CreativeCard Project Documentation

A Greeting Card Cover Designer using Generative AI

Prepared by:
Jethro P. Moleño

Acknowledgement

I sincerely express my gratitude to Mr. Greg Sarmiento and all the team leaders for their valuable feedback, continuous guidance, and support throughout the development of this project. Their leadership and dedication for me to inspire and accomplish this project have greatly contributed to the development phase. Also, I would like to thank my family and friends for their encouragement and understanding during the challenging times of this project.

Table of Contents

List of Figures

A Abstract

This documentation provides a comprehensive overview of CreativeCard application. The project focuses on image generation using trained AI model to give users an efficient solution in creating front or cover pages for greeting cards.

The documentation outlines the software's features, system requirements, installation instructions, and user manual describing on how to utilize the application effectively. Also, it also includes the detailed progress that the developer encountered throughout the development process such as limitations and work progress.

The CreativeCard was designed with user-friendly, flexibility and compatibility principles, making it a versatile tool for the users in Ollopa Corporation. This document serves as a reference for users, providing step-by-step instructions and insights to help them maximize the software's functionality.

B Purpose and Overview

The users or employees of Ollopa Corporation will be able to generate different images based on the input text prompt using the application. Also, the user can also save the generated image on their chosen directory.

This application uses a Stable Diffusion XL Model, which is fine-tuned using a dataset containing images from one of the websites from Ollopa, FibeiGreetings. By using default parameters when training the model, the application can generate high quality images depending on the text prompt provided by the user.

The greeting card cover image generator speeds up the design process for card covers, making it easier and more efficient. Employees no longer need to spend time thinking about or piecing together designs from Canva; instead, they can quickly generate covers, allowing them to focus more on other tasks and work more smoothly.

C Scope and Features

C.1 Generate Image

This allows the user/employees to generate images with a click of a button. After loading is finished, the image is generated and displayed in the image frame.

C.2 High Quality Image

Users can generate high quality images with a resolution of 1500 x 2000, which is a default resolution for 5 x 7" inches cover page of the card.

C.3 Image Prompt

Images can be generated by typing a text prompt to describe the image that they want to generate. This involves the elements, background and styles of the image that they want to include in the image.

C.4 Save Image

Users can choose what file directory will the image be saved.

C.5 Software Compatibility

Users can run in most of the computer with minimum specifications of the software.

C.6 Portability

Users can run the application without having a need to install any third-party software or packages from python.

D Limitations

D.1 Generate Image

The application can't generate images with text as it is one of the limitations of the base model, Stable Diffusion XL (SDXL). Also, the user must wait for the image to be displayed before generating image again since it can only generate one image at a time. Also, it cannot generate factual people or events and does not generate photorealism.

D.2 High Quality Image

Image generated by the application are only limited to 1500 x 2000-pixel resolution which may result in higher file size.

D.3 Image Prompt

The text prompt input needs to be detailed so that the image generated is aligned with the user's preferences or expectations.

D.4 Save Image

Generated image can only be saved locally or in the same machine.

D.5 Software Compatibility

The application needs a higher specification than the average office computer might not have. Also, it can only run on a machine with the Windows Operating System.

D.6 Portability

The application needs a high storage capacity.

E Project Development Process

This section discusses the development of the application which provides different flowcharts, block diagrams, GUI and testing phase of the application. Furthermore, the application development consists of different parts: Training and Deployment.

E.1 Dataset Preparation

E.1.1 Image Resizing

The 10 dataset images were cropped using a website called BIRME, which was used for resizing the images to 1024 x 1024 resolution as shown in the Figure E.1 below. The images were saved as a zip file renamed as image.

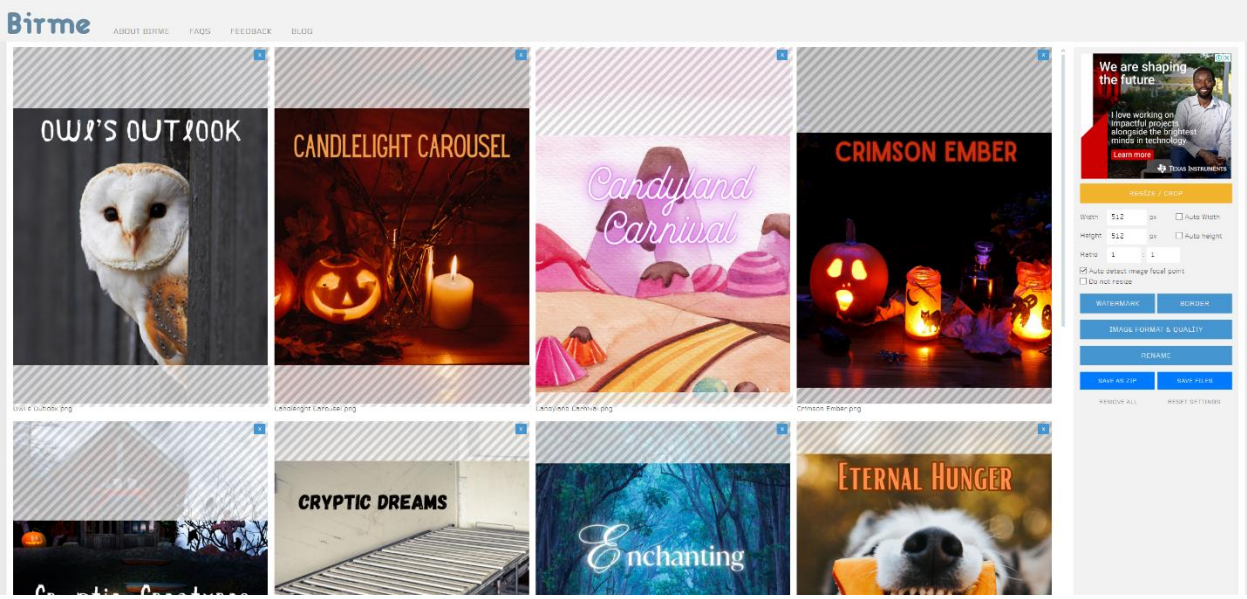

Figure E.1 Image Resizing in BIRME Website

E.1.2 Image Captioning

The captioning of images was done using a python library called Kohya_SS, implemented using Automatic1111's web interface. When the software was opened, it displayed a command prompt, shown in Figure E.2, where the python libraries and CUDA Nvidia Toolkit were initialized. Also, the local URL was also displayed and accessed using the browser illustrated in Figure E.3. The images were captioned using BLIP Captioning tool from the utilities of the web interface. Then, the image directory of saved dataset images was included in the "image to caption" section in the GUI. With default settings, the images were captioned when the caption images button was pressed. Afterwards, the captions were placed in a .txt file as indicated in the GUI. The developer reviewed the txt file and added a few texts to the caption to describe the image as much as possible. Furthermore, the captions for each image were formatted in a way that every new line corresponds to the image captioned, as shown in Figure E.4.

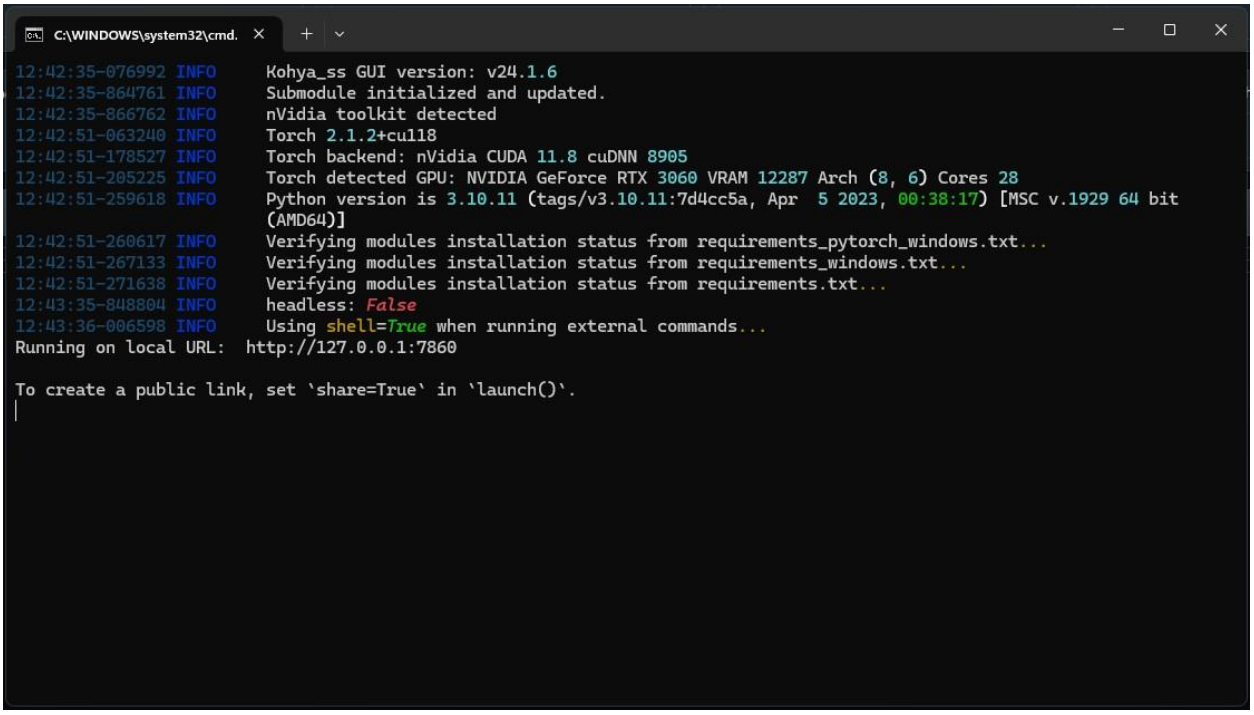

Figure E.2 Kohya_ss installation

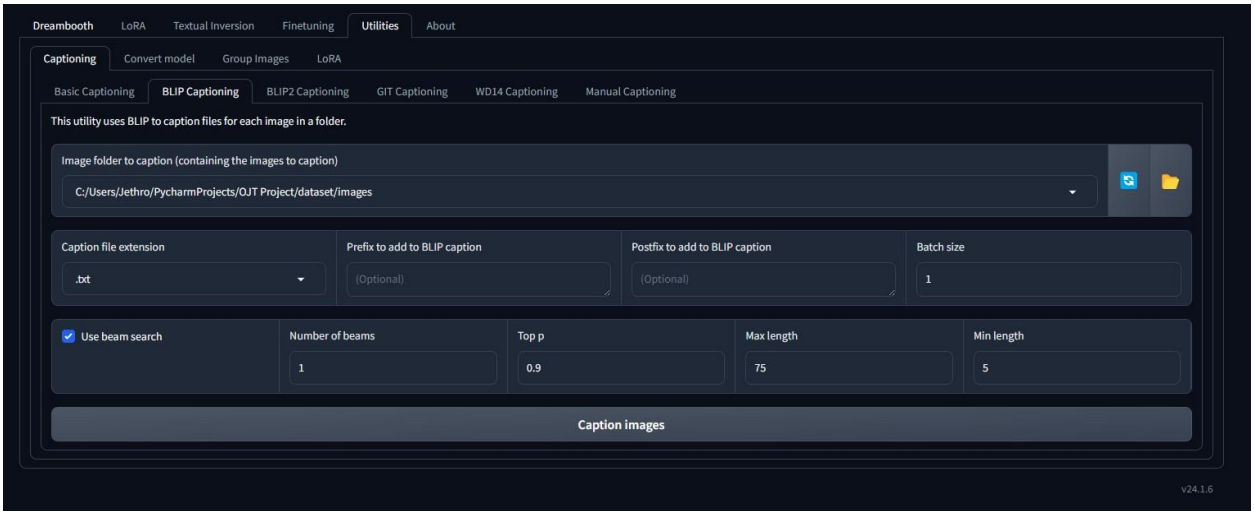

Figure E.3 BLIP Captioning

```
"a Halloween greeting card with entitled english text "Owl's Outlook" and with a barn owl peeks out of a black wooden fence"
"a Halloween greeting card with entitled english text "Candlelight Carousel" and with a candlelight carousel with three carved pumpkins"
"a Halloween greeting card with entitled english text "Candyland Carnival" and with a candy quest carnival with candy and candy balls"
"a Halloween greeting card with entitled english text "Crimson Ember" and with a group of halloween pumpkins sitting on top of a table"
"a Halloween greeting card with entitled english text "Cryptic Creatures" and with a stone path with a lit candle on it"
"a Halloween greeting card with entitled english text "Cryptic Dreams" and with a metal bed sitting in a room next to a wall"
"a Halloween greeting card with entitled english text "Enchanting Enchantment" and with a road with trees and a person walking down it"
"a Halloween greeting card with entitled english text "Eternal Hunger" and with a dog with a pumpkin in its mouth"
"a Halloween greeting card with entitled english text "Halloween Whisper" and with a group of carved pumpkins sitting on top of a table"
"a Halloween greeting card with entitled english text "Haunted Glow" and with a group of skeletons sitting on the steps of a house"
```

