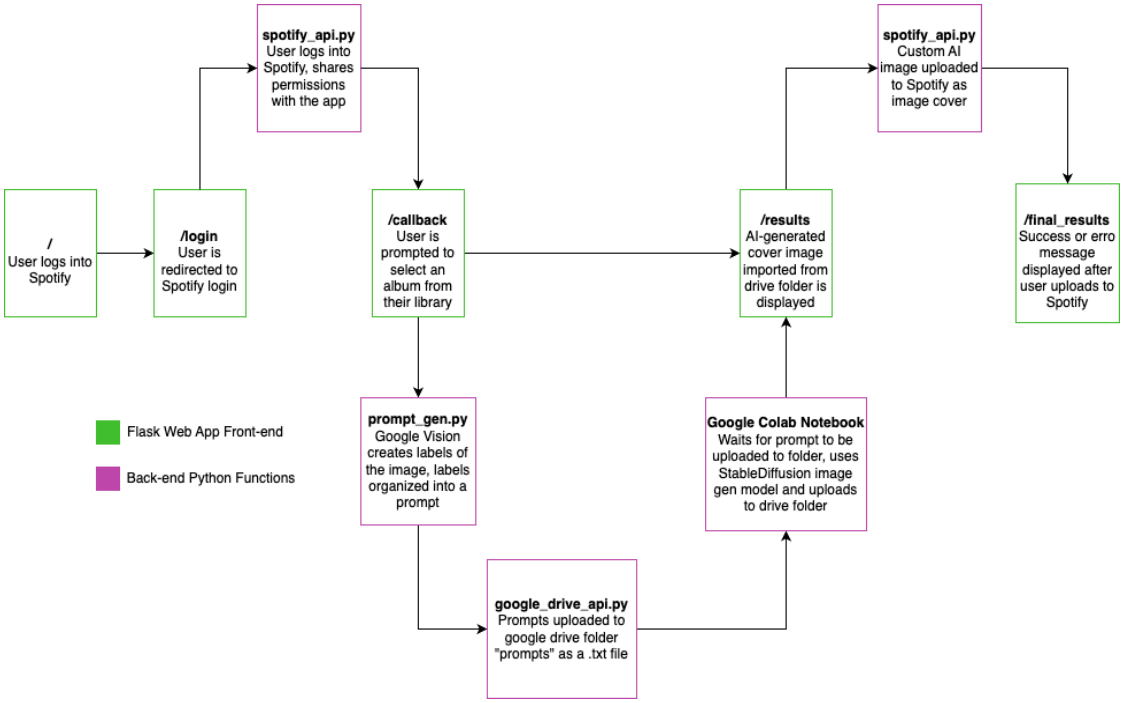
PIC 16A Project Technical Report

Tom Seifert, 6/13/25

1. Project Overview

I created the Spotify Image Generator Project to provide Spotify users with customized images to help stylize their playlists, building their listening aesthetic. Any Spotify user is free to use the web application, as long as they agree to offer permissions for the app to read their playlist and sing data and (only upon user request) upload images to Spotify on their behalf.

If you are interested in using the app, simply open the site at https://playlist-cover-image-gen.uw.r.appspot.com and begin generating images for your playlists! (Note: As of 6/13/25, public access is limited, as the images may only be generated while a Google Colab notebook is running on my local machine, and a finite amount of GPU credits remain—cloud-based image generation to allow for consistent public access coming soon). Here is a flowchart of how my app, built with Flask, is structured:



The user begins at the home page, where they are prompted to log into Spotify. Once pressing “Log in”, the app directs the user to a spotify login page, which is configured to work with the app via Spotify API (see spotify_api.py for backend logic). Once logged in, the user is automatically redirected to a page displaying all of the user’s playlists, with their names and images included. The user is able to select one of the playlists and press “Generate Playlist Cover

