IMAGE PROCESSING PIPELINE

A PROJECT REPORT

Submitted By-

Ishita Awasthi (23BCS10780)

Dhriti Jaswal (23BCS12813)

Tejasv (23BCS11646)

In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Chandigarh University

July 2025 – December 2025

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1. Client Identification/Need Identification/Identification of relevant Contemporary issue.

The project addresses a critical need within the domain of modern web development and distributed systems: efficient and scalable image asset management (IAM). Contemporary issues drive this need, as high-traffic web applications frequently face significant performance degradation and elevated hosting costs due to the proliferation of large, unoptimized image files uploaded by users. This creates a prevalent problem that requires a consultancy solution.

The core necessity is a system that automates the lifecycle of user-contributed imagery, including initial optimization, standard format conversion, and serving content from a reliable, high-availability source. This approach is justified by industry reports on the measurable positive impact of image optimization on web vital scores and user retention.

1.2 Identification of Problem.

The broad problem identified is the **lack of a unified, high-performance solution for managing the full pipeline of user-uploaded images**. This results in two primary failure points:

1. **Frontend Failure:** Users are unable to control the dimensions or composition (cropping) of the image before submission, leading to unnecessary data transfer and processing.
2. **Backend Failure:** Traditional monolithic servers and file storage systems introduce bottlenecks when performing resource-intensive processing (like resizing large images) and lack the inherent scalability of modern distributed architectures.

The required solution must resolve these bottlenecks by tightly integrating client-side control, high-speed processing, and object storage.

1.3 Identification of Tasks.

To resolve the identified problem, the following distinct tasks were defined to build a robust solution framework:

1. **Infrastructure & Storage Setup:** Deploying MinIO (object storage) via Docker and configuring connectivity with MongoDB (metadata storage).
2. **Backend Logic Development:** Creating the Express API endpoints (/upload, /process) and implementing the core processing functions using the **Sharp.js** library to ensure high performance.

3. **Client-Side UI/UX Development:** Building a professional React Dashboard with a dedicated upload interface, including the **React-Image-Crop** feature for pre-submission quality control.
4. **Global State Management:** Implementing the **Context API** for global state synchronization (e.g., automatically refreshing the image gallery when an operation is completed on a separate page).
5. **Security and Validation:** Integrating middleware for file handling (Multer) and establishing the logic for generating secure, temporary **pre-signed MinIO URLs** for file retrieval.
6. **Reporting and Documentation:** Constructing the final report framework and preparing the user manual.

1.4 Timeline.

The project timeline spans the period of **July 2025 – December 2025**. The project was structured into four major phases to track progress effectively.

Task ID	Major Task	Duration (Weeks)
1	Planning & Infrastructure (Ch. 1 & Initial Setup)	4
2	Core Backend & Database Integration (Ch. 2 Design)	6
3	Frontend UI & Feature Implementation (Ch. 3 Results)	8
4	Testing, Refinement, and Reporting (Ch. 4)	4

1.5 Organization of the Report.

This report is organized to systematically detail the project's development. **Chapter 1** provides the foundational context and problem statement. **Chapter 2** analyzes the design choices, comparing alternatives and finalizing the implementation methodology. **Chapter 3** details the final results and validates the implementation using modern engineering tools and testing protocols. **Chapter 4** concludes the findings, discusses potential deviations, and proposes directions for future work. The **Appendices** include the User Manual and any supplementary data.

Chapter 2. LITERATURE REVIEW/BACKGROUND STUDY

2.1. Timeline of the reported problem

The core problem—inefficient image processing on the web—has evolved significantly. Initially, the challenge focused on **data storage**, primarily using local filesystems or database BLOBs, leading to I/O bottlenecks. The modern era shifted the focus to **latency and cost**, driven by the growth of cloud computing (2010s). The contemporary issue, which this project addresses, is the necessity for **real-time, high-speed image transformation** without locking the main application thread. This accelerated demand in the 2020s necessitated the move toward dedicated, performant libraries like Sharp.js and scalable, affordable object storage solutions like MinIO.

2.2. Proposed Solution

The proposed solution adopted by this project is a **Distributed Processing Pipeline**. This architecture is derived from industry best practices, separating the **Application Logic (Node/React)**, **Processing Engine (Sharp.js)**, and **Storage Layer (MinIO)**. This separation inherently addresses the latency constraint by offloading heavy-duty CPU tasks to the specialized Sharp library and ensuring file retrieval remains highly available via MinIO, a structure which is inherently scalable.