Figure E.4 prompts.txt

E.2 Model Training

E.2.1 Training Flowchart

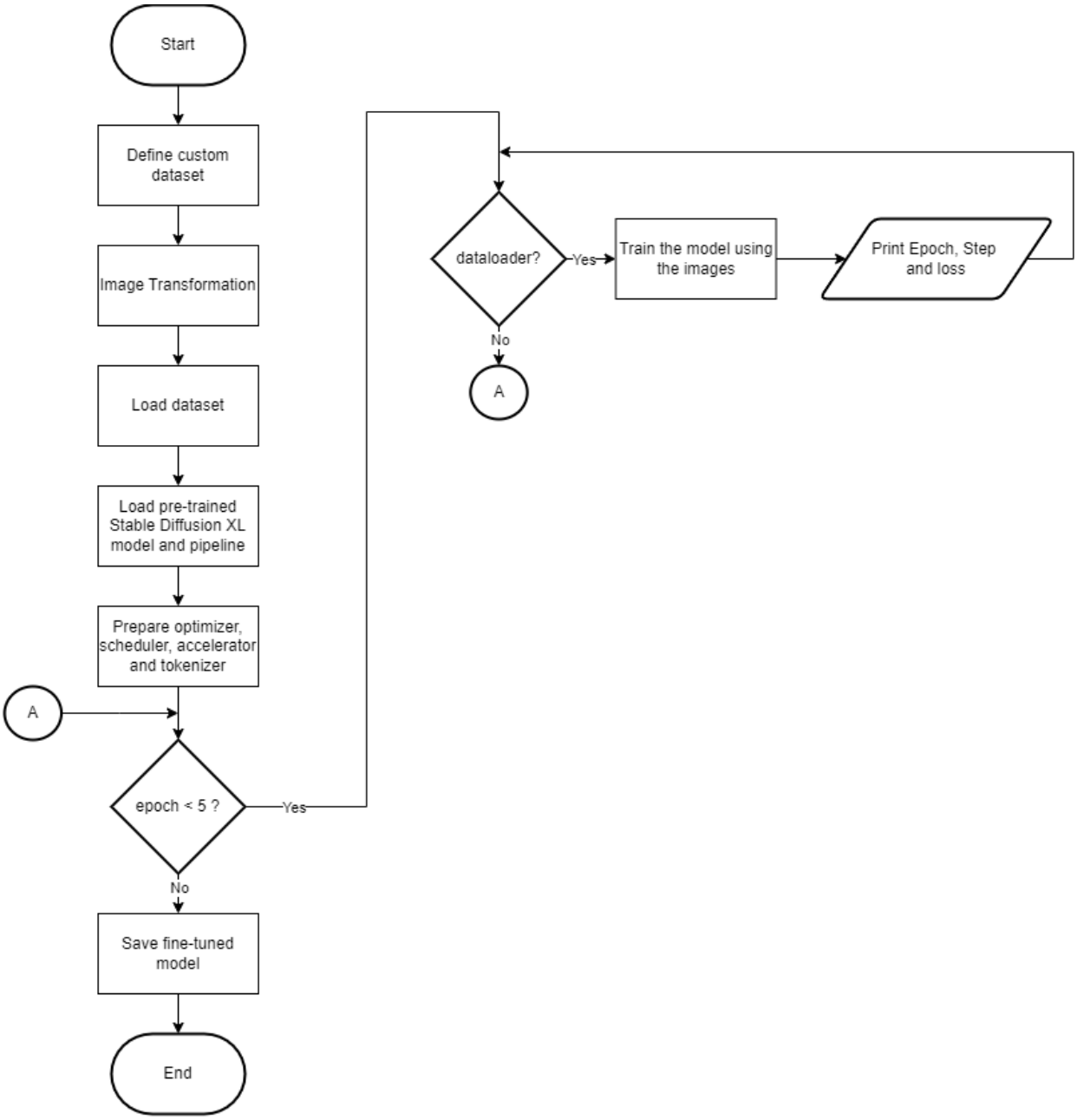

Figure E.5 Training Flowchart

Figure E.5 illustrates the flowchart for model training. Firstly, the dataset is defined through a series of parameters. The image directory, prompt file and image transform parameters are defined. Also, the images are opened, read, and sorted based on the number inside the parentheses so that both image and the captions inside the prompt file are matched. The image and their respective prompts are retrieved with a format compatible for training. Then, the images are pre-processed by resizing them to 1024 x 1024 resolution, which is suitable for training the Stable Diffusion XL (SDXL) model.

Also, the images are also converted to a tensor format compatible with model computations when training. The pixel values of the images are also normalized with a standard deviation of 0.5 to ensure the training stability. Afterwards, the SDXL model and its pipeline are loaded. The dataloader, optimizer, scheduler and tokenizers are also loaded together with the model to initialize model configurations. Furthermore, accelerator is also loaded to speed up the training process. Afterwards, it will proceed to the training loop. The epoch is set to 5 by the developer to which corresponds to how many times the dataset is trained in a complete cycle. Since the dataset contains 10 images and the batch size is set to 1, there will be 50 iterations (10 steps for each epoch) for training. Also in training, the mse (Mean Squared Error) loss is also computed and displayed together with the Step and Epoch. After the training finished, the fine-tuned model is saved in the model directory.

E.2.2 Code

```
from diffusers import StableDiffusionXLPipeline, DDPMScheduler
from transformers import CLIPTokenizer
import torch
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms as Trans
from accelerate import Accelerator
from PIL import Image
import os
```

Firstly, the tools and libraries were imported in the modeltrain.py which are essential for training the SDXL model, as shown in the code above.

From diffuser's library:

'StableDiffusionXLPipeline' - used for loading the model and its pipeline, which provides methods to generate images using text prompts.

'DDPMScheduler' - used to manage the timesteps and noise schedules while training.

From transformer's library:

'CLIPTokenizer' - transforms text prompts into numerial values or Token ID format that the model can process on during its training.

From torch's library:

Torch - used to provide tensor and neural network operations functionalities which is used for model training.

'DataLoader' - handles the batch size parameter for the model and operates loading and batching of dataset.

'Dataset' - used as a class to create a custom dataset that converts the dataset images into a readable format for model training.

'transforms' – used for image preprocessing such as cropping, resizing, normalization and tensor conversion.

From accelerate's library:

'accelerator' – used to simplify model training process by managing training workflows on the GPU (graphic processing unit) or other hardware used for training.

From PIL's library:

'Image' – used for image saving, opening and operations.

From OS' library:

'os' – used for file and directory operations.

```
# Define the custom dataset class
class GreetingCardDataset(Dataset):
    def __init__(self, image_directory, prompts_file,
transforms=None):
        self.image_directory = image_directory
        self.transforms = transforms
        with open(prompts_file, "r") as file:
            self.prompts = file.readlines()
            # Sort the images considering the format "image
(1).jpg"
            self.image_files = sorted(
                os.listdir(image_directory),
                key=lambda x: int(x.split("(")[1].split(")")[0])
            )

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, index):
        image_path = os.path.join(self.image_directory,
self.image_files[index])
        img = Image.open(image_path).convert("RGB")
        if self.transforms:
            img = self.transforms(img)
        prompt = self.prompts[index].strip()
        return prompt, img
```

Then, the custom dataset class was defined by the 'GreetingCardDataset' class, as shown in the code above. It contained image loading using '__init__', '__len__' and '__getitem__' method. In 'init' method, the parameters for image directory, prompt file and image transform were initialized. Also, the prompt file was read per line into 'self.prompts' which expects that each line corresponds to the caption per each image. Afterwards, the sorting of image was done using 'sorted' function based on the numbers inside the parentheses to ensure that both the image and caption in the prompt file matched correctly.

```
# Define transformations for the images
image_transforms = Trans.Compose([
    Trans.Resize((1024, 1024)),
    Trans.ToTensor(),
    Trans.Normalize([0.5]*3, [0.5]*3),
])
```

The code shown above defines the image transformation that was done with the dataset images. Firstly, the 'compose' modules allows the transformations to occur in a sequential manner. The 'resize' module changes the image resolution to 1024 x 1024 pixels. On the other hand, 'ToTensor' module converts the PIL images to tensor values

which are values expected by the model for training. Also, 'Normalize' module ensures that the tensor image's normalization by assigning each channel (RGB values) were set to 0.5 mean and standard deviation. This resulted to an even distribution of pixels and consistent values which helped the model train faster.

```
# Load dataset
dataset = GreetingCardDataset(
    image_directory=r"C:\Users\Jethro\PycharmProjects\OJT
Project\dataset\images",
    prompts_file=r"C:\Users\Jethro\PycharmProjects\OJT
Project\dataset\prompts.txt",
    transforms=image_transforms
)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

In the code above, dataset was loaded and assigned the path for both image directories and prompt file. The dataset images are also loaded using the 'DataLoader' module which set the batch size to 1. Also in this part, the image transformation shown earlier was done in this section.

```
# Load the pre-trained Stable Diffusion XL model and pipeline
pipeline =
StableDiffusionXLPipeline.from_pretrained("stabilityai/stable-
diffusion-xl-base-1.0")
pipeline.to("cuda")  # Move to GPU
```

Then, the pre-trained SDXL model was loaded using the 'StableDiffusionXLPipeline'. The model is loaded and fetched from "stabilityai/stable-diffusion-xl-base-1.0" repository. Then the model and its components is moved to the GPU's computing platform 'cuda' since it will be trained faster than moving it to the CPU.

```
# Prepare optimizer and scheduler
optimizer = torch.optim.AdamW(pipeline.unet.parameters(), lr=1e-
5)
scheduler = DDPMScheduler(num_train_timesteps=1000)
```

Afterwards, the optimizer and scheduler for the model was initialized. The optimizer was set as AdamW, which is also called Adam optimizer with a weight decay. This normalizes the model and prevent overfitting. It also set the model's learning rate parameter to 0.00005. On the other hand, the scheduler was used for the model's reverse diffusion process. It also set the timestep to 1000 which means that the model goes through 1000 iterations before the reverse diffusion process occurs.

```
# Initialize the accelerator
accelerator = Accelerator(mixed_precision="fp16")
```

Then, the accelerator was initialized, and the mixed precision is set to 'fp16' or 16-bit floating point precision. This enhances the speed of training, thus increasing the performance by improving the memory and computation efficiency.

```
# Tokenizer
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-base-patch32")
```

The code shown above initializes the pre-trained tokenizer from OpenAI's pre-trained CLIP model which was used to convert text prompt into Token ID or numerical representations which the model can process.

```
# Training loop
for epoch in range(5):
    for step, (text_prompt, image_data) in enumerate(dataloader):
        # Tokenize the text prompt
        text_inputs = tokenizer(text_prompt, return_tensors="pt", padding=True, truncation=True).input_ids.to("cuda")

        # Convert text inputs back to list of strings for the pipeline
        text_prompt_list = [tokenizer.decode(ids, skip_special_tokens=True) for ids in text_inputs]

        # Forward pass through the model
        with torch.cuda.amp.autocast():
            # Generate image based on prompts and input image
            output = pipeline(prompt=text_prompt_list, image=image_data)
            # Access the generated image
            generated_image = output.images[0]  # PIL Image

        # Convert PIL Image to tensor
        generated_image_tensor = Trans.ToTensor()(generated_image).unsqueeze(0).to("cuda")
        image_data = image_data.to("cuda")

        # Resize generated image tensor to match input image size
        if generated_image_tensor.size() != image_data.size():
            generated_image_tensor = Trans.Resize(image_data.size()[2:])(generated_image_tensor)

        # Clamp tensor values to [0, 1] before scaling
        generated_image_tensor = torch.clamp(generated_image_tensor, 0.0, 1.0)
        image_data = torch.clamp(image_data, 0.0, 1.0)

        # Ensure gradients are enabled for the model parameters
        for param in pipeline.unet.parameters():
            param.requires_grad = True

        # Ensure the tensors used for loss calculation require gradients
        generated_image_tensor.requires_grad = True
        image_data.requires_grad = True
```

```
        # Compute the loss between the generated image and the
original image
        loss =
torch.nn.functional.mse_loss(generated_image_tensor, image_data)

        # Backpropagation
        accelerator.backward(loss)
        optimizer.step()
        optimizer.zero_grad()

        # Print progress
        if step % 10 == 0:
            print(f"Epoch {epoch}, Step {step}, Loss:
{loss.item()}")
```

The training loop code is shown above iterates depending on the number of epochs. In this case, the epoch is set to 5, since the developer tried to test with 10 epochs, but it takes more time. Each epoch passes through the entire dataset and processes in small steps to train the model. In the code, for loop was used to iterate five times, which was the epoch value. Then each iteration, the dataloader, which handles the retrieval of both image data and text prompt, loads one pair of data at a time. Then the text prompt was tokenized and moved to the GPU for processing. It was also decoded back as list of strings, to be used for image generation by the model. Afterwards, the model tries to generate an image as a 'PIL' image and stored in generated_image variable. It was then converted into a tensor format and moved into the GPU for computation. To ensure that the size of both input image tensor and generated image tensor was the same, the size of both is matched using if statement. The pixel values are clamped with a range of [0,1] to ensure validity during processing. Afterwards, the gradients were enabled so that backpropagation can be done. Also, the MSE loss is also calculated which is the difference in the generated image from the true image. This loss value is backpropagated through the model and updates the model's weights. Subsequently, for debugging purposes, the training log was displayed by displaying the epoch, step and MSE loss every 10 steps.

