the data. This would involve creating a new table that maps userId to fileId, where fileId is a file shared with the given user. This way, when a user opens our site, we can quickly get the list of files shared with them by querying the `SharedFiles` table for all of the files with a `userId` of the user.

So you would create a new table, `SharedFiles`, that looks like this:

```
| userId (PK) | fileId (SK) |
|-------------|-------------|
| user1       | fileId1     |
| user1       | fileId2     |
| user2       | fileId3     |
```

In this design, we no longer need the `sharelist` in the file metadata. We can simply query the `SharedFiles` table for all of the files that have a `userId` matching the requesting user, removing the need to keep the `sharelist` in sync with the `sharedFiles` list.

Challenges

This query is slightly less efficient than the previous approach since now we query using an index instead of a simple key-value lookup. However, the tradeoff may be worth it since we no longer need to keep the `sharelist` in sync with the `sharedFiles` list.

4) Users can automatically sync files across devices

Last up, we need to make sure that files are automatically synced across different devices. At a high level, this works by keeping a copy of a particular file on each client device (locally) and also in remote storage (i.e., the "cloud"). As such, there are two directions we need to sync in:

1. Local -> Remote
2. Remote -> Local

Local -> Remote

When a user updates a file on their local machine, we need to sync these changes with the remote server. We consider the remote server to be the source of truth, so it's important that we get it consistent as soon as possible so that other local devices can know when there are changes they should pull in.

To do this, we need a client-side sync agent that:

1. Monitors the local Dropbox folder for changes using OS-specific file system events (like FileSystemWatcher on Windows or FSEvents on macOS)
2. When it detects a change, it queues the modified file for upload locally
3. It then uses our upload API to send the changes to the server along with updated metadata
4. Conflicts are resolved using a "last write wins" strategy - meaning if two users edit the same file, the most recent edit will be the one that's saved

> Versioning is out of scope for this write-up, but note that you would typically not overwrite the only file. Instead, you'd add a new file (or at least the new chunks) and update a version number and pointer on the metadata.

Remote -> Local

For the other direction, each client needs to know when changes happen on the remote server so they can pull those changes down.

There are two main approaches we could take:

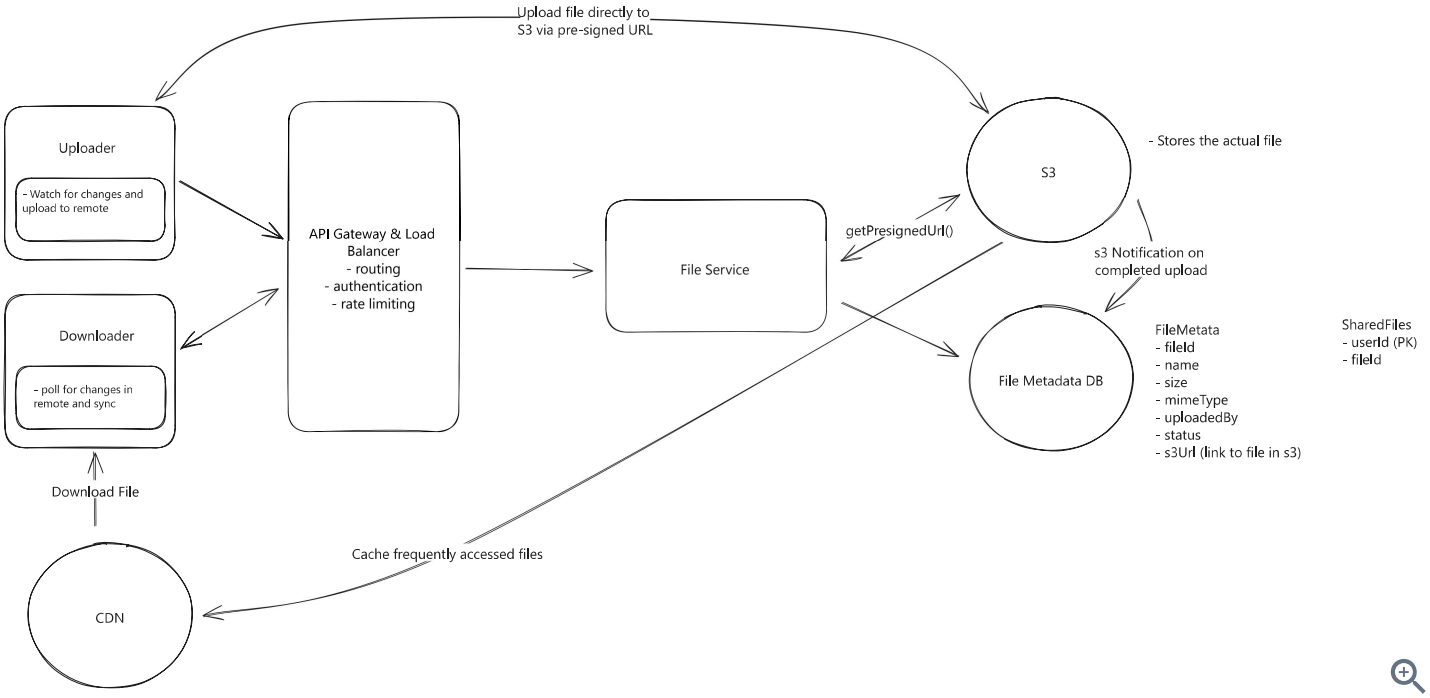
1. **Polling**: The client periodically asks the server "has anything changed since my last sync?" The server would query the DB to see if any files that this user is watching has a updatedAt timestamp that is newer than the last time they synced. This is simple but can be slow to detect changes and wastes bandwidth if nothing has changed.

