PROJECT REPORT

Mosaic Generator

Bachelor of Engineering in Computer Engineering

Department of Computer Engineering
Goa College of Engineering
Farmagudi, Goa

May 2025

Name	Roll No.
Soham Ghotge	22B-CO-030
Shubham Chede	22B-CO-057
Vishnu Variyar	22B-CO-072

Academic Year: 2024 - 25

Contents

Chapter 1

Introduction

1.1 Problem Definition

Creating mosaics from digital images is a visually compelling technique that requires arranging small image tiles in a manner that collectively replicates the appearance of a larger, target image. Traditional mosaic generation tools often fall short in customization, visual quality, and user interaction. They typically provide limited control over resolution, color fidelity, and tile selection, which can result in mosaics that are either computationally inefficient or aesthetically unrefined. There exists a need for a more flexible, accurate, and user-friendly solution that allows users to generate artistic mosaics with advanced control over image quality and process flow.

1.2 Motivation

The growing interest in digital art, personalized design, and AI-assisted creativity has led to a demand for tools that can convert ordinary images into artistic forms while maintaining high visual coherence. Mosaics, as a medium, offer a unique blend of structure and abstraction, making them suitable for artistic prints, educational visuals, and media content. This project was motivated by the opportunity to improve upon existing mosaic generation methods by integrating modern web technologies with image processing algorithms to create a more powerful and accessible tool for both casual users and professionals.

1.3 Objectives

The primary objectives of our Mosaic Generator project are:

1. **Color Mosaic Generation**
    - Implement color mosaics using RGB images through dynamic color mapping.
    - Process each RGB channel independently for precise color adjustments.
    - Generate both dynamic (color-adjusted) and simple (unaltered) mosaic outputs for comparison.

2. **Fixed Block Size (16×16):** The system uses a standard block size of 16×16 pixels for tile placement, offering a balanced resolution suitable for most images.

3. **Visual Effect Filters**
   - Integrate a variety of post-processing filters (e.g., sepia, vintage, pop art) for artistic enhancement.
   - Display filter previews before application to support informed user choices.
   - Apply selected filters via backend API to enhance the final mosaic output.

4. **Quality Metrics**
   - Evaluate mosaic accuracy using Structural Similarity Index (SSIM).
   - Compute Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR) for numerical fidelity assessment.
   - Visualize quality metrics in the frontend to provide feedback on mosaic output quality.

5. **Step-by-Step Processing Workflow**
   - Divide the mosaic generation process into discrete steps: Upload → Preprocess → Generate → Enhance.
   - Enable better user control and understanding through sequential execution.
   - Support both guided and legacy one-step processing modes.

6. **Job Status Tracking**
   - Maintain job state information (status, progress, outputs) using server-side tracking.
   - Use a polling mechanism to update frontend UI with real-time job progress.
   - Collect and manage all intermediate and final outputs for traceability and user reference.

1.4 Tech Stack & Requirements

1.4.1 Backend

- **Python 3.6+**: Core programming language
- **Flask**: Web framework for the API
- **Flask-CORS**: Cross-Origin Resource Sharing support
- **NumPy**: Matrix operations for image processing
- **Pillow (PIL)**: Image loading, manipulation, and saving
- **OpenCV (cv2)**: Advanced image processing algorithms
- **scikit-image**: Image quality metrics (SSIM)
- **matplotlib**: Graph generation for metric comparisons

1.4.2 Frontend

- **Next.js**: React framework for the UI
- **Tailwind CSS**: Utility-first CSS framework for styling
- **React Hooks**: State management (useState, useEffect)
- **Fetch API**: Communication with the backend

Chapter 2

Literature Study

The **Enhanced Mosaic Generator** is based on core concepts from image processing, supported by modern web technologies and algorithms for image transformation. This literature study explores the foundational techniques and tools that influenced the development of the system.