```
# Save the fine-tuned model
pipeline.save_pretrained(r"C:\Users\Jethro\PycharmProjects\OJT
Project\ModelV2")
```

When the training loop is completed, the fine-tuned model is saved to the specified category.

E.2.3 Execution

When the modeltrain.py is executed, it first loads the model's pipeline components. Then proceeds with the training. The developer used the machine with the following specifications:

Processor: AMD Ryzen 5 5600 6-Core

GPU: NVIDIA RTX 3060

GPU VRAM: 12 GB

RAM: 32 GB

Operating System: Windows 11 Pro 64-bit (version 10.0)

After the training, the overall training time was around 10 hours 6 minutes and 58 seconds, with an average MSE loss of 0.0695, which the GPU's computations was around an average of 14.53 seconds per iteration. The runtime log in the terminal as shown in Figure E.6, wherein it displayed the epoch, step and loss every 10 steps.

Figure E.6 Model Training Log

After the model training is finished, the fine-tuned model is saved in the ModelV2 folder. As shown in Figure E.7, it provided several components with different parameters such as feature extractor, scheduler, text encoder, tokenizer, unet, vae and model index.

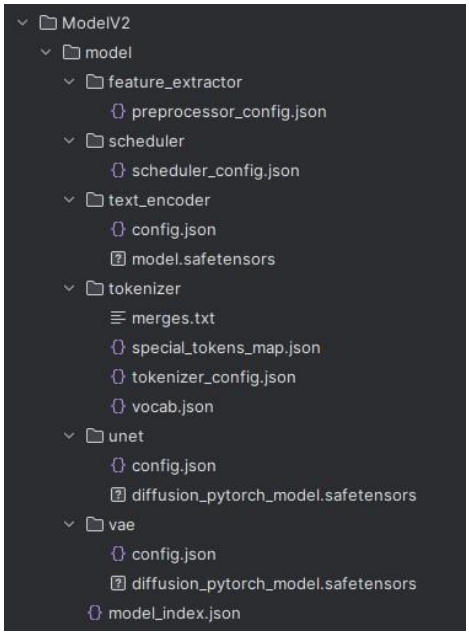

Figure E.7 Fine-tuned model folder directory

E.3 Model Deployment

E.3.1 System Flowchart

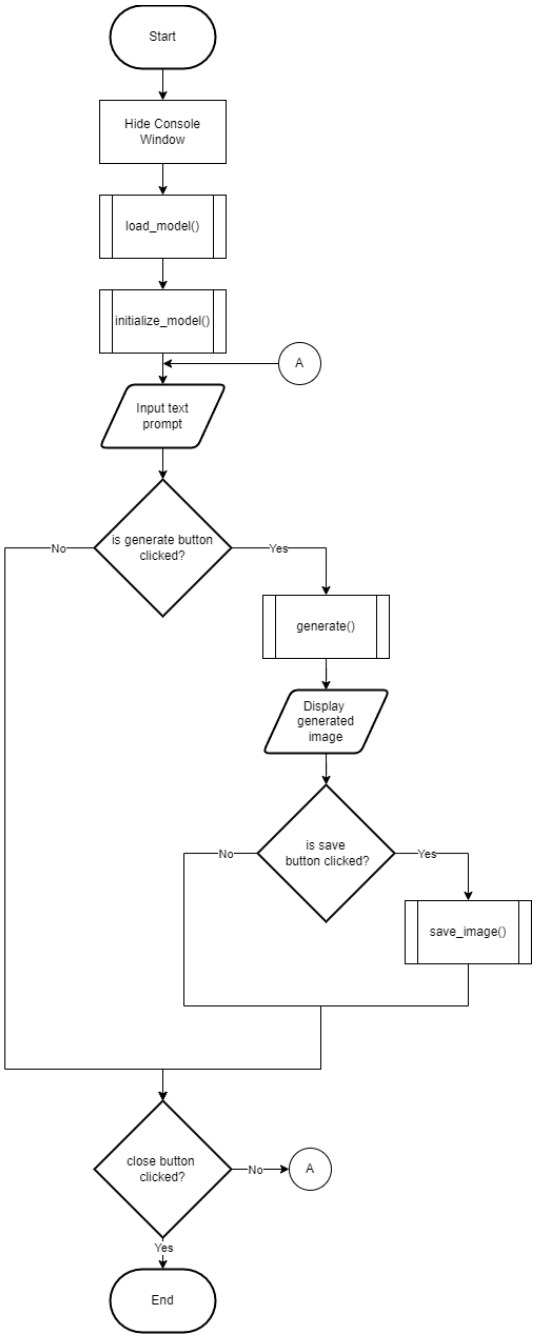

Figure E.8 System Flowchart

The system flowchart, as shown in Figure E.8, is the main flow of the application. Firstly, the console window is hidden from the user. Then, the model is loaded and initialized using the load_model() and initialize_model() function. Afterwards, the user must input a text prompt. When the generate button is clicked, it will proceed to the generate() function wherein the image will be generated using the fine-tuned model. Then, the generated image is displayed in the image frame. Furthermore, the user has the option to save the generated image when the save image button is pressed. Then, it will continue to save_image() function where the user has the option to select the location of where the image will be saved. While the close button of the application is not clicked, the user can still input text prompt and generate images.

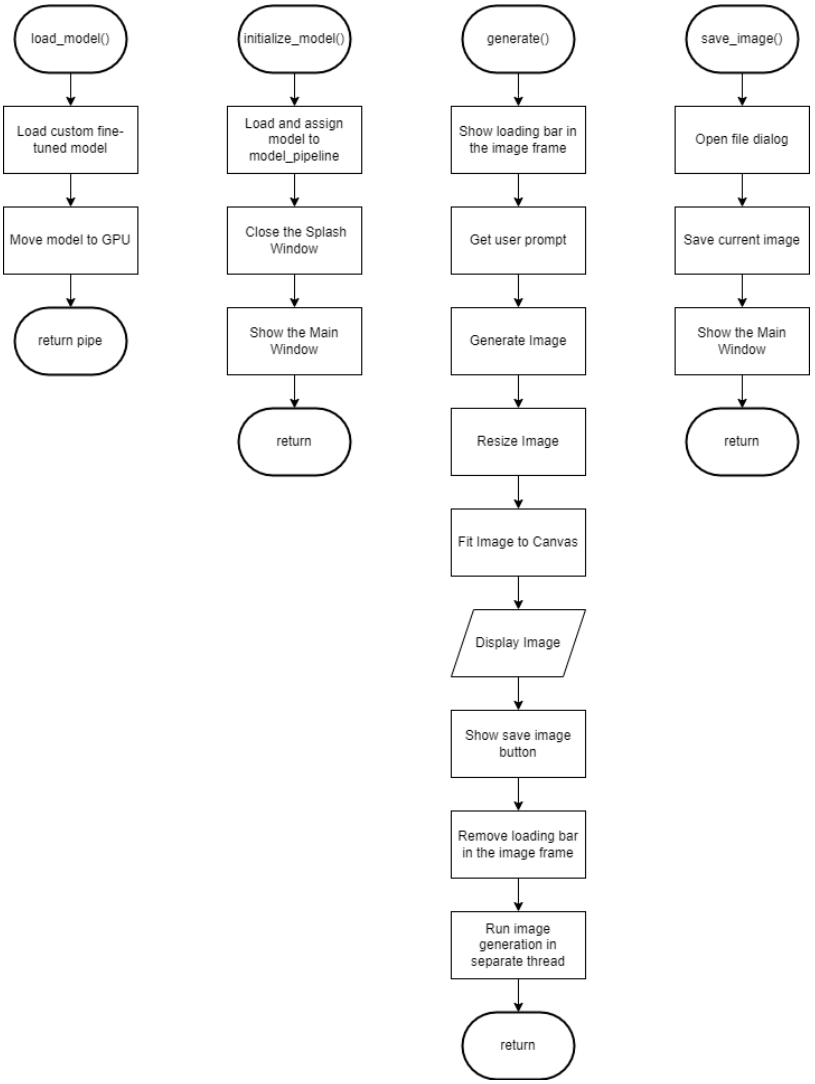

Figure E.9 load_model(), initialize_model(), generate() and save_image() flowchart

In Figure E.9, the functions used by the main application are illustrated. The following definition of each function are the following:

load model() – function that defines the directory path of the fine-tuned model. It also loads the model and specifies the parameters to be used. Then it is moved to the gpu for faster processing when generating images.

initialize_model() – function that loads and assigns the model to model_pipeline. It also defines and declares the model_pipeline variable where the model is stored. It also closes the splash window which is the loading screen window for loading model pipelines. Then the main application window is shown.

generate(): - function that generates the image based on the user's input text prompt. It also resizes the image with dimensions of 5x7 inches and displays it in the image frame. Also while generating the image, it will show a loading bar to inform the user that the application is still in the process of generating the image. After generating the image, the loading bar disappears.

save_image(): - function that allows the user to save the generated image locally in a chosen file directory.

E.3.2 Code

```
import ctypes
import customtkinter
from tkinter import ttk, filedialog
import torch
from diffusers import StableDiffusionXLPipeline
from PIL import Image, ImageTk
import threading
```

The tools and libraries were imported in the CreativeCardV3.py which are essential for training the SDXL model, as shown in the code above.

From ctype's library:

'ctypes' – allows the console window to be hidden.

From customtkinter's library:

'customtkinter' – allows the developer to create buttons and widgets with enhanced appearance and functionality of Tkinter for the application.

From tkinter's library:

'ttk' – allows the developer to use progress bar.

'filedialog' – allows the utilization of dialog for displaying file selection when opening or saving files.

From diffuser's library:

'StableDiffusionXLPipeline' - used for loading the model and its pipeline, which provides methods to generate images using text prompts.

From torch's library:

Torch - used to provide tensor and neural network operations functionalities which is used for model training.

From PIL's library:

'Image' – used for image saving, opening and operations.

'ImageTk' – converts PIL images into format that can be displayed in TKinter widgets.

From threading's library:

'threading' – handles the creation and management of threads in python to enable running of tasks asynchronously.

```
# Hide the console window (Windows only)
if __name__ == "__main__":

ctypes.windll.user32.ShowWindow(ctypes.windll.kernel32.GetConsole
Window(), 0)
```

The code above uses the ctype library to hide the console window. The ShowWindow module's parameter is set to 0, which hides the terminal window. This is useful when the application is packaged and bundled into a standalone application and the user utilizes the application.

```
customtkinter.set_appearance_mode("dark")
customtkinter.set_default_color_theme("dark-blue")

app = customtkinter.CTk()
app.title("Creative Card")
app.geometry("768x1024")
app.resizable(False, False)  # Disable window resizing

# Splash screen with a progress bar for loading the model
splash = customtkinter.CTkToplevel()
splash.title("Creative Card")
splash.geometry("400x200")
splash_label = customtkinter.CTkLabel(splash, text="Loading model
pipeline components, please wait...", font=("Roboto", 14))
splash_label.pack(pady=20)
splash.resizable(False, False)  # Disable window resizing

# Progress bar for loading the model
progress = ttk.Progressbar(splash, mode="indeterminate")
progress.pack(pady=20, padx=20)
progress.start()
```

With the utilization of both customtkinter and tkinter, it allows the creating of the GUI application window. The appearance and theme of the GUI application is set to dark and dark blue to look modern and simple. The app variable is for the main application window and set the resizable() value to false, which disables window resizing of user to maintain the application's default appearance. The splash variable which denotes to the loading screen window is also shown when the application is opened first. It displays the progress bar and a label so that the user is informed that the application is already opened and loading the model pipelines. The loading screen window is also not resizable just like the main application window.

