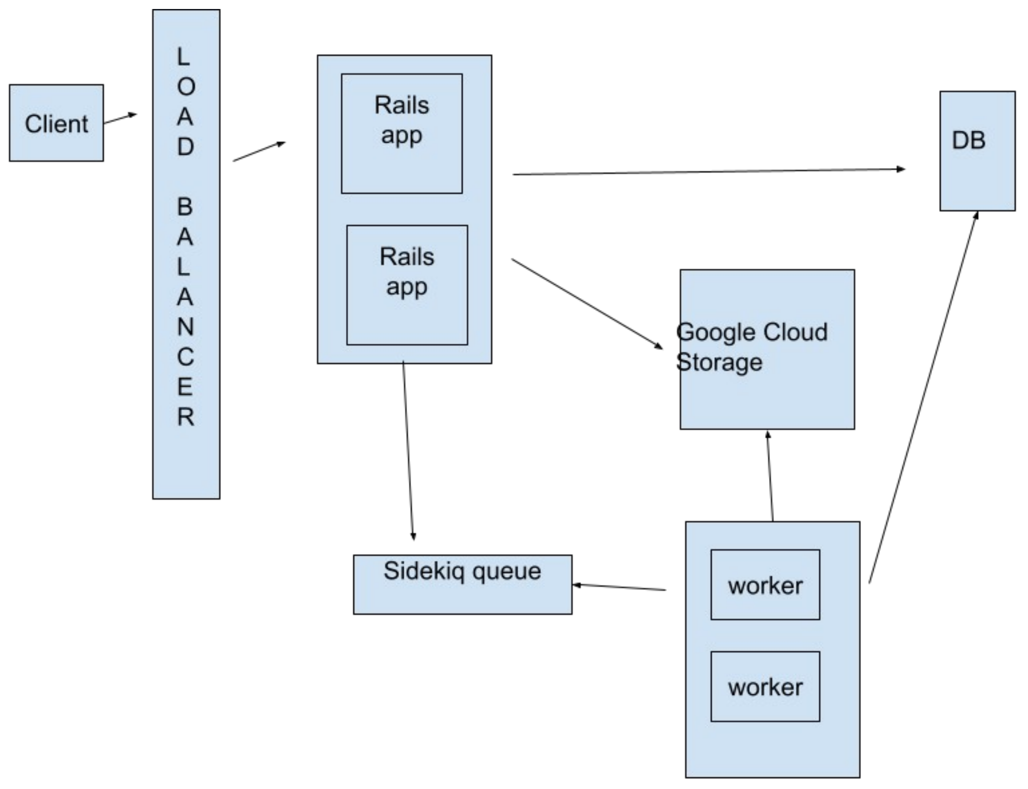
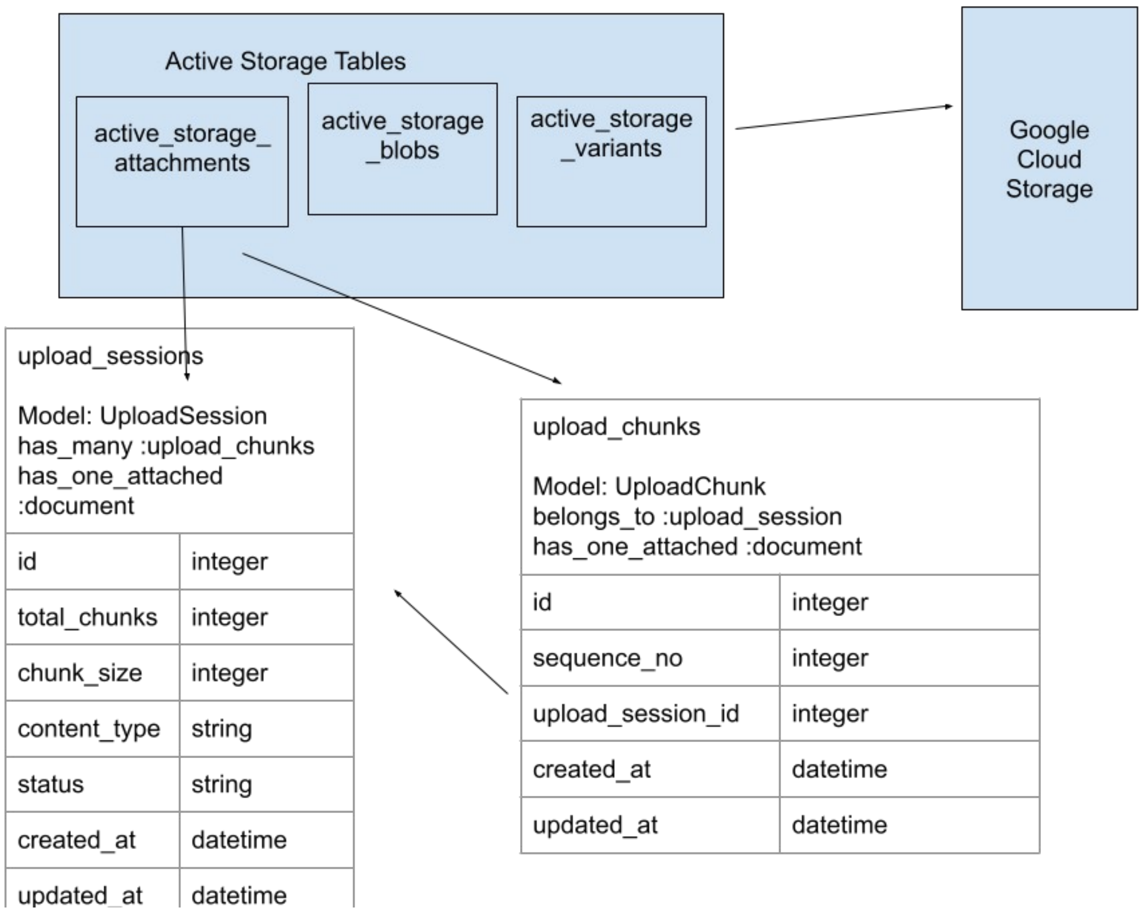
High Level Architecture Diagram