2. **WebSocket or SSE**: The server maintains an open connection with each client and pushes notifications when changes occur. This is more complex but provides real-time updates.

For Dropbox, we can use a hybrid approach. We can classify files into two categories:

- **Fresh files**: Files that have been recently edited (within the last few hours). For these, we maintain a WebSocket connection to ensure near real-time sync.
- **Stale files**: Files that haven't been modified in a while. For these, we can fall back to periodic polling since immediate updates are less critical.

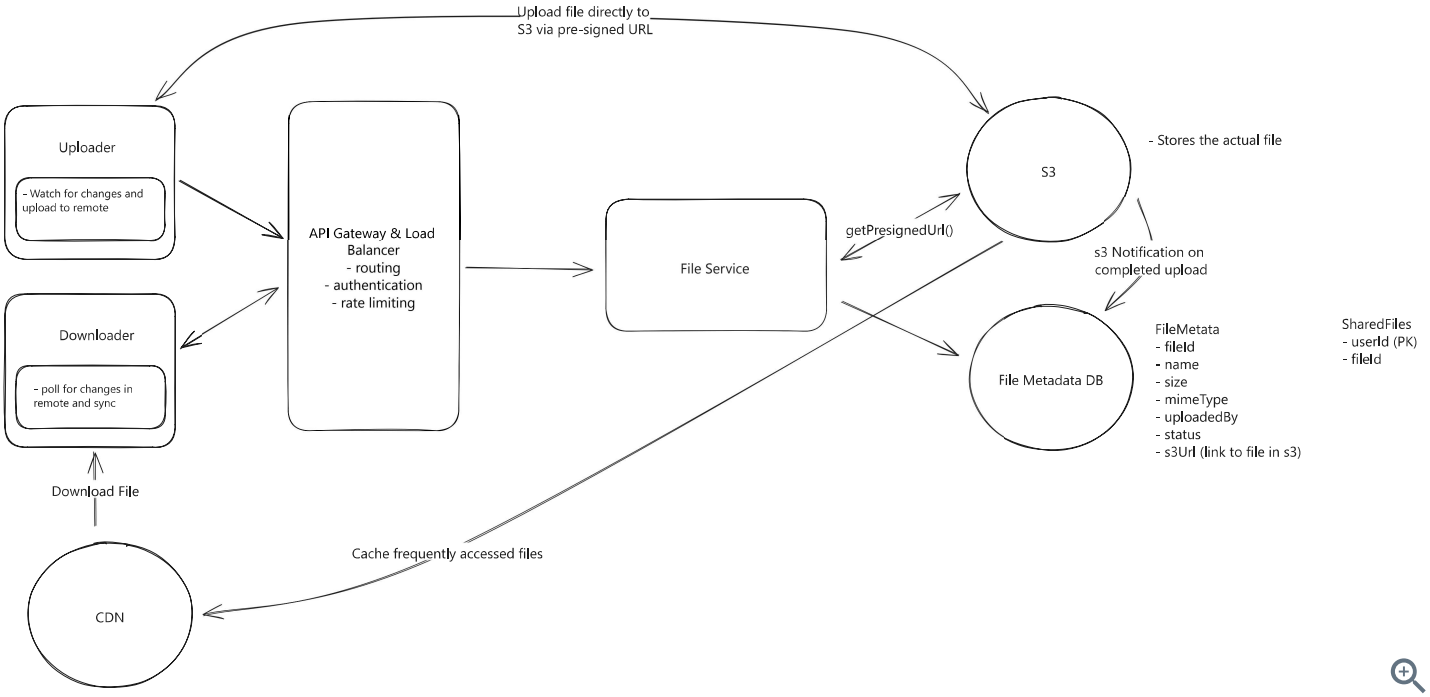
This hybrid approach gives us the best of both worlds - real-time updates for active files while conserving resources for inactive ones.