2.1 Image Processing Concepts

- **Mosaic Generation**: The concept of using small image tiles to reconstruct a larger image is inspired by traditional photomosaics. The key technique involves dividing the target image into blocks and replacing each with a version of the tile image that best matches its color or intensity.
- **Color Matching**: RGB channel processing is commonly used in photomosaic generation. Each channel (Red, Green, Blue) is analyzed separately, and the average pixel intensity is used to adjust the tile images for accurate color reproduction.
- **Post-Processing Filters**: Applying visual filters like sepia, grayscale, and vintage effects is a popular enhancement step in digital image editing. These are typically achieved using pixel-wise transformations and blending techniques.

2.2 Quality Metrics in Image Processing

To evaluate the similarity between the mosaic and the original image:

- **SSIM (Structural Similarity Index)** is used to measure perceptual similarity.
- **MSE (Mean Squared Error)** and **PSNR (Peak Signal-to-Noise Ratio)** are standard metrics in digital image comparison, helping assess how closely the generated mosaic resembles the input.

These metrics are supported by libraries like **scikit-image** and are commonly referenced in academic research on image quality assessment.

2.3 Technologies and Software Tools

- **Python & Flask**: The backend processing is implemented using Python, with Flask serving as the lightweight web framework to handle APIs and image operations.
- **Pillow (PIL)**: An image processing library in Python used for resizing, filtering, and manipulating images.
- **Next.js**: A modern React framework used for building the frontend interface, allowing seamless interaction with the backend and dynamic updates.
- **scikit-image**: A Python library used for image analysis, especially for calculating metrics like SSIM, MSE, and PSNR.

2.4 Relevant Research and Applications

- **”Photomosaic Creation Using Image Processing Techniques” (IEEE, 2019)** – Highlights the use of color matching and block-based replacement for creating photo mosaics.
- **”A Study on Mosaic Image Generation Algorithms” (IJCA, 2020)** – Explores various matching algorithms and filter enhancements used in artistic mosaics.
- **Open-source tools like Metapixel and AndreaMosaic** – Earlier tools that demonstrated basic tile-based image reconstruction but lacked advanced features such as dynamic filtering or quality metrics.

Chapter 3

Design

3.1 System Architecture

This chapter outlines the architectural design and workflow of the Mosaic Generator project. The system follows a client-server architecture with clear separation between frontend and backend components.

3.2 Workflow

The mosaic generation process follows these key steps:

1. **Image Uploading**: Users upload a target image and optionally a tile image
2. **Image Preprocessing**: The system analyzes and prepares both images
3. **Mosaic Generation**: The algorithm creates a mosaic by mapping tiles to the target image
4. **Filter Application**: Optional post-processing effects are applied
5. **Quality Assessment**: Metrics are calculated to evaluate the mosaic quality
6. **Result Delivery**: The final mosaic and metrics are presented to the user

3.3 Block Diagram

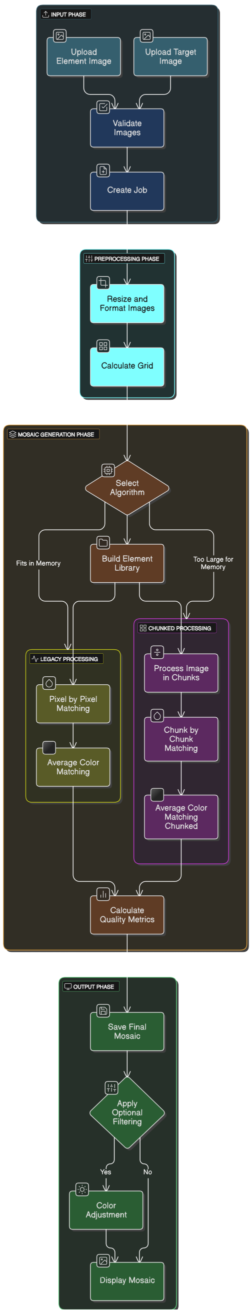

Figure 3.1: System Architecture Block Diagram

3.4 Algorithm Design

The mosaic generator employs two distinct algorithms for image transformation, intelligently selecting between them based on image size and memory requirements:

3.4.1 Legacy Mosaic Algorithm

This algorithm is simple but memory-intensive, best suited for smaller images:

- Creates a mosaic by replacing blocks in the target image with color-adjusted versions of the element image
- Processes the entire image at once, requiring significant memory for large images
- Performs pixel-by-pixel color adjustment for each block

3.4.2 Chunked Mosaic Algorithm

This algorithm offers memory-efficient processing for larger images:

- Divides processing into manageable chunks to reduce memory usage
- Processes the image row by row, with configurable chunk sizes
- Uses a pre-built element library for faster block matching
- Provides progress tracking with partial output generation

3.4.3 Algorithm Selection Logic

The system automatically chooses the appropriate algorithm based on:

- Estimated output image size (in pixels)
- Available system memory
- Processing time constraints

Chapter 4

Implementation

4.1 Test Cases

We evaluated the mosaic generator with three specific test cases to verify its performance with different image dimensions and configurations.

4.1.1 Test Case 1: Both Images Square

In this test case, both the target image and element image have square dimensions. This represents the ideal scenario for mosaic generation, as square images typically produce the most balanced and visually consistent results.

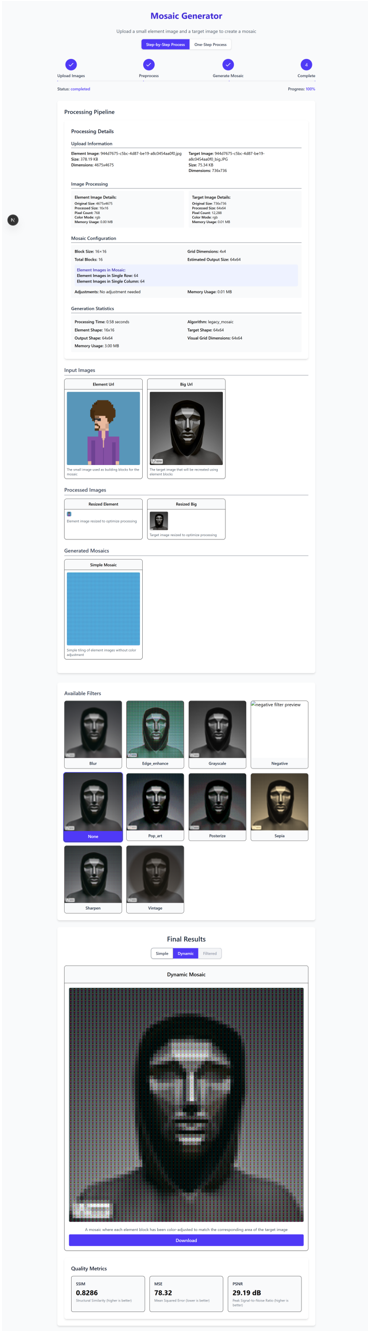

Figure 4.1: Test Case 1: Square Target and Element Images

4.1.2 Test Case 2: Target Image Non-Square

This test case evaluates how the system handles a non-square target image while using a square element image. The algorithm must properly map the element tiles across the rectangular dimensions of the target.

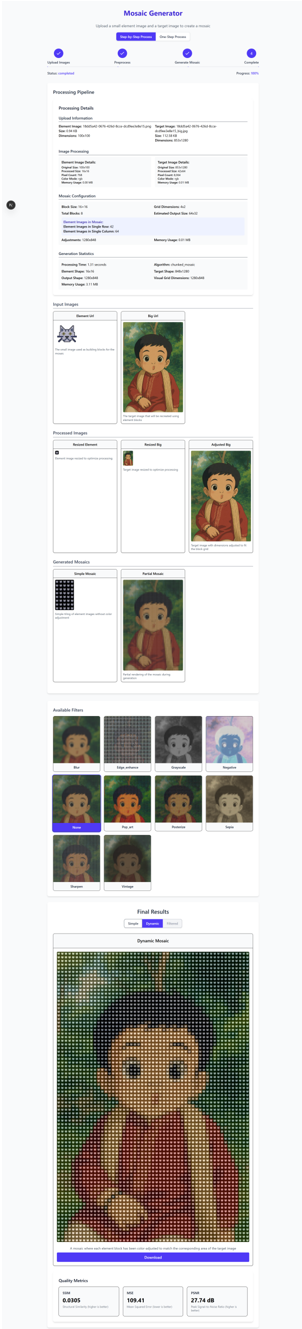

Figure 4.2: Test Case 2: Non-Square Target Image

4.1.3 Test Case 3: Element Image Non-Square

In this test, we use a square target image but a non-square element image. This tests the system's ability to properly handle and transform rectangular building blocks.

