CIS5810 Final Project Part 5 | Fall 2025

Team: Vision Totem Lab (Liwen He, Xinyu Liu, Ivy Zhong)
Project Title: *Animorphic Triptych: Feature-Driven Persona Generation*[1]

1. Project Summary

We extend the baseline *Spirit-Animal Generation* (Idea 5, Track 1) pipeline and implement **Animorphic Triptych**, a multi-stage computer vision pipeline that generates three animal portraits (a triptych) from a human portrait while preserving the user's geometric facial structure. Our approach separates **structure** (facial geometry) from **texture** (skin/fur) and uses classical CV methods (landmarking, edge detection) to constrain modern generative models (Stable Diffusion + ControlNet). Semantic selection of animal archetypes is performed with GPT-4o Vision. The pipeline enforces biometric constraints (e.g., inter-ocular distance, jawline curvature) so the generated animals reflect the user's underlying facial geometry while adopting distinct animal textures.

2. Problem Statement & Target Users

2.1 Problem Statement

We framed the problem as **Structure-Texture Separation**: the need to preserve facial geometry while changing surface appearance to an animal texture. Existing pixel-based transfer methods either (a) paste human texture onto animal forms producing uncanny artifacts, or (b) generate pleasing images that ignore the subject's geometry entirely.

Key obstacles:

- **Geometric Misalignment:** Pixel blending preserves inappropriate human textures, causing uncanny results.
- **Loss of Spatial Constraints:** Pure text-to-image generation discards spatial information and produces geometry that does not correspond to the input face.

2.2 Stakeholders & Use Cases

- Social media & gaming: privacy-preserving avatars that retain a user's geometry without exposing raw biometrics.
- Digital creators & concept artists: structural control when visualizing characters from human references.

3. Our Solution

We implement a **Feature-Driven Structural Morphing** pipeline:

[1] Notes: For the purpose of preserving conceptual and stylistic consistency, Part 5 draws upon and adapts content presented in the preceding parts.

1. **Semantic Decomposition (GPT-4o Vision):** Suggest three animal archetypes and the facial feature to emphasize (e.g., *Owl — eyes*).
2. **Structural Isolation (MediaPipe + Canny):** Use MediaPipe Face Mesh to extract landmarks and isolate high-frequency edge maps for specific features (eyes, jawline, mouth). Create *sparse* Canny masks that preserve geometry while removing texture.
3. **Generative Reconstruction (Stable Diffusion + ControlNet):** Use the sparse edge maps as boundary conditions (ControlNet-Canny) and prompt the diffusion model to synthesize animal textures that align to the human wireframe.

The output is a triptych: three animal images that preserve the same human geometry while each showing a different biological texture.

4. Pipeline and Baseline

4.1 Baseline

Our initial baseline used an Img2Img Stable Diffusion setup: input = human photo + prompt like "a photo of a lion." Results either ignored geometry or applied fur as a superficial filter, confirming the need for structural conditioning.

4.2 Final Pipeline (Four Stages)

Stage I — Preprocessing & Alignment (MediaPipe)

- Detect face bounding box (MediaPipe Face Detection), perform a tight crop and expand by ~10% to include hair and jawline, and rotate using eye landmarks to standardize frontal orientation. Resize to 512×512.

Stage II — Semantic Reasoning (GPT-4o Vision)