Sync

Tying it all together

Let's take a step back and look at our system as a whole. At this point, we have a simple design that satisfies all of our functional requirements.



Final

- **Uploader**: This is the client that uploads the file. It could be a web browser, a mobile app, or a desktop app. It is also responsible for proactively identifying local changes and pushing the updates to remote storage.
- **Downloader**: This is the client that downloads the file. Of course, this can be the same client as the uploader, but it doesn't have to be. We separate them in our design for

clarity. It is also responsible for determining when a file it has locally has changed on the remote server and downloading these changes.

- **LB & API Gateway**: This is the load balancer and API Gateway that sits in front of our application servers. It's responsible for routing requests to the appropriate server and handling things like SSL termination, rate limiting, and request validation.
- **File Service**: The file service is only responsible for writing to and from the file metadata db as well as requesting presigned URLs from S3. It doesn't actually handle the file upload or download. It's just a middleman between the client and S3.
- **File Metadata DB**: This is where we store metadata about the files. This includes things like the file name, size, MIME type, and the user who uploaded the file. We also store a shared files table here that maps files to users who have access to them. We use this table to enforce permissions when a user tries to download a file.
- **S3**: This is where the files are actually stored. We upload and download files directly to and from S3 using the presigned URLs we get from the file server.
- **CDN**: This is a content delivery network that caches files close to the user to reduce latency. We use the CDN to serve files to the downloader.

Potential Deep Dives

1) How can you support large files?

The first thing you should consider when thinking about large files is the user experience. There are two key insights that should stick out and ultimately guide your design:

1. **Progress Indicator**: Users should be able to see the progress of their upload so that they know it's working and how long it will take.
2. **Resumable Uploads**: Users should be able to pause and resume uploads. If they lose their internet connection or close the browser, they should be able to pick up where they left off rather than redownloading the 49GB that may have already been uploaded before the interruption.

This is, in some sense, the meat of the problem and where I usually end up spending the most time with candidates in a real interview.

Before we go deep on solutions, let's take a moment to acknowledge the limitations that come with uploading a large file via a single POST request.