```
def load_model():
    # Load the fine-tuned Stable Diffusion XL model
    fine_tuned_model_dir = r"C:\Users\Jethro\PycharmProjects\OJT
Project\Model"
    pipe = StableDiffusionXLPipeline.from_pretrained(
        fine_tuned_model_dir,
        torch_dtype=torch.float16,
        use_safetensors=True  # Automatically handles loading
```

```
from safetensors files if applicable
    )

    pipe = pipe.to("cuda")  # Use GPU for faster generation
    return pipe
```

The code for the load_model() function is shown above. Firstly, the fine-tuned SDXL model is loaded first by opening the model directory. Then it uses StableDiffusionXLPipeline to load the model and set some configurations. Torch.float16 enables the model to load with a 16-bit floating point precision which reduces memory utilization when generating images. Since the trained model uses .safetensors format, the use_safetensors is set to True to inform the pipeline that it will be handling safetensor files. Then, the model is moved to the GPU's computation platform 'cuda' for faster image generation.

```
def initialize_model():
    global model_pipeline
    model_pipeline = load_model()
    splash.destroy()  # Close the loading screen once the model
is loaded
    app.deiconify()  # Show the main application window
    return
```

Then the initialize_model() function is executed. The model is loaded and initialized to the applications UI from the GPU. Then the splash or the loading screen window is closed when the model is loaded. Also, the main application window is showed.

```
# Load the model in a separate thread to avoid freezing the GUI
thread = threading.Thread(target=initialize_model)
thread.start()
```

To prevent the main GUI application to freeze or unresponsive when an error occurs, the model is loaded in a separate thread. This means that when the model is generating image, the application is still responsive. The developer added this feature since the application becomes not responding when the model is generating an image.

```
def generate():
    # Display the loading bar inside the image frame
    progress = ttk.Progressbar(image_frame, mode="indeterminate")
    progress.place(relx=0.5, rely=0.5, anchor="center")
    progress.start()

    def generate_image():
        prompt = entry.get()

        # Generate the image
        with torch.no_grad():
            generated_image = model_pipeline(prompt).images[0]

        # Resize the image to 5x7 inches at 300 DPI
        dpi = 300
```

```
        target_size_in_inches = (5, 7)
        target_size_in_pixels = (int(target_size_in_inches[0] *
dpi), int(target_size_in_inches[1] * dpi))
        upscaled_image =
generated_image.resize(target_size_in_pixels,
Image.Resampling.LANCZOS)

        # Fit the image to the canvas while maintaining aspect
ratio
        frame_width = image_frame.winfo_width()
        frame_height = image_frame.winfo_height()
        image_aspect = upscaled_image.width /
upscaled_image.height
        frame_aspect = frame_width / frame_height

        if image_aspect > frame_aspect:
            new_width = frame_width
            new_height = int(frame_width / image_aspect)
        else:
            new_height = frame_height
            new_width = int(frame_height * image_aspect)

        resized_image = upscaled_image.resize((new_width,
new_height), Image.Resampling.LANCZOS)
        global photo
        photo = ImageTk.PhotoImage(resized_image)

        # Clear the canvas and display the resized image
        canvas.delete("all")
        canvas.create_image(frame_width // 2, frame_height // 2,
anchor="center", image=photo)
        canvas.config(scrollregion=canvas.bbox("all"))

        # Show the Save Image button
        save_button.grid(row=2, column=1, padx=10, pady=10)

        # Stop and remove the loading bar after the image is
generated
        progress.stop()
        progress.place_forget()


    # Run the image generation in a separate thread
    threading.Thread(target=generate_image).start()
```

This section of the code is responsible for generating the image itself. Firstly, the progress bar is created to inform the user that the application is in process of generating image. Afterwards, the text prompt from the user is obtained and saved to the prompt variable. Then the gradient calculation is disabled since it is not need for model inference. Then the application uses the model_pipeline for image generation based on the user's text prompt. It only shows a list of images but only one image is displayed at the end of the function. Then the image is resized to 1500x2000 pixel resolution, which is the same as a 5 x 7" inches with 300 dpi (dots per inch) image. Also, the image is upscaled using LANCZOS resampling filter to maintain high-quality resizing. Then the generated image is fitted to the frame by resizing the image to fit

into the screen without downscaling the image. Then the image is centered and displayed in the image frame. Also, the save image button is enabled and visible to the user. Furthermore, the loading bar is hidden and a separate thread is created for the next image generation of the model.

```
def save_image():
    # Open a file dialog to save the image
    file_path = filedialog.asksaveasfilename(
        defaultextension=".png",
        filetypes=[("PNG files", "*.png"), ("All files", "*.*")]
    )
    if file_path:
        # Save the current image
        global upscaled_image
        upscaled_image.save(file_path)
```

In the save_image() function, the file dialog is opened and asks the user to save the image in a specific folder or directory. If the user selects a file path for the image to be saved, the upscaled image file will be saved in that folder with a PNG file extension.

```
frame = customtkinter.CTkFrame(master=app)
frame.pack(pady=20, padx=60, fill="both", expand=True)

label = customtkinter.CTkLabel(master=frame, text="Creative Card
Cover Image Generator",
                               font=("Roboto", 24))
label.grid(row=0, column=0, columnspan=2, pady=12, padx=10)

entry = customtkinter.CTkEntry(master=frame, height=30,
width=768,
                               placeholder_text="Enter Prompt:
Ex. A Halloween greeting card with three scary "
                                                "pumpkins in a
dark background")
entry.grid(row=1, column=0, columnspan=2, pady=12, padx=10,
sticky="ew")

# Generate button with fixed width
button = customtkinter.CTkButton(master=frame, text="Generate",
command=generate, fg_color="#275da3", width=150)
button.grid(row=2, column=0, pady=10, padx=10)

# Save Image button with fixed width (initially hidden)
save_button = customtkinter.CTkButton(master=frame, text="Save
Image", text_color="#010010", hover_color="#e9edf0",
command=save_image, fg_color="#9fcfe6", width=150)
save_button.grid_forget()  # Hide the button initially

# Frame for displaying the image
image_frame = customtkinter.CTkFrame(master=frame,
border_color="#e9edf0", border_width=2)
image_frame.grid(row=3, column=0, columnspan=2, pady=10, padx=10,
```

```
sticky="nsew")

# Canvas widget for displaying the image
canvas = customtkinter.CTkCanvas(master=image_frame,
bg="#4b5355")
canvas.pack(fill="both", expand=True)

# Configure grid to expand properly
frame.grid_rowconfigure(3, weight=1)
frame.grid_columnconfigure(0, weight=1)
frame.grid_columnconfigure(1, weight=1)

app.withdraw()  # Hide the main window while the model is loading
app.mainloop()
```

This section of code is responsible for some of the widgets and buttons used in the application. The main frame is created using CTkFrame which is where the widgets will be placed. Then the title for the application "Creative Card Cover Image Generator" is saved in the label variable. The textbox on where the user inputs the text prompt is placed in the entry variable. There is also a placeholder text to inform the user an example of a text prompt. Then the button variable is responsible for the generation of image and will proceed to generate() function when clicked. Also, the save image button allows the user to save the image when clicked. However, it will only be displayed if an image is generated since the user can't save any image file if there is no image displayed. Then the image frame is placed in the canvas widget and is created where the generated image is displayed. Afterwards, the Grid layout manager is configured so that certain rows and columns can expand. This ensures that the application maintains a sleek and well-designed user interface.

E.3.3 Graphical User Interface

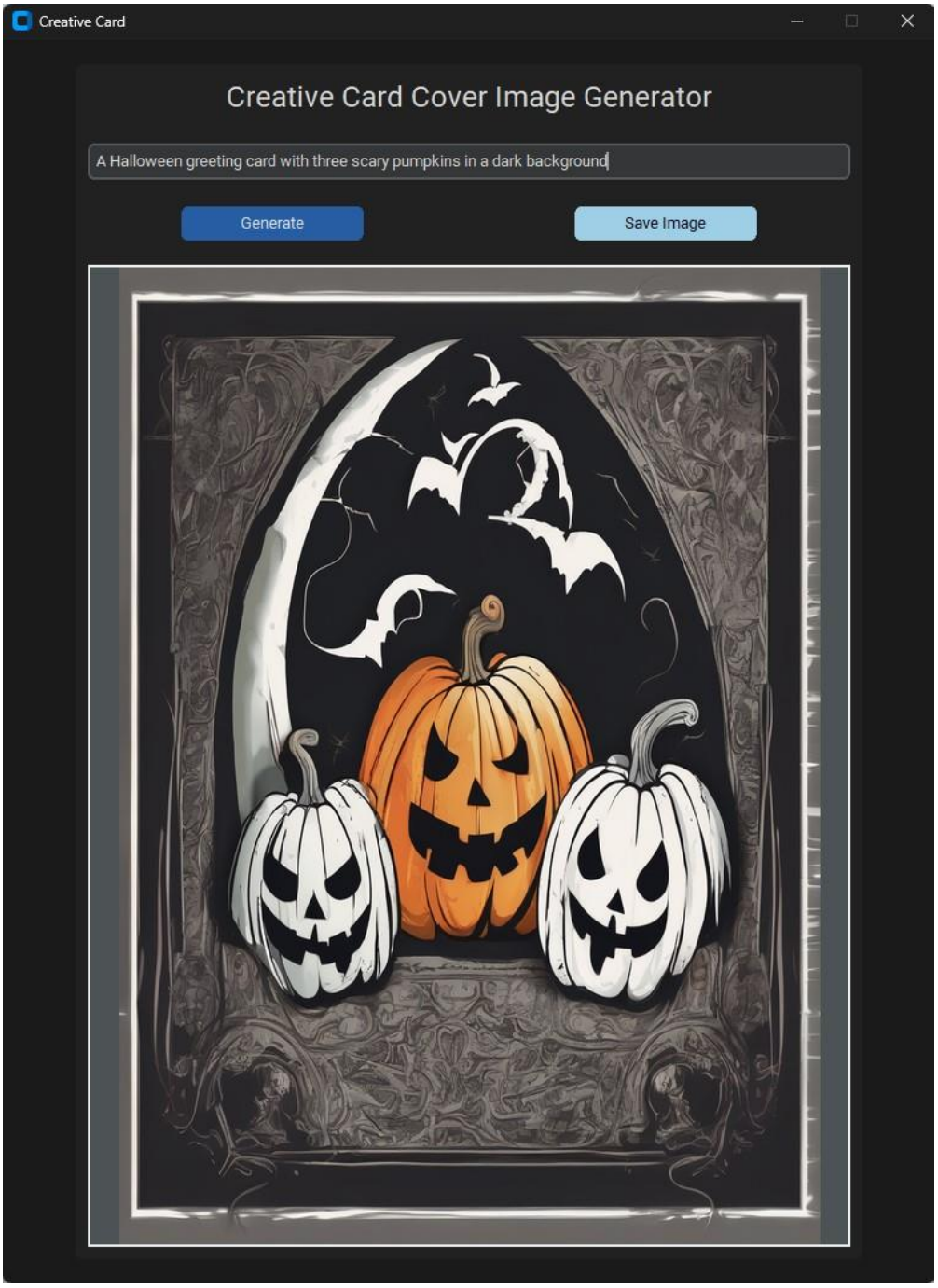

Figure E.10 CreativeCard Main GUI

The graphical user interface of the main window of the application is shown in Figure 10. The project title "Creative Card Cover Image Generator" is shown at the top of the window. Also, the text field is where the user can enter the text prompt for image generation. The generate and save image buttons are placed below the text box field. Note that the save image button will only appear if there is an image in the

frame which is below the two buttons. The GUI of the main application gives a modern look and sleek design with dark-blue, black and light-blue themes.

E.3.4 Model Inference

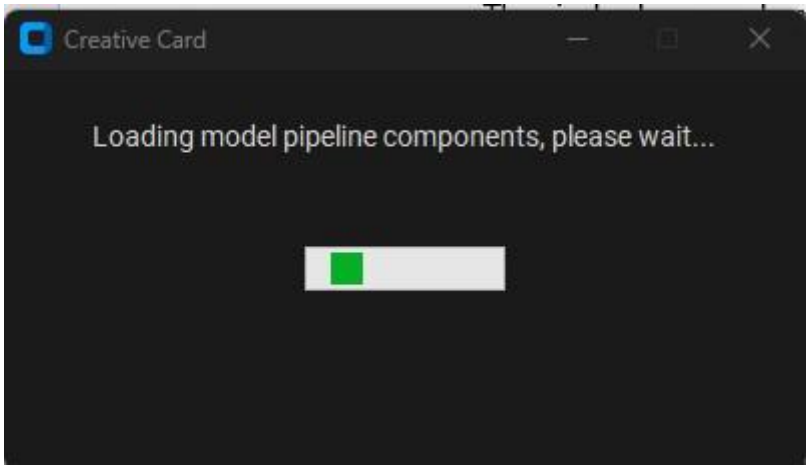

Figure E.11 Loading Screen Window

For model inference, the application was opened using Pycharm IDE by running the CreativeCardV3.py file. Afterwards, the loading screen window, shown in Figure E.11, is displayed with a progress bar while the model pipelines are loading. Then, the loading screen window disappears, and the main application window is displayed, as shown in Figure E.12.

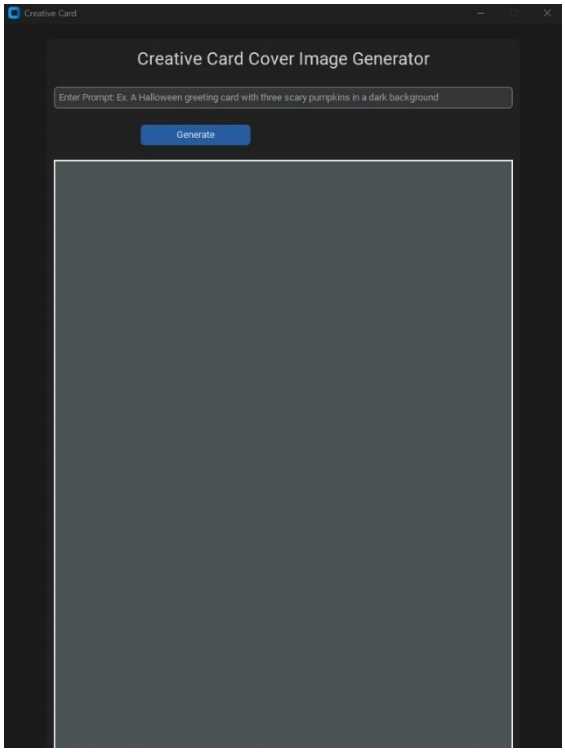

Figure E.12 CreativeCard Main Application Window

Then a text prompt "A Halloween greeting card with three scary pumpkins in a dark background" was used to test the model and the generate button is clicked, shown in Figure E.13. The loading bar is also displayed while the model is generating. Afterwards, the generated image is displayed in the image frame as shown in Figure E.14. The terminal log is also seen in the terminal window in Figure E.15, which shows that the generated image took 37 seconds to generate.

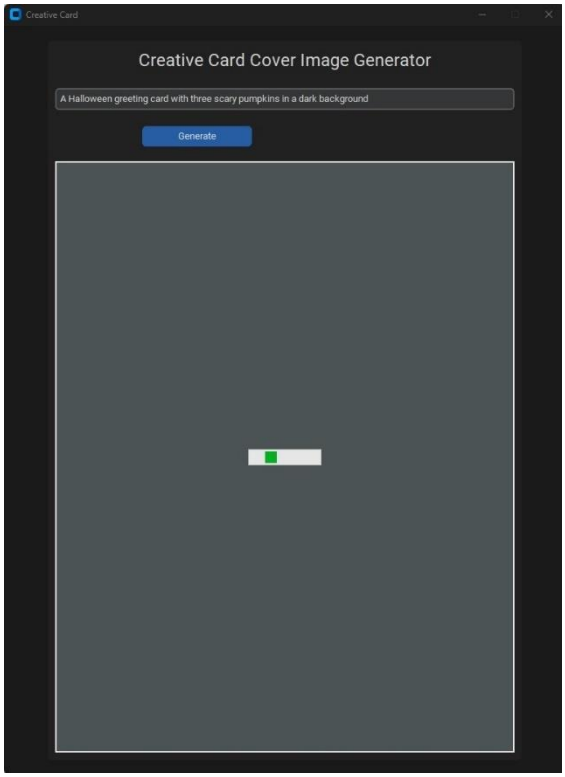

Figure E.23 Application in the process of generating image based on text prompt

Figure E.14 Generated image seen in the image frame