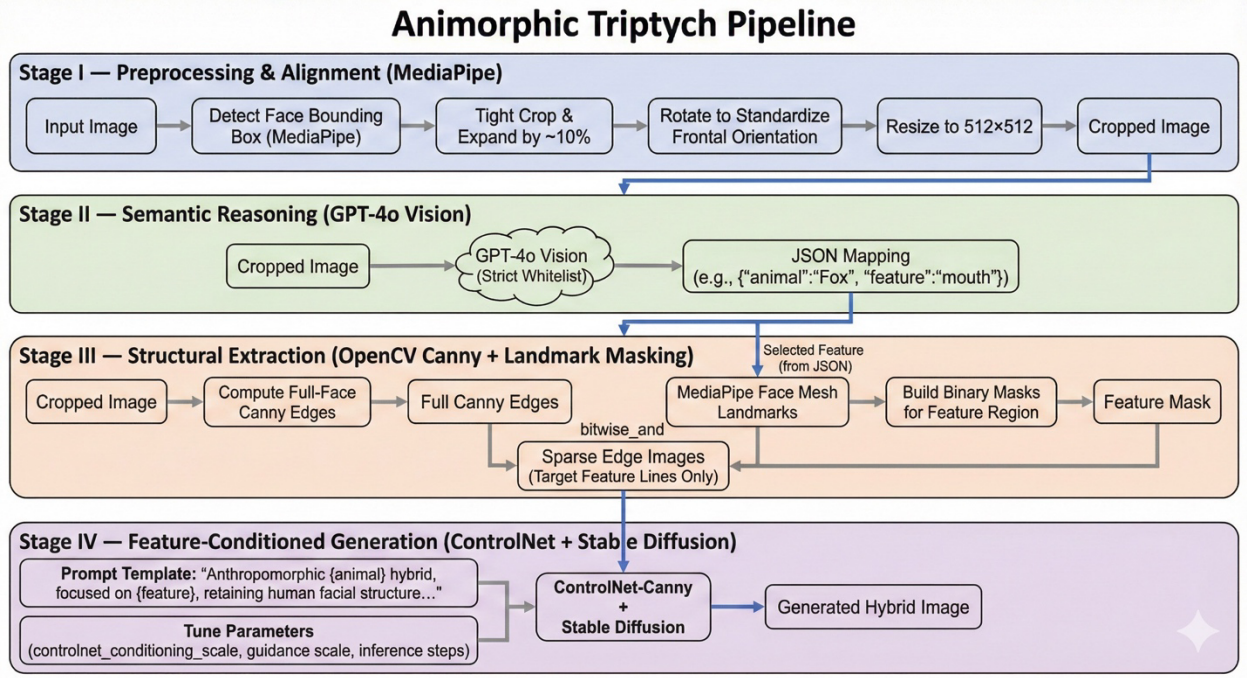
- Send cropped image to GPT-4o with a strict whitelist to restrict animals to geometrically compatible species. GPT returns a JSON mapping animals to features (e.g., {"animal":"Fox", "feature":"mouth"}).

Stage III — Structural Extraction (OpenCV Canny + Landmark Masking)

- Compute full-face Canny edges. Use MediaPipe Face Mesh landmarks to build binary masks for the selected feature region and perform a bitwise_and to produce *sparse* edge images containing only the target feature lines.

Stage IV — Feature-Conditioned Generation (ControlNet + Stable Diffusion)

- For each animal: feed the feature-specific sparse Canny into ControlNet-Canny and a prompt template like:
  Anthropomorphic {animal} hybrid, focused on {feature}, retaining human facial structure...
- Tune controlnet_conditioning_scale (λ), guidance scale, and number of inference steps to balance geometry adherence and animal texture realism.

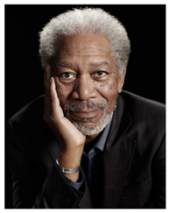


5. Experiments and Results

Experiment A - Sementic Analysis & Feature-Mapping

A core challenge in Stage II is balancing output *reproducibility* (getting the same result for the same image) with *variability* (getting different results for different images).

Our initial tests used a deterministic setup (model: gpt-4o-mini-2024-07-18, temperature: 0.0, top_p: 1.0, seed: 42). This configuration successfully achieved reproducibility when the same image was input multiple times. However, it exhibited very low variability, often returning the exact same animal archetypes (e.g., Owl, Fox, Deer) for different input portraits, as seen below.