- **Timeouts**: Web servers and clients typically have timeout settings to prevent indefinite waiting for a response. A single POST request for a 50GB file could easily exceed these timeouts. In fact, this may be an appropriate time to do some quick math in the interview. If we have a 50GB file and an internet connection of 100Mbps, how long will it take to

upload the file? `50GB * 8 bits/byte / 100Mbps = 4000 seconds` then `4000 seconds / 60 seconds/minute / 60 minutes/hour = 1.11 hours`. That's a long time to wait without any response from the server.

- **Browser and Server Limitation**: In most cases, it's not even possible to upload a 50GB file via a single POST request due to limitations configured in the browser or on the server. Both browsers and web servers often impose limits on the size of a request payload. For instance, popular web servers like Apache and NGINX have configurable limits, but the default is typically set to less than 2GB. Most modern services, like Amazon API Gateway, have default limits that are much lower and cannot be increased. This is just 10MB in the case of Amazon API Gateway which we are using in our design.
- **Network Interruptions**: Large files are more susceptible to network interruptions. If a user is uploading a 50GB file and their internet connection drops, they will have to start the upload from scratch.
- **User Experience**: Users are effectively blind to the progress of their upload. They have no idea how long it will take or if it's even working.

To address these limitations, we can use a technique called "chunking" to break the file into smaller pieces and upload them one at a time (or in parallel, depending on network bandwidth). Chunking needs to be done on the client so that the file can be broken into pieces before it is sent to the server (or S3 in our case). A very common mistake candidates make is to chunk the file on the server, which effectively defeats the purpose since you still upload the entire file at once to get it on the server in the first place. When we chunk, we typically break the file into 5-10 MB pieces, but this can be adjusted based on the network conditions and the size of the file.

With chunks, it's rather straightforward for us to show a progress indicator to the user. We can simply track the progress of each chunk and update the progress bar as each chunk is successfully uploaded. This provides a much better user experience than the user simply staring at a spinning wheel for an hour.

The next question is: **how will we handle resumable uploads?** We need to keep track of which chunks have been uploaded and which haven't. We can do this by saving the state of the upload in the database, specifically in our `FileMetadata` table. Let's update the `FileMetadata` schema to include a `chunks` field.

```
{
  "id": "123",
  "name": "file.txt",
  "size": 1000,
  "mimeType": "text/plain",
  "uploadedBy": "user1",
  "status": "uploading",
```

```
  "chunks": [
    {
      "id": "chunk1",
      "status": "uploaded"
    },
    {
      "id": "chunk2",
      "status": "uploading"
    },
    {
      "id": "chunk3",
      "status": "not-uploaded"
    }
  ]
}
```

When the user resumes the upload, we can check the `chunks` field to see which chunks have been uploaded and which haven't. We can then start uploading the chunks that haven't been uploaded yet. This way, the user doesn't have to start the upload from scratch if they lose their internet connection or close the browser.

But how should we ensure this `chunks` field is kept in sync with the actual chunks that have been uploaded?

There are two approaches we can take:

Good Solution: Update based on client PATCH request

Approach

The most obvious approach is to use the client to orchestrate chunk statuses. So the flow would look like:

1. The client takes the file, chunks it, and uploads the chunks directly to S3.
2. S3 responds to each chunk upload with a success message.
3. Upon success, the client sends a PATCH request to our backend to update the `chunks` field in the `FileMetadata` table.

```
PATCH /files/{fileId}/chunks
Request:
{
  "chunks": [
    {
      "id": "chunk1",
```

```
        "status": "uploaded"
      },
    ]
  }
```

Challenges

The issue here is that we are trusting the client to keep the `chunks` field in sync with the actual chunks that have been uploaded which is a security risk. A malicious user could send a PATCH request to our backend to mark all of the chunks as uploaded without actually uploading them. While they'd only be able to corrupt their own file download in this case and not anyone else's, it is still a risk that can lead to an inconsistent state, which is difficult to debug. We can address this issue by using a server-side approach to ensure that the 'chunks' field stays in sync with the actual uploaded chunks.