```
D:\anaconda\envs\machinelearning\python.exe "C:\Users\Jethro\PycharmProjects\OJT Project\CreativeCardV3.py"
Loading pipeline components...: 100%|██████████| 7/7 [00:34<00:00,  4.88s/it]
100%|██████████| 50/50 [00:37<00:00,  1.35it/s]
```

Figure E.35 Terminal Log for model inference

In Figure E.16 below are the generated images with the same text prompt. The model showed high-quality images as a result and even though each image is different, it showed the same style as the training images used.

Figure E.16 Generated images using the same prompt

E.3.5 Software Packaging

After the model inference, the application is bundled and packaged into a standalone executable file. Using pyinstaller, the user/employee can run the packaged up without installing or downloading Python libraries or modules. Here are the following procedures that was done by the developer:

1. Opened the folder where the python file to be bundled is located. In the developer's case, it is located at this directory: `C:\Users\Jethro\PycharmProjects\OJT Project`

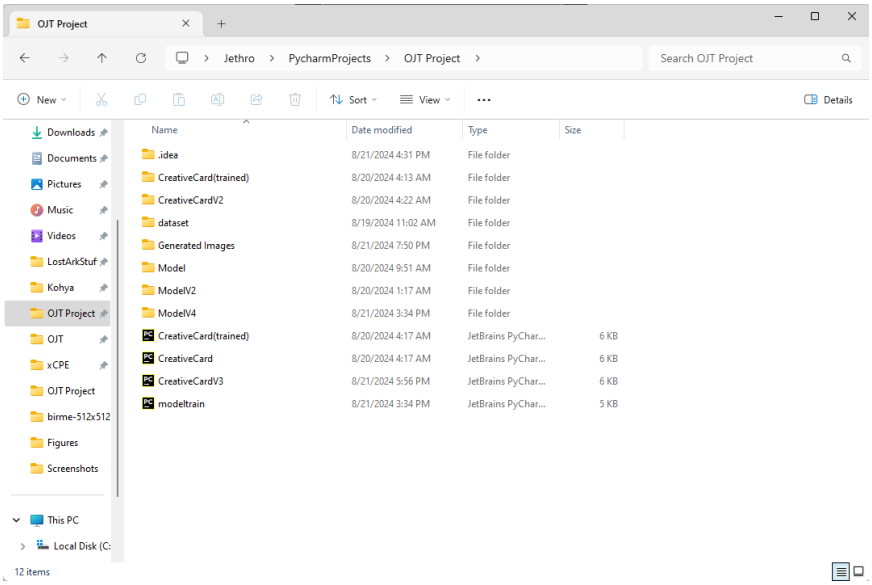

Figure E.4 CreativeCard.py file location

2. Opened command prompt and the file directory where the python file is located using the command:
`cd C:\Users\Jethro\PycharmProjects\OJT Project`

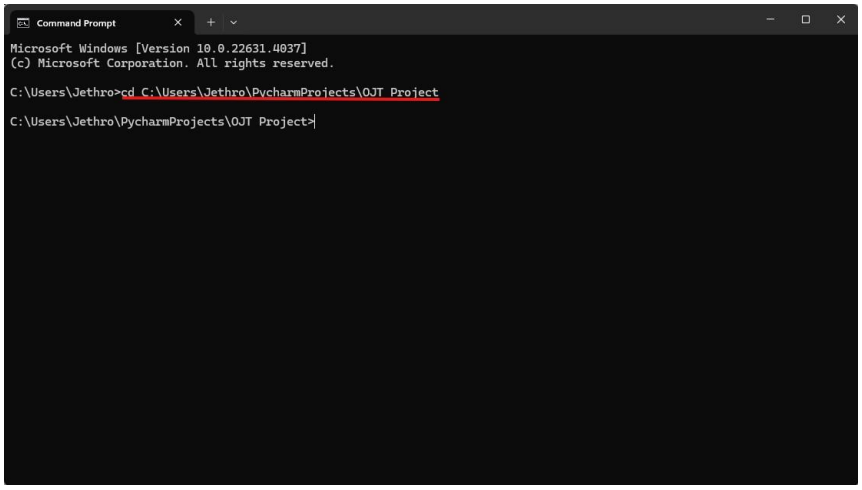

Figure E.18 Command Prompt

3. Installed pyinstaller using the following command: `pip install -U pyinstaller`

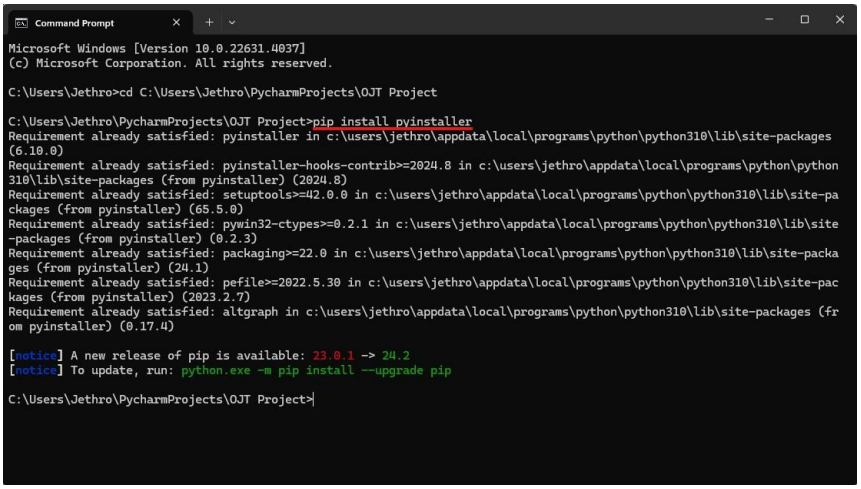

Figure E.19 pyinstaller installation in command prompt

4. Started the application to be packaged using the command: `pyinstaller CreativeCardV3.py`

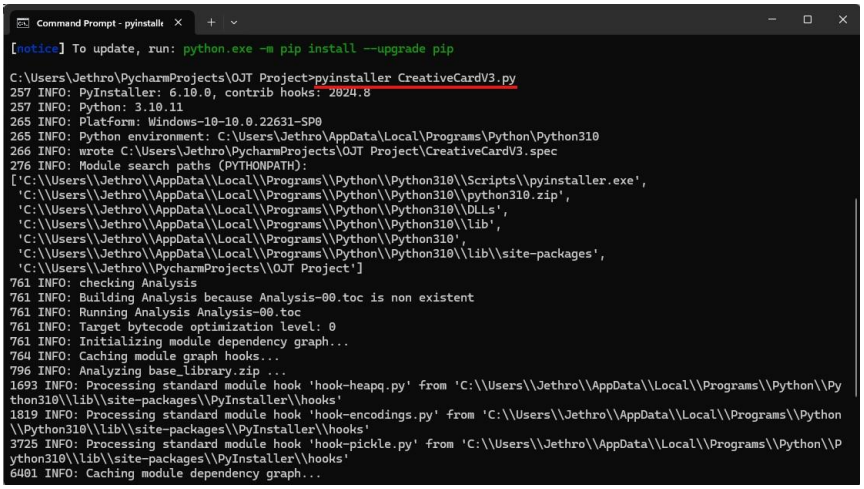

Figure E.20 CreativeCardV3.py software packaging using pyinstaller

5. After packaging the application, the developer added the package contents (folders named *dist* and *build*, file named *CreativeCardV3.spec*) to a zip file together with the user manual.

F System Requirements

F.1 Hardware Requirements

F.1.1 Processor

Processor Core Count	: 4 cores or above

F.1.2 Memory

Memory Capacity	: 8 GB or above

F.1.3 Video Card

VRAM Capacity	: 8 GB or above

F.1.4 Storage

Disk Size	: 5 GB

F.2 Software Requirements

F.2.1 Operating System

Operating System Name	: Windows 10
Architecture Type	: 64-bit
Edition	: Windows 10 Home Edition

G User Manual

G.1 Opening the CreativeCard application

1. Navigate to the CreativeCard zip file that you downloaded.

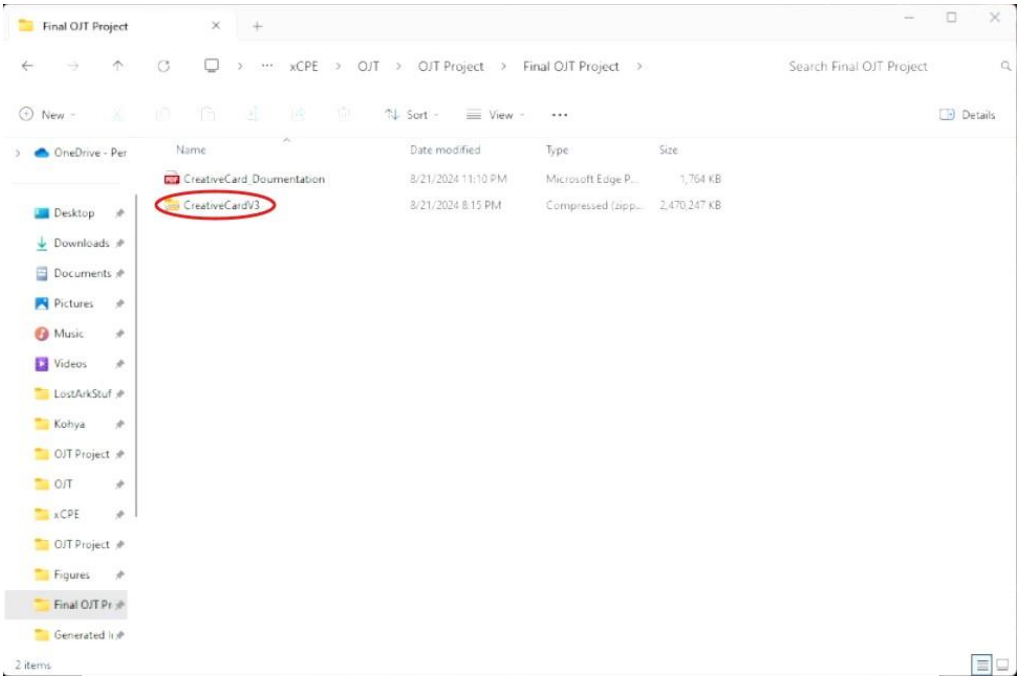

Figure G.21 Navigate CreativeCard

2. Right click the zip file and choose *Extract All*.

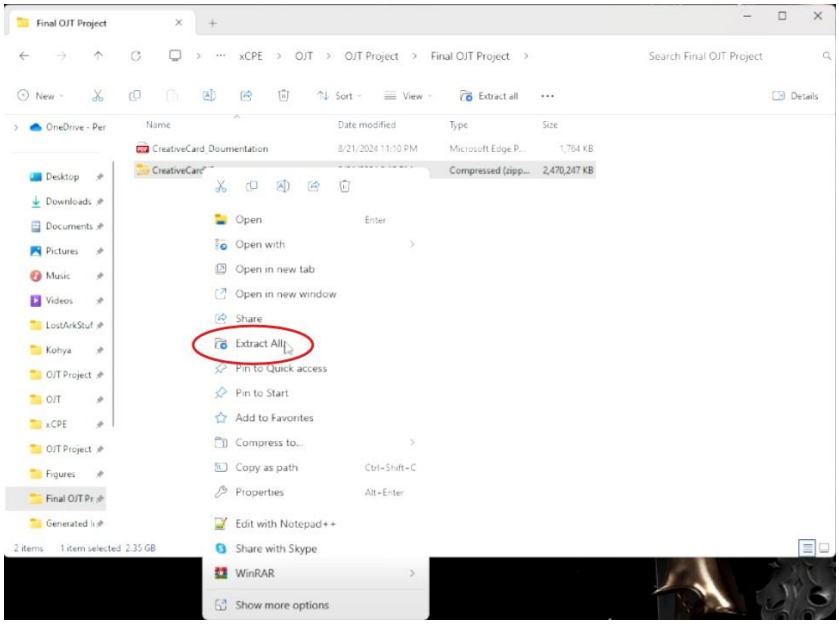

Figure G.22 Open zip file