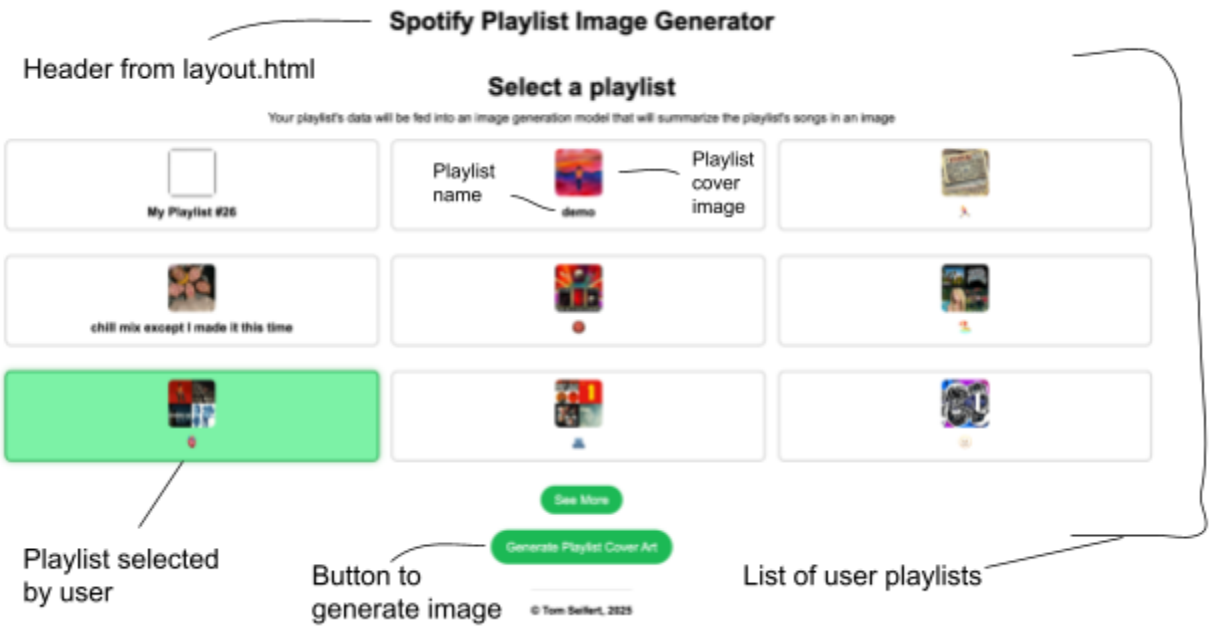
Art" to begin generating an image based on the album covers of the songs within the selected playlist.

Next, a series of backend functions occur while the user waits for the image generation to finish. First, spotify_api.py uses the ID of the selected playlist to pull the song-level data of the playlist from Spotify API. This data is then cleaned, and a list of urls of the songs' album covers is extracted. Second, prompt_gen.py feeds the images into Google Vision's Image Annotator, which creates a list of image labels for each song's album cover. A prompt is then constructed using the resulting labels, with randomness introduced at 2 different points. Images are randomized—5 of the top 10 most frequently occurring album covers are randomly selected—and then labels from the selected images are randomized—5 of the top 10 most frequently occurring labels across all images are chosen as the final labels to pass into the prompt. Here is the template for the prompt: *"An abstract digital album cover featuring {label1}, {label2}, {label3}, {label 4} and {label 5} with a modern aesthetic in a photorealistic style"*

Once this prompt is crafted, google_drive_api.py uploads the prompt, with a unique file name based on the date and time of creation, to a Google Drive Folder called "prompts." Meanwhile, a Google Colab notebook, project.ipynb, is running a loop to continuously check the drive folder for new prompts. When a new prompt is detected, the notebook reads the .txt file and passes the prompt to the public StableDiffusion model via Hugging Face (Link). The model generates an image that the notebook then uploads to the "prompts" folder, named with the date and time signature corresponding to the prompt's file name. The app waits for an image with the correct date and time signature to be uploaded, and once it is found, uploads the image to a temporary folder in google cloud. Finally, the image is read in from the cloud and displayed for the user to see. The user can then choose to press "Upload to Spotify," which triggers spotify_api.py to utilize Spotify API to change the selected playlist's cover image to the AI-generated image

2. Technical Components

Technical Component 1: Flask Routing for Displaying Spotify Playlists



This component allows the user of the Flask web application to select from a list of their playlists to proceed with creating an AI-generated cover image. The user is routed to /callback automatically from the Spotify login page, and the access token returned by the login is used to retrieve playlist names and images to display. This is important because in order to get the image urls for each song in a given playlist, the app needs a playlist ID to pass into the request for this info from Spotify API. After the user logs in and the list of user playlists is extracted, the authorized.html template prompts the user to select a playlist, which will provide the app with the desired playlist ID.