Great Solution: Rely on S3 Event Notifications

Approach

A better approach is to use S3 event notifications to keep the `chunks` field in sync with the actual chunks that have been uploaded. **S3 event notifications** are a feature of S3 that allow you to trigger a Lambda function or send a message to an SNS topic when a file is uploaded to S3. We can use this feature to send a message to our backend when a chunk is successfully uploaded and then update the `chunks` field in the `FileMetadata` table without relying on the client.

Next, let's talk about how to uniquely identify a file and a chunk. When you try to resume an upload, the very first question that should be asked is: (1) Have I tried to upload this file before? and (2) If yes, which chunks have I already uploaded? To answer the first question, we cannot naively rely on the file name. This is because two different users (or even the same user) could upload files with the same name. Instead, we need to rely on a unique identifier that is derived from the file's content. This is called a fingerprint.

A **fingerprint** is a mathematical calculation that generates a unique hash value based on the content of the file. This hash value, often created using cryptographic hash functions like SHA-256, serves as a robust and unique identifier for the file regardless of its name or the source of the upload. By computing this fingerprint, we can efficiently determine whether the file, or any portion of it, has been uploaded before.

For resumable uploads, the process involves not only fingerprinting the entire file but also generating fingerprints for each individual chunk. This chunk-level fingerprinting allows the system to precisely identify which parts of the file have already been transmitted.

Taking a step back, we can tie it all together. Here is what will happen when a user uploads a large file:

1. The client will chunk the file into 5-10Mb pieces and calculate a fingerprint for each chunk. It will also calculate a fingerprint for the entire file, this becomes the fileId.
2. The client will send a GET request to fetch the `FileMetadata` for the file with the given fileId (fingerprint) in order to see if it already exists -- in which case, we can resume the upload.
3. If the file does not exist, the client will POST a request to `/files/presigned-url` to get a presigned URL for the file. The backend will save the file metadata in the `FileMetadata` table with a status of "uploading" and the `chunks` array will be a list of the chunk fingerprints with a status of "not-uploaded".
4. The client will then upload each chunk to S3 using the presigned URL. After each chunk is uploaded, S3 will send a message to our backend using S3 event notifications. Our backend will then update the `chunks` field in the `FileMetadata` table to mark the chunk as "uploaded".
5. Once all chunks in our `chunks` array are marked as "uploaded", the backend will update the `FileMetadata` table to mark the file as "uploaded".