3. Choose the location you want to extract the file then click *Extract*.

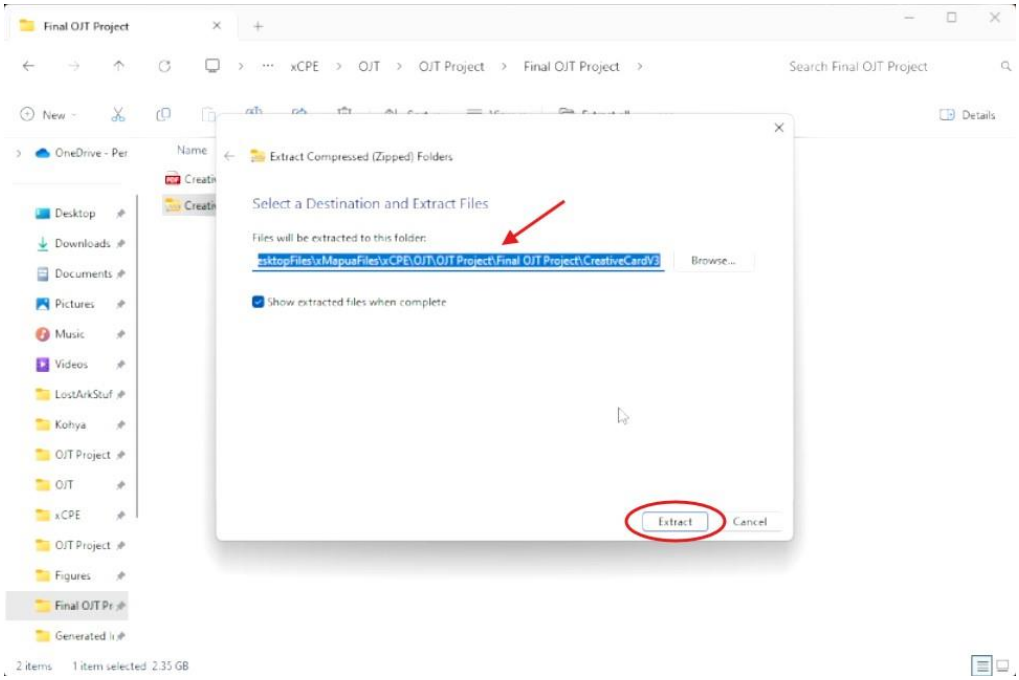

Figure G.23 Extract file

4. Open the extracted folder of CreativeCard app. Navigate to *dist* > *CreativeCardV3*.

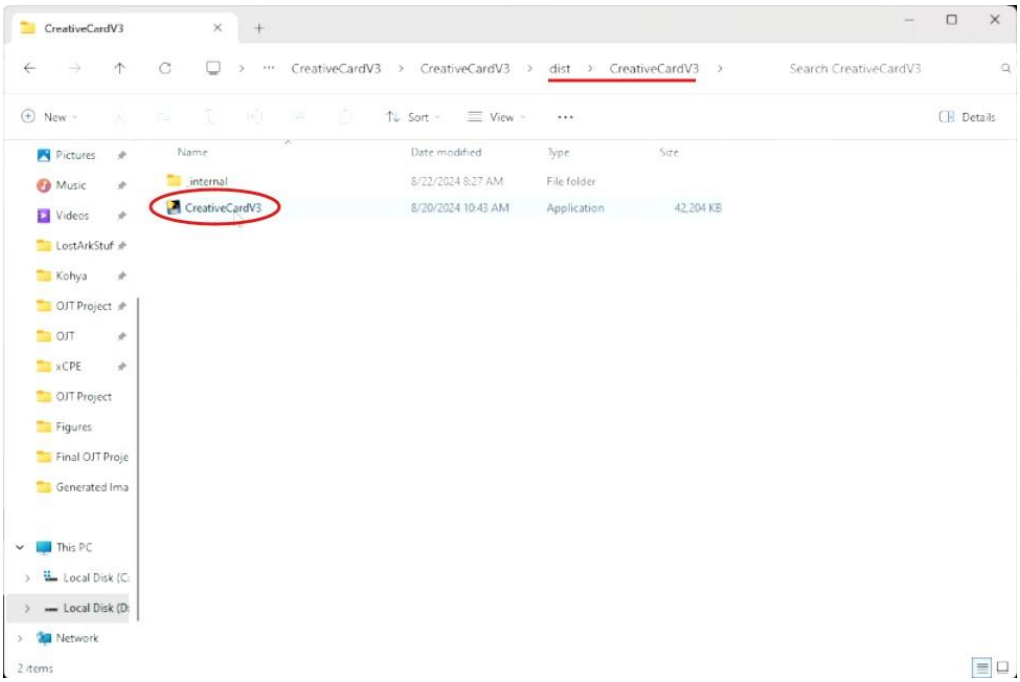

Figure G.24 Open extracted folder

5. Right click **CreativeCardV3.exe** file and click **Open**.

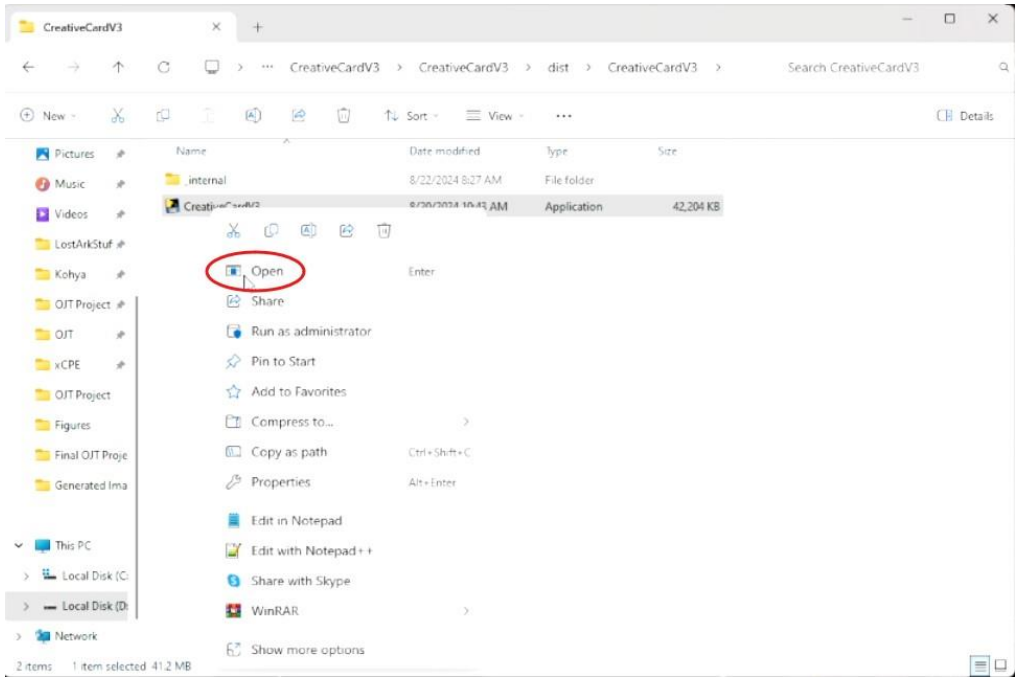

Figure G.25 Open CreativeCard.exe

G.2 Generating card cover image

1. Open **CreativeCardV3.exe** file.

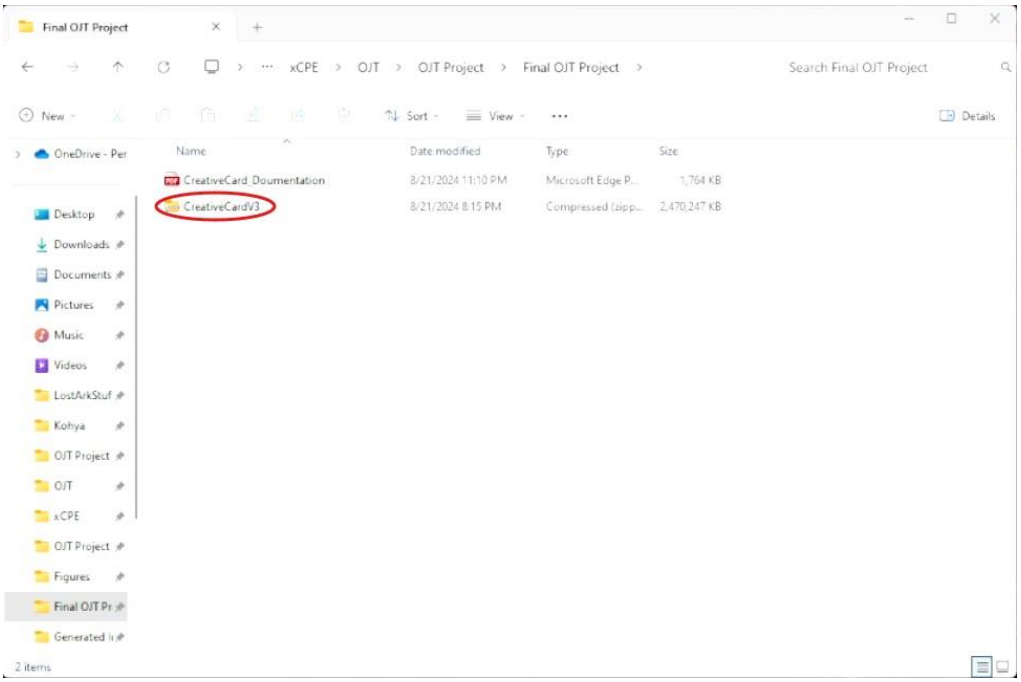

Figure G.26 Open file

2. Wait for the loading screen to finish and minimize the terminal window it is not hidden.

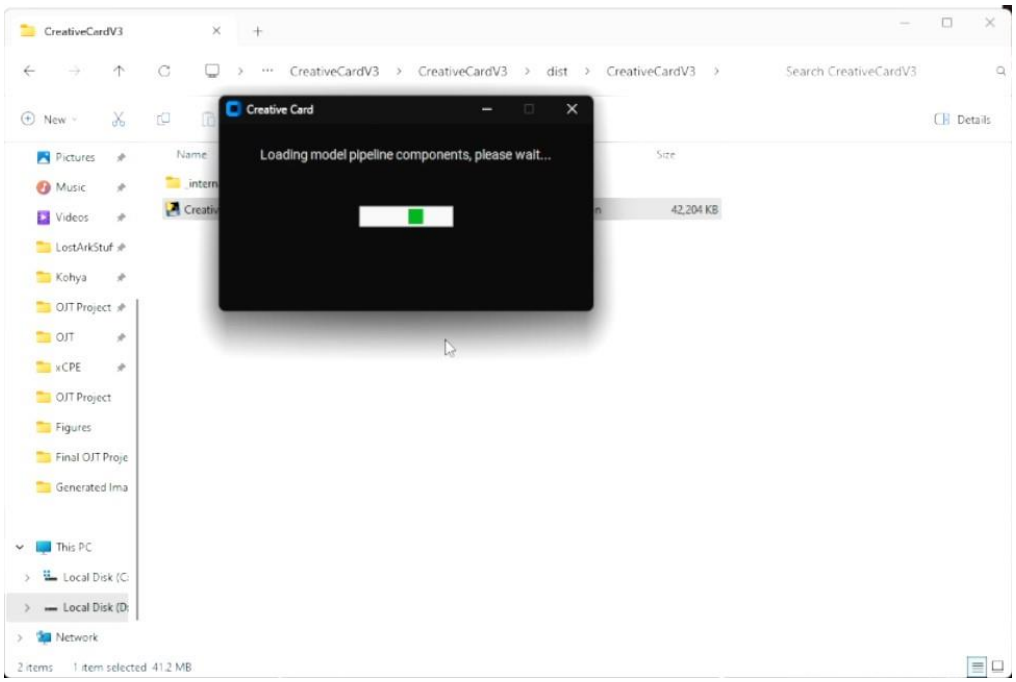

Figure G.27 Wait Loading screen

3. When the loading is finished, the main application window will open.

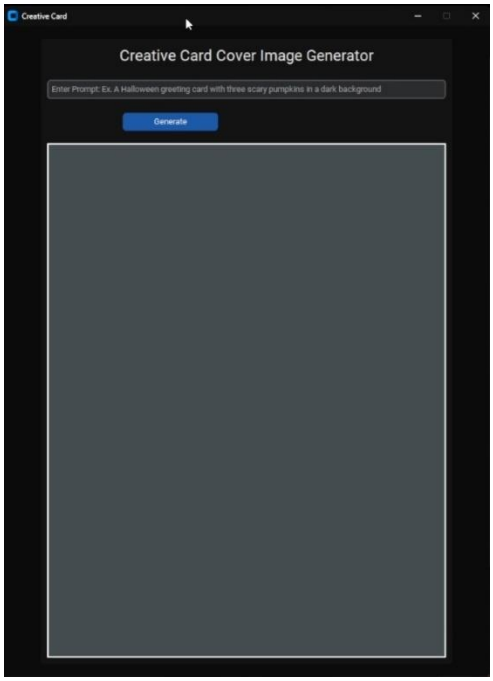

Figure G.28 CreativeCard App

4. Input a text prompt describing the card design that you want in the image. The more detailed the description is, the better.

Figure G.29 Input text prompt

5. Click on the **Generate** button.

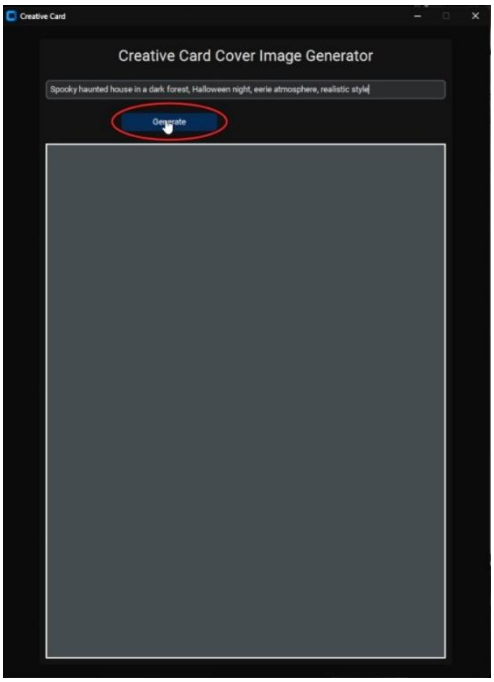

Figure G.30 Click Generate button

6. The application will start generating the image based on your text prompt. The loading bar also appears in the rectangular frame below the **Generate** button. The image generation is dependent on your computer specifications as it will use your computer's video card or GPU (Graphics Processing Unit).

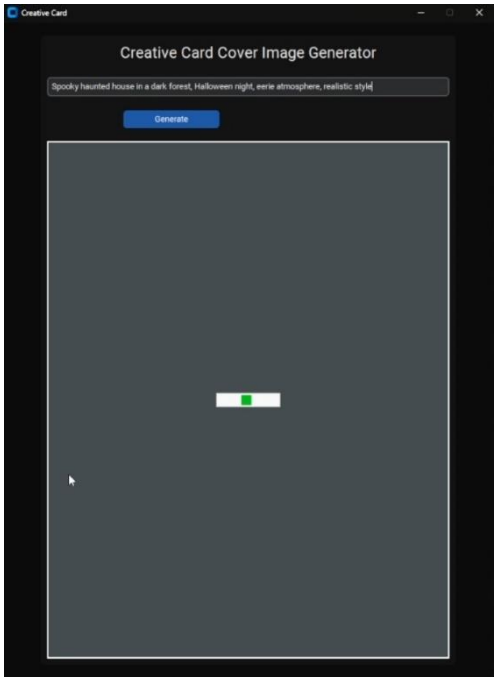

Figure G.31 App generating image

7. If the image generation is taking too long, open the terminal window to see the progress of image generation of the model.

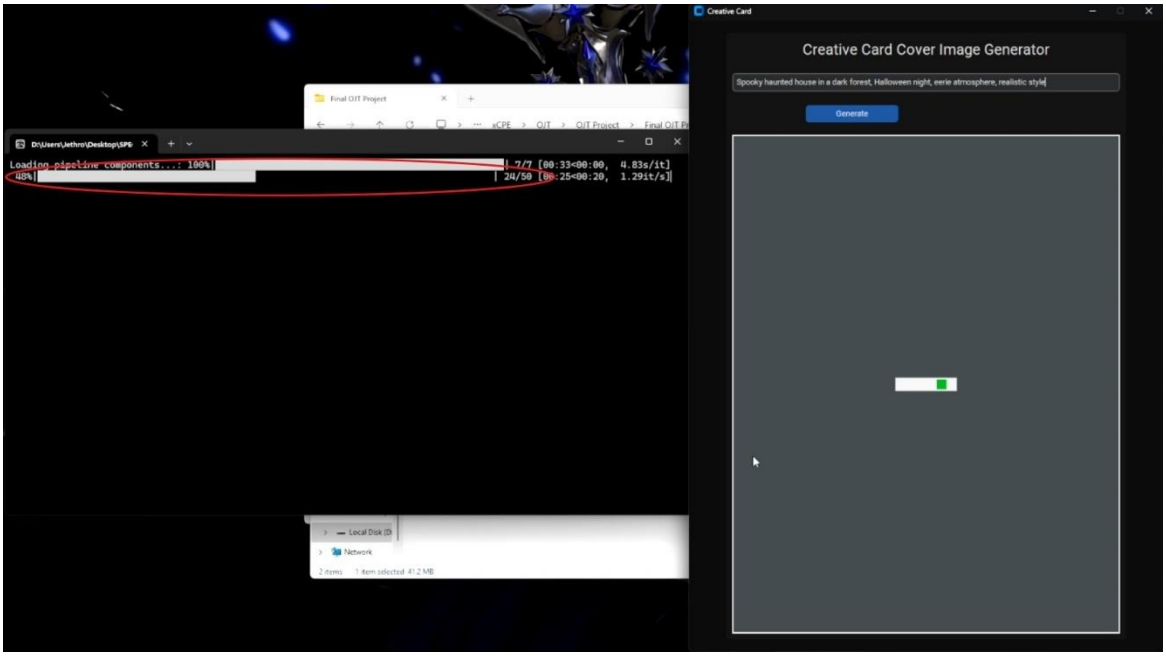

Figure G.32 Opening Terminal Window

8. When the image generation is finished, the image will be displayed in the frame. If you prefer other designs, just click the **Generate button** again to start generating another image. If you want to use another text prompt, go to Step 4.

G.3 Saving the generated image

1. In the application window, click the **Save Image** button.

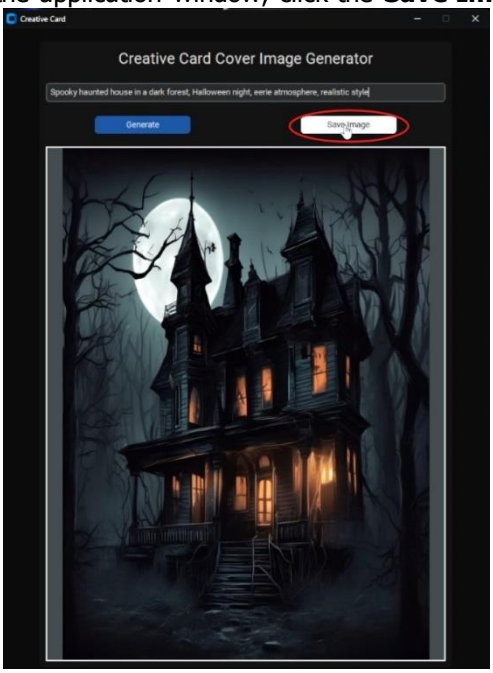

Figure G.33 Click save image

2. Navigate to the folder you wish to save the image, then click **Save**.

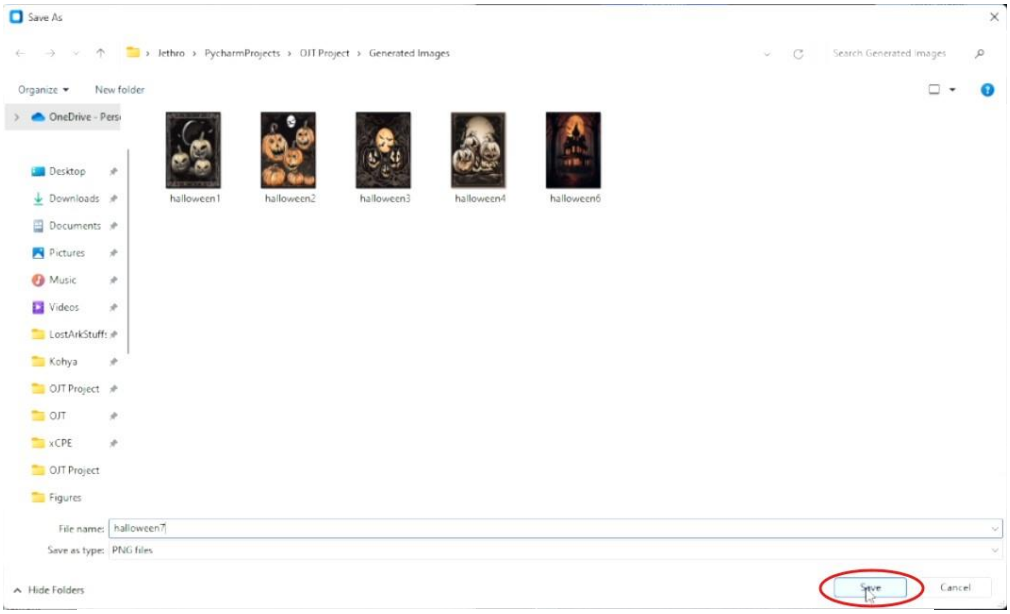

Figure G.34 Navigate file directory and click save

G.4 Tips to insert proper text prompts

1. Describe the image in detail.

   Example of detailed text prompts:

   a. Spooky haunted house in a dark forest, Halloween night, eerie atmosphere, realistic style.
   b. Realistic image of a witch flying on a broomstick across a full moon, Halloween night.