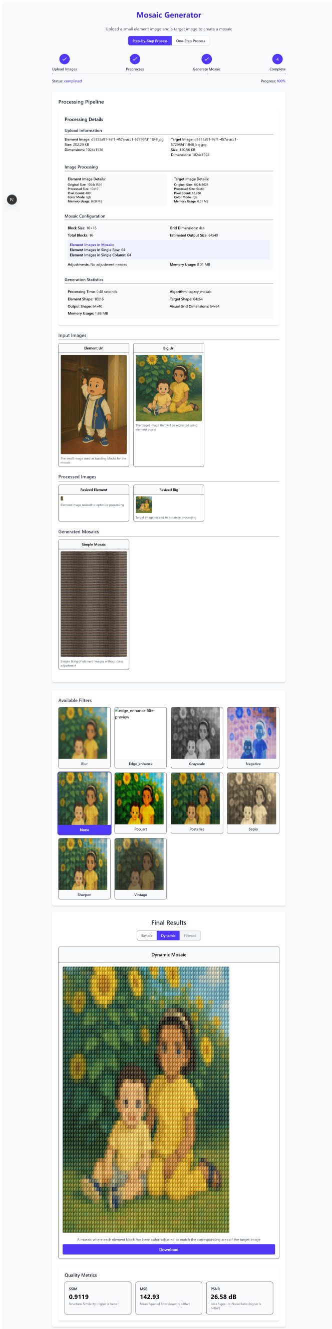

Figure 4.3: Test Case 3: Non-Square Element Image

4.2 Code Implementation

Our implementation consists of both backend Python code for image processing and frontend React components for user interaction.

4.2.1 Backend Code

Below are the key components of our backend implementation:

Quality Metrics Implementation

Listing 4.1: Quality Metrics Implementation

```
# From core/metrics.py
def evaluate_mosaic_quality(target_img, mosaic_img):
    """
    Calculate quality metrics between target image and generated mosaic.

    Args:
        target_img: Target image as numpy array
        mosaic_img: Generated mosaic as numpy array

    Returns:
        dict: Dictionary containing SSIM, MSE, and PSNR values
    """
    # Resize images to same dimensions if needed
    if target_img.shape != mosaic_img.shape:
        from PIL import Image
        mosaic_pil = Image.fromarray(mosaic_img)
        mosaic_pil = mosaic_pil.resize((target_img.shape[1], target_img.shape
            ↪ [0]), Image.LANCZOS)
        mosaic_img = np.array(mosaic_pil)

    # Calculate Structural Similarity Index
    ssim_value = compute_ssim(target_img, mosaic_img)

    # Calculate Mean Squared Error
    mse_value = compute_mse(target_img, mosaic_img)

    # Calculate Peak Signal-to-Noise Ratio
    psnr_value = compute_psnr(target_img, mosaic_img, mse_value)

    return {
        'ssim': ssim_value,
        'mse': mse_value,
        'psnr': psnr_value
    }
```

Structural Similarity Index (SSIM)