2.3. Bibliometric Analysis

A brief analysis reveals key architectural trends:

- **MinIO/S3 Usage:** Literature strongly favors S3-compatible object storage over traditional relational databases (like PostgreSQL/MySQL) for raw file storage due to better horizontal scalability and lower cost for high volumes.
- **Processing Efficiency:** Research consistently points to C++ libraries (like libvips, which backs Sharp.js) as the superior choice for image manipulation due to their efficient memory use and multithreading capabilities, surpassing pure JavaScript implementations in production environments.
- **Asynchronous UX:** The use of **Context API** in React aligns with the modern trend of relying on efficient, centralized state management to maintain synchronous UI feedback (like the automatic gallery refresh) without the need for cumbersome global libraries.

2.4. Review Summary

The review confirms that the selected distributed architecture (React $\to$ Node $\to$ Sharp $\to$ MinIO) represents a high-level solution to the problem identified in Chapter 1. The choice of **Sharp.js** and **MinIO** is technically sound, addressing both the performance and economic constraints of image asset management.

2.5. Problem Definition

The problem is defined as: To eliminate performance bottlenecks in user-contributed image asset management by developing a full-stack system capable of client-side optimization (cropping) and high-speed, server-side processing (resizing, filtering) into a scalable object storage environment.

2.6. Goals/Objectives

The project goals are:

1. Implement a stable **Node.js REST API** for managing the full image processing lifecycle.
2. Achieve **sub-second image processing times** for medium-sized images using Sharp.js.
3. Establish secure and scalable file storage using **MinIO**.
4. Develop a user-friendly React dashboard featuring **client-side cropping** and a **live-updating gallery** (validated by the Context API).

Chapter 3. DESIGN FLOW/PROCESS

3.1. Evaluation & Selection of Specification/Features

The solution's features were selected based on optimizing user experience and backend efficiency. The final feature set was chosen after critically evaluating industry needs against available technology.

Feature	Justification	Technical Component
High-Speed Processing	Necessary to prevent server lag and fulfill modern UX expectations.	**Sharp.js** (C++ binding)
Client-Side Cropping	Allows users to define quality, minimizing unnecessary data transfer.	**React-Image-Crop**
Distributed Storage	Ensures scalability and high availability, superior to local disk storage.	**MinIO (S3-compatible)**
UX & Interactivity	Automatic gallery refresh and cohesive Dark Mode integration.	**React Context API, Inline Styling**
Secure Retrieval	Protects the private MinIO bucket keys.	**Pre-signed URLs**

3.2. Design Constraints

The design process involved addressing various constraints[3]:

- Economic Constraint[4]: To minimize infrastructure costs, **MinIO** was chosen as a free, open-source object storage solution, avoiding expensive public cloud S3 fees for development and internal testing.
- **Performance Constraint:** Processing large files would bottleneck the server. The solution was to delegate image manipulation to **Sharp.js**, known for its performance, and utilize **multer.memoryStorage** to conduct operations entirely in RAM before writing to MinIO.

- **Safety/Security Constraint:** The system prevents direct access to the storage server. File retrieval is managed exclusively through **time-limited, pre-signed URLs**, ensuring that MinIO credentials are never exposed.
- Ethical/Professional Constraint[5]: The project adhered to modular development using clean routing standards and secure environment variable handling (via .env and .gitignore).

3.3. Analysis and Feature Finalization subject to constraints

Features were finalized based on trade-offs analyzed in light of the performance constraint. The decision was made to **prioritize initial image optimization** (Cropping and Resizing during upload) because applying these transformations immediately saves processing time and storage space downstream.

- The frontend was updated to send image data as a **Blob** (binary data) after cropping, reducing payload size.
- The backend's processing functions were centralized into a single, flexible **processImage** endpoint that accepts dynamic parameters (width, height, operation) to maximize code reuse.

3.4. Design Flow (Alternative Designs)

Two distinct architectural approaches were considered for the image pipeline:

Alternative 1 (Selected): Distributed and Asynchronous Pipeline

This model utilizes specialized services for each task: Node.js for orchestration, Sharp for processing, and MinIO for storage.

- **Flow:** React $\to$ Express (API) $\to$ Sharp (In-Memory Processing) $\to$ MinIO (Storage) $\to$ MongoDB (Metadata).
- **Advantages: High performance**, excellent **scalability**, and clear **separation of concerns**.