   c. Realistic image of a ghostly figure in an old, abandoned mansion, Halloween night.
   d. Realistic image of a werewolf howling at the moon, dark forest, Halloween night.

2. Add depth to the image by describing the clothing, color, texture, perspective of the subject.

3. Add and describe the description of the environment your subject. Describe the landscape, time of day, architecture, terrain, natural elements, weather or primary setting of the image.

4. Add an image style. Examples of image styles are: Realistic, Cartoonish, Photographic, Fantasy Art, Pixel Art, Watercolor etc.

H References

H.1 Reference # 1

Reference Title : Stable Diffusion XL

Reference Author : Stability AI

Reference Page Source : https://huggingface.co/stabilityai/stable-diffusion-xl-base-1.0

H.2 Reference # 2

Reference Title : Diffusers python library

Reference Author : Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, Berman, Yiyi Xu, Steven Liu and Thomas Wolf

Reference Page Source : https://github.com/huggingface/diffusers

H.3 Reference # 3

Reference Title : Transformers python library

Reference Author : Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest and Alexander M. Rush

Reference Page Source : https://pypi.org/project/transformers/

H.4 Reference # 4

Reference Title : Pytorch

Reference Author : Soumith Chintala

Reference Page Source : https://pypi.org/project/torch/

H.5 Reference # 5

Reference Title : Python Imaging Library (Fork)

Reference Author : Jeffrey A. Clark and contributors

Reference Page Source : https://pypi.org/project/pillow/

H.6 Reference # 6

Reference Title : OS python library

Reference Author : Python

Reference Page Source : https://docs.python.org/3/library/os.html

H.7 Reference # 7

Reference Title : Accelerate python library

Reference Author : Hugging Face Team

Reference Page Source : https://pypi.org/project/accelerate/

H.8 Reference # 8

Reference Title : tkinter python library

Reference Author : Fredrik Lundh and John Ousterhout

Reference Page Source : https://docs.python.org/3/library/tkinter.html

H.9 Reference # 9

Reference Title : customtkinter python library

Reference Author : Tom Schimansky

Reference Page Source : https://pypi.org/project/customtkinter/0.3/

H.10 Reference # 10

Reference Title : ctypes python library

Reference Author : Thomas Heller

Reference Page Source : https://docs.python.org/3/library/ctypes.html

H.11 Reference # 11

Reference Title : threading python library

Reference Author : Guido van Rossum

Reference Page Source : https://docs.python.org/3/library/threading.html

I Appendices

I.1 modeltrain.py