DB Schema



upload_sessions

Model: UploadSession
has_many :upload_chunks
has_one_attached :document

id	integer
total_chunks	integer
chunk_size	integer
content_type	string
status	string
created_at	datetime
updated_at	datetime

upload_chunks

Model: UploadChunk
belongs_to :upload_session
has_one_attached :document

id	integer
sequence_no	integer
upload_session_id	integer
created_at	datetime
updated_at	datetime

Client-side Upload Flow

1.Client sends a Upload Session api call. In the payload, it sends file content type and total chunks and chunk size.
Backend validates that the file size is upto 5GB and content type is valid like image or video. If it’s valid, then backend creates a session and responds with session’s id. This session id will be used in the subsequent requests to upload the chunks and track the upload status.

2. Client slices the large file into smaller chunks and starts uploading chunks.
In the UploadChunk api call, it sends session id, chunk’s sequence_no and the chunk data as request payload.
Backend receives the chunk api call. It validates that the chunk size is within limit. It stores the chunk on google cloud storage as well as stores the sequence_no. The sequence_no will be used to determine the order of the chunks when these are assembled into single file.

3. When all chunks have been uploaded, client sends the “complete session” api call.
Backend marks the session as complete and triggers a background job “FileProcessingJob”

4. The FileProcessingJob is picked by a sidekiq worker. In this job, the worker will assemble all the chunks into single file. It’s important to note that the ActiveStorage supports google’s compose api, that means, the worker will not download individual chunks. Instead, the files will be composed within the Google Cloud Storage. The worker will just send the object keys ordered by sequence_no in the google compose’s api.
ActiveStorage will also generate the thumbnail image after the file is assembled.

5. After the file is assembled, the status will be marked as assembled. Then the worker will proceed with virus scan. If the virus scan is successful, the session’s status is marked as success, otherwise, the session’s status is marked as scan failure.

6. At any stage, the client can query the upload’s status.

- If the upload is incomplete, the status api responds with all the sequence_no of the chunks that have been uploaded. For e.g, the file is divided into 10 chunks with sequence_no from 0..9 and till now, chunks 0, 1, 3, 5 have been uploaded. The status api will respond with 0,1,3 and 5. The client can use this information to upload the remaining chunks.Client can send chunks in parallel and chunks can arrive out of sequence on backend.
- If the upload is complete and all the files have been assembled, the status api also responds with document url.

Technical Specification:

- **Chunked Upload Strategy:** How will you handle large files?
    - The large files will be handled in two phases:
        - First, the client will split the files into smaller chunks. These smaller chunks can be uploaded parallelly to save upload time.
        - Then, the backend will assemble all chunks into single file through a background job after all chunks have been uploaded
- **Concurrency Management:** How will you handle 1000+ simultaneous uploads?
    - We can have multiple nodes/instances running behind load balancer to handle concurrent load. The system is designed such that each chunk can be independently uploaded of each other. It does major processing work like assembling the chunks, thumbnail generation, metadata extraction through background jobs.
    - For future considerations, we can change the strategy to support direct uploads to the cloud storage from client through pre-signed urls. This will give more

- **Error Handling & Recovery:** What happens when uploads fail?
    - If the upload fails, then the server responds with error. The client can re-upload the chunk. The client can also query the status of the upload session to determine how many chunks have been uploaded successfully so far, and which ones are remaining.