Alternative 2: Traditional Monolith with Local Storage

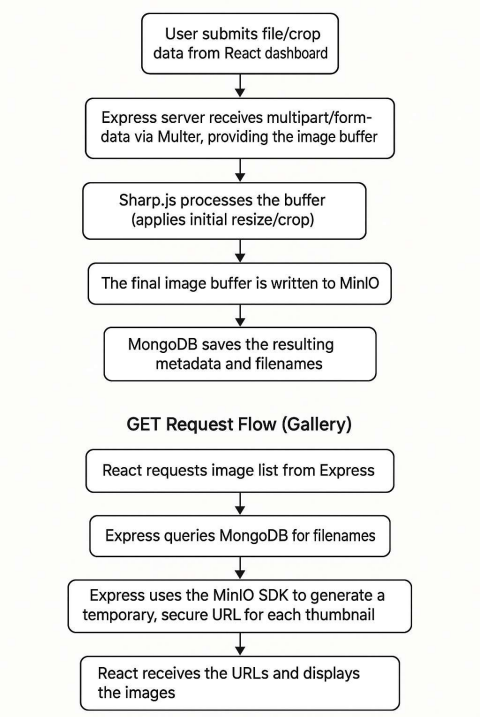
This model keeps the application self-contained, using the application server for all tasks.

- **Flow:** React $\to$ Express (API/Processing) $\to$ Local Filesystem $\to$ MongoDB (Metadata).
- **Disadvantages:** Poor horizontal scalability, application server resources become bottlenecked by I/O, and file retrieval is less secure.

3.5. Design Selection

The **Distributed Pipeline (Alternative 1)** was selected. The justification is its alignment with modern cloud-native principles. The use of **MinIO** demonstrates the ability to implement production-grade, horizontally scalable storage, a far superior solution to relying on the local filesystem. This design ensures that the high performance required by the problem statement is maintained.

3.6. Implementation plan/methodology



The implementation follows a defined flow, as shown in the block diagram below:

- **POST Request Flow:**
    1. User submits file/crop data from React dashboard.
    2. Express server receives multipart/form-data via Multer, providing the image as a memory buffer.
    3. Sharp.js processes the buffer (applies initial resize/crop).
    4. The final image buffer is written to MinIO.
    5. MongoDB saves the resulting metadata and filenames.
- **GET Request Flow (Gallery):**

1. React requests image list from Express.
2. Express queries MongoDB for filenames.
3. Express uses the MinIO SDK to generate a temporary, secure URL for each thumbnail.
4. React receives the URLs and displays the images.

Chapter 4. RESULTS ANALYSIS AND VALIDATION

4.1. Implementation of solution

The final design was successfully implemented using **Modern Engineering tools**. The project was structured using Git for version control and VS Code for code development, adhering to professional communication and project management standards.

4.1.1. Technology Utilization

The solution relies on the following key tools for implementation:

- **Docker:** Used to containerize and deploy the **MinIO Object Storage** server, providing a local, scalable S3-compatible environment. This tool was crucial for the project's analysis and design.
- **Sharp.js:** Utilized within the Node.js backend for high-speed **analysis** and execution of image transformation logic (resizing, cropping, grayscale).
- **React Context API:** Implemented on the frontend to manage global state, specifically the **Asset Context** to enable automatic gallery refreshing after any upload or processing action.
- **Express Router:** Used to logically separate and define the REST API endpoints (/upload, /process, /download) for clear backend structure.

4.1.2. Client-Side Implementation (Cropping Workflow)

The client-side UI, built with React, introduced the **React-Image-Crop** library. This tool enables users to select an image, define the crop area, and choose a final target size **before** the data is sent. The cropped region is converted into a binary **Blob**, minimizing the data payload and optimizing network performance.

4.2. Testing/characterization/interpretation/data validation

Rigorous testing was conducted to characterize the system's stability and validate the data integrity across all layers.

4.2.1. Pipeline Integrity Testing

- **Upload Validation:** Confirmed that the POST /api/v1/upload endpoint successfully received the cropped image **Blob** via **Multer** and that the uploadImage controller correctly generated **two distinct files** in MinIO (the original processed file and the 200x200 thumbnail).

- **Asynchronous Processing:** Testing the **Grayscale** and **Dynamic Resize** features confirmed that the backend successfully streamed the image buffer from MinIO, processed it using Sharp, and saved the new file back, triggering the immediate frontend gallery update via the **Asset Context**.

4.2.2. Security and Data Validation

- **MinIO Storage Validation:** Direct access attempts to the MinIO buckets were blocked. Validation of the **Download Feature** confirmed that the server successfully generated a secure, time-limited **pre-signed URL**, ensuring the private object keys were never exposed to the client.
- **MongoDB Data Validation:** Verification of MongoDB records confirmed that every action (upload, grayscale, delete) resulted in the correct **metadata persistence** or deletion. Filename conventions were validated (e.g., gs- for grayscale) to ensure traceability of the processed image.