```
from diffusers import StableDiffusionXLPipeline, DDPMScheduler
from transformers import CLIPTokenizer
import torch
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms as Trans
from accelerate import Accelerator
from PIL import Image
import os


# Define the custom dataset class
class GreetingCardDataset(Dataset):
    def __init__(self, image_directory, prompts_file, transforms=None):
        self.image_directory = image_directory
        self.transforms = transforms
        with open(prompts_file, "r") as file:
            self.prompts = file.readlines()
            # Sort the images considering the format "image (1).jpg"
            self.image_files = sorted(
                os.listdir(image_directory),
                key=lambda x: int(x.split("(")[1].split(")")[0])
            )

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, index):
        image_path = os.path.join(self.image_directory,
self.image_files[index])
        img = Image.open(image_path).convert("RGB")
        if self.transforms:
            img = self.transforms(img)
        prompt = self.prompts[index].strip()
        return prompt, img


# Define transformations for the images
image_transforms = Trans.Compose([
    Trans.Resize((1024, 1024)),
    Trans.ToTensor(),
    Trans.Normalize([0.5]*3, [0.5]*3),
])

# Load dataset
dataset = GreetingCardDataset(
    image_directory=r"C:\Users\Jethro\PycharmProjects\OJT
Project\dataset\images",
    prompts_file=r"C:\Users\Jethro\PycharmProjects\OJT
Project\dataset\prompts.txt",
    transforms=image_transforms
)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

```
# Load the pre-trained Stable Diffusion XL model and pipeline
pipeline = StableDiffusionXLPipeline.from_pretrained("stabilityai/stable-
diffusion-xl-base-1.0")
pipeline.to("cuda")  # Move to GPU

# Prepare optimizer and scheduler
optimizer = torch.optim.AdamW(pipeline.unet.parameters(), lr=1e-5)
scheduler = DDPMScheduler(num_train_timesteps=1000)

# Initialize the accelerator
accelerator = Accelerator(mixed_precision="fp16")

# Tokenizer (Moved out of the training loop to avoid re-initialization)
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-base-patch32")

# Training loop
for epoch in range(5):
    for step, (text_prompt, image_data) in enumerate(dataloader):
        # Tokenize the text prompt
        text_inputs = tokenizer(text_prompt, return_tensors="pt",
padding=True, truncation=True).input_ids.to("cuda")

        # Convert text inputs back to list of strings for the pipeline
        text_prompt_list = [tokenizer.decode(ids, skip_special_tokens=True)
for ids in text_inputs]

        # Forward pass through the model
        with torch.cuda.amp.autocast():
            # Generate image based on prompts and input image
            output = pipeline(prompt=text_prompt_list, image=image_data)
            # Access the generated image
            generated_image = output.images[0]  # PIL Image

        # Convert PIL Image to tensor
        generated_image_tensor =
Trans.ToTensor()(generated_image).unsqueeze(0).to("cuda")
        image_data = image_data.to("cuda")

        # Resize generated image tensor to match input image size
        if generated_image_tensor.size() != image_data.size():
            generated_image_tensor =
Trans.Resize(image_data.size()[2:])(generated_image_tensor)

        # Clamp tensor values to [0, 1] before scaling
        generated_image_tensor = torch.clamp(generated_image_tensor, 0.0,
1.0)
        image_data = torch.clamp(image_data, 0.0, 1.0)

        # Ensure gradients are enabled for the model parameters
        for param in pipeline.unet.parameters():
            param.requires_grad = True

        # Ensure the tensors used for loss calculation require gradients
        generated_image_tensor.requires_grad = True
        image_data.requires_grad = True
```

```
        # Compute the loss between the generated image and the original image
        loss = torch.nn.functional.mse_loss(generated_image_tensor,
image_data)

        # Backpropagation
        accelerator.backward(loss)
        optimizer.step()
        optimizer.zero_grad()

        # Print progress
        if step % 10 == 0:
            print(f"Epoch {epoch}, Step {step}, Loss: {loss.item()}")

# Save the fine-tuned model
pipeline.save_pretrained(r"C:\Users\Jethro\PycharmProjects\OJT
Project\ModelV4")
```

I.2 CreativeCardV3.py

```
import ctypes
import customtkinter
from tkinter import ttk, filedialog
import torch
from diffusers import StableDiffusionXLPipeline
from PIL import Image, ImageTk
import threading

# Hide the console window (Windows only)
if __name__ == "__main__":

ctypes.windll.user32.ShowWindow(ctypes.windll.kernel32.GetConsoleWindow(), 0)

customtkinter.set_appearance_mode("dark")
customtkinter.set_default_color_theme("dark-blue")

app = customtkinter.CTk()
app.title("Creative Card")
app.geometry("768x1024")
app.resizable(False, False)  # Disable window resizing

# Splash screen with a progress bar for loading the model
splash = customtkinter.CTkToplevel()
splash.title("Creative Card")
splash.geometry("400x200")
splash_label = customtkinter.CTkLabel(splash, text="Loading model pipeline
components, please wait...", font=("Roboto", 14))
splash_label.pack(pady=20)
splash.resizable(False, False)  # Disable window resizing

# Progress bar for loading the model
progress = ttk.Progressbar(splash, mode="indeterminate")
progress.pack(pady=20, padx=20)
progress.start()

def load_model():
    # Load the fine-tuned Stable Diffusion XL model
```

```
    fine_tuned_model_dir = r"C:\Users\Jethro\PycharmProjects\OJT
Project\Model"
    pipe = StableDiffusionXLPipeline.from_pretrained(
        fine_tuned_model_dir,
        torch_dtype=torch.float16,
        use_safetensors=True  # Automatically handles loading from
safetensors files if applicable
    )

    pipe = pipe.to("cuda")  # Use GPU for faster generation
    return pipe

def initialize_model():
    global model_pipeline
    model_pipeline = load_model()
    splash.destroy()  # Close the loading screen once the model is loaded
    app.deiconify()  # Show the main application window
    return

# Load the model in a separate thread to avoid freezing the GUI
thread = threading.Thread(target=initialize_model)
thread.start()

def generate():
    # Display the loading bar inside the image frame
    progress = ttk.Progressbar(image_frame, mode="indeterminate")
    progress.place(relx=0.5, rely=0.5, anchor="center")
    progress.start()

    def generate_image():
        prompt = entry.get()

        # Generate the image
        with torch.no_grad():
            generated_image = model_pipeline(prompt).images[0]

        # Resize the image to 5x7 inches at 300 DPI
        dpi = 300
        target_size_in_inches = (5, 7)
        target_size_in_pixels = (int(target_size_in_inches[0] * dpi),
int(target_size_in_inches[1] * dpi))
        upscaled_image = generated_image.resize(target_size_in_pixels,
Image.Resampling.LANCZOS)

        # Fit the image to the canvas while maintaining aspect ratio
        frame_width = image_frame.winfo_width()
        frame_height = image_frame.winfo_height()

        image_aspect = upscaled_image.width / upscaled_image.height
        frame_aspect = frame_width / frame_height

        if image_aspect > frame_aspect:
            new_width = frame_width
            new_height = int(frame_width / image_aspect)
        else:
            new_height = frame_height
            new_width = int(frame_height * image_aspect)
```

```
        resized_image = upscaled_image.resize((new_width, new_height),
Image.Resampling.LANCZOS)
        global photo
        photo = ImageTk.PhotoImage(resized_image)

        # Clear the canvas and display the resized image
        canvas.delete("all")
        canvas.create_image(frame_width // 2, frame_height // 2,
anchor="center", image=photo)
        canvas.config(scrollregion=canvas.bbox("all"))

        # Show the Save Image button
        save_button.grid(row=2, column=1, padx=10, pady=10)

        # Stop and remove the loading bar after the image is generated
        progress.stop()
        progress.place_forget()


    # Run the image generation in a separate thread
    threading.Thread(target=generate_image).start()

def save_image():
    # Open a file dialog to save the image
    file_path = filedialog.asksaveasfilename(
        defaultextension=".png",
        filetypes=[("PNG files", "*.png"), ("All files", "*.*")]
    )
    if file_path:
        # Save the current image
        global upscaled_image
        upscaled_image.save(file_path)

frame = customtkinter.CTkFrame(master=app)
frame.pack(pady=20, padx=60, fill="both", expand=True)

label = customtkinter.CTkLabel(master=frame, text="Creative Card Cover Image
Generator",
                               font=("Roboto", 24))
label.grid(row=0, column=0, columnspan=2, pady=12, padx=10)

entry = customtkinter.CTkEntry(master=frame, height=30, width=768,
                               placeholder_text="Enter Prompt: Ex. A
Halloween greeting card with three scary "
                                                "pumpkins in a dark
background")
entry.grid(row=1, column=0, columnspan=2, pady=12, padx=10, sticky="ew")

# Generate button with fixed width
button = customtkinter.CTkButton(master=frame, text="Generate",
command=generate, fg_color="#275da3", width=150)
button.grid(row=2, column=0, pady=10, padx=10)

# Save Image button with fixed width (initially hidden)
save_button = customtkinter.CTkButton(master=frame, text="Save Image",
text_color="#010010", hover_color="#e9edf0", command=save_image,
```

```
fg_color="#9fcfe6", width=150)
save_button.grid_forget()  # Hide the button initially

# Frame for displaying the image
image_frame = customtkinter.CTkFrame(master=frame, border_color="#e9edf0",
border_width=2)
image_frame.grid(row=3, column=0, columnspan=2, pady=10, padx=10,
sticky="nsew")

# Canvas widget for displaying the image
canvas = customtkinter.CTkCanvas(master=image_frame, bg="#4b5355")
canvas.pack(fill="both", expand=True)

# Configure grid to expand properly
frame.grid_rowconfigure(3, weight=1)
frame.grid_columnconfigure(0, weight=1)
frame.grid_columnconfigure(1, weight=1)

app.withdraw()  # Hide the main window while the model is loading
app.mainloop()
```