# **Technical Design Document**

## **1. Overview**

The system is designed to process image data from CSV files asynchronously. Users can upload a CSV file that contains product information along with image URLs. The system validates the CSV, processes each image (compressing it by 50% quality), stores the results in a database, and provides a downloadable output CSV. Optionally, the system notifies an external webhook upon completion.

## **2. System Architecture**

### **2.1. High-Level Components**

* **API Server (FastAPI):** Provides endpoints for CSV upload, status checking, and downloading the output CSV. It is responsible for receiving user input, validating the data, and returning a unique request ID.
* **Asynchronous Worker:** Processes the CSV file in a background thread/process. It downloads, compresses, and saves images; updates the processing status; and handles webhook notifications.
* **Database (PostgreSQL):** Stores the processing requests and product data. The schema includes tables for processing requests and products with relevant metadata.
* **Static File Storage:** A directory within the application (or external storage) used to store the processed images.
* **Webhook Receiver (Optional):** An external system endpoint (e.g., RequestBin) that gets notified upon processing completion.

### **2.2. Visual Diagram**

Below is a textual representation of the architecture. You can create a visual diagram using Draw.io or a similar tool based on this description is present in Architecture Diagram.drawio file.

## **3. Component Details**

### **3.1. API Server (FastAPI)**

* **Endpoints:**
  + **Upload API (/upload):** Accepts a CSV file and an optional webhook URL. Validates the CSV format and returns a unique request ID. It triggers asynchronous processing via a background executor.
  + **Status API (/status/{request\_id}):** Allows clients to query the processing status of the uploaded CSV using the request ID.
  + **Download API (/download/{request\_id}):** Generates an output CSV containing original and processed image URLs for the given request ID and returns it as a downloadable file.
* **Request Validation:** The API validates that:  
  + The uploaded file is a CSV.
  + The CSV has the required headers: "S. No.", "Product Name", and "Input Image Urls".

### **3.2. Asynchronous Worker**

* **Processing Workflow:**
  1. **Start Processing:** The worker is triggered asynchronously by the API after a CSV is uploaded. It retrieves the request from the database and sets its status to "processing".
  2. **Image Processing:** The worker parses the CSV, iterates over each row, downloads the images from the provided URLs, compresses them by 50% quality using the PIL library, and stores the images in a designated directory.
  3. **Database Update:** Processed image URLs and corresponding product data are saved into the database.
  4. **Finalization & Notification:** Upon successful completion, the worker updates the request status to "completed". If a webhook URL was provided, it sends a POST request to notify the external system. In case of errors, the status is updated to "failed", and an error notification is sent to the webhook.
* **Error Handling:** The worker logs all errors and performs database rollbacks in case of exceptions. It also implements retry logic for webhook notifications if required.

### **3.3. Database Schema**

* **ProcessingRequest Table:**
  + **id (UUID):** Unique identifier for each processing request.
  + **status (String):** Current status (e.g., pending, processing, completed, failed).
  + **created\_at / updated\_at (DateTime):** Timestamps for record creation and updates.
  + **webhook\_url (String):** Optional webhook endpoint provided by the user.
* **Product Table:**
  + **id (Integer):** Primary key.
  + **request\_id (UUID):** Foreign key linking to the processing request.
  + **serial\_number (Integer):** Serial number from the CSV.
  + **product\_name (String):** Product name.
  + **input\_urls (ARRAY of Text):** List of input image URLs.
  + **output\_urls (ARRAY of Text):** List of processed image URLs.

## **4. API Documentation**

### **4.1. Endpoints**

1. **Upload Endpoint**
   * **URL:** /upload
   * **Method:** POST
   * **Request Parameters:**
     + file (CSV file): The CSV file to be processed.
     + webhook\_url (optional, string): The URL for receiving webhook notifications.
   * **Response:**
     + request\_id: Unique identifier for the processing request.
2. **Status Endpoint**
   * **URL:** /status/{request\_id}
   * **Method:** GET
   * **Request Parameters:**
     + request\_id: The unique ID returned during CSV upload.
   * **Response:**
     + status: Current processing status (pending, processing, completed, or failed).
3. **Download Endpoint**
   * **URL:** /download/{request\_id}
   * **Method:** GET
   * **Request Parameters:**
     + request\_id: Unique identifier for the processing request.
   * **Response:**
     + Returns a downloadable CSV file containing columns: "S. No.", "Product Name", "Input Image Urls", and "Output Image Urls".

## **5. Asynchronous Worker Documentation**

### **5.1. Worker Responsibilities**

* **CSV Processing:** The worker reads the CSV content and iterates over each row to extract product details and image URLs.
* **Image Download & Processing:** For each image URL:  
  + Downloads the image with a timeout.
  + Compresses the image using a utility function (compress\_and\_save\_image).
  + Saves the compressed image to the specified directory.
* **Database Interaction:** The worker updates the status of the processing request and saves product details along with input and output image URLs.
* **Webhook Notification:** After processing:  
  + Constructs a JSON payload including the request\_id and status (and error details, if any).
  + Sends an HTTP POST request to the provided webhook URL.
  + Implements basic error handling for the webhook call.

### **5.2. Error Handling**

* **Database Errors:** Transactions are rolled back if errors occur during database updates.
* **Image Processing Errors:** Errors during image download or processing are logged, and processing continues for remaining images.
* **Webhook Errors:** Any errors during webhook notification are logged, and the system may implement retry mechanisms if needed.

## **6. Deployment and Testing**

### **6.1. Deployment**

* **Docker Deployment:** A Dockerfile is provided to containerize the application. The image includes Python 3.12-slim, necessary system dependencies, and sets up the working directory. Processed images are stored in a dedicated folder, and the application runs using Uvicorn.
* **Environment Variables:** Use a .env file to configure variables such as DATABASE\_URL and PROCESSED\_IMAGES\_DIR.

### **6.2. Testing**

* **API Testing:**
  + Use Postman or cURL to test the /upload, /status/{request\_id}, and /download/{request\_id} endpoints.
  + Validate CSV input and check response formats.
* **Webhook Testing:**
  + Use RequestBin or a similar service to act as a dummy webhook receiver.
  + Upload a CSV with a webhook URL, monitor the RequestBin log to confirm receipt of notifications.
* **Unit & Integration Tests:** Write tests for:  
  + CSV validation logic.
  + Image processing and compression.
  + Database operations.
  + Asynchronous worker functionality.
  + Webhook notification handling.

## **7. Conclusion**

This technical design document outlines the overall architecture and component interactions for the image processing system. The use of asynchronous processing combined with webhook notifications allows the system to handle tasks efficiently and provide real-time updates to external systems. The detailed API and worker documentation ensure that developers and testers have clear guidelines for implementation and validation.

You can use this document as the foundation for further technical discussions and improvements to the system architecture.