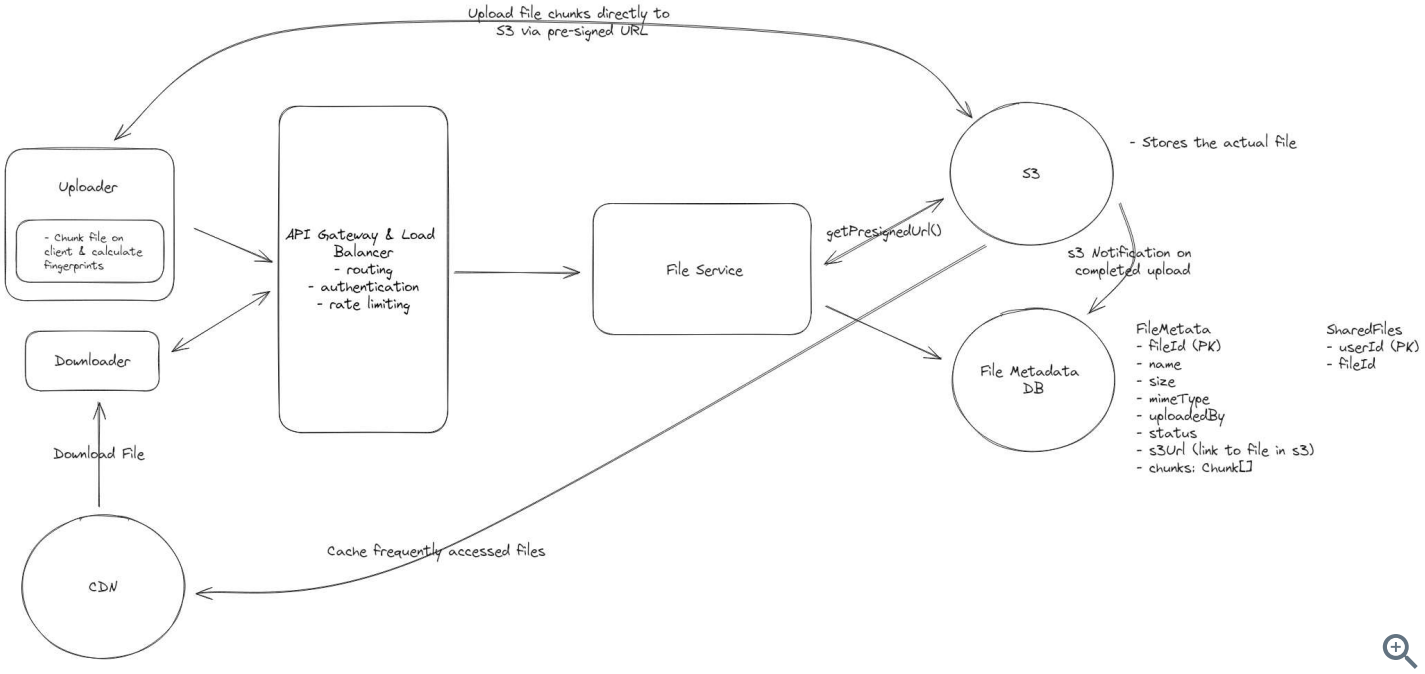
All throughout this process, the client is responsible for keeping track of the progress of the upload and updating the user interface accordingly so the user knows how far in they are and how much longer it will take.

AWS Multipart Upload

The approach we just described is not novel. In fact, this is a problem that has been solved by cloud storage providers like Amazon S3. They have a feature called Multipart Upload that allows you to upload large objects in parts. This is exactly what we just described. The client breaks the file into parts and uploads each part to S3. S3 then combines the parts into a single object. They even provide a handy JavaScript SDK which will handle all of the chunking and uploading for you.

In practice, you'd rely on this API when designing a system like Dropbox. However, it's almost certainly the case that you could not get away with just saying, "I'd use the S3 Multipart Upload API" in your interview without being able to explain how it works and how you would implement it yourself if you had to. Making the interviewer aware that you are familiar with multipart upload is a good idea, however, as it shows hands on experience.

Show More



2) How can we make uploads, downloads, and syncing as fast as possible?

We've already touched on a few ways to speed up both download and upload respectively, but there is still more we can do to make the system as fast as possible. To recap, for download we used a CDN to cache the file closer to the user. This made it so that the file doesn't have to travel as far to get to the user, reducing latency and speeding up download times. For upload, chunking, beyond being useful for resumable uploads, also plays a significant role in speeding up the upload process. While bandwidth is fixed (put another way, the pipe is only so big), we can use chunking to make the most of the bandwidth we have. By sending multiple chunks in parallel, and utilizing adaptive chunk sizes based on network conditions, we can maximize the use of available bandwidth. The same chunking approach can be used for syncing files - when a file changes, we can identify which chunks have changed and only sync those chunks rather than the entire file, making syncing much faster.

Beyond that which we've already discussed, **we can also utilize compression to speed up both uploads and downloads**. Compression reduces the size of the file, which means fewer bytes need to be transferred. We can compress a file on the client before uploading it and then decompress it on the server after it's uploaded. We can also compress the file on the server before sending it to the client and then rely on the client to decompress it.