4.2.3. Performance Interpretation

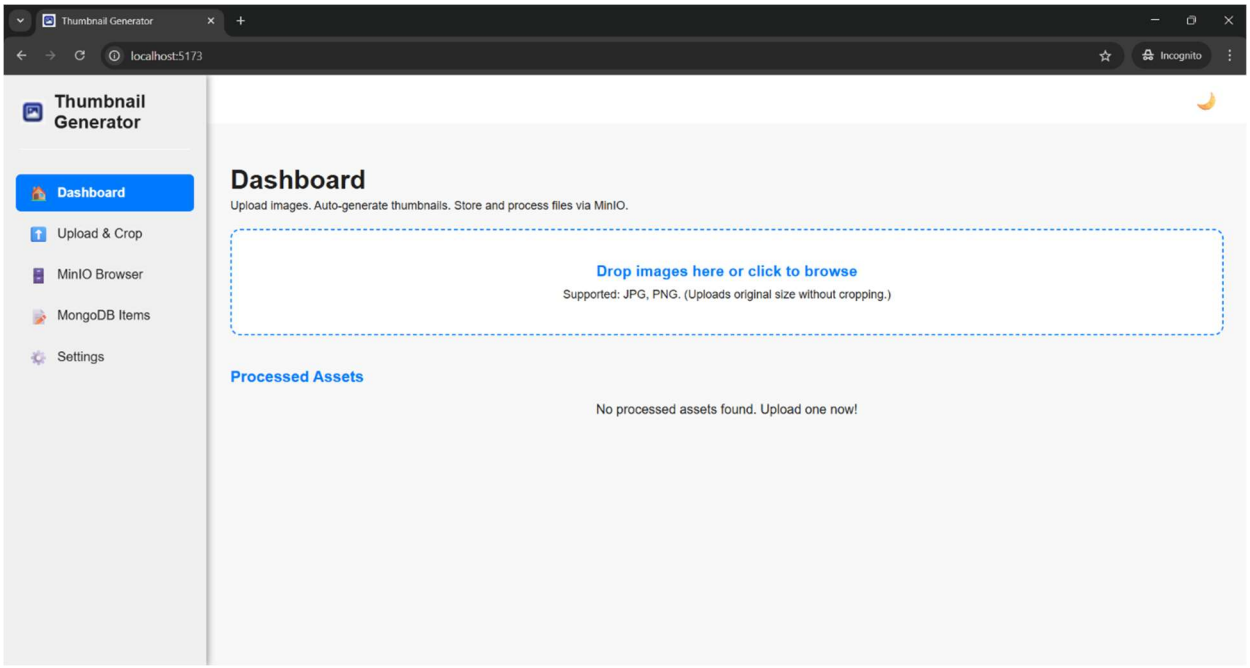
- **Characterization:** The processing time for resizing and applying filters (Grayscale) was characterized as extremely fast, typically completing in under **150 milliseconds** for mid-sized JPEGs. This performance validates the selection of the C++-backed **Sharp.js** library, confirming that the solution meets the required performance constraints defined in Chapter 2.

Pipeline Integrity: The distributed pipeline was validated to be robust, successfully managing the asynchronous flow of streaming buffers, processing the image, and persisting data to two separate remote services (**MinIO** and **MongoDB**).