I implemented this component by utilizing Spotify's API documentation, GET requests, decorators, and rendering html templates. The Spotify API supports GET requests for getting an authorization code upon login by the user. This code can then be used to retrieve an access token, which is used in all requests to the API that involve the user's data, which is needed to read in a list of the user's playlists (this is all done within retrieve_playlists). I use flask_session to keep the access token available for the entire session across different routes of the app so that I can extract song-level data once a playlist is selected. Once I read in the list of playlists, I render authorized.html, which displays all of the playlists using Jinja2 syntax with the variables that are passed into the template: playlists_info and user_id. In the template, I used CSS styling to style the page, and Jinja2 syntax to extend the layout.html file to thai file, keeping the website's layout consistent.

Code Snippet:

```
# define callback route - where Spotify authentication page routes to after
successful login (Spotify API docs 15 May 2025)
@app.route("/callback", methods = ["GET", "POST"])
def callback():
    # retrieve authorization code from Spotify API authentication and error if
applicable
    code = request.args.get("code")
    error = request.args.get("error")
    # error handling
    if error:
        return render_template("error.html")
    if not code:
        return render_template("error.html")
    # get access token, a dictionary containing a user's playlist info, and the
user's id - see spotify_api.py
    access_token, playlists_info, user_id = retrieve_playlists(CLIENT_ID,
CLIENT_SECRET, REDIRECT_URI, code)
    # set access_token as a session variable to use in other routes
    session["access_token"] = access_token
    # render page that prompts user to select a playlist - see authorized.html
    return render_template("authorized.html", playlists_info=playlists_info,
user_id=user_id)
```

Technical Component 2: AI-Generated Images

The main purpose of my project is to create AI-generated playlist cover images. All of the information extracted from the Spotify API and provided by the user is aimed at creating the final image. In order for the image to be generated, I had to use Google Colab, as this was the only way that I could access free GPU support needed to run the AI model. The Colab notebook imports the free, open-source StableDiffusion model via HuggingFace to use for image generation. The notebook then utilizes Google Drive to connect to the "prompts" Google Drive folder, where the Flask app uploads the prompts as .txt files.

The core component of the notebook is a continuous while loop that checks the "prompts" folder for prompt files that have not been seen by the model thus far by utilizing set membership validation. When an unseen prompt is detected, it is passed into the StableDiffusion model, which creates an image. The image is then saved into the "prompts" folder, where the Flask app then retrieves it from to display to the user. I have shown this code as well as some of the images generated by the project so far.

Code Snippet:

```
import time
import os
from google.colab import files
import re


# run pre-trained stable diffusion image gen model
(https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5 28 May
2025)

from diffusers import StableDiffusionPipeline
import torch
pipe = StableDiffusionPipeline.from_pretrained(
    "runwayml/stable-diffusion-v1-5",
    cache_dir="/content/model_cache",
    torch_dtype=torch.float16
).to("cuda")

# access google drive from colab
(https://colab.research.google.com/notebooks/snippets/drive.ipynb 28 May 2025)
from google.colab import drive
drive.mount('/content/drive')

# use loop to continuously check for prompts added to the folder and generate
an image when a prompt is detected

prompt_folder = "/content/drive/MyDrive/prompts"
seen_prompts = set()
while True:
    files = os.listdir(prompt_folder)
    new_prompts = [f for f in files if f.startswith("prompt_") and f not in
seen_prompts]
    for fname in new_prompts:
        prompt_path = os.path.join(prompt_folder, fname)
        with open(prompt_path, 'r') as f:
            prompt = f.read().strip()
        print(f"Processing: {fname}")
        image = pipe(prompt).images[0]
        timestamp = re.search(r'\d+', fname).group()
        output_path = os.path.join(prompt_folder, f"output_{timestamp}.png")
        image.save(output_path)
        print("File uploaded!")
        seen_prompts.add(fname)
    time.sleep(2)
```

3. Concluding Remarks

My experience creating this project was challenging but incredibly rewarding. It felt daunting to successfully create a functioning image generation application, especially after running into some challenges that caused me to have to change my methodology. My main takeaways from this experience are that it is possible to achieve an ambitious project as long as you are willing to experience lots of trial and error to get to the end goal. While the general purpose of my project persisted, my original plan for achieving the final project changed plenty of times, which required flexibility and perseverance

The ethical ramifications of my project are mostly associated with a user's personal Spotify account data. While the Spotify API requires valid credentials, the app itself is not built with secure measures to protect a user's information once they log in, so it may be susceptible to leaking private Spotify account information. Further, feeding information from album covers into an image generation model may raise issues about who owns the resulting images, as existing art is used to inform the new creations.