Listing 4.2: SSIM Implementation

```
# From core/metrics.py
def compute_ssim(img1, img2):
    """
    Compute Structural Similarity Index between two images.
    Higher values indicate greater similarity (range: 0-1).

    Args:
```

```
        img1, img2: Images to compare as numpy arrays

    Returns:
        float: SSIM value
    """
    # Constants for stabilization
    C1 = (0.01 * 255) ** 2
    C2 = (0.03 * 255) ** 2

    # Convert to float
    img1 = img1.astype(np.float64)
    img2 = img2.astype(np.float64)

    # Compute means
    mu1 = img1.mean()
    mu2 = img2.mean()

    # Compute standard deviations
    sigma1 = np.std(img1)
    sigma2 = np.std(img2)

    # Compute covariance
    sigma12 = ((img1 - mu1) * (img2 - mu2)).mean()

    # Compute SSIM
    ssim = ((2 * mu1 * mu2 + C1) * (2 * sigma12 + C2)) / \
           ((mu1 ** 2 + mu2 ** 2 + C1) * (sigma1 ** 2 + sigma2 ** 2 + C2))

    return ssim
```

Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR)

Listing 4.3: MSE and PSNR Implementations

```
# From core/metrics.py
def compute_mse(img1, img2):
    """
    Compute Mean Squared Error between two images.
    Lower values indicate higher similarity.

    Args:
        img1, img2: Images to compare as numpy arrays

    Returns:
        float: MSE value
    """
    return np.mean((img1 - img2) ** 2)
```

```
def compute_psnr(img1, img2, mse=None):
    """
    Compute Peak Signal-to-Noise Ratio between two images.
    Higher values indicate higher quality (measured in dB).

    Args:
        img1, img2: Images to compare as numpy arrays
        mse: Pre-computed MSE value (optional)

    Returns:
        float: PSNR value in decibels
    """
    if mse is None:
        mse = compute_mse(img1, img2)

    if mse == 0:
        return float('inf')

    # Assuming images with 8 bits per pixel (max value of 255)
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))

    return psnr
```

Legacy Mosaic Algorithm (Simple but Memory-Intensive)

Listing 4.4: Legacy Mosaic Algorithm

```
# From core/legacy_mosaic.py
def create_mosaic(element_img, big_img):
    """
    Generate a mosaic image that looks like big_img but is made of element_img
        ↪ blocks

    Args:
        element_img: The building block image (numpy array)
        big_img: The target image (numpy array)

    Returns:
        tuple: (mosaic, simple_mosaic)
    """
    # Handle RGB images
    if len(element_img.shape) == 3 and len(big_img.shape) == 3:
        N, M, C = element_img.shape
        H, W, _ = big_img.shape
```

```
        # Generate a simple mosaic
        simple_mosaic = create_image_matrix(element_img, (H, W))
        mosaic = simple_mosaic.copy()

        # Adjust each element block to match the target image's colors
        for i in range(H):
            for j in range(W):
                # For each RGB channel
                for c in range(C):
                    # Adjust the mean of the element_img to match the target pixel
                    element = adjust_element_mean(element_img[:,:,c], big_img[i, j
                        ↪ , c])
                    mosaic[i*N:(i+1)*N, j*M:(j+1)*M, c] = element

    return mosaic, simple_mosaic
```

Chunked Mosaic Algorithm (Memory-Efficient)

Listing 4.5: Memory-Efficient Chunked Mosaic Algorithm

```
# From core/mosaic.py
def create_mosaic(element_img, target_img, block_size, color_method='
    ↪ average_rgb',
                  adjust_colors=True, alpha=0.7, job_id=None, job_states=None):
    """Generate a mosaic with memory-efficient processing by working in chunks
        ↪ """
    # Get dimensions and calculate number of blocks
    target_h, target_w, channels = target_img.shape
    n_blocks_h = target_h // block_size
    n_blocks_w = target_w // block_size

    # Estimate memory requirements
    estimated_mem_mb = (n_blocks_h * block_size) * (n_blocks_w * block_size) *
        ↪ channels / (1024 * 1024)

    # Define chunk size to limit memory usage
    MAX_CHUNK_BLOCKS = 20 # Process 20 rows at a time
    chunk_size_h = min(MAX_CHUNK_BLOCKS, n_blocks_h)

    # Build element library for matching
    element_library = build_element_library(element_img, block_size, method=
        ↪ color_method)

    # Create a simple mosaic at smaller scale for preview
    preview_scale = 0.1 # 10% of full size
    preview_blocks_h = max(5, int(n_blocks_h * preview_scale))
    preview_blocks_w = max(5, int(n_blocks_w * preview_scale))
```

```
simple_mosaic = create_image_matrix(element_img, (preview_blocks_h,
    ↪ preview_blocks_w), block_size)

# Initialize the final mosaic
mosaic = np.zeros((n_blocks_h * block_size, n_blocks_w * block_size, 3),
    ↪ dtype=np.uint8)

# Process mosaic in chunks to limit memory usage
blocks_processed = 0
total_blocks = n_blocks_h * n_blocks_w

for chunk_start in range(0, n_blocks_h, chunk_size_h):
    chunk_end = min(chunk_start + chunk_size_h, n_blocks_h)

    # Process this chunk of rows
    for i in range(chunk_start, chunk_end):
        for j in range(n_blocks_w):
            # Update progress
            blocks_processed += 1
            if job_id is not None and job_states is not None:
                if blocks_processed % max(1, total_blocks // 50) == 0:
                    progress = 30 + (blocks_processed / total_blocks) * 60
                    job_states[job_id]['progress'] = progress

            # Extract target block
            h_start = i * block_size
            h_end = min((i + 1) * block_size, target_h)
            w_start = j * block_size
            w_end = min((j + 1) * block_size, target_w)

            target_block = target_img[h_start:h_end, w_start:w_end]

            # Get color feature and find best matching block
            color_feature = get_average_color(target_block)
            best_match = find_best_matching_block(color_feature,
                ↪ element_library)

            # Adjust colors if requested
            matched_block = best_match['block'].copy()
            if adjust_colors:
                matched_block = adjust_block_colors(matched_block,
                    ↪ color_feature, alpha)

            # Place in mosaic
            mosaic[h_start:h_end, w_start:w_end] = matched_block[:h_end-
                ↪ h_start, :w_end-w_start]

    # Force garbage collection after each chunk
```

```
        gc.collect()

        # Save partial mosaic for progress display
        if job_id is not None and job_states is not None:
            partial_filename = f"{job_id}_partial_mosaic.png"
            partial_path = get_file_path(partial_filename, 'temp')
            save_image(mosaic, partial_path)
            job_states[job_id]['intermediate_outputs']['partial_mosaic'] =
                ↪ get_file_url(partial_filename, 'temp')

    return mosaic, simple_mosaic
```

Algorithm Selection Logic

Listing 4.6: Intelligent Algorithm Selection Logic

```
# From api/generation.py
# Check if we should use the new memory-efficient implementation
from config import MAX_MOSAIC_PIXELS
estimated_pixels = output_h * output_w

job_states[job_id]['memory_info'] = {
    'estimated_output_size': f"{output_h}x{output_w}",
    'estimated_memory_gb': mem_gb,
    'estimated_pixels': estimated_pixels,
    'max_allowed_pixels': MAX_MOSAIC_PIXELS
}

# Use memory-efficient implementation if output would be too large
if estimated_pixels > MAX_MOSAIC_PIXELS / 2:
    # Use memory-efficient implementation with chunked processing
    mosaic, simple_mosaic = create_mosaic(
        element_img,
        big_img,
        block_size,
        color_method=color_method,
        adjust_colors=True,
        job_id=job_id,
        job_states=job_states
    )
else:
    # Generate mosaic using legacy implementation for smaller images
    mosaic, simple_mosaic = legacy_create_mosaic(element_img, big_img)
```

4.2.2 Frontend Code

The frontend implementation includes React components for user interaction and displaying quality metrics:

Quality Metrics Display Component

Listing 4.7: Quality Metrics UI Component

```
// From frontend/src/app/components.js
export function QualityMetrics({ jobId }) {
  const [metrics, setMetrics] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    if (jobId) {
      const fetchMetrics = async () => {
        setLoading(true);
        try {
          const result = await getQualityMetrics(jobId);
          setMetrics(result);
        } catch (error) {
          console.error("Failed to load quality metrics:", error);
        } finally {
          setLoading(false);
        }
      };

      fetchMetrics();
    }
  }, [jobId]);

  return (
    <div className="bg-white p-6 rounded-lg shadow-md mt-6">
      <h3 className="text-lg font-medium mb-3 text-gray-900">Quality Metrics</h3
        ↪ >

      <div className="grid grid-cols-1 md:grid-cols-3 gap-4">
        <div className="bg-gray-50 p-4 rounded-lg border">
          <div className="text-sm font-medium text-gray-900 mb-1">SSIM</div>
          <div className="text-2xl font-bold">
            {typeof metrics?.ssim === 'number' ? metrics.ssim.toFixed(4) : 'N/A
              ↪ '}
          </div>
          <div className="text-xs text-gray-500 mt-1">
            Structural Similarity (higher is better)
          </div>
        </div>
```

```
        <div className="bg-gray-50 p-4 rounded-lg border">
          <div className="text-sm font-medium text-gray-900 mb-1">MSE</div>
          <div className="text-2xl font-bold">
            {typeof metrics?.mse === 'number' ? metrics.mse.toFixed(2) : 'N/A'}
          </div>
          <div className="text-xs text-gray-500 mt-1">
            Mean Squared Error (lower is better)
          </div>
        </div>

        <div className="bg-gray-50 p-4 rounded-lg border">
          <div className="text-sm font-medium text-gray-900 mb-1">PSNR</div>
          <div className="text-2xl font-bold">
            {typeof metrics?.psnr === 'number' ? metrics.psnr.toFixed(2) + ' dB'
                ↪ : 'N/A'}
          </div>
          <div className="text-xs text-gray-500 mt-1">
            Peak Signal-to-Noise Ratio (higher is better)
          </div>
        </div>
      </div>
    </div>
  );
}
```

Chapter 5

Conclusion

The Enhanced Mosaic Generator successfully transforms input images into artistic mosaic representations using a fixed 16×16 tile approach. By combining a modular Flask backend with a responsive Next.js frontend, the system delivers both creative control and processing efficiency. Key features such as color-adjusted mosaics, visual filters, and quality metrics provide users with a rich and engaging image transformation experience.

This project demonstrates the potential of integrating traditional image processing techniques with modern web technologies to produce visually compelling digital art. The system is scalable, user-friendly, and offers a solid foundation for future enhancements in creative AI and image manipulation.

5.1 Future Work

To further enhance the capabilities and user experience of the Enhanced Mosaic Generator, the following improvements are proposed:

1. **Generative Mosaic Art via GPT/Claude APIs**: Integrate AI language models (like GPT or Claude) to generate intelligent mosaic design ideas, suggest tile types, or automatically pick filters based on user prompts.

2. **Drag-and-Drop Image Upload**: Streamline user interaction by enabling drag-and-drop functionality for uploading both the target image and tile image.

3. **Basic Tile Library**: Provide a built-in collection of element tiles (e.g., emojis, cartoon tiles, abstract textures) for quick mosaic generation without needing custom uploads.

4. **Image Crop & Zoom Tool**: Allow users to crop or zoom into a specific area of the target image before processing, which can reduce memory load and focus on meaningful sections.

5. **Randomized Tile Arrangement Option**: Introduce an optional artistic mode that applies slight random rotations or placements of tiles to produce abstract and creative mosaics.

References

[1] **Python Imaging Library (PIL / Pillow):** `https://pillow.readthedocs.io/en/stable/`

[2] **scikit-image – Image Quality Metrics:** `https://scikit-image.org/docs/stable/`

[3] **Flask Framework:** `https://flask.palletsprojects.com/`

[4] **Next.js Framework:** `https://nextjs.org/`

[5] **OpenAI – ChatGPT (Generative Idea Support):** `https://openai.com/chatgpt`

[6] **Claude AI by Anthropic (Design and code enhancement):** `https://claude.ai/`

[7] **Google & YouTube Tutorials (for development references):**

- YouTube keyword: *"Python mosaic generator tutorial"*
- Google Search: *"Flask image processing mosaic project"*