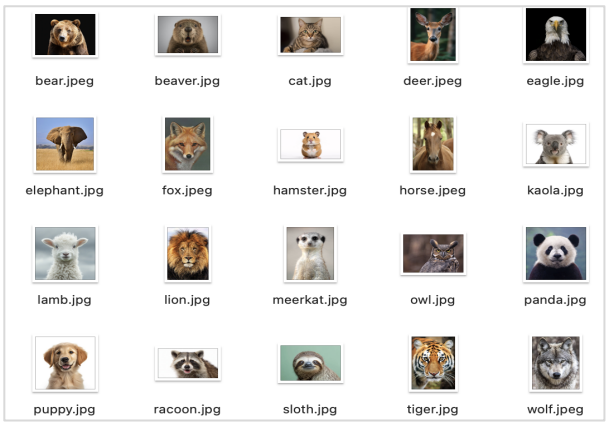
Both Input A and Input B produced the identical JSON output:

```
{
  "animals":[
    {"animal":"Owl", "trait": "wise, observant", "feature": "eyes"},
    {"animal": "Fox", "trait": "clever, playful", "feature": "mouth"},
    {"animal": "Deer", "trait": "gentle, alert", "feature": "jawline"}
  ]
}
```

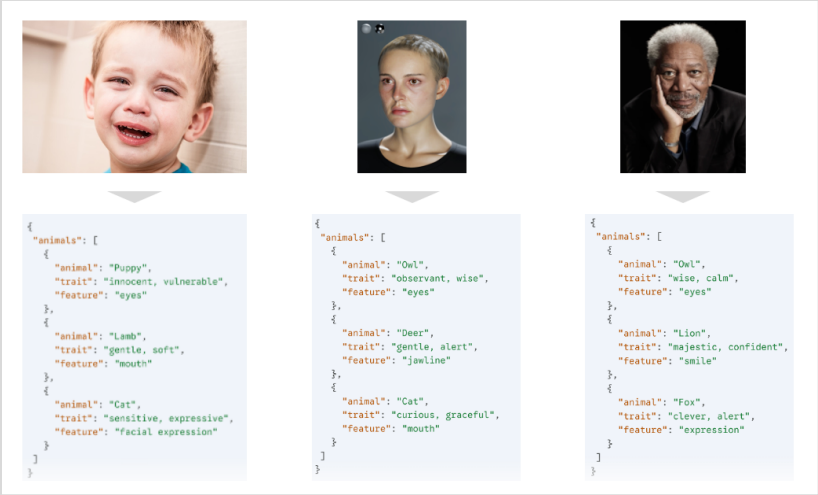
To solve this, we implemented two key improvements:

1. **Image-Dependent Seed:** We replaced the fixed seed with a deterministic, image-dependent seed, calculated by hashing the input image: seed = int(img_hash, 16) % (2**31). This ensures reproducibility for a single image while guaranteeing a unique starting point for different images.

2. **Constrained Animal Pool:** To control the output and align with our project's visual goals, we restricted the model to a predefined animal pool of 20 species. This was enforced through a new, highly-structured prompt that explicitly lists the <ALLOWED ANIMAL POOL>.



This new approach successfully balances consistency with variability, producing diverse and context-appropriate archetypes for different faces, as shown in the examples below.



Experiment B - Prototype Fusion Testing

Experimental Method: This experiment explores the fusion of human and animal faces using Stable Diffusion and ControlNet for image-to-image transformation, with OpenCV handling face alignment. We tested text prompts such as **"half human, half wolf"** to guide the latent-space blending. For fixed-pair tests (e.g., Ronaldo–Wolf), we also experimented with region-based warping, LAB color tone adjustment, and texture overlay, adjusting the blending ratio (e.g., 50%, 70%, 85%) to balance realism and clarity.

Results: The generated images present convincing combinations of human and animal features. We can successfully integrate **fur texture, animalistic eye shapes, and muzzle outlines** while maintaining coherent lighting and smooth transitions to human skin.

Varying the diffusion strength and CFG scale allowed for precise control over the fusion intensity. Our experiments indicate that balanced prompt tuning and staged blending produce the most natural

outcomes. As shown in the figures below, excessive emphasis on animal traits (e.g., high blending ratio or CFG) tends to distort the original human facial identity, validating our project's approach of using *sparse* Canny maps to *gently guide* the structure rather than completely overwriting it.

Experiment C — "Ghost Edges" Failure

- **Hypothesis:** Faint full-head outlines would provide shape context.
- **Result:** Model interpreted the faint outline as a human head and generated animals with human necks/clothing or human skin textures.
- **Correction:** We switched to **Sparse Masking** (only target feature edges), forcing the model to hallucinate animal head shapes while preserving local geometry.

Experiment D — ControlNet Conditioning Scale (λ)