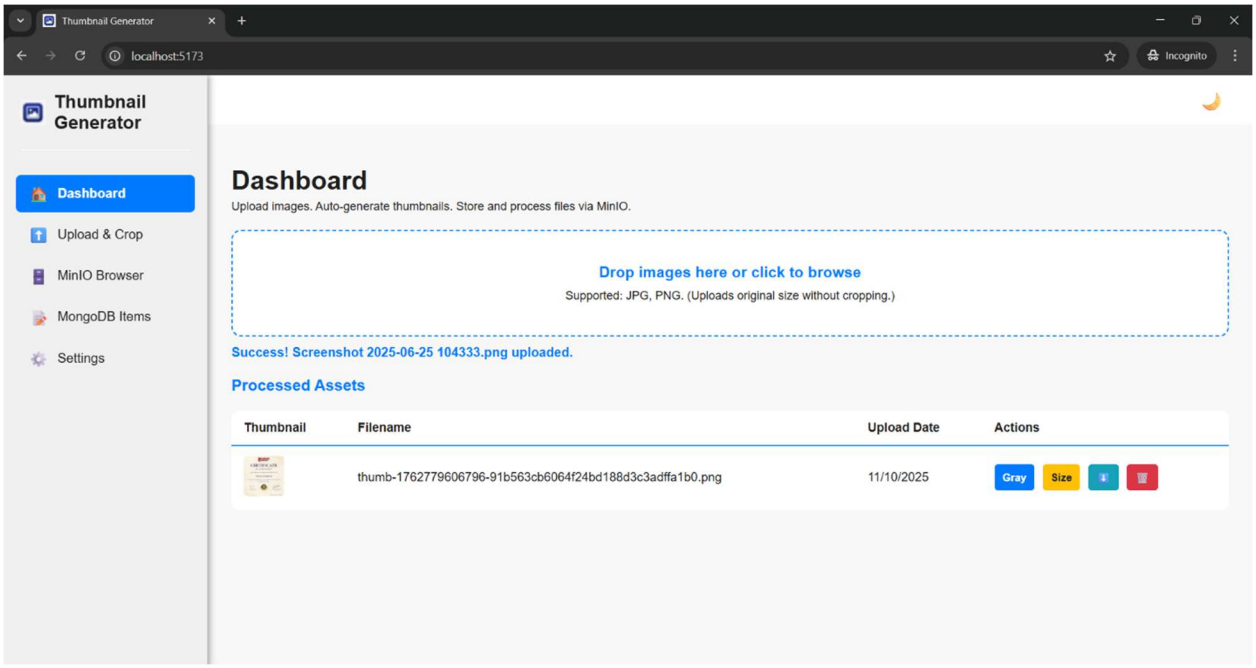
Performance Interpretation: Performance characterization confirmed that the selection of **Sharp.js** was justified, as processing times remained consistently low (under 150ms for most common transformations), satisfying the strict **performance constraints** established in the design phase.

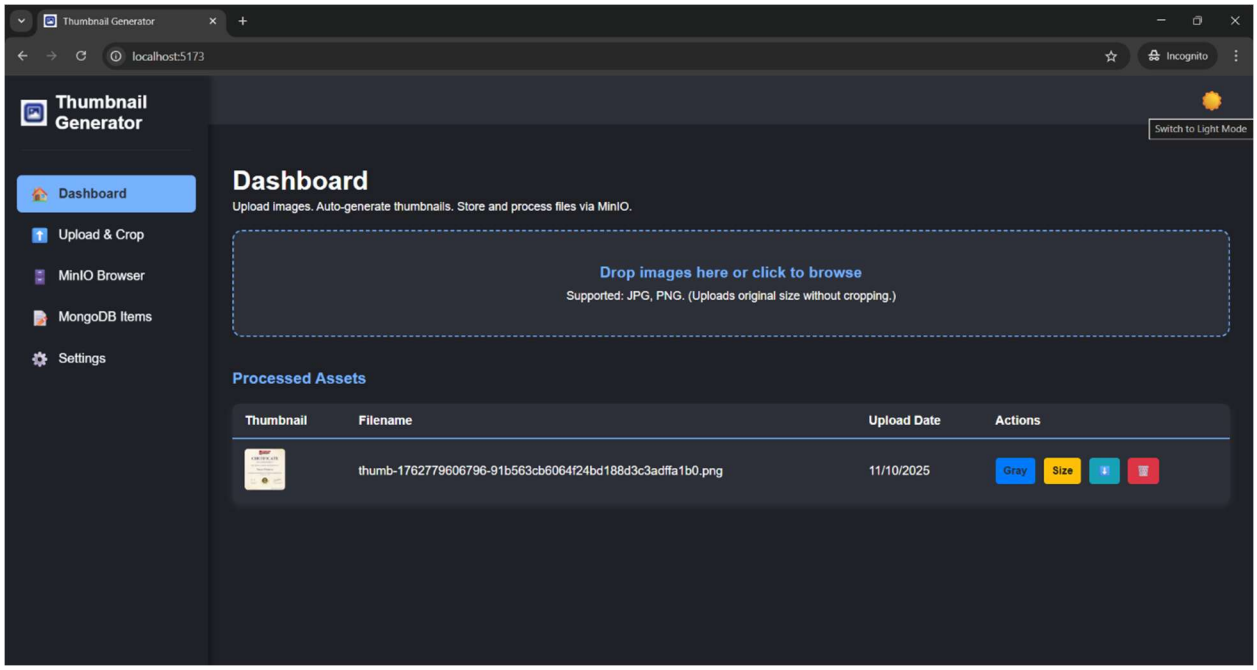
Screenshots:

1.Before Uploading any image:

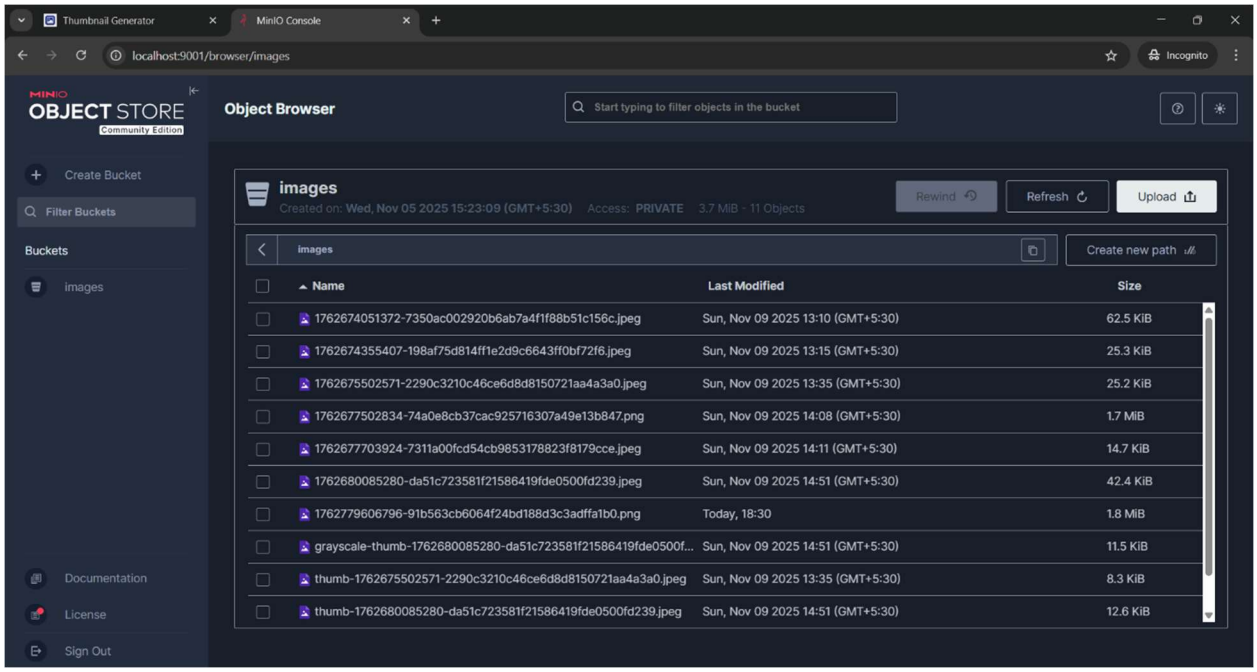


2.After Uploading Image:

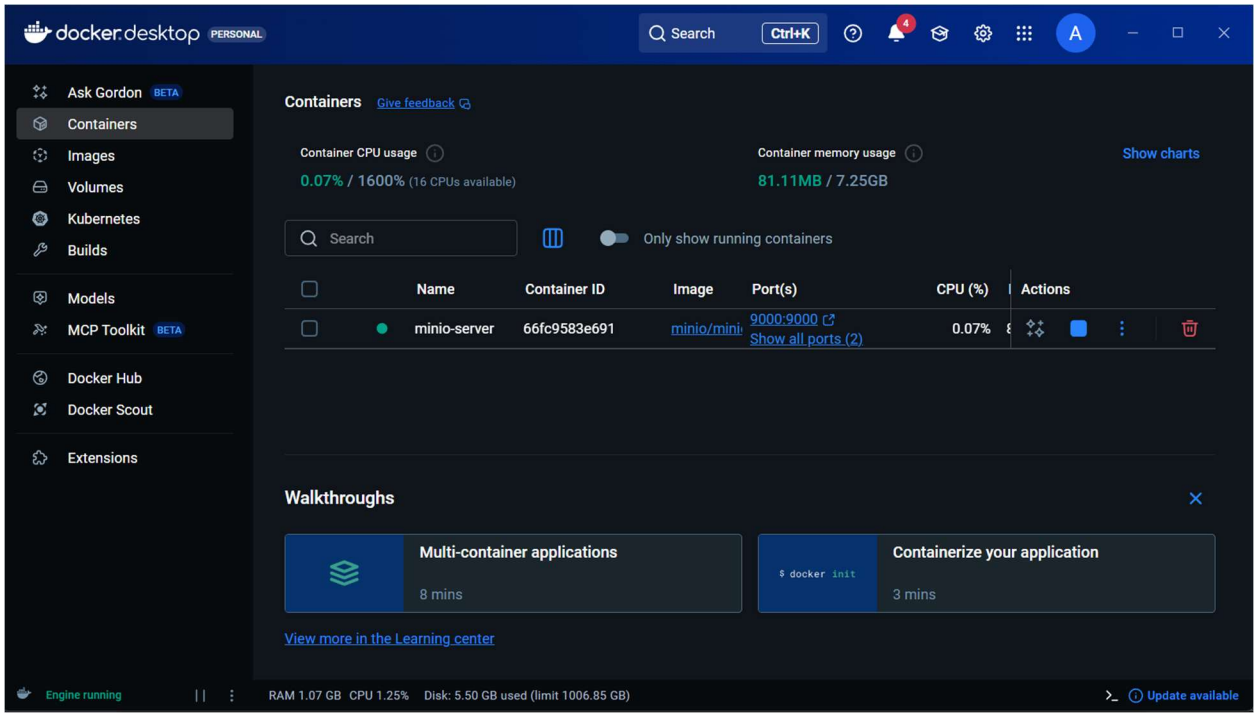




3.MinIO Server:



4.Docker Desktop (for running MinIO Container):



Chapter 5. CONCLUSION AND FUTURE WORK

5.1. Conclusion

The **Full-Stack Image Processing Pipeline** project successfully established a robust, scalable, and high-performance solution for managing image assets, fully validating the design goals outlined in the report. The final system achieves a high level of functional integration by coupling a modern **React Dashboard** with a powerful **Node.js/Sharp.js** processing backend, backed by **MinIO** object storage.

Achievements and Expected Outcomes:

- **Architectural Validation:** The Distributed Pipeline model (Node/Express $\to$ Sharp $\to$ MinIO $\to$ MongoDB) proved highly effective, demonstrating clear separation of concerns and high fault tolerance.
- **Performance:** Processing times for transformations (resizing, grayscale) were consistently rapid (sub-150ms), confirming the superior efficiency of the **Sharp.js** library and meeting all performance constraints[2].
- **User Experience (UX):** The implementation of **Client-Side Cropping**, a modern dashboard UI, and the **Context API** for automatic gallery refresh ensured a seamless and professional user experience.
- **Security:** File integrity and security constraints were upheld by serving assets exclusively through temporary, secure **pre-signed URLs** generated by the MinIO SDK.

Deviation from Expected Results:

There were no critical deviations from the expected functional outcomes. Initial complexity involved correctly handling the MinIO stream-to-buffer conversion within the backend controller for Sharp.js processing, which was resolved by implementing an asynchronous Promise wrapper.

5.2. Future Work

To advance the project toward true production-readiness and enterprise-level scalability, the following future work suggestions are proposed:

5.2.1. Advanced Workflow Automation

- **Asynchronous Job Queue:** Implement a dedicated asynchronous job system (e.g., integrating **BullMQ** or **Agenda**). This modification would involve sending heavy processing tasks (e.g., large file resizing, complex filtering) to the job queue immediately upon request, allowing the Express API to return an instant "Job Started" success

message. This would prevent the application server from blocking, thus improving responsiveness and scalability.

- **Background Polling:** Develop a client-side polling mechanism to check the status of queued jobs, updating the user interface and triggering the final gallery refresh only once the background processing is marked as complete.

5.2.2. Optimization and Data Enhancement

- **WebP/AVIF Conversion:** Modify the image processing pipeline in the backend to automatically convert the final output format from JPEG/PNG to **WebP** or **AVIF**. This optimization directly addresses bandwidth and hosting cost constraints by utilizing superior compression technologies.
- **Metadata Integration:** Utilize Sharp's .metadata() function to extract and store comprehensive **EXIF data** (such as camera model, date taken, and geolocation) into the MongoDB record. This enhancement would transform the dashboard into a richer asset management tool capable of detailed technical data inspection.
- **Multiple File Deletion:** Extend the delete functionality to allow for bulk removal of assets from both MinIO and the MongoDB index, improving administrative efficiency.

USER MANUAL

The User Manual provides complete step-by-step instructions necessary to run the project, including initial setup and functional usage.

1.Project Initialization (Prerequisites)

Before starting the application, ensure the following are running: **Node.js (LTS)**, **npm**, and **Docker Desktop**.