We'll need to be smart about when we compress though. Compression is only useful if the speed gained from transferring fewer bytes outweighs the time it takes to compress and decompress the file. For some file types, particularly media files like images and videos, the compression ratio is so low that it's not worth the time it takes to compress and decompress the file. If you take a `.png` off your computer right now and compress it, you'll be lucky to have decreased the file size by more than a few percent -- so it's not worth it. For text files, on the other hand, the

compression ratio is much higher and, depending on network conditions, it may very well be worth it. A 5GB text file could compress down to 1GB or even less depending on the content.

In the end, you'll want to implement logic on the client that decides whether or not to compress the file before uploading it based on the file type, size, and network conditions.

Compression Algorithms

There are a number of compression algorithms that you can use to compress files. The most common are Gzip, Brotli, and Zstandard. Each of these algorithms has its own tradeoffs in terms of compression ratio and speed. Gzip is the most widely used and is supported by all modern web browsers. Brotli is newer and has a higher compression ratio than Gzip, but it's not supported by all web browsers. Zstandard is the newest and has the highest compression ratio and speed, but it's not supported by all web browsers. You'll need to decide which algorithm to use based on your specific use case.

One important fact about compression is that you should always compress before you encrypt in cases where encryption is necessary. This is because encryption naturally introduces randomness into the file, which makes it difficult to compress. By compressing before encrypting, you will achieve a much higher compression ratio.

Show More

3) How can you ensure file security?

Security is a critical aspect of any file storage system. We need to ensure that files are secure and only accessible to authorized users.

1. **Encryption in Transit**: Sure, to most candidates, this is a no-brainer. We should use HTTPS to encrypt the data as it's transferred between the client and the server. This is a standard practice and is supported by all modern web browsers.
2. **Encryption at Rest**: We should also encrypt the files when they are stored in S3. This is a feature of S3 and is easy to enable. When a file is uploaded to S3, we can specify that it should be encrypted. S3 will then encrypt the file using a unique key and store the key separately from the file. This way, even if someone gains access to the file, they won't be able to decrypt it without the key. You can learn more about S3 encryption **here**.
3. **Access Control**: Our `shareList` or separate share table/cache is our basic ACL. As discussed earlier, we make sure that we share download links only with authorized users.

But what happens if an authorized user shares a download link with an unauthorized user? For example, an authorized user may, intentionally or unintentionally, post a download link to a public forum or social media and we need to make sure that unauthorized users cannot download the file.

This is where those signed URLs we talked about early come back into play. When a user requests a download link, we generate a signed URL that is only valid for a short period of time (e.g. 5 minutes). This signed URL is then sent to the user, who can use it to download the file. If an unauthorized user gets a hold of the signed URL, they won't be able to use it to download the file because it will have expired.

They also work with modern CDNs like CloudFront and are a feature of S3. Here is how:

1. Generation: A signed URL is generated on the server, including a signature that typically incorporates the URL path, an expiration timestamp, and possibly other restrictions (like IP address). This signature is encrypted with a secret key known only to the content provider and the CDN.
2. Distribution: The signed URL is distributed to an authorized user, who can use it to access the specified resource directly from the CDN.
3. Validation: When the CDN receives a request with a signed URL, it checks the signature in the URL against the expected format and parameters, including verifying the expiration timestamp. If the signature is valid and the URL has not expired, the CDN serves the requested content. If not, it denies access.

What is Expected at Each Level?

Ok, that was a lot. You may be thinking, “how much of that is actually required from me in an interview?” Let’s break it down.

Mid-level

Breadth vs. Depth: A mid-level candidate will be mostly focused on breadth (80% vs 20%). You should be able to craft a high-level design that meets the functional requirements you've defined, but many of the components will be abstractions with which you only have surface-level familiarity.

Probing the Basics: Your interviewer will spend some time probing the basics to confirm that you know what each component in your system does. For example, if you add an API Gateway, expect that they may ask you what it does and how it works (at a high level). In short, the interviewer is not taking anything for granted with respect to your knowledge.

Mixture of Driving and Taking the Backseat: You should drive the early stages of the interview in particular, but the interviewer doesn’t expect that you are able to proactively recognize problems in your design with high precision. Because of this, it’s reasonable that they will take over and drive the later stages of the interview while probing your design.

The Bar for Dropbox: For this question, an E4 candidate will have clearly defined the API endpoints and data model, landed on a high-level design that is functional for all of uploading, downloading, and sharing. I don't expect candidates to know about pre-signed URLs or uploading/downloading to/from S3 directly, or to immediately know about chunking. However, I expect that when I ask probing questions like, "You're uploading the file twice right now, how can we avoid that?" or "How can you show a user's progress while allowing them to resume an upload?" that they can reason through the problem and come to a solution via some back and forth.

Senior

Depth of Expertise: As a senior candidate, expectations shift towards more in-depth knowledge — about 60% breadth and 40% depth. This means you should be able to go into technical details in areas where you have hands-on experience. It's crucial that you demonstrate a deep understanding of key concepts and technologies relevant to the task at hand.

Advanced System Design: You should be familiar with advanced system design principles. For example, knowing how to utilize blob storage for large files, or how to implement a CDN for faster downloads. You should be able to discuss the trade-offs involved in different design choices and justify your decisions based on your experience.

Articulating Architectural Decisions: You should be able to clearly articulate the pros and cons of different architectural choices, especially how they impact scalability, performance, and maintainability. You justify your decisions and explain the trade-offs involved in your design choices.

Problem-Solving and Proactivity: You should demonstrate strong problem-solving skills and a proactive approach. This includes anticipating potential challenges in your designs and suggesting improvements. You need to be adept at identifying and addressing bottlenecks, optimizing performance, and ensuring system reliability.

The Bar for Dropbox: For this question, E5 candidates are expected to quickly go through the initial high-level design so that they can spend time discussing, in detail, how to handle uploading large files, in particular. I expect them to be more proactive here than mid-level candidates, thinking through several options and arriving at a reasonable solution. While not strictly required, many candidates will have experience with file uploads and can speak directly about certain APIs (like multipart upload) and how they work.

Staff+

Emphasis on Depth: As a staff+ candidate, the expectation is a deep dive into the nuances of system design — I'm looking for about 40% breadth and 60% depth in your understanding. This

level is all about demonstrating that, while you may not have solved this particular problem before, you have solved enough problems in the real world to be able to confidently design a solution backed by your experience.

You should know which technologies to use, not just in theory but in practice, and be able to draw from your past experiences to explain how they'd be applied to solve specific problems effectively. The interviewer knows you know the small stuff (REST API, data normalization, etc) so you can breeze through that at a high level so you have time to get into what is interesting.

High Degree of Proactivity: At this level, an exceptional degree of proactivity is expected. You should be able to identify and solve issues independently, demonstrating a strong ability to recognize and address the core challenges in system design. This involves not just responding to problems as they arise but anticipating them and implementing preemptive solutions. Your interviewer should intervene only to focus, not to steer.

Practical Application of Technology: You should be well-versed in the practical application of various technologies. Your experience should guide the conversation, showing a clear understanding of how different tools and systems can be configured in real-world scenarios to meet specific requirements.

Complex Problem-Solving and Decision-Making: Your problem-solving skills should be top-notch. This means not only being able to tackle complex technical challenges but also making informed decisions that consider various factors such as scalability, performance, reliability, and maintenance.

Advanced System Design and Scalability: Your approach to system design should be advanced, focusing on scalability and reliability, especially under high load conditions. This includes a thorough understanding of distributed systems, load balancing, caching strategies, and other advanced concepts necessary for building robust, scalable systems.

The Bar for Dropbox: For a staff-level candidate, expectations are high regarding the depth and quality of solutions, especially for the complex scenarios discussed earlier. Exceptional candidates delve deeply into each of the topics mentioned above and may even steer the conversation in a different direction, focusing extensively on a topic they find particularly interesting or relevant. They are also expected to possess a solid understanding of the trade-offs between various solutions and to be able to articulate them clearly, treating the interviewer as a peer.