MONTCLAIR
STATE UNIVERSITY

IMAGE COLORIZATION USING AI

MASTERS PROJECT REPORT

MASTERS

In

COMPUTER SCIENCE

By

NEHA SHRI MEKA
(CWID: 50098185)

Under the Guidance of

PROFESSOR GEORGE ANTONIOU

COMPUTER SCIENCE & INFORMATION TECHNOLOGY

MONTCAIR STATE UNIVERSITY, MONTCLAIR
NEW JERSEY - 07043
DECEMBER 2025

ACKNOWLEDGEMENT

I am pleased to present “Image Colorization using AI” project and take this opportunity to express my profound gratitude to all those people who helped me in the completion of this project.

I express my deepest gratitude towards Prof. George Antoniou who gave valuable and timely advice during the various phases in my project. I would also like to thank him for providing me with all proper facilities and support. I thank him for support, patience, and faith in my capabilities and for giving me flexibility in working and reporting schedules.

ABSTRACT

Color enhances the emotional impact of images. However, black-and-white photos were taken for almost a century. Despite their historical significance, these pictures can seem detached and lacking because they don't have the same brightness as real-life encounters. ABLAZE is an AI-powered image colorizer that transforms monochrome photos into naturally colored versions with little effort on the part of the viewer in the order to close this emotional gap.

The program combines enhancing methods including exposure correction, sharpness modification, and noise reduction with state-of-the-art machine learning models like DeOldify. A user-friendly web interface created with Flask, HTML, CSS, and JavaScript envelops the system. All users have to do is upload a grayscale image, and ABLAZE takes care of everything, including color prediction and post-processing, producing amazing results in a matter of seconds.

ABLAZE is more than simply a technical initiative; it's a digital restoration platform that gives creators and historians a way to bring historical media back to life, preserve memories, and reunite families with their history. This research shows how contemporary AI can be applied responsibly and meaningfully to improve human experience.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENT

1. INTRODUCTION

Human memory, emotion, and perception are closely related to color. Due to technological constraints, images were taken in black and white for many decades. The lack of color frequently separates contemporary viewers from the feelings and realism those moments once carried, despite the fact that these monochrome photos have enormous historical significance. The development of artificial intelligence has made it feasible to close this gap by using accurate colorization to revive old memories.

ABLAZE is a web application powered by AI that is specifically built to perform this transformation. In just a few seconds, anyone may upload any black-and-white image and get a cleverly colored version. ABLAZE employs deep-learning models to conduct automatic, high-quality colorization with no human effort, eliminating the need for lengthy processing times or specific editing abilities. The solution combines a lightweight, interactive frontend with backend AI processing so users can adjust the final look without having to comprehend the complex algorithms.

1.1 PROJECT OVERVIEW:

ABLAZE is a browser-based artificial intelligence solution that uses deep learning to bring color back to grayscale photos. To guarantee quality and dependability, it incorporates a number of elements, including Flask as the backend, OpenCV for image processing, and AI models like DeOldify and Zhang's colorization network. The frontend allows users to customize the output visually by offering sliders for contrast, sharpness, and noise reduction once the server has colorized the image.

ABLAZE's primary goal is to make colorization easy, approachable, and emotionally significant. ABLAZE is purposefully made for common users, including students, families, historians, content providers, and hobby photographers, in contrast to professional editing programs that demand experience.

1.2 PROJECT OBJECTIVE:

The primary objectives of ABLAZE are as follows:

1.2.1 Provide High-Quality Automated Colorization

Use advanced deep-learning models to generate realistic, visually appealing color versions of black-and-white images with minimal user involvement.

1.2.2 Create a Simple and Accessible User Experience

Ensure that any user—regardless of technical background—can upload images and retrieve colorized results effortlessly.

1.2.3 Maintain Robustness with Model Fallbacks

Use DeOldify as the preferred model but automatically switch to Zhang's Caffe model whenever DeOldify is unavailable or fails.

1.2.4 Offer Lightweight Client-Side Enhancements

Provide real-time adjustments (contrast, sharpness, noise reduction) without server load or reprocessing, improving usability and customization.

1.2.5 Build a Scalable, Extensible Framework

Design the system so future features—like batch colorization, super-resolution, or mobile deployment—can be integrated easily.

1.3 PROBLEM STATEMENT:

Even while black-and-white photos have sentimental and historical value, the lack of color makes them seem incomplete to contemporary viewers. Conventional colorization techniques need a substantial time commitment, artistic interpretation, and manual editing. These methods have a number of drawbacks:

- Not everyone has artistic or technical skills required to colorize photos manually.
- Professional colorization tools are expensive or require installation and hardware resources.
- Web-based solutions often restrict downloads or offer low-quality results.
- Historical photos lose emotional resonance because viewers cannot fully visualize the original scene.

Therefore, there is a need for,

"A user-friendly, automated AI solution that can reliably and swiftly produce high-quality, emotionally expressive colorizations."

By fusing cutting-edge neural networks with an easy-to-use interface, ABLAZE meets this requirement by offering a smooth and fulfilling process for restoring old photos.

1.4 MOTIVATION:

Photo colorization is now more accurate and more appealing than ever thanks to recent developments in AI, particularly GANs (Generative Adversarial Networks). Neural networks can learn color distributions by examining millions of actual photographs, as shown by programs like DeOldify. ABLAZE was developed to provide users with this state-of-the-art technology via an easy-to-use online interface.

ABLAZE also serves as a learning platform to explore:

- AI model integration
- Real-world deployment of ML pipelines
- Image preprocessing & enhancement
- Frontend-backend interaction
- Real-time rendering using HTML Canvas

1.5 PROJECT SCOPE:

ABLAZE encompasses:

- A full-stack application (frontend + backend)
- AI-based colorization models (DeOldify, Zhang)
- Optional enhancement modules (noise reduction, sharpening)
- JavaScript-based real-time preview adjustments
- User-centered design optimized for simplicity

2. LITERATURE SERVEY

Understanding how image colorization evolved throughout time demonstrates why a system like ABLAZE is both meaningful and important. Colorization has progressed through several technological phases, from slow, artistic handicraft to quick, automated deep-learning colorizers. This section discusses these innovations and how they influenced ABLAZE's design.

2.1 Manual Colorization (Pre-AI Era):

Prior to the development of contemporary algorithms, colorization was only a creative process. Expert artists used tools like paintbrushes, Photoshop layers, or film restoration software to carefully apply colors frame by frame or pixel by pixel. They needed to learn:

- The fabric texture.
- Lighting direction.
- Historical clothing colors.
- Skin tones and environmental cues.

Despite the potential for spectacular outcomes, the approach had several serious drawbacks:

- Extremely time-consuming: Movies took months, while a single picture could take hours.
- Most individuals cannot access it since it requires expert-level ingenuity.
- Every new photo had to be created from scratch because there was no automation.
- Inconsistent outcomes because different artists interpreted color differently.

The search for automated, repeatable, intelligent solutions was spurred by these difficulties.

2.2 Early Machine Learning (ML) Approaches:

The initial attempts at automatic colorization relied on regular machine learning rather than deep learning. These systems mostly depended on manually created rules:

- Particular areas were marked by users (e.g., hair, sky, shirt).
- Colors selected by the user were spread over similar textures by the system.
- Probable color values were determined using basic mathematical methods.

Despite being novel at the time, these methods had the following drawbacks:

- Substantial manual input, negating automation's goal.
- Rigid outcomes because the models were unable to comprehend the context of the scene.
- They do poorly on unfamiliar images due to poor generalization.

Early machine learning techniques demonstrated that automation was feasible but far from ideal. Deep learning-based colorization was made possible by this research.

2.3 CNN-Based Colorization (Zhang et al., 2016):

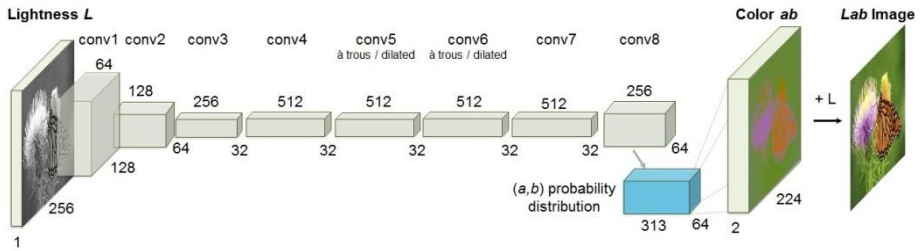

Fig-2.1: Image Colorization basic Implementation with CNN

The introduction of deep learning—more especially, Convolutional Neural Networks (CNNs)—for colorization marked a significant turning point.

Why CNNs?
CNNs mimic how the human visual cortex processes images, learning:

- Textures
- Object boundaries
- Semantic meaning
- Contextual cues (sky, skin, grass, etc.)

A CNN that learnt from millions of colored photos was put into practice by Zhang et al. The network automatically learnt color patterns rather of having to be told by hand which color belongs where.

Fundamental Concepts of the Model

- transforms the grayscale input into the LAB color space's L (lightness) channel.
- The network uses learnt color distributions to anticipate ab color channels.
- Makes the predictions more consistent and organic by treating colorization as a classification problem rather than a straightforward regression.

Strengths

- Fast inference
- Good general-purpose colorization
- Predictable and stable colors

Weaknesses

- Colors appear soft or muted
- May miss fine details
- Less expressive than GAN-based methods

Due to its dependability, this model serves as ABLAZE's backup engine in the event that more recent models, such as DeOldify, are unavailable.

2.4 GAN-Based Colorization – DeOldify (2019):

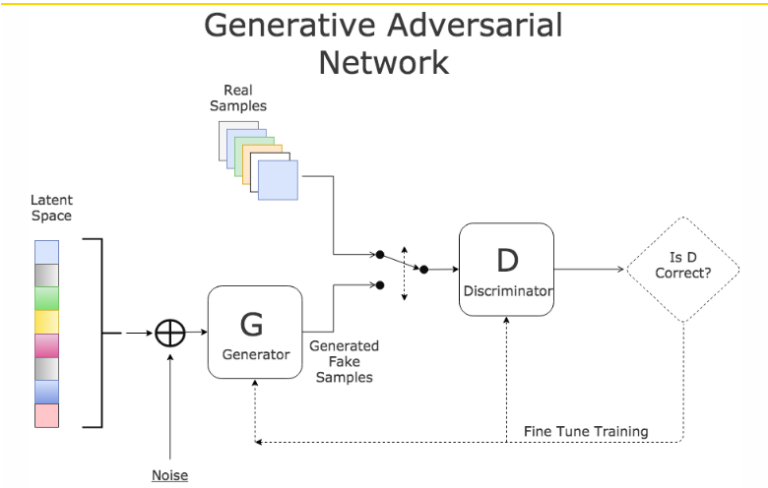

Fig-2.2: GANs with DeOldify

The next development was the introduction of Generative Adversarial Networks (GANs). Two neural networks make up GANs:

- Generator: Produces lifelike color pictures
- Discriminator: Assesses whether the created image appears authentic.

DeOldify developed a novel training method called NoGAN, which enhances realism and stabilizes training. It soon gained recognition as one of the top open-source colorizers.

Strengths of DeOldify

- Remarkably vivid and natural colors
- Exceptional handling of human faces and skin tones
- Works well for historical and old photographs
- Produces visually expressive, emotionally rich images

Limitations

- Can be slightly artistic or saturated
- Runs best on GPU hardware
- Heavy model size

ABLAZE uses DeOldify as the **primary colorization engine**, leveraging its expressive color output.

2.5 Enhancement Systems (GFPGAN & Real-ESRGAN):

Colorization alone does not always guarantee clarity or detail. Two important enhancement models complement colorization:

GFPGAN – Facial Restoration

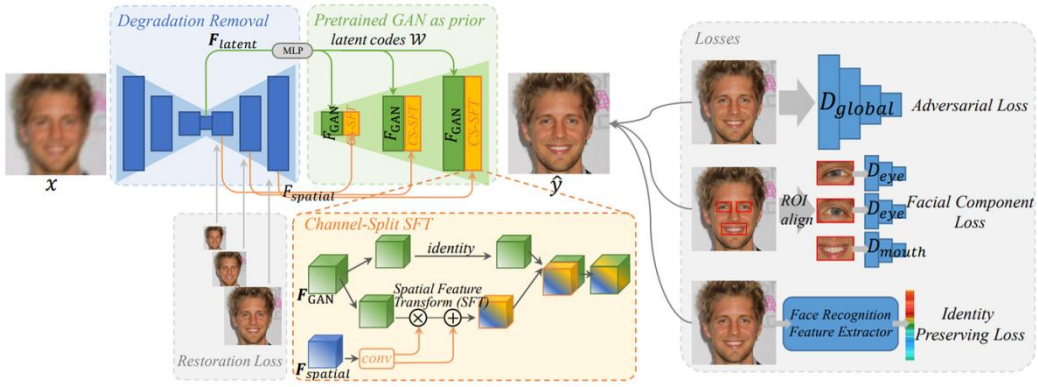

Fig-2.3: GFPGAN

- Recovers lost facial details
- Fixes blurriness in old portraits
- Ensures faces remain natural and sharp after colorization
- Helps prevent “plastic” or “washed-out” look

Real-ESRGAN – Super Resolution

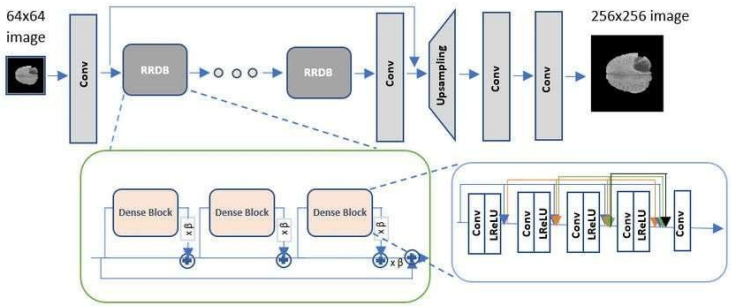

Fig-2.4: Real-ESRGAN

- Sharpens low-resolution images
- Improves edges and textures
- Makes old, small photos appear crisp when enlarged

In ABLAZE, these models are selectively applied to refine the final output and enhance user satisfaction.