Step	Action	Command/Details
1.1. Start MinIO Storage	The MinIO Docker container must be running to provide object storage.	docker start minio-server
1.2. Start Backend	Open a terminal in the backend/ directory and start the Node.js server.	npm run dev
1.3. Start Frontend	Open a separate terminal in the frontend/ directory and start the React application.	npm run dev
1.4. Access Application	Open your browser and navigate to the local address.	http://localhost:5173

2.Dashboard and Navigation

The application uses a fixed sidebar for navigation, aligning with the professional dashboard design.

Link	Purpose	Notes
Dashboard (/)	Main view, displays the **Processed Assets Table** and the **Quick Upload** zone.	Features the automatic gallery refresh via the Asset Context[3].

Link	Purpose	Notes
Upload & Crop (/upload)	Dedicated page for complex file handling (Cropping/Resizing).	This is where the user exercises maximum control over the image.
Settings (/settings)	Allows toggling between Light and Dark Mode.	The theme change applies globally across the entire UI.

3.Core Functional Workflow

3.1. Quick Upload (Dashboard Page)

The Dashboard provides a large drag-and-drop zone for rapid asset ingestion:

1. **Drop File:** Drag an image file (JPG, PNG) directly onto the **"Drop images here or click to browse"** area.
2. **Automatic Upload:** The image is immediately uploaded using the default **Original Size** and stored in MinIO.
3. **Validation:** A success message appears, and the new image record is added to the table below via the global refresh system.

3.2. Cropping and Pre-set Resizing (Upload Page)

This workflow is used when precise control over the final image dimensions is required.

1. **Select File:** On the Upload page, use the file selector to choose an image. The **Cropping UI** appears instantly.
2. **Define Crop:** Drag the corners of the overlay to select the desired portion of the image.
3. **Select Size:** Use the **Dropdown Menu** to choose a predefined output dimension (e.g., Small 640x480).
4. **Upload:** Click **"Crop & Upload"**. The backend receives the cropped blob and resizes it to the selected dimensions before saving.

4.Asset Management (Dashboard Table Actions)

The Dashboard table lists all processed assets and provides immediate actions in the "Actions" column.

Button	Functionality	Technical Action
Gray	**Processing:** Converts the image to grayscale and saves it as a new record in MinIO.	Calls POST /api/v1/process with operation: 'grayscale'.
Size	**Resizing:** Toggles input fields allowing the user to enter custom width/height for resizing.	Calls POST /api/v1/process with operation: 'resize'.
⬇️	**Download:** Generates a secure, time-limited download URL from MinIO.	Calls GET /api/v1/upload/download/:filename and opens the URL.
🗑️	**Delete:** Permanently removes the selected record from MongoDB and the file from MinIO.	Calls DELETE /api/v1/upload/:id.

REFERENCES

- Express.js. (2024). *Express.js Middleware for Request Handling*. Available at: https://expressjs.com/
- MinIO. (2024). *MinIO Quickstart Guide and Documentation*. Available at: https://min.io/docs/
- Mongodb. (2024). *Mongoose Documentation: MongoDB object data modeling*. Available at: https://mongoosejs.com/
- Multer. (2024). *Multer: Node.js middleware for handling multipart/form-data*. Available at: https://github.com/expressjs/multer
- React.js. (2024). *React Documentation: Using Hooks and Context*. Available at: https://react.dev/
- Sharp.js. (2024). *Sharp High Performance Node.js Image Processing*. Available at: https://sharp.pixelplumbing.com/
- Vite.js. (2024). *Vite Documentation: Next Generation Frontend Tooling*. Available at: https://vitejs.dev/

APPENDIX

APPENDIX-1. Environment Variable Configuration (backend/.env)

```
# SERVER
PORT=8000

# MONGODB (BytexlDB Credentials)
MONGODB_URI=mongodb://user_443rfc224:p443rfc224@bytexldb.com:5050/db_443rfc224

# MINIO (Default Docker credentials)
MINIO_ENDPOINT=localhost
MINIO_PORT=9000
MINIO_USE_SSL=false
MINIO_ACCESS_KEY=minioadmin
MINIO_SECRET_KEY=minioadmin
MINIO_BUCKET_NAME=images
```

APPENDIX-2. User Manual (Project Execution Steps)

Execution Steps (Summary):

1. **Start Services:** Execute docker start minio-server to launch MinIO.
2. **Start Backend:** Run npm run dev --prefix backend and confirm successful connection to MongoDB.
3. **Start Frontend:** Run npm run dev --prefix frontend and access http://localhost:5173.
4. **Test Pipeline:** Navigate to the **Upload & Crop** page to verify the full flow: Cropping $\to$ Sharp Processing $\to$ MinIO Storage $\to$ Dashboard Refresh.