

The service processes CSV files with image URLs, compresses the images asynchronously, and stores the results. Users can submit CSVs via an API and receive a unique request ID to track the status of processing. Once processing is complete, a webhook can be triggered.

**System Components**

1. **API Layer (Flask)**
   * **Purpose**: Handles user interaction and API requests.
   * **Endpoints**:
     + /upload: Accepts the CSV file, initiates asynchronous image processing, and returns a unique request ID.
     + /status/<request\_id>: Allows users to check the processing status using the request ID.
   * **Flow**:
     + Validates the input CSV file.
     + Saves the initial request details to the MongoDB database.
     + Delegates image processing tasks to Celery asynchronously.
2. **Asynchronous Processing (Celery)**
   * **Purpose**: Processes each image asynchronously and reduces its quality.
   * **Task Flow**:
     + Reads the image URLs from the CSV file.
     + Downloads the images from URLs and compresses them by 50%.
     + Stores the compressed images in MongoDB’s GridFS.
     + Updates the status and processed data in MongoDB after each task is completed.
3. **Database (MongoDB + GridFS)**
   * **MongoDB**: Stores metadata such as request ID, input image URLs, output image URLs, and processing status.
   * **GridFS**: Stores the binary compressed images.
4. **Task Queue (Redis + Celery Worker)**
   * **Redis**: Acts as a broker for Celery tasks.
   * **Celery Worker**: Listens for tasks submitted by Flask (via the /upload API) and processes images asynchronously.
5. **Webhook**
   * **Purpose**: Notifies an external system when all images have been processed.
   * **Flow**: Once image processing is complete, Celery triggers a POST request to the webhook URL (if provided).

**2. System Workflow**

1. **CSV File Upload:**
   * A user submits a CSV file with image URLs using the /upload API.
   * Flask validates the CSV format and generates a unique request ID.
   * The request details (CSV metadata) are stored in MongoDB.
2. **Asynchronous Image Processing:**
   * Flask uses Celery to trigger the image processing task in the background.
   * Celery downloads each image using the URLs in the CSV.
   * Images are compressed by 50% quality using the Pillow library.
   * Compressed images are stored in GridFS (MongoDB).
   * Once all images are processed, the database is updated with output image URLs and status is marked as "Completed."
3. **Webhook Trigger:**
   * If a webhook URL is provided in the initial request, the system sends a POST request to that URL once processing is completed.
4. **Status Tracking:**
   * The user can use the /status/<request\_id> API to check the processing status (Pending, Processing, Completed).
   * Once completed, the response includes both input and output image URLs.

**3. Visual Diagram of the System**

The visual architecture diagram of this system can be created using Draw.io (or any similar tool). Below is a detailed structure of the components:

1. **Client (User)**:
   * Submits the CSV file via the **Upload API**.
   * Receives the **Request ID** for status tracking.
2. **Flask Application**:
   * Validates CSV and starts asynchronous tasks.
   * Saves initial request data to MongoDB.
   * Sends request to **Celery** for background processing.
3. **Celery + Redis**:
   * Celery worker picks the image-processing task.
   * Each image is downloaded and compressed.
   * Stores compressed image in GridFS.
   * Updates status in MongoDB after processing.
4. **MongoDB (Database)**:
   * Stores request information (input and output image URLs, status).
   * Uses **GridFS** to store image files in binary format.
5. **Webhook** (Optional):
   * Once processing is complete, Celery triggers the **Webhook** to notify an external system of completion.
6. **Client** (Status API):
   * Queries the **Status API** to get current processing status.

**4. Component Roles & Functions**

* **Flask Application**:
  + Provides API endpoints for CSV upload and status tracking.
  + Initiates image processing via Celery and updates MongoDB.
  + Receives webhook URL (if provided) for callback notification.
* **Celery**:
  + Manages background processing asynchronously.
  + Downloads images from provided URLs, compresses them, and stores them in MongoDB.
  + Updates request status as processing progresses and triggers the webhook upon completion.
* **Redis**:
  + Acts as the broker for Celery tasks, queueing up the image-processing jobs.
* **MongoDB**:
  + Stores request metadata (request ID, input image URLs, output image URLs, status).
  + Uses GridFS for storing compressed image files as binary objects.
* **GridFS**:
  + Handles storage of binary data (compressed images) that exceed MongoDB’s default size limit (16 MB).
* **Webhook**:
  + Notifies an external system upon completion of image processing, delivering the final status and URLs of processed images.

**5. Technical Considerations**

* **Scalability**:
  + By leveraging Celery and Redis, the system can scale horizontally, processing multiple image-processing tasks simultaneously.
* **Error Handling**:
  + Celery tasks will handle download or processing errors by marking failed tasks and continuing with other tasks.
* **Performance**:
  + Using asynchronous processing prevents blocking API calls. Large CSV files with many images will not overwhelm the system.

### Conclusion:

This system efficiently processes image data from CSV files asynchronously, with scalable design, tracking capabilities, and webhook support. You can enhance the system further by integrating additional features like retries for failed tasks, security improvements (such as file validation), and more sophisticated status reporting.

Let me know if you need help with generating the visual diagram!