- **Security Considerations:**
    - File validation: It validates maximum file size of 5 GB with each chunk of max size 5MB and upto 1000 chunks
    - Authentication & Authorization: The application can support user based authentication and authorization but is not implemented yet as that is outside the scope of this assignment as it requires additional setup like registrations, login and signout.
- **Performance Optimization:**
    - We can put a CDN in front of our cloud storage to serve these assets faster. We will need to do configuration changes. It will improve latency as the content will be served from the nearest edge location as well as offer imporved website performance.
    - Database optimization: We can analyze through instrumentation and profiling to see what queries are taking time. As needed, we can put cache frequently accessed data. We can go with some techniques like data archival or partitioning as the data size grows.
- **Monitoring & Observability**:
    - To identify key metrics like response times, error rates, we can use some tools like new relic, open telemetry. These will provide us with insights on where our system performance lacks. These tools provide us to define SLOs and setup alerts when SLO breach occurs

Technology Stack Justification

Technologies used in this project:
ActiveStorage - It provides out of the box support for things like connecting to s3, gcs. It automatically enqueues jobs for thumbnail processing. It provides methods for composing files on gcs.
Ruby version - It uses latest stable ruby version - 3.4.4
Infrastructure -
DB - mysql - It is widely used database system due to it's reliability, scalibility and ease-of-use. It serves the purpose for this project
Files Storage - We can go with GCS or S3. Both offers good integration with rails and are similar in terms of costing and availability and durability.
Background workers - Sidekiq - It is widely used and provides good performance

Scalability & Future Growth -

For future, we can consider some change in the upload flow. Instead of uploading from client to server and then server to cloud storage. We can allow client to directly upload to cloud storage through pre-signed url. This will take the load off from our server and will provide more bandwidth without increasing the infrastructure.

Database can become a bottleneck as the site users grow. We can opt for some techniques like data archival, partitioning, caching.

Part 3: Technical Leadership & Code Review

Task 3.1: Code Review & Mentoring

- **Issues**:
  - using user provided data like filename inside shell commands like convert and exif can lead to shell injection vulnerability
  - users can overwrite each other's file if they upload the file with same name
  - storing user supplied files in public/uploads directory can lead to Executable Code vulnerability. Please read https://guides.rubyonrails.org/security.html#executable-code-in-file-uploads
  - validate mime-type before processing the file
  - using Thread.new for asynchronous processing of the files is not a good idea. Use a background job like delayed jobs, sidekiq for asychronous processing
  - when creating the UploadedFile record, code should check whether it was succesful or not and accordingly return validation errors or success message.
  - It violates separation of concerns. The code for metadata extraction and thumbnail generation should be part of separate service or class.
  - Storing the file in local directory is fine for development but in production, application could be running in multiple nodes. So, the file stored in one instance will not be available in another instance.

- **Refactored solution**:
  Models:
  - UploadedFile
    - Add validation which checks for file presence, size and content-type
    - Add a callback on create which triggers a background job to process thumbnail and extract metadata from the file
    - Add a callback for saving the file in some storage service like disk, s3, gcs
  - Image utility
    - Extract the exiftool and convert shell commands into a ruby wrapper class like ImageUtils and call these methods from the background job like this:
      - ImageUtils.metadata(file_path) -> returns { "width" => 100, "height" => 200 }
      - ImageUtils.generate_thumbnail(file_path) -> returns thumbnail file object
  - Background Job
    - use this job to extract metadata and generate thumbnail as explained above.
  - Controller
    - In the controller, create the UploadedFile record using the create method and check if it's successful or not. If successful, returns success response otherwise returns validation errors

Task 3.2: Technical Decision Making

As Principal Engineer, you need to make a critical decision for the AssetFlow team:
Scenario:
The current file upload system handles 100 uploads/minute during peak hours.
Product requirements demand scaling to 2000 uploads/minute within 6 months.
The current Rails monolith is showing signs of strain.

Your Options:
1. Optimize Current Monolith: Scale vertically, optimize queries, add caching
2. Microservices Architecture: Extract upload service into separate microservice
3. Hybrid Approach: Keep Rails for API, move processing to separate services
4. Event-Driven Architecture: Implement event sourcing with message queues

Answer:
Option 1: Optimize Current Monolith: Scale vertically, optimize queries, add caching

Pros:
This option is good for long term to continually optimize and fix performance related problem
Cons:
This option will require a lot of changes within the entire application and thus will subsequently require lots of effort to determine impact and testing efforts will increase significantly.
Within the given timeframe of 6 months, it may or may not be a feasible approach depending on how big the monolith application is.

Option 2: Microservices Architecture: Extract upload service into separate microservice

Pros:
It has a lot of advantages like loose coupling, simplified code and can be independently deployed and scaled.
Cons:
Implementing communication between monolith and upload service will require additional development efforts. Existing data migration into the upload service and syncing between them can be challenging.

Option 3: Hybrid Approach: Keep Rails for API, move processing to separate services

This option only impacts the File Upload component, leaving the rest of the components unaffected and also requires less development and testing efforts

Option 4: Event-Driven Architecture: Implement event sourcing with message queues
This option offers lots of advantages like loose coupling, scalibility, fault tolerance. It has some disadvantages like:
It will lead to additional complexity like managing and handling event flows.
Debugging can become a challenge because event-driven systems are distributed.

Recommended Approach:
The recommended approach for the given problem will be to go with Hybrid Approach i.e Keep Rails for API, move processing to separate services as it directly addresses the problem and has minimal impact and can be completed within the time frame.