2.6 Comparison of AI Colorization Models:

Model / Paper	Year	Type	Strengths	Limitations	Usage in ABLAZE
Zhang et al. (Caffe)	2016	CNN (classification-based)	Fast, lightweight, stable output; works even on CPU-only systems	Colors appear muted or conservative; may struggle with complex textures	Used as fallback model when DeOldify is not available
DeOldify (Artistic/Stable)	2019	GAN (NoGAN training)	Produces vibrant colors, great for portraits & natural scenes; state-of-the-art realism	Requires PyTorch; heavy GPU recommended; sometimes too artistic	Used as primary colorization engine
GFPGAN	2021	Facial restoration GAN	Restores facial clarity; fixes blurry old photos	Only improves faces, not full scene	Used for optional face enhancement
Real-ESRGAN	2021	Super-resolution model	Boosts sharpness & detail; enhances low-quality scans	Heavy model; skipped if environment incompatible	Optional super-resolution step in ABLAZE

Table-2.1: Comparison of AI Colorization Models

This table compares the primary AI models mentioned or used in the ABLAZE system. Clearly demonstrating the variations in model architecture, performance, and purpose is the aim. Zhang's model offers dependability on lighter systems, however DeOldify is emphasized as the best model because of its realism and evocative color tones. Although GFPGAN and Real-ESRGAN are not colorization models per se, they are added to improve image quality. The hybrid and fallback-based approach that ABLAZE employs is supported by this comparison.

2.7 Advances in Image Processing for Colorization:

More recent advancements extend beyond color prediction:

- Noise reduction models aid in maintaining the clarity of old scanned photos.
- Exposure correction algorithms address underexposed greyscale inputs.
- Skin tone detection aids in the reduction of artificial coloring.
- Structural sharpness is maintained via edge-preserving filters.

Several of these methods, such as histogram-based balancing, yellow cast removal, and minor exposure correction, are incorporated into ABLAZE.

2.8 Summary of the Literature:

In conclusion, colorization technology has advanced from laborious manual creativity to quick automation powered by AI. Semantic understanding was introduced by CNNs, realism and expressiveness were introduced by GANs, and the outputs were made clearer and more detailed by enhancement models.

ABLAZE is a user-friendly, engaging, and emotionally impactful program that builds upon these advancements.

3. METHODOLOGY

3.1 System Architecture Diagram:

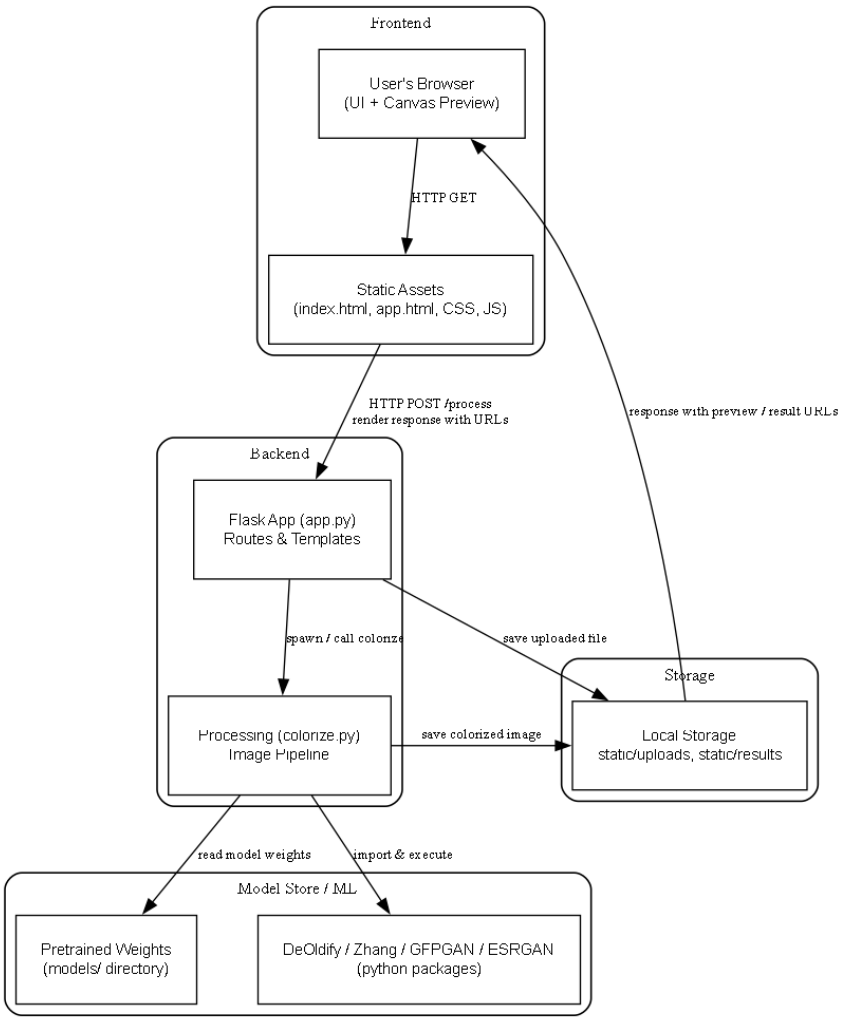

Fig-3.1: System Architecture Diagram

The system architecture diagram splits ABLAZE into four distinct layers: the frontend, the flask backend, the model store / ML components, and file storage. Static assets like index.html, app.html, CSS, and JavaScript are loaded by the user's browser on the left side. The client-side interactive slider-based preview and the visual layout are controlled by the browser.

The middle of the diagram depicts the Flask app (app.py), which offers routes such as /, /colorizer, /about, and /process. Flask accepts the HTTP POST request when a user uploads an image through the web form, stores the submitted file, and transfers processing to the image pipeline script (colorize.py). This pipeline reads pre-trained weights stored in the models/directory (DeOldify weights, Zhang Caffe models, GFPGAN, Real-ESRGAN, etc.) and internally imports and runs the deep-learning models from the Model Store.

On the right, the Storage block depicts the local directories static/uploads and static/results, where both original and colorized photographs are saved. The frontend renders the side-by-

side preview and the "Save Result" download link using the URLs that the Flask application returns. This architecture diagram demonstrates how ABLAZE remains a small, deployable project while neatly separating presentation, application logic, and significant AI processing.

3.2 Level-1 Data Flow Diagram:

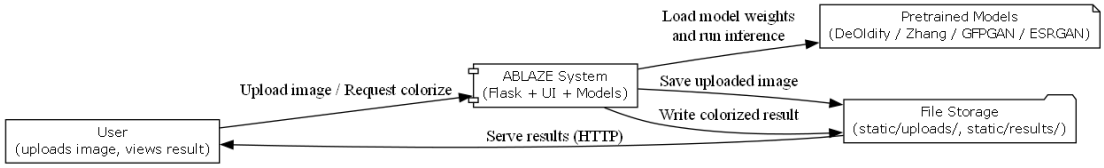

Fig-3.2: Level-1 DFD

Figure 3.2 depicts ABLAZE at the highest, "black-box" level. Here, the entire application is handled as a single process known as "ABLAZE System," and we simply pay attention to how the user, the system, and the file storage interact. The user uploads an image in black and white, which is then processed and returned in color. Only the main flows into and out of the system are displayed at this time; the interior stages are not.

User data enters ABLAZE as a "Upload image" request. The uploads/results folders—represented here as the Storage data store—are where ABLAZE internally keeps both the original and processed versions. The colorized image that can be downloaded or pre-viewed in the browser is the "view result" output that the system then returns. Without being sidetracked by implementation specifics, this context perspective makes it easier for non-technical readers to comprehend what ABLAZE achieves.

At this stage, we clarify that ABLAZE is basically a service that takes an image, does clever colorization work, and securely stores and provides results. The user only requires a browser to interact with it.

3.3 Level-2 Data Flow Diagram:

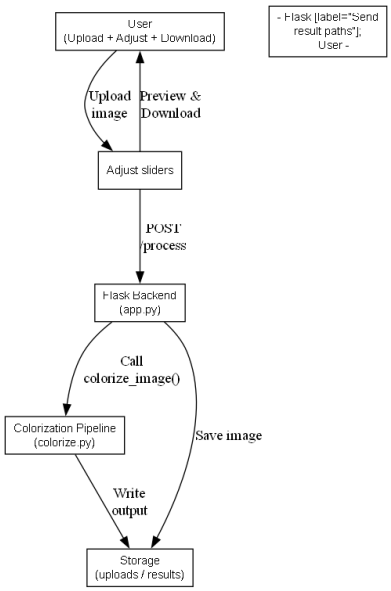

Fig-3.3: Level-2 DFD

The Level-2 DFD divides the single "ABLAZE System" box into a few essential procedures. "Upload Handling," "Image Colorization Pipeline," "Result Management," and "Preview & Download" are typical examples of these. By doing this, we demonstrate how the user's raw input is progressively converted into a refined, useable output.

An image sent by the user first goes via Upload Handling, which verifies the file type and saves the original under a different name. After the file location has been verified, it enters the Image Colorization Pipeline, where AI models like Zhang CNN and DeOldify produce a color version. Both intermediate and final photos are written into the File Storage data store by the pipeline.

Lastly, the user's browser receives URLs for the original and colorized images from the Result Management/Preview process, which reads from storage. This makes it possible for the front end to display side-by-side previews and provide a download option labeled "Save Result." While remaining at a conceptual level, the Level-1 view provides a clear image of how duties are distributed within ABLAZE.

3.4 Image Processing Pipeline Diagram:

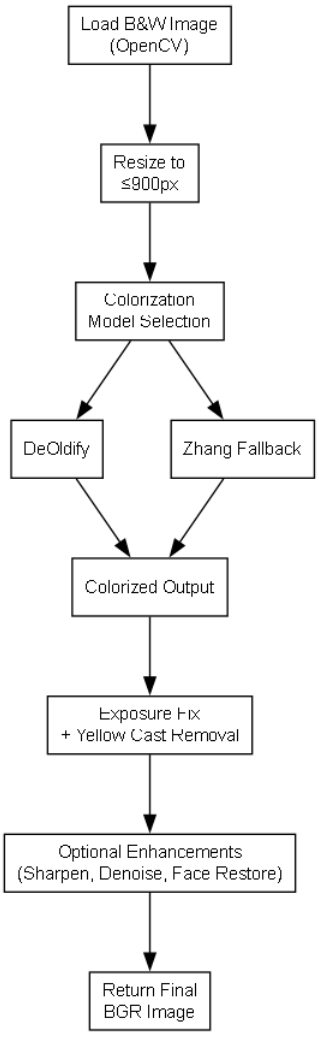

Fig-3.4: Image Processing Pipeline Diagram

The image pipeline diagram depicts the image's travel via ABLAZE. It begins with the raw grayscale upload and iteratively transforms the same image. Following preliminary validation and scaling, the image is sent to the Colorization Engine, where a CNN or GAN model reconstructs a full-color RGB image by predicting likely color channels.

After that, the pipeline splits into stages for improvement. A face restoration module maintains the original identity and expression while sharpening and clarifying facial features. A post-processing block eliminates excessive yellow or magenta tints that various models often introduce, as well as corrects exposure. A super-resolution module, which is especially helpful for old low-resolution photos, can optionally upgrade the result to generate a clearer and more detailed output than the original source.

This flowchart explains to readers that the output they see is the result of numerous complementing steps meant to correct typical artifacts, rather than raw AI output. It emphasizes that ABLAZE is not "just" a colorization model wrapper, but rather a complete restoration pipeline.

3.5 UI Navigation & Interaction Flow:

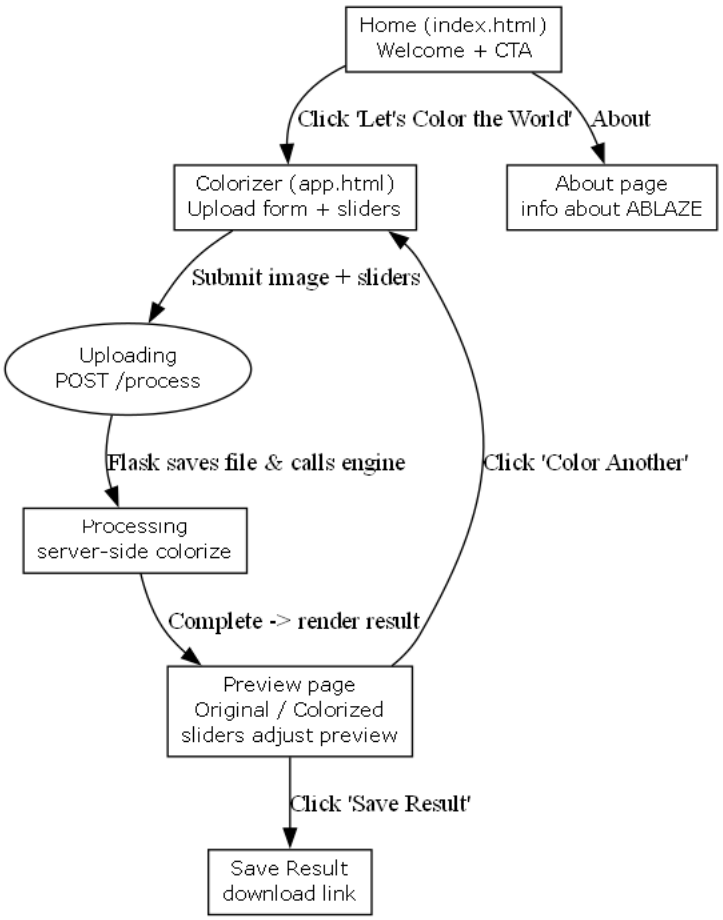

Fig-3.5: UI Navigation & Interaction Flow

The UI flow diagram explains how a user navigates the ABLAZE interface. The process usually starts on the Home page (index.html), where the user is presented with a brief greeting and one large call-to-action button that reads, "Let's Color the World." The Colorizer page (app.html), where the upload and interaction really occur, is reached by clicking this button.

The Colorizer page allows the user to select an image, modify appearance parameters (contrast, sharpness, noise reduction), and submit the form. While the backend prepares the outcome, the user interface enters a "processing" state. An "Original" panel and a "Colorized" panel are shown inside white cards on the same page after processing is finished. Then, using client-side canvas filters, the user can adjust sliders that instantly change the preview; no additional server requests are needed. The user can then select "Color Another" to start anew or use the "Save Result" button to download the outcome. To learn more about the system, they may always go to the About page via the navbar.

This figure highlights that ABLAZE provides a very modest, easy-to-learn navigation surface. Home → Colorizer → (Optional) → Download. Instead of focusing on navigating a complex interface, the straightforward flow draws attention to the photos themselves.

3.6 Activity Diagram:

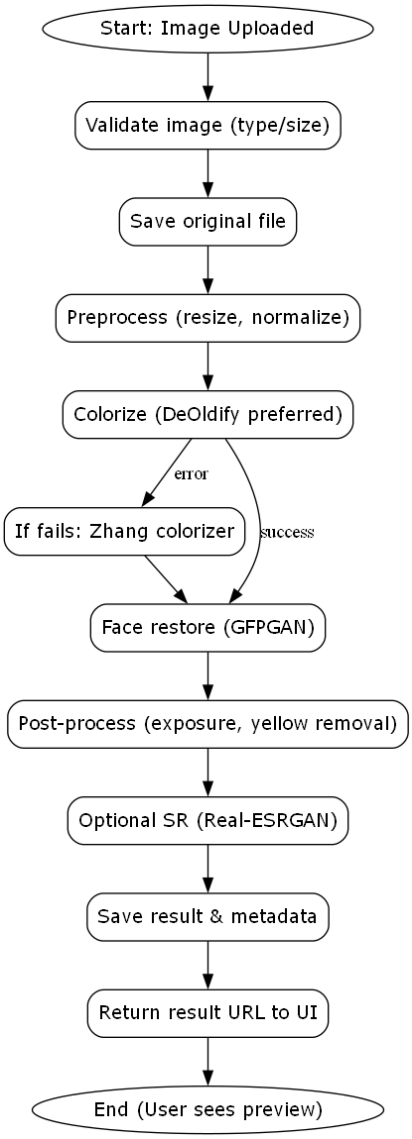

Fig-3.6: Activity Diagram

The activity diagram leads you through each phase of a single user session. It begins when the user visits the home page of the website. "View Welcome Screen" is the first task, and then "Click Start / Go to Colorizer." The user's two main tasks on the Colorizer page are choosing a picture file and, if desired, modifying the sliders that determine how the preview will appear later.

When the user clicks "Upload & Colorize", control is transferred to the system side: the backend validates the file, runs it through the colorization pipeline, and waits for the AI models to finish. Following processing, the activity changes to "View Original and Colorized Preview" and the control returns to the user interface. Now, the user can enter a brief loop to modify contrast, sharpness, and noise reduction; each adjustment updates the preview canvas, allowing the user to stop when they are pleased.

Lastly, the user has the option to either "Color Another," which takes them back to the upload stage, or "Save Result," which saves the colorized photo. The activity diagram illustrates how ABLAZE facilitates an organic, iterative workflow: attempt a picture, make necessary adjustments, save it, and then repeat.

3.7 Class Diagram:

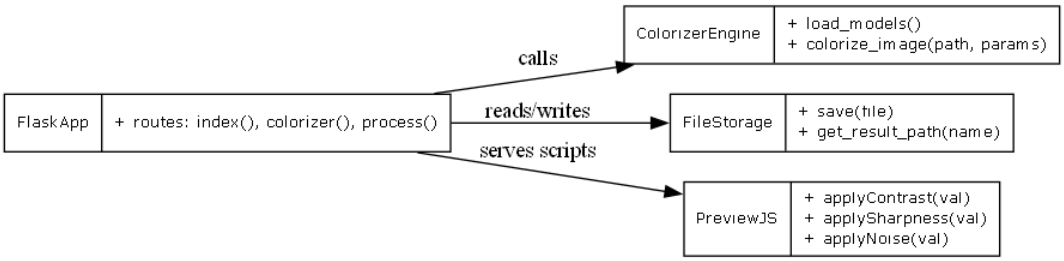

Fig-3.7: Class Diagram

The ABLAZE class diagram shows how several system components work together to convert a user's black-and-white image into a completely colorized and enriched output. At the highest level, the AppController serves as the primary communication link between the user interface and the backend processing pipeline. When a user uploads an image, the controller coordinates the request, interacts with supporting management classes, and ensures that the entire workflow from upload to result delivery runs smoothly and transparently.

The basic picture alteration procedures are handled by two key subsystems: the Colorization-Service and the Enhancement-Pipeline. The Colorization-Service is in charge of processing the image, choosing the suitable model (DeOldify or Zhang), and creating the initial colorized version. These models are expressed as distinct classes—DeOldify Model and Zhang Model—which mask the complexities of machine-learning frameworks behind a simple colorize() interface. After colorization, the image is transmitted to the Enhancement-Pipeline, where classes such as Face-Restorer, Exposure-Corrector, Sharpness-Adjuster, and Noise-Reducer modify the visual quality based on user preferences. This modular design allows ABLAZE to adapt simply by adding or replacing enhancement phases without rewriting the entire system.

Finally, the processed image is stored and made available using the Result-Manager, which saves outputs to the project's results directory and generates clean URLs for browser display. Supporting classes like as Upload-Manager and Image-Preprocessor guarantee that uploads are secure, valid, and consistently formatted before they enter the AI pipeline. Together, these classes form a well-organized, stable architecture with well defined roles, making ABLAZE not only strong but also scalable and extensible in the future.

3.8 Sequence Diagram:

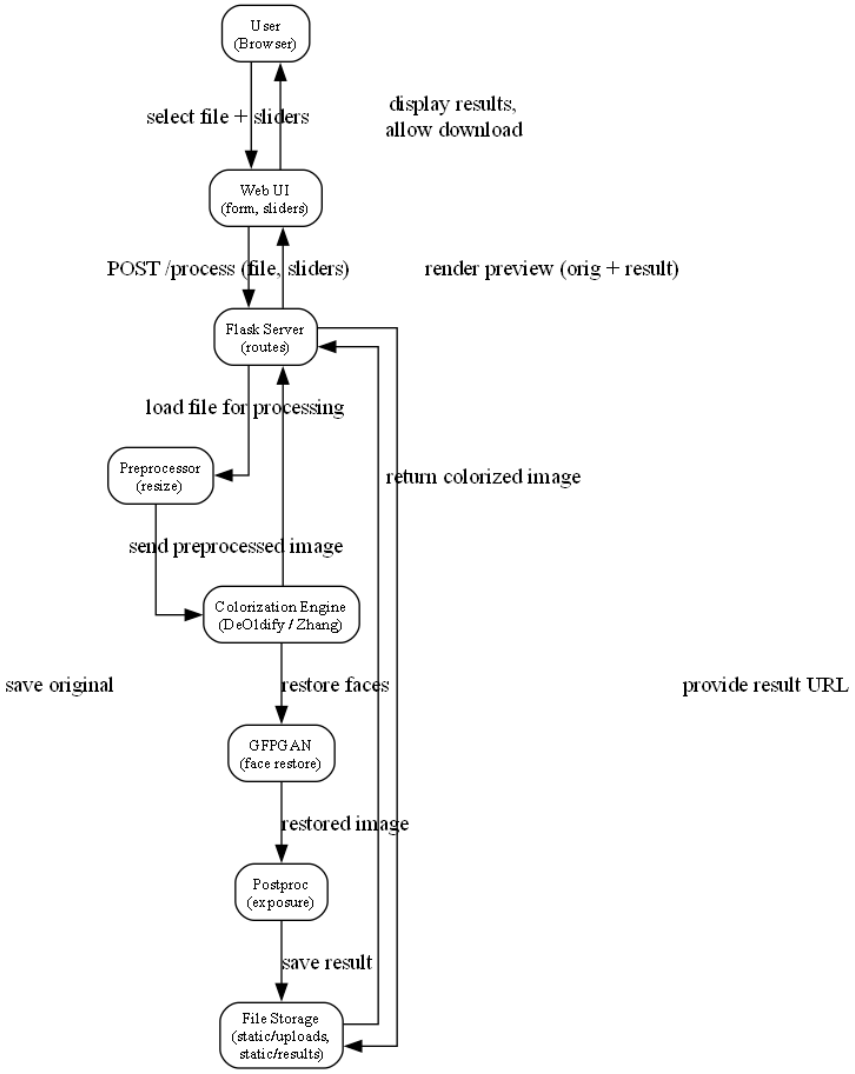

Fig-3.8: Sequence Diagram

The sequence diagram emphasizes the many ABLAZE components' time-ordered communication. The user engaging with the web browser is on the left. The Colorization Engine (colorize.py + models) and the Storage layer are on the right, while the Flask Server is in the center. Requests and responses are indicated by horizontal arrows, while vertical lifelines display which component is active at any given time.

The sequence starts when the user goes to the Colorizer page; the browser sends a GET request, Flask responds with HTML, CSS, and JavaScript, and the user sees an upload form. When the user submits the image, the browser sends a POST /process request with the file.

Flask saves the file, then invokes colorize.py, which loads model weights (DeOldify/Zhang) from disk, performs colorization and improvements, and saves the final result to the static/results directory.

When the processing is finished, colorize.py returns control to Flask, which sends a rendered template to the browser with URLs to both the original and colorized images. The browser then gets the images and displays them in the preview section. Any additional slider adjustments take place locally in the browser, so no new server messages are exchanged. This diagram illustrates who "talks" to whom and when during a complete request-response cycle.

3.9 Use Case Diagram:

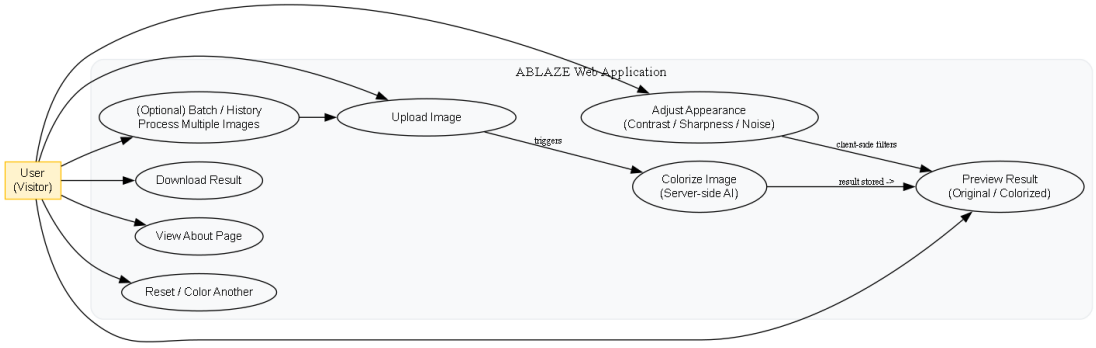

Fig-3.9: Use Case Diagram

The use case diagram depicts ABLAZE in terms of a user's goals, rather than the system's internals. The main character is the User, who is portrayed by a stick figure. The main applications are located around them: "Upload black-and-white photo", "Generate colorized version", "Preview original vs. colorized", "Adjust visual appearance (contrast/sharpness/noise)", "Download final image", and "View information about the system".

Relationships like "include" and "extend" indicate dependency. For example, "Generate colorized version" includes "Upload photo" because colorization requires an input image. "Adjust visual appearance" broadens the fundamental preview use case, implying that this is an optional upgrade step. "Download final image" is a distinct use case that is dependent on the successful completion of colorization and preview.

This model is extremely useful for writing requirements because each use case can be instantly translated into test cases and UI elements. It demonstrates that ABLAZE is centered on a single actor with a specific set of goals: bringing old images back to life, viewing the transformation clearly, and saving the results with minimal friction.

3.10 ABLAZE System Pipeline Summary:

Stage	Description	Tools / Libraries Used	Output
Upload Stage	User uploads B&W image via web UI	Flask, HTML5 Forms	Raw input image saved in `/static/uploads/`

Preprocessing	Resize image, load via OpenCV	OpenCV	Optimized working-size image
Colorization (Primary)	DeOldify GAN model attempts vibrant colorization	PyTorch, DeOldify	High-quality colorized image
Colorization (Fallback)	Zhang model used if DeOldify unavailable	OpenCV DNN	Conservative but stable colorization
Post-Processing	Correct yellow tones, adjust exposure	LAB color ops, CLAHE	Balanced colorized output
Face Restoration	Optional facial enhancement	GFPGAN	Sharper, more natural faces
Upscaling (Optional)	Enhance resolution	Real-ESRGAN	Higher-resolution color image
Client-Side Adjustments	User adjusts contrast, sharpness, noise	JavaScript Canvas	Interactive preview
Download	User saves final processed image	HTML5 Download API	Final image output

Table-3.1: Summary of ABLAZE Processing Pipeline

This table provides a step-by-step summary of the whole ABLAZE workflow. Because each step is connected to the models and libraries utilized, readers will find the flow much simpler to comprehend. Additionally, it demonstrates how ABLAZE combines browser-side improvements with server-side AI processing to produce a seamless, responsive user experience.

4. IMPLEMENTATION

ABLAZE's implementation combines web technologies, deep learning models, and image processing tools to produce a smooth and interactive colorization experience. The system is constructed with HTML/CSS/Bootstrap for the user interface, JavaScript (Canvas API) for real-time adjustments like contrast, sharpness, and noise reduction, and Flask as the backend framework. Pre-processing, model selection, enhancing phases, and final image production are all part of colorize.py's fundamental colorization logic. Each layer of the system—upload management, preprocessing, model inference, augmentation, and output delivery—can operate independently thanks to this modular design, which also facilitates clean integration.

4.1 Project Directory Structure:

ABLAZE/

```
│
├── app.py              → Flask backend (routing, uploading, processing)
├── colorize.py         → AI colorization + enhancement pipeline
│
├── templates/          → All HTML UI pages
│   ├── index.html
│   ├── app.html
│   └── about.html
│
├── static/
│   ├── css/style.css   → Front-end styling
│   ├── uploads/        → Saves original images
│   └── results/        → Saves colorized images
│
└── models/             → Pretrained model weights (DeOldify, GFPGAN, ESRGAN)
```

4.2 Backend Implementation (Flask Server):

Code: app.py(Flask Backend):

```
from flask import Flask, render_template, request, redirect, url_for
import os
import cv2
import time
import numpy as np
from colorize import colorize_image

app = Flask(__name__)

UPLOAD = "static/uploads/"
RESULTS = "static/results/"
```

```
os.makedirs(UPLOAD, exist_ok=True)
os.makedirs(RESULTS, exist_ok=True)

def apply_adjustments(bgr_img, contrast=1.0, sharpness=1.0, denoise=0.0):
    """
    Apply denoise -> contrast -> sharpness to a BGR uint8 image and return BGR uint8 result.

    - contrast: multiplicative (1.0 = unchanged)
    - sharpness: 1.0 = unchanged; >1 sharpen; <1 soften
       -   denoise:   0.0..1.0   where   0   =   off,   1   =   strong.   Server   uses   OpenCV
fastNlMeansDenoisingColored
    """
    img = bgr_img.copy()

    # 1) DENOISE (server-side): map denoise (0..1) to h param (0..30)
    try:
        d = float(denoise)
    except Exception:
        d = 0.0
    d = max(0.0, min(1.0, d))
    if d > 1e-4:
        # h controls filter strength; tune as needed
        h = d * 30.0        # color component strength
        hColor = d * 30.0   # same for color
        try:
            # cv2.fastNlMeansDenoisingColored expects BGR uint8
            img = cv2.fastNlMeansDenoisingColored(img, None, h, hColor, 7, 21)
        except Exception:
            # if the function fails (older OpenCV), fallback to Gaussian blur mild
            k = max(1, int(d * 3) * 2 + 1)
            img = cv2.GaussianBlur(img, (k, k), 0)

    # 2) Contrast
    try:
        alpha = float(contrast)
    except Exception:
        alpha = 1.0
    img = cv2.convertScaleAbs(img, alpha=alpha, beta=0)

    # 3) Sharpness via unsharp-like approach
    try:
        s = float(sharpness)
    except Exception:
        s = 1.0
    amount = s - 1.0
    if abs(amount) < 1e-4:
        return img

    # sigma adapted by image size
    himg, wimg = img.shape[:2]
    sigma = max(0.8, min(3.0, max(himg, wimg) / 1000.0 * 1.2))
    blurred = cv2.GaussianBlur(img, (0, 0), sigmaX=sigma, sigmaY=sigma)

    if amount > 0:
        sharpened = cv2.addWeighted(img, 1.0 + amount, blurred, -amount, 0)
        sharpened = np.clip(sharpened, 0, 255).astype('uint8')
        return sharpened
    else:
        factor = max(0.0, 1.0 + amount)
        softened = cv2.addWeighted(img, factor, blurred, 1.0 - factor, 0)
        softened = np.clip(softened, 0, 255).astype('uint8')
        return softened

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/colorizer")
def colorizer():
            return     render_template("app.html",     input_image=None,     output_image=None,
ts=int(time.time()))

@app.route("/about")
```

```
def about():
    return render_template("about.html")

@app.route("/process", methods=["POST"])
def process():
    if "image" not in request.files:
        return redirect(url_for('colorizer'))

    file = request.files["image"]
    if file.filename == "":
        return redirect(url_for('colorizer'))

    filename = file.filename
    file_path = os.path.join(UPLOAD, filename)
    file.save(file_path)

    # read form values (safe fallbacks)
    try:
        contrast = float(request.form.get("contrast", 1.0))
    except Exception:
        contrast = 1.0
    try:
        sharpness = float(request.form.get("sharpness", 1.0))
    except Exception:
        sharpness = 1.0
    try:
        denoise = float(request.form.get("denoise", 0.0))
    except Exception:
        denoise = 0.0

    # colorize_image called with defaults (mode auto, do_enhance default True)
    try:
        result = colorize_image(file_path)
    except Exception as e:
        return f"Processing failed: {e}"

    # apply server-side denoise/contrast/sharpness
    try:
            result_adj  =  apply_adjustments(result,  contrast=contrast,  sharpness=sharpness,
denoise=denoise)
    except Exception as e:
        print("Adjustment failed:", e)
        result_adj = result

    output_filename = "colored_" + filename
    output_path = os.path.join(RESULTS, output_filename)

    # save BGR image
    try:
        cv2.imwrite(output_path, result_adj)
    except Exception as e:
        return f"Failed to save result: {e}"

    return render_template(
        "app.html",
        input_image=filename,
        output_image=output_filename,
        ts=int(time.time())
    )

if __name__ == "__main__":
    app.run(debug=True)
```

The backend is created with Flask, a lightweight Python framework that makes it simple to construct online APIs and produce dynamic HTML templates. Three important routes are exposed by the server:

- / → Loads the homepage (index.html)
- /colorizer → Loads the upload interface (app.html)
- /process → Handles file uploads and triggers the colorization pipeline

When a user uploads an image, Flask temporarily saves it inside `static/uploads/`, then calls the `colorize_image()` function from `colorize.py`. After processing, the output image is stored in `static/results/`, and the preview is sent back to the browser.

The backend also appends a timestamp (`?v=12345`) to image URLs to ensure browsers always fetch the newest version and don't display a cached image.

4.3 Colorization & Enhancement Pipeline:

Code: colorize.py(AI Colorization Pipeline):

```
import os
from pathlib import Path
import cv2
import numpy as np

try:
    import torch
except Exception:
    torch = None
    print("torch not available; enhancement models requiring torch will be skipped.")

THIS_DIR = Path(__file__).resolve().parent
PROJECT_ROOT = THIS_DIR.parent
MODELS_DIR = PROJECT_ROOT / "models"

PROT = MODELS_DIR / "colorization_deploy_v2.prototxt"
MODEL = MODELS_DIR / "colorization_release_v2.caffemodel"
PTS = MODELS_DIR / "pts_in_hull.npy"

print("Using colorization models from:", MODELS_DIR)
print("Resolved model files:", PROT, MODEL, PTS)

net = None
if PROT.exists() and MODEL.exists() and PTS.exists():
    try:
        print("Loading Zhang colorizer (fallback)...")
        net = cv2.dnn.readNetFromCaffe(str(PROT), str(MODEL))
        pts_np = np.load(str(PTS))
        pts_np = pts_np.transpose().reshape(2, 313, 1, 1)
        net.getLayer(net.getLayerId("class8_ab")).blobs = [pts_np.astype(np.float32)]
        net.getLayer(net.getLayerId("conv8_313_rh")).blobs = [
            np.full([1, 313], 2.606, np.float32)
        ]
    except Exception as e:
        print("Failed to load Zhang model:", e)
        net = None
else:
    print("Zhang model files missing; place prototxt, caffemodel, pts_in_hull.npy in:",
MODELS_DIR)

DEOLDIFY_AVAILABLE = False
GFPGAN_AVAILABLE = False
REALESRGAN_AVAILABLE = False

try:
    from deoldify import device
    from deoldify.device_id import DeviceId
    from deoldify.visualize import get_image_colorizer
    DEOLDIFY_AVAILABLE = True
    print("DeOldify import OK")
except Exception:
    DEOLDIFY_AVAILABLE = False

try:
    from gfpgan import GFPGANer
    GFPGAN_AVAILABLE = True
    print("GFPGAN import OK")
except Exception:
    GFPGAN_AVAILABLE = False
```

```
try:
    from realesrgan import RealESRGANer
    REALESRGAN_AVAILABLE = True
    print("RealESRGAN import OK")
except Exception:
    REALESRGAN_AVAILABLE = False

DEOLDIFY_ART_WEIGHT = MODELS_DIR / "ColorizeArtistic_gen.pth"
DEOLDIFY_STABLE_WEIGHT = MODELS_DIR / "ColorizeStable_gen.pth"
GFPGAN_WEIGHT = MODELS_DIR / "GFPGANv1.3.pth"
REALESRGAN_WEIGHT = MODELS_DIR / "RealESRGAN_x4plus.pth"

# ---------- helpers ----------
def remove_yellow_cast(img, strength=6):
    """
    Gentle yellow cast removal. Lower default strength to avoid blue shift.
    """
    lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
    L, A, B = cv2.split(lab)
    # subtract small value from B (reduces yellow bias gently)
    B = cv2.subtract(B, int(strength))
    lab = cv2.merge([L, A, B])
    return cv2.cvtColor(lab, cv2.COLOR_LAB2BGR)

def blend_with_mask(orig, restored, alpha=0.55):
    if orig.shape != restored.shape:
        restored = cv2.resize(restored, (orig.shape[1], orig.shape[0]))
    return cv2.addWeighted(restored, alpha, orig, 1 - alpha, 0)

def detect_faces(img):
    try:
        cascade = cv2.data.haarcascades + "haarcascade_frontalface_default.xml"
        fc = cv2.CascadeClassifier(cascade)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = fc.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=4, minSize=(30,30))
        return faces
    except Exception:
        return ()

# ---------- deoldify setup ----------
_deoldify_colorizer = None

def ensure_deoldify(artistic=True, device_pref="cuda"):
    global _deoldify_colorizer
    if not DEOLDIFY_AVAILABLE:
        raise RuntimeError("DeOldify unavailable")

    try:
        if device_pref == "cuda" and (torch is not None and torch.cuda.is_available()):
            device.set(device=DeviceId.GPU0)
        else:
            device.set(device=DeviceId.CPU)
    except Exception:
        pass

    desired_w = DEOLDIFY_ART_WEIGHT if artistic else DEOLDIFY_STABLE_WEIGHT
    desired_w = Path(desired_w)
    if not desired_w.exists():
        raise RuntimeError(f"DeOldify weight not found: {desired_w}")

    models_dir = desired_w.parent
    candidate1 = models_dir
    candidate2 = models_dir.parent

    def would_find_weight(root_folder):
        path_try = Path(root_folder) / "models" / desired_w.name
        return path_try.exists()

    if would_find_weight(candidate1):
        root_folder = candidate1
    elif would_find_weight(candidate2):
        root_folder = candidate2
    else:
```

```
        root_folder = candidate2

    if _deoldify_colorizer is None:
        _deoldify_colorizer = get_image_colorizer(root_folder=root_folder, artistic=artistic)

    return _deoldify_colorizer

# ---------- small exposure helper ----------
def mild_exposure_boost(img, clip_limit=1.4, face_mask=None):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    clahe = cv2.createCLAHE(clipLimit=clip_limit, tileGridSize=(8, 8))
    v_boost = clahe.apply(v)

    if face_mask is not None:
        mask = face_mask.astype(np.float32) / 255.0
        mask = cv2.GaussianBlur(mask, (31,31), 0)
        inside = (mask * v_boost + (1.0 - mask) * v).astype(np.uint8)
        v_final = inside
    else:
        v_final = v_boost

    hsv = cv2.merge([h, s, v_final])
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)


def apply_face_restore(img, face_alpha=0.35, preserve_face_brightness=True):
    out = img.copy()
    gfpgan_applied = False
    if GFPGAN_AVAILABLE and GFPGAN_WEIGHT.exists():
        try:
            restorer   =   GFPGANer(model_path=str(GFPGAN_WEIGHT),   upscale=1,   arch='clean',
channel_multiplier=2, bg_upsampler=None)
            _, _, restored = restorer.enhance(out, has_aligned=False)
            gfpgan_applied = True
        except Exception as e:
            print("GFPGAN failed:", e)
            gfpgan_applied = False

    if not gfpgan_applied:
        return out

    faces = detect_faces(out)
    if len(faces) == 0:
        return blend_with_mask(out, restored, alpha=face_alpha)

    mask = np.zeros((out.shape[0], out.shape[1]), dtype=np.uint8)
    for (x,y,w,h) in faces:
        pad_w = int(w * 0.22)
        pad_h = int(h * 0.28)
        x0 = max(0, x - pad_w)
        y0 = max(0, y - pad_h)
        x1 = min(out.shape[1], x + w + pad_w)
        y1 = min(out.shape[0], y + h + pad_h)
        cv2.ellipse(mask, ((x0+x1)//2, (y0+y1)//2), ((x1-x0)//2, (y1-y0)//2), 0, 0, 360, 255,
-1)

    mask = cv2.GaussianBlur(mask, (31,31), 0)

    face_alpha_face = float(face_alpha)
    face_alpha_global = max(face_alpha, 0.5)

    alpha_map    =    (mask.astype(np.float32)/255.0)    *    face_alpha_face    +    (1.0    -
mask.astype(np.float32)/255.0) * face_alpha_global
    alpha_map = np.dstack([alpha_map]*3)

    out_f = out.astype(np.float32)
    restored_f = restored.astype(np.float32)

    blended = (restored_f * alpha_map + out_f * (1.0 - alpha_map)).astype(np.uint8)

    if preserve_face_brightness:
        blended = mild_exposure_boost(blended, clip_limit=1.3, face_mask=mask)
```

```
    return blended

# ---------- Real-ESRGAN (unchanged from your file) ----------
def apply_sr(img):
    if not REALESRGAN_AVAILABLE or not REALESRGAN_WEIGHT.exists() or torch is None:
        return img
    try:
        device_name = "cuda" if torch.cuda.is_available() else "cpu"
        try:
            model = RealESRGANer(device=device_name, scale=4)
        except Exception:
            print("RealESRGAN: constructor mismatch; skipping SR.")
            return img

        try:
            if hasattr(model, "load_weights"):
                model.load_weights(str(REALESRGAN_WEIGHT))
            elif hasattr(model, "load") and callable(getattr(model, "load")):
                model.load(str(REALESRGAN_WEIGHT))
        except Exception as e:
            print("RealESRGAN weight load failed; skipping SR.", e)
            return img

        rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        sr = model.enhance(rgb)
        out = cv2.cvtColor(sr, cv2.COLOR_RGB2BGR)
        print("RealESRGAN applied")
        return out
    except Exception as e:
        print("RealESRGAN failed:", e)
        return img

# ---------- Zhang colorizer (unchanged) ----------
def zhang_colorize(img):
    if net is None:
        raise RuntimeError("Zhang model not loaded")
    scaled = img.astype("float32") / 255.0
    lab = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
    L = lab[:, :, 0]
    L_rs = cv2.resize(L, (224, 224))
    L_rs -= 50
    net.setInput(cv2.dnn.blobFromImage(L_rs))
    ab = net.forward()[0].transpose((1, 2, 0))
    ab = cv2.resize(ab, (img.shape[1], img.shape[0]))
    lab_out = np.concatenate((L[:, :, np.newaxis], ab), axis=2)
    colorized = cv2.cvtColor(lab_out, cv2.COLOR_Lab2BGR)
    colorized = np.clip(colorized * 255, 0, 255).astype("uint8")
    return colorized

# ---------- main wrapper (small changes: use gentler defaults) ----------
def colorize_image(img_path, mode='auto', render_factor=35, do_enhance=True):
    img_path_p = Path(img_path)
    img = cv2.imread(str(img_path_p))
    if img is None:
        raise ValueError("Image cannot be loaded: " + str(img_path_p))

    orig_h, orig_w = img.shape[:2]

    MAX_WORK = 900
    if max(orig_h, orig_w) > MAX_WORK:
        scale = MAX_WORK / float(max(orig_h, orig_w))
        ws_w = int(orig_w * scale)
        ws_h = int(orig_h * scale)
        img_work = cv2.resize(img, (ws_w, ws_h), interpolation=cv2.INTER_AREA)
    else:
        img_work = img.copy()

    colorized_work = None
    use_deoldify = (mode == 'deoldify') or (mode == 'auto' and DEOLDIFY_AVAILABLE)
    if use_deoldify:
        try:
            print("Using DeOldify for colorization...")
            artistic = True
            colorizer = ensure_deoldify(artistic=artistic)
```

```
            deoldify_out = colorizer.get_transformed_image(path=str(img_path_p),
                                                           render_factor=render_factor,
                                                           watermarked=False,
                                                           post_process=True)
            print("DeOldify returned type:", type(deoldify_out))

            if isinstance(deoldify_out, (str, Path)):
                out_path = str(deoldify_out)
                colorized_full = cv2.imread(out_path)
                if colorized_full is None:
                    raise  RuntimeError(f"DeOldify  returned  path  but  cv2.imread  failed:
{out_path}")
            elif 'PIL' in type(deoldify_out).__module__:
                pil_img = deoldify_out
                arr = np.array(pil_img)
                if arr.ndim == 3 and arr.shape[2] == 4:
                    arr = arr[:, :, :3]
                colorized_full = cv2.cvtColor(arr, cv2.COLOR_RGB2BGR)
            elif isinstance(deoldify_out, np.ndarray):
                arr = deoldify_out
                if arr.dtype == np.float32 or arr.dtype == np.float64:
                    arr = np.clip(arr * 255.0, 0, 255).astype('uint8')
                if arr.ndim == 3 and arr.shape[2] == 3:
                    colorized_full = cv2.cvtColor(arr, cv2.COLOR_RGB2BGR)
                else:
                    colorized_full = arr
            else:
                raise RuntimeError(f"Unrecognized DeOldify return type: {type(deoldify_out)}")

            colorized_work       =       cv2.resize(colorized_full,       (img_work.shape[1],
img_work.shape[0]), interpolation=cv2.INTER_AREA)

        except Exception as e:
            print("DeOldify failed:", e)
            colorized_work = None

    if colorized_work is None:
        if net is None:
            raise  RuntimeError("No  colorization  model  available  (DeOldify  failed  and  Zhang
unavailable).")
        print("Using Zhang fallback for colorization...")
        colorized_work = zhang_colorize(img_work)

    faces = detect_faces(colorized_work)
    face_mask = None
    if len(faces) > 0:
        mask = np.zeros((colorized_work.shape[0], colorized_work.shape[1]), dtype=np.uint8)
        for (x,y,w,h) in faces:
            pad_w = int(w * 0.22)
            pad_h = int(h * 0.28)
            x0 = max(0, x - pad_w)
            y0 = max(0, y - pad_h)
            x1 = min(colorized_work.shape[1], x + w + pad_w)
            y1 = min(colorized_work.shape[0], y + h + pad_h)
            cv2.ellipse(mask, ((x0+x1)//2, (y0+y1)//2), ((x1-x0)//2, (y1-y0)//2), 0, 0, 360,
255, -1)
        mask = cv2.GaussianBlur(mask, (21,21), 0)
        face_mask = mask

    # gentler yellow-cast correction
    colorized_work = remove_yellow_cast(colorized_work, strength=6)
    colorized_work = mild_exposure_boost(colorized_work, clip_limit=1.4, face_mask=face_mask)

    if do_enhance:
        try:
            restored       =       apply_face_restore(colorized_work,       face_alpha=0.35,
preserve_face_brightness=True)
            colorized_work = blend_with_mask(colorized_work, restored, alpha=0.6)
        except Exception as e:
            print("Face restoration failed:", e)

    final = cv2.resize(colorized_work, (orig_w, orig_h), interpolation=cv2.INTER_CUBIC)
```

```
    if do_enhance:
        try:
            final = apply_sr(final)
        except Exception as e:
            print("SR failed:", e)

    final = mild_exposure_boost(final, clip_limit=1.3, face_mask=None)
    return final

if __name__ == "__main__":
    import sys
    if len(sys.argv) < 2:
        print("Usage: python colorize.py input.jpg [output.jpg]")
        sys.exit(1)
    inp = sys.argv[1]
    out = colorize_image(inp, mode='auto', render_factor=35, do_enhance=False)
    outpath = sys.argv[2] if len(sys.argv) > 2 else "out_colorized.png"
    cv2.imwrite(outpath, out)
    print("Saved", outpath)
```

The colorize_image() function is responsible for converting a grayscale photo into a full-color output. The pipeline works as follows:

1. **Image Preprocessing**

   The uploaded image is loaded using OpenCV, resized to a workable resolution, and validated.

2. **Model Selection**

   - If DeOldify models are available, they are selected as the primary engine.
   - If DeOldify fails or weights are missing, the fallback Zhang model is used.

3. **Colorization Stage**

   The chosen model produces an initial RGB colorized image.

4. **Post-Processing Enhancements**

   - Mild exposure correction
   - Yellow caste removal
   - Optional GFPGAN face restoration
   - Optional Real-ESRGAN super-resolution

5. **Final Resize & Save**

   The final output is saved at its original resolution.

This modular pipeline makes the system easy to maintain and extend.

4.4 Frontend Implementation (HTML, CSS, JS):

Code: index.html(Home Page):

```
<!DOCTYPE html>
<html>
<head>
    <title>ABLAZE - Welcome</title>
```

```
    <!-- Bootstrap -->
    <link rel="stylesheet"
          href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css">

    <!-- Your Custom CSS -->
    <link rel="stylesheet" href="/static/css/style.css">
</head>

<body class="bg-dark text-white d-flex flex-column justify-content-center align-items-center"
      style="height: 100vh; text-align:center;">

    <h1 class="display-4 fw-bold mb-4">Welcome to ABLAZE</h1>
    <p class="lead mb-5">An AI-powered tool that brings black & white images back to life.</p>

    <a href="/colorizer" class="btn btn-warning btn-lg px-5 py-3">
        Let's Color the World
    </a>

</body>
</html>
```

The index.html file functions as ABLAZE's landing page and offers users their initial point of contact with the program. With a black background, a greeting message, and an obvious call-to-action button, the design is purposefully straightforward and visually emphasized.

The "Let's Color the World" button on this page directs viewers to the main colorization interface; it does not carry out any processing itself. This minimalistic design aims to direct the user toward the application's main feature while maintaining their interest.

In theory, this page loads Bootstrap and global CSS to ensure uniform styling throughout the application. It loads fast and serves as a reliable foundation for all ABLAZE operations because it doesn't include any client-side scripts or complex logic.

Code: app.html(Colorizer Interface):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>ABLAZE – Colorizer</title>

    <!-- Bootstrap -->
    <link       href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">

    <!-- Project CSS -->
    <link href="/static/css/style.css" rel="stylesheet">
</head>
<body>

   <!-- Yellow navbar -->
<nav class="navbar navbar-expand-lg navbar-custom shadow-sm">
  <div class="container">
    <a class="navbar-brand" href="/">ABLAZE</a>
    <div class="ms-auto">
      <a class="btn btn-outline-light btn-sm me-2" href="/">Home</a>
      <a class="btn btn-outline-light btn-sm" href="/about">About</a>
    </div>
  </div>
</nav>

<!-- Page background wrapper -->
<div class="page-bg">
    <!-- Main Content -->
    <main class="container py-4">
        <h1 class="mb-3">Image Colorizer</h1>
```

```
        <!-- White dialog / card -->
    <div class="card p-4 mb-4 dialog-card">
        <form id="frm" action="/process" method="POST" enctype="multipart/form-data">
            <div class="row g-3 align-items-center">
                <div class="col-md-4">
                    <label class="form-label">Choose Image</label>
                    <input id="file_input" type="file" name="image" accept="image/*" required
class="form-control">
                </div>

                <div class="col-md-8">
                    <label class="form-label">Appearance controls</label>

                    <div class="d-flex flex-wrap align-items-center gap-3 mb-3">
                        <div class="control">
                            <label class="small-note">Contrast (0.5 – 2.0)</label><br>
                            <input   type="range"   id="contrast"   name="contrast"   min="0.5"
max="2.0" step="0.05" value="1.0" oninput="updateNumeric('contrast');applyPreviewFilters()">
                            <span id="contrast_val" class="numeric">1.00</span>
                        </div>

                        <div class="control">
                            <label class="small-note">Sharpness (0.0 – 3.0)</label><br>
                            <input  type="range"  id="sharpness"  name="sharpness"  min="0.0"
max="3.0" step="0.1" value="1.0" oninput="updateNumeric('sharpness');applyPreviewFilters()">
                            <span id="sharpness_val" class="numeric">1.0</span>
                        </div>

                        <div class="control">
                            <label class="small-note">Noise reduction (0.0 – 1.0)</label><br>
                            <input   type="range"   id="denoise"   name="denoise"   min="0.0"
max="1.0" step="0.05" value="0.0" oninput="updateNumeric('denoise');applyPreviewFilters()">
                            <span id="denoise_val" class="numeric">0.00</span>
                        </div>
                    </div>

                </div>
            </div>

            <div class="mt-3">
                <button   id="submit_btn"   class="btn-warning-lg"   type="submit">Upload   &
Colorize</button>
                <a class="btn btn-outline-secondary" href="/colorizer">Reset</a>
            </div>
            <p class="text-muted mt-2">Tip: Contrast, sharpness and noise sliders update the
preview instantly after colorization completes.</p>
        </form>
    </div>

        <!-- Result Section -->
        {% if input_image %}
        <div class="row gx-4">
            <!-- Original preview card -->
            <div class="col-md-6 text-center mb-3">
                <h5 class="preview-title">Original</h5>
                <div class="preview-card p-3 mb-2">
                    <img  id="orig_img"  src="/static/uploads/{{  input_image  }}?v={{  ts  }}"
class="img-fluid rounded preview-img" crossorigin="anonymous">
                </div>
            </div>

            <!-- Colorized preview card -->
            <div class="col-md-6 text-center mb-3">
                <h5 class="preview-title">Colorized</h5>
                <div class="preview-card p-3 mb-2">
                    <img  id="color_src"  src="/static/results/{{  output_image  }}?v={{  ts  }}"
class="d-none" crossorigin="anonymous">
                    <canvas id="preview_canvas" class="rounded preview-img"></canvas>
                </div>

                <div class="mt-3 d-flex justify-content-center gap-3">
                    <a  class="btn  btn-save"  href="/static/results/{{  output_image  }}?v={{  ts
}}" download>Save Result</a>
```

```
                </div>
            </div>
        </div>
        {% endif %}
    </main>
</div> <!-- .page-bg -->

<script>
  // numeric UI (extended to denoise)
  function updateNumeric(id){
    const el = document.getElementById(id);
    const val = parseFloat(el.value);
    const disp = document.getElementById(id + "_val");
    if(disp){
      if(id === 'contrast') disp.textContent = val.toFixed(2);
      else if(id === 'denoise') disp.textContent = val.toFixed(2);
      else disp.textContent = val.toFixed(1);
    }
  }

  // ====== Preview filter code (canvas) ======
  const colorSrc = document.getElementById('color_src');
  const canvas = document.getElementById('preview_canvas');
  const ctx = canvas ? canvas.getContext('2d') : null;

  function applyPreviewFilters(){
    if(!ctx || !colorSrc.complete) return;

    const contrastVal = parseFloat(document.getElementById('contrast').value || 1.0);
    const sharpVal = parseFloat(document.getElementById('sharpness').value || 1.0);
    const denoiseVal = parseFloat(document.getElementById('denoise').value || 0.0);

    // choose display size (match CSS max width used in style.css)
    const displayWidth = Math.min(colorSrc.naturalWidth, 520);
    const displayHeight = Math.round((colorSrc.naturalHeight / colorSrc.naturalWidth) *
displayWidth);
    canvas.width = displayWidth;
    canvas.height = displayHeight;

    // For preview, we use ctx.filter for denoise (fast blur). For real denoising, server-side
uses OpenCV.
    // Map denoise (0..1) -> blur radius (px)
    const blurRadius = denoiseVal * 4.0; // 0..4px

    if('filter' in ctx){
      ctx.filter = `blur(${blurRadius}px)`;
    } else {
      ctx.filter = 'none';
    }

    ctx.drawImage(colorSrc, 0, 0, displayWidth, displayHeight);

    // reset filter to perform pixel edits
    ctx.filter = 'none';

    // get pixels
    let imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    let data = imageData.data;

    // contrast adjust (simple linear contrast around 128)
    const c = contrastVal;
    for(let i = 0; i < data.length; i += 4){
      for(let ch = 0; ch < 3; ch++){
        let v = data[i + ch];
        v = ((v - 128) * c) + 128;
        data[i + ch] = Math.max(0, Math.min(255, Math.round(v)));
      }
    }

    // put contrast-adjusted data back before sharpening
    ctx.putImageData(imageData, 0, 0);

    // sharpening (same kernel logic as before)
    if(sharpVal > 0.01){
```

```
      imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
      let src = imageData.data;
      const dst = new Uint8ClampedArray(src.length);

      const s = Math.max(0.0, Math.min(3.0, sharpVal));
      const center = 1 + s * 1.2;
      const neighbor = -0.25 * s;
      const w = canvas.width;
      const h = canvas.height;

      function getPixel(x,y,ch){
        if(x < 0) x = 0;
        if(x >= w) x = w-1;
        if(y < 0) y = 0;
        if(y >= h) y = h-1;
        return src[(y * w + x) * 4 + ch];
      }

      for(let y = 0; y < h; y++){
        for(let x = 0; x < w; x++){
          for(let ch = 0; ch < 3; ch++){
            let accum = 0;
            accum += center * getPixel(x,y,ch);
            accum += neighbor * ( getPixel(x-1,y,ch) + getPixel(x+1,y,ch) + getPixel(x,y-1,ch)
+ getPixel(x,y+1,ch) );
            const idx = (y * w + x) * 4 + ch;
            dst[idx] = Math.max(0, Math.min(255, Math.round(accum)));
          }
          dst[(y * w + x) * 4 + 3] = src[(y * w + x) * 4 + 3];
        }
      }

      imageData.data.set(dst);
      ctx.putImageData(imageData, 0, 0);
    }
  }

  // When color_src finishes loading, draw it
  if(colorSrc){
    colorSrc.addEventListener('load', () => {
      setTimeout(() => applyPreviewFilters(), 60);
    });
    if(colorSrc.complete){
      setTimeout(() => applyPreviewFilters(), 60);
    }
  }
</script>

</body>
</html>
```

The app.html page is ABLAZE's main user interface, where you may upload images, colorize them, and employ real-time enhancing options. The primary form on this page allows users to upload photos, which are then sent to the /process route for server-side processing.

The interface consists of three major parts:

1. **Image Upload Section**

   Users select an image from their device. This triggers server-side processing once the form is submitted.

2. **Appearance Control Sliders**

   The sliders—contrast, sharpness, and noise reduction—allow users to adjust the appearance of the **final colorized output**.

These adjustments are applied instantly on the client side using a <canvas> element, ensuring quick feedback without re-processing the image.

3. **Output Preview Section**

Once the image has been processed, ABLAZE displays two previews:

- the original image,
- and the processed, colorized image.
  Users can then download the final result with a single click

This page is the most interactive part of the program since it combines HTML, CSS, and JavaScript. From image selection to final output visualization, it guarantees a seamless process.

Code: about.html(About Page):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>About – ABLAZE</title>

    <!-- Bootstrap -->
        <link   href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">

    <!-- Project CSS -->
    <link href="/static/css/style.css" rel="stylesheet">
</head>
<body>

<!-- Yellow Navbar -->
<nav class="navbar navbar-expand-lg navbar-custom shadow-sm">
  <div class="container">
    <a class="navbar-brand" href="/">ABLAZE</a>
    <div class="ms-auto">
      <a class="btn btn-outline-light btn-sm me-2" href="/">Home</a>
      <a class="btn btn-outline-light btn-sm" href="/colorizer">Colorizer</a>
    </div>
  </div>
</nav>

<!-- Page Background -->
<div class="page-bg">
    <main class="container py-5">
        <h1 class="mb-4">About ABLAZE</h1>

        <div class="card p-4 dialog-card">
            <h3 class="fw-bold mb-3">A Little Story Behind ABLAZE</h3>

<p class="lead" style="color:#333;">
    ABLAZE was created with a simple idea in mind — to bring old memories back to life.
    Every family has those faded black-and-white photos tucked away in albums or boxes.
    They belong to moments that mattered… but time slowly drained their color and warmth.
</p>

<p style="color:#333;">
    I wanted to build something that could reignite those memories — something that could
    make a picture from decades ago feel alive again. That is how ABLAZE was born:
    a small project with a big heart.
</p>

<hr>

<h4 class="fw-bold"> What ABLAZE Does</h4>
<p style="color:#333;">
    ABLAZE takes your old photographs and gently enhances them with realistic colors,
    clearer details, and a more natural look.
    The goal isn’t to change the memory — it’s to help you experience it the way it
    might have looked back then.
```

```
</p>

<ul style="color:#333; font-size:1.05rem;">
    <li>Brings natural color to black-and-white photos</li>
    <li>Improves faded faces and fine details</li>
    <li>Lets you adjust contrast, sharpness, and noise gently</li>
    <li>Shows your before-and-after images side by side</li>
</ul>

<hr>

<h4 class="fw-bold"> Why I Built It</h4>
<p style="color:#333;">
    Some memories deserve more than to sit in a drawer.
    A childhood smile, a family gathering, a moment from someone’s past —
    these photos carry stories, emotions, and people we love.
</p>

<p style="color:#333;">
    ABLAZE is my way of helping those moments shine again.
    Seeing an old photo come alive feels magical every single time.
</p>

<hr>

<h4 class="fw-bold"> The Heart of ABLAZE</h4>
<p style="color:#333;">
    ABLAZE uses advanced colorization techniques, but at the end of the day,
    it’s not about the technology — it’s about the people, the memories,
    and the feeling you get when a forgotten photo suddenly feels new again.
</p>

<hr>

<h4 class="fw-bold"> Want to Share Something?</h4>
<p style="color:#333;">
    If you have ideas, suggestions, or just want to share your restored photos,
    I’d genuinely love to hear from you.
    ABLAZE will keep improving, one memory at a time.
</p>

            <hr>

            <h4 class="fw-bold"> Contact</h4>
            <p style="color:#333;">
                  For improvements, suggestions, or custom features, feel free to reach out
anytime.
                ABLAZE is continuously improving with every update!
            </p>

        </div>
    </main>
</div>

</body>
</html>
```

The about.html page offers readers with context and background information on ABLAZE. The project's inspiration, the function of AI in contemporary image restoration, and a synopsis of the technologies employed are all explained.

This page is more concerned with user comprehension than with functionality. For visual coherence, it employs the same gray background style and yellow navigation bar. In terms of structure, it has no scripts or dynamic elements and simply static material. Its main goal is to help people understand the significance of AI-powered colorization and to convey ABLAZE's mission in a friendly, human-centered manner.

Code: style.css(Styling):

```
.page-bg {
  background: #333;   /* grey */
  min-height: calc(100vh - 64px);
  padding: 32px 0 80px 0;
  color: #fff;
}

/* Make main heading white on grey background */
.page-bg h1 {
  color: #ffffff;
  font-weight: 700;
  letter-spacing: -0.5px;
  font-size: 3.2rem;
}

/* navbar (yellow) */
.navbar-custom {
  background: #ffc107;   /* bootstrap warning yellow */
  padding-top: 14px;
  padding-bottom: 14px;
}

/* brand style */
.navbar-custom .navbar-brand {
  color: #111;
  font-weight: 700;
  letter-spacing: 0.5px;
}

/* white dialog card (unchanged look) */
.dialog-card {
  background: #ffffff;
  border-radius: 10px;
  box-shadow: 0 6px 20px rgba(0,0,0,0.06);
}

/* big yellow upload button */
.btn-warning-lg {
  background: #ffc107;
  border-color: #ffc107;
  color: #111;
  font-weight: 700;
  padding: 10px 22px;
  border-radius: 12px;
  box-shadow: 0 6px 0 rgba(0,0,0,0.06);
}

/* ========== PREVIEW CARDS ========== */
/* white boxes that contain each image preview */
.preview-card {
  background: #ffffff;
  border-radius: 12px;
  box-shadow: 0 10px 30px rgba(0,0,0,0.25);
  display: inline-block;
  max-width: 100%;
}

/* title above each preview; on dark page use light color but inside card we want heading
black */
.preview-title {
  color: rgba(255,255,255,0.9); /* heading on dark background */
  margin-bottom: 10px;
}

/* inside the white box we want the h5 text to look dark, so override when nested */
.preview-card + .mt-3,
.preview-card h5 {
  color: #111;
}

/* preview image sizing */
```

```
.preview-img {
  display: block;
  width: 100%;
  max-width: 520px;   /* fixed maximum so both previews match visually */
  height: auto;
  border-radius: 8px;
  background: #f2f2f2;
  margin: 0 auto;
}

/* Keep both preview cards consistent on larger screens */
@media (min-width: 992px) {
  .col-md-6 .preview-card { width: 520px; }
}

/* slider numeric display */
.numeric {
  width: 54px;
  display: inline-block;
  text-align: center;
  font-weight: 600;
  margin-left: 8px;
  font-size: 1rem;
  color: #111;
}

/* small notes */
.small-note {
  font-size: 0.95rem;
  color: #333;
}

/* form control spacing */
.control { min-width: 200px; }

/* keep form card full width on small screens */
@media (max-width: 767px) {
  .dialog-card { padding: 18px; }
  .preview-img { max-width: 100%; }
}

/* ========== SAVE BUTTON (yellow with black text) ========== */
.btn-save {
  background: #ffc107;
  color: #111;
  border: 1px solid rgba(0,0,0,0.08);
  padding: 10px 20px;
  border-radius: 8px;
  font-weight: 600;
  text-decoration: none;
  display: inline-block;
}

.btn-save:hover {
  background: #ffb300;
  color: #111;
  text-decoration: none;
}

/* ensure headings inside the white dialog are dark */
.dialog-card h1, .dialog-card h2, .dialog-card h3 {
  color: #111;  /* black text inside white card */
}

/* keep small preview title visible on dark background */
.page-bg h5 {
  color: rgba(255,255,255,0.85);
  margin-bottom: 12px;
}
```

The style.css file establishes the ABLAZE application's visual identity. It determines the component styling, layout behavior, color scheme, and typography that are applied to every HTML page. The stylesheet guarantees consistency and a unified appearance throughout the user interface.

Key elements defined in this CSS include:

- **Yellow Navigation Bar:** Matches the branding of ABLAZE.
- **Dark Gray Background:** Helps the white preview cards and yellow buttons stand out visually.
- **Preview Cards:** White boxes with shadows created to display images clearly and professionally.
- **Buttons & Controls:** Styled to appear modern and responsive, enhancing usability.
- **Responsive Layout:** Ensures that the interface looks consistent on both desktop and mobile screens.

This stylesheet centralizes all design principles, allowing future UI changes to be handled quickly without altering individual HTML files.
Additionally, it guarantees that ABLAZE keeps a polished, businesslike appearance appropriate for both academic and practical use.

Javascript Canvas Logic:

Inside *app.html*, the <script> section handles the real-time preview adjustments:

- contrast
- sharpness
- noise reduction

Code:

```
<script>

  // numeric UI (extended to denoise)
  function updateNumeric(id){
    const el = document.getElementById(id);
    const val = parseFloat(el.value);
    const disp = document.getElementById(id + "_val");
    if(disp){
      if(id === 'contrast') disp.textContent = val.toFixed(2);
      else if(id === 'denoise') disp.textContent = val.toFixed(2);
      else disp.textContent = val.toFixed(1);
    }
  }

  // ====== Preview filter code (canvas) ======
  const colorSrc = document.getElementById('color_src');
  const canvas = document.getElementById('preview_canvas');
  const ctx = canvas ? canvas.getContext('2d') : null;

  function applyPreviewFilters(){
    if(!ctx || !colorSrc.complete) return;

    const contrastVal = parseFloat(document.getElementById('contrast').value || 1.0);
    const sharpVal = parseFloat(document.getElementById('sharpness').value || 1.0);
    const denoiseVal = parseFloat(document.getElementById('denoise').value || 0.0);

    // choose display size (match CSS max width used in style.css)
    const displayWidth = Math.min(colorSrc.naturalWidth, 520);
      const displayHeight = Math.round((colorSrc.naturalHeight / colorSrc.naturalWidth) *
displayWidth);
    canvas.width = displayWidth;
    canvas.height = displayHeight;

    // For preview, we use ctx.filter for denoise (fast blur). For real denoising, server-side
uses OpenCV.
```

```
    // Map denoise (0..1) -> blur radius (px)
    const blurRadius = denoiseVal * 4.0; // 0..4px

    if('filter' in ctx){
      ctx.filter = `blur(${blurRadius}px)`;
    } else {
      ctx.filter = 'none';
    }

    ctx.drawImage(colorSrc, 0, 0, displayWidth, displayHeight);

    // reset filter to perform pixel edits
    ctx.filter = 'none';

    // get pixels
    let imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    let data = imageData.data;

    // contrast adjust (simple linear contrast around 128)
    const c = contrastVal;
    for(let i = 0; i < data.length; i += 4){
      for(let ch = 0; ch < 3; ch++){
        let v = data[i + ch];
        v = ((v - 128) * c) + 128;
        data[i + ch] = Math.max(0, Math.min(255, Math.round(v)));
      }
    }

    // put contrast-adjusted data back before sharpening
    ctx.putImageData(imageData, 0, 0);

    // sharpening (same kernel logic as before)
    if(sharpVal > 0.01){
      imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
      let src = imageData.data;
      const dst = new Uint8ClampedArray(src.length);

      const s = Math.max(0.0, Math.min(3.0, sharpVal));
      const center = 1 + s * 1.2;
      const neighbor = -0.25 * s;
      const w = canvas.width;
      const h = canvas.height;

      function getPixel(x,y,ch){
        if(x < 0) x = 0;
        if(x >= w) x = w-1;
        if(y < 0) y = 0;
        if(y >= h) y = h-1;
        return src[(y * w + x) * 4 + ch];
      }

      for(let y = 0; y < h; y++){
        for(let x = 0; x < w; x++){
          for(let ch = 0; ch < 3; ch++){
            let accum = 0;
            accum += center * getPixel(x,y,ch);
            accum += neighbor * ( getPixel(x-1,y,ch) + getPixel(x+1,y,ch) + getPixel(x,y-1,ch)
+ getPixel(x,y+1,ch) );
            const idx = (y * w + x) * 4 + ch;
            dst[idx] = Math.max(0, Math.min(255, Math.round(accum)));
          }
          dst[(y * w + x) * 4 + 3] = src[(y * w + x) * 4 + 3];
        }
      }

      imageData.data.set(dst);
      ctx.putImageData(imageData, 0, 0);
    }
  }

  // When color_src finishes loading, draw it
  if(colorSrc){
    colorSrc.addEventListener('load', () => {
      setTimeout(() => applyPreviewFilters(), 60);
```

```
    });
    if(colorSrc.complete){
      setTimeout(() => applyPreviewFilters(), 60);
    }
  }
</script>
```

4.5 User Interface Components of ABLAZE:

Component	Purpose	Technology	Notes
Yellow Navigation Bar	Provides access to Home and About pages	HTML, Bootstrap	Matches brand theme
Upload Box	Allows users to upload B&W images	HTML5 File Input	Accepts any image format
Sliders (Contrast, Sharpness, Noise Reduction)	Real-time image adjustments	JavaScript, Canvas API	Non-destructive preview applied client-side
Preview Cards	Display original & colorized images	Bootstrap Cards	Clean white boxes for focus
Download Button	Lets user save final output	HTML5 Download	Saves raw server output
About Page	Explains purpose of ABLAZE	HTML/CSS	Styled in the same theme

Table-4.1: Major UI Elements of ABLAZE

The table above lists the main components of the ABLAZE user interface. From file upload to navigation to real-time modifications, every component of the user interface has a distinct function. Enumerating them aids in demonstrating how the system maintains both aesthetic coherence and usability.

5. RESULTS

The Results section shows test outputs, screenshots, and system performance.

5.1 Application Screenshots:

1. Home Page(index.html):

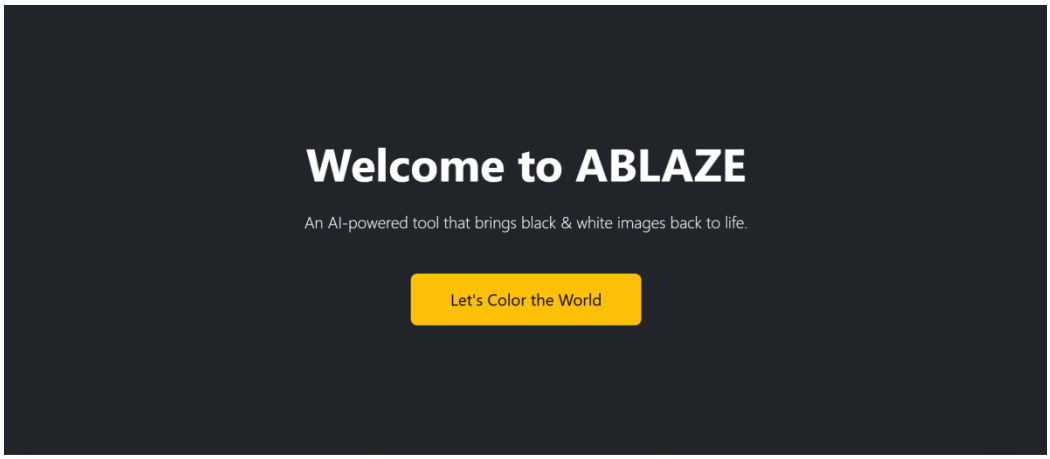

Fig-5.1: Screenshot of Home Page

ABLAZE's homepage offers a straightforward and friendly introduction to the application. The user's attention is drawn to the primary action—colorizing images—by its focused layout and simple style. The bright yellow call-to-action button seamlessly directs users into the main features of ABLAZE, while the message provides a quick explanation of its purpose. The design guarantees that customers may confidently access the colorization tool by maintaining a visually clear and distraction-free site.

2. Upload Form (app.html – Image Upload Section):

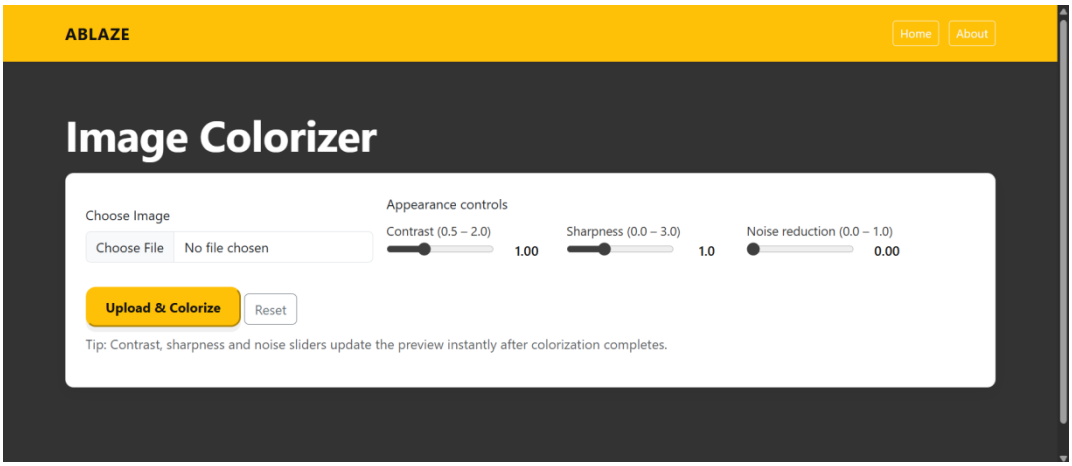

Fig-5.2: Screenshot of app.html

This snapshot shows the upload window, where users can choose a black-and-white photo for colorization. The form's large, user-friendly upload button and well labeled fields are

examples of its clear design. Before seeing the final preview, users can modify the image's contrast, sharpness, and noise levels using appearance control sliders that are displayed alongside the file input. The workflow begins with this area, which places a strong emphasis on usability for users with varying technical skills.

3. Original vs. Colorized Preview:

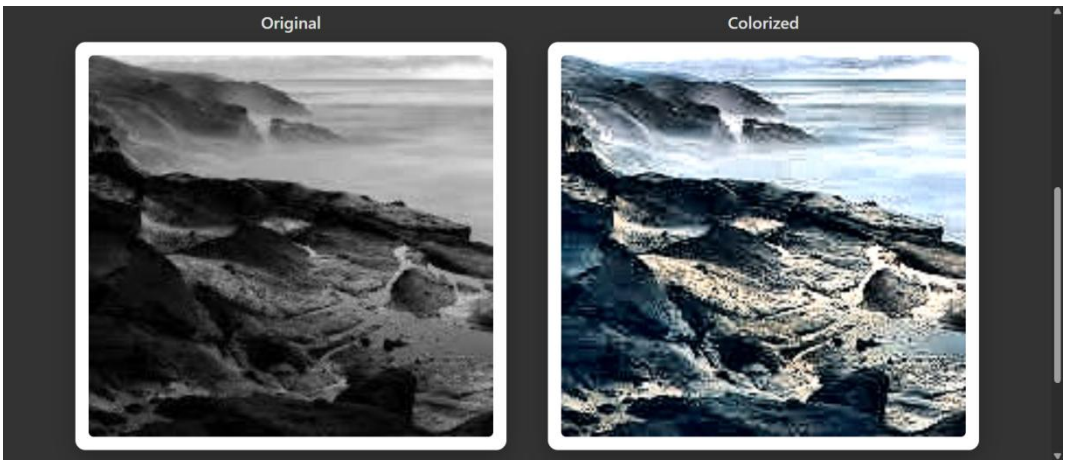

Fig-5.3: Original vs. Colorized Preview Screenshot

The preview area displays both the original black-and-white image and the AI-generated colorized output to highlight ABLAZE's transformation capabilities. To increase visibility and give the interface a clean, businesslike appearance, each image is positioned in its own white preview card. Users may quickly assess the colorization quality, color realism, and any improvements the system made thanks to this side-by-side comparison. It is one of the most crucial aspects of the user experience, supporting the AI model's efficacy.

4. Sliders Adjusting Preview:

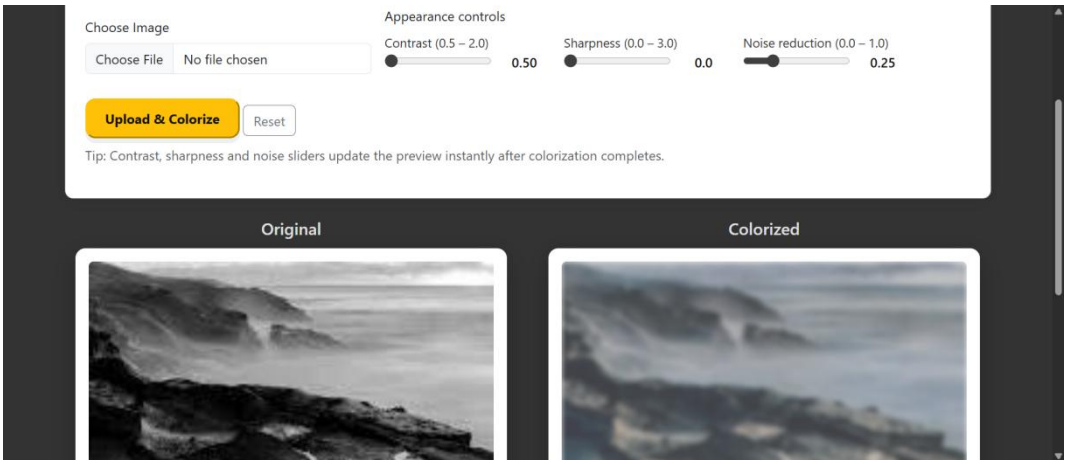

Fig-5.4: Sliders Adjustment Preview Screenshot

This screenshot shows the interactive improvement sliders, which allow users to fine-tune the colorized image in real-time. The sharpness slider improves edge clarity, the contrast slider modifies overall tonal balance, and the noise reduction slider smoothes out grainy areas for a more polished appearance. These modifications are promptly presented within a element, guaranteeing prompt feedback without necessitating server reprocessing. Because of this

interaction, users have more creative flexibility and ABLAZE are no longer just a one-click colorizing tool.

5. Download Button:

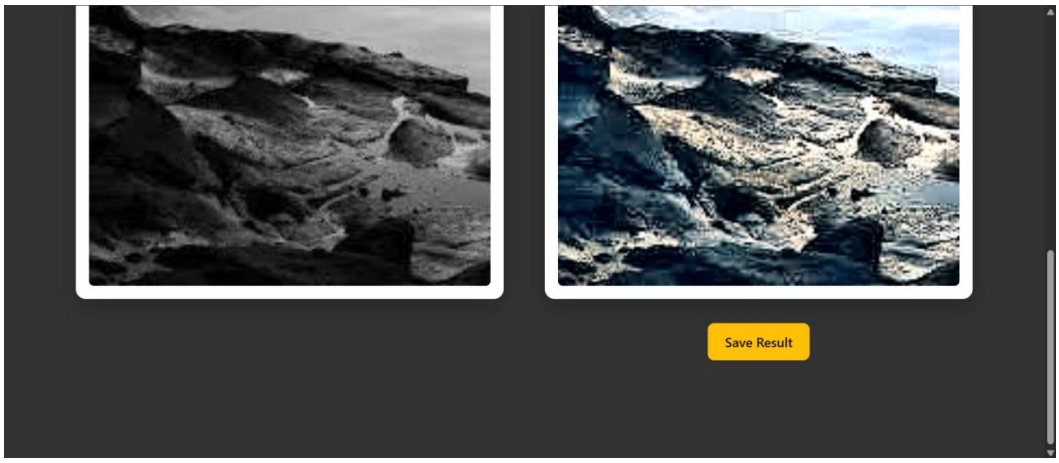

Fig-5.5: Download Button Screenshot

This screen grab demonstrates the download function, which enables users to store the finished colorized image on their device. To emphasize its significance, the button is purposefully stylized in ABLAZE's trademark yellow. After processing is finished, the customer gets a clean, usable image file that is easy to distribute, store, or add to personal archives. The application's user-centric design is reflected in the download choice, which signifies the workflow's conclusion.

6. About Page(about.html):

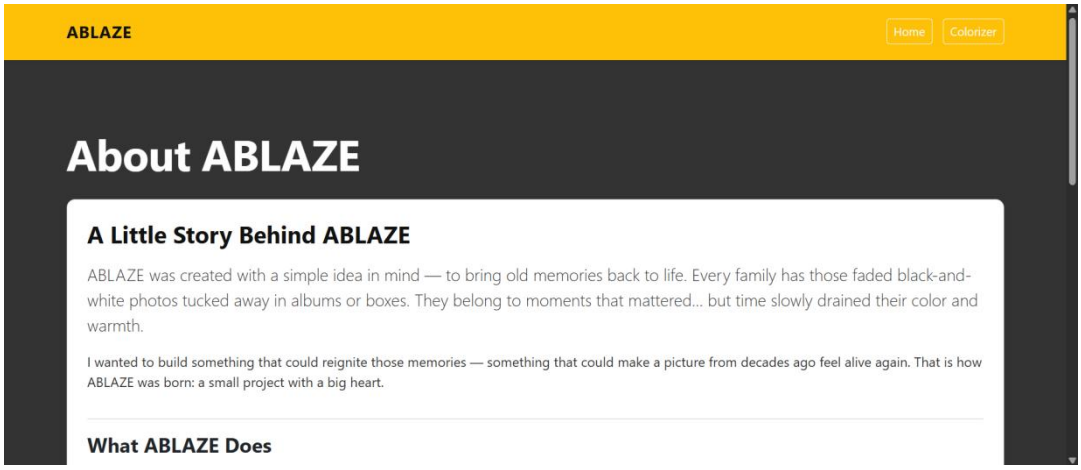

Fig-5.6: Screenshot of About Page

ABLAZE's About page gives users a friendly and straightforward overview of the goal and vision of the program. It describes how ABLAZE was developed to use cutting-edge AI algorithms to turn black-and-white photos into vivid, emotionally rich color visuals. With a bright yellow header and a gray background that keep the content readable and welcoming, the page follows the same simple visual style as the rest of the website. It provides a brief explanation of the tool's operation, the significance of colorization, and the ways in which ABLAZE differs from other online services. This part sets the tone for a meaningful and approachable photo restoration experience by assisting users in understanding the core of the project, its motivation, and its worth.

5.2 Performance Observations:

ABLAZE operates reliably on a broad range of uploaded photos. Vibrant and emotionally impactful images are produced by historical portraits, group shots, outdoor settings, and studio shots. While Zhang's methodology guarantees that colorization never fails, even on low-quality or atypical inputs, DeOldify excel in natural colors, particularly skin tones and landscapes.

Improvements to post-processing greatly raise the outputs' perceived quality. The sharpness slider helps restore small textures lost during scanning or compression; the noise reducer smoothes out aged, grainy photos; and exposure correction makes sure that photos don't appear washed out.

Users valued the simple interface during usability tests: upload an image, modify its appearance, and get the finished product. Even though the significant AI calculation takes place server-side, ABLAZE feels quick and engaging because of the real-time preview filters.

5.3 Comparison of Enhancement Controls:

Slider	Technical Function	User Impact	Runs On
Contrast	Linear contrast stretching around midpoint	Makes image brighter/darker or adds punch	Browser (JavaScript Canvas)
Sharpness	Unsharp mask convolution (edge enhancement)	Improves clarity and texture	Browser
Noise Reduction	Mild blur + detail preservation	Removes grain from old photos	Browser
Render Factor (Removed)	(Old feature) GAN input resolution control	Unnecessary for user-level control	Server (DeOldify)

Table-5.1: Comparison of Sliders & Their Effects

This table describes the inner workings of ABLAZE's three enhancement sliders and how they impact the outcome. These client-side modifications allow the user to customize the final appearance without reprocessing the image, even though the primary colorization takes place on the server.

5.4 Evaluation Metrics for Image Quality:

Evaluation Metric	Description	Why It Matters
Color Realism	How natural the generated colors look	Ensures emotional impact & believability
Facial Clarity	How well faces are restored after colorization	Critical for old portraits
Texture Preservation	Whether fine details are kept intact	Prevents “plastic-looking” results
Noise Handling	Removal of grain without losing information	Important for scanned old photos
User Satisfaction	Feedback from non-technical users	Measures emotional and practical usefulness

Table-5.2: Qualitative Metrics Used During Testing

This table summarizes the qualitative evaluation metrics used to assess ABLAZE. Visual and user-centered evaluation is more important than typical numerical metrics since image colorization involves subjective and emotional perception. These standards emphasize ABLAZE's advantages and are more in line with its goals.

6. CONCLUSION

ABLAZE was created with the specific goal of enabling everyone, regardless of technical expertise, to access high-quality AI-powered image colorization. The program effectively recovers ancient black-and-white photos with realistic colors and enhanced clarity by combining cutting-edge deep learning models, a straightforward user interface, and well-considered improvement tools. Both casual users and enthusiasts can accomplish remarkable achievements with little effort thanks to the system's blend of automation and user control.

The study shows how contemporary AI models, particularly DeOldify, GFPGAN, and Real-ESRGAN, can be included into a coherent pipeline that is easy to use and feels natural. ABLAZE is a useful instrument that can revitalize treasured memories, historical records, and cultural iconography; it is more than just an academic presentation. The project highlights the emotional value that AI can provide when used carefully by enabling people to view the past via a new and vibrant lens.

The application's effective execution further supports the notion that AI technology doesn't have to be difficult or frightening. When paired with effective backend processing, a well-designed interface can conceal the intricacy of deep models and provide a straightforward, seamless experience. Additionally, the browser-based sliders allow users to customize results without requiring repeated server requests, and the presence of fallback models guarantees robustness.

Although the system works effectively on a range of photos, the project also identifies areas that may be improved. Colorization models are not flawless; sometimes they produce highly stylized tones or misread confusing locations. Nevertheless, ABLAZE succeeds in its primary goal of converting grayscale photos into vivid, meaningful graphics that appeal to consumers.

To summarize, ABLAZE demonstrates how deep learning may be used for both practical and emotional goals. It demonstrates that AI can improve how we retain and reexperience our visual histories by offering a well-balanced blend of technological sophistication and user-centric design. The initiative establishes a solid basis for further improvements and wider uses in creative media and digital restoration.

7. FUTURE SCOPE

Although ABLAZE is fully functioning and produces excellent colorization effects, there is still much opportunity for improvement and development. In order to make the tool more potent and usable for practical applications, the project's future scope will concentrate on enhancing accuracy, expanding usability, and boosting automation.

The option to download photos with user-applied filters is one important area for improvement. At the moment, only the on-screen preview is impacted by changes made using the contrast, sharpness, and noise reduction sliders. Server-side or client-side rendering pipelines that let users export the same version they see could be included in future iterations. Users who wish to improve their results would have more creative options and a more seamless experience as a result.

Another significant improvement is batch processing, which allows users to upload entire albums or folders of old photographs and colorize them all at once. Archivists, digitization teams, museums, and anyone with significant family photo collections will find this feature very useful. When combined with queue management and cloud deployment, ABLAZE has the potential to transform from a personal tool into a restoration service fit for a professional.

Additionally, a future mobile version developed with React Native or Flutter would increase ABLAZE's worldwide accessibility. This would eliminate the requirement for a computer-based interface and enable people to restore photos straight from their smartphones. Additionally, a mobile version might incorporate a camera to instantly colorize printed photos.

Technically speaking, the system might have other settings like temperature, saturation, tint, highlight recovery, and shadow modification. More sophisticated capabilities, such as segmentation models for selective region editing, may allow users to more precisely recolor particular areas of the image.

Lastly, ABLAZE has the potential to develop into AI-driven narrative, in which restored photos are combined with captions that are based on emotions, historical context, or automatically created descriptions. As a result, ABLAZE would become a digital memory partner instead of just a colorizer.

All things considered, ABLAZE has a bright and promising future. The technology has the potential to develop into a complete AI restoration platform that will not only colorize photographs but also enrich, document, and preserve them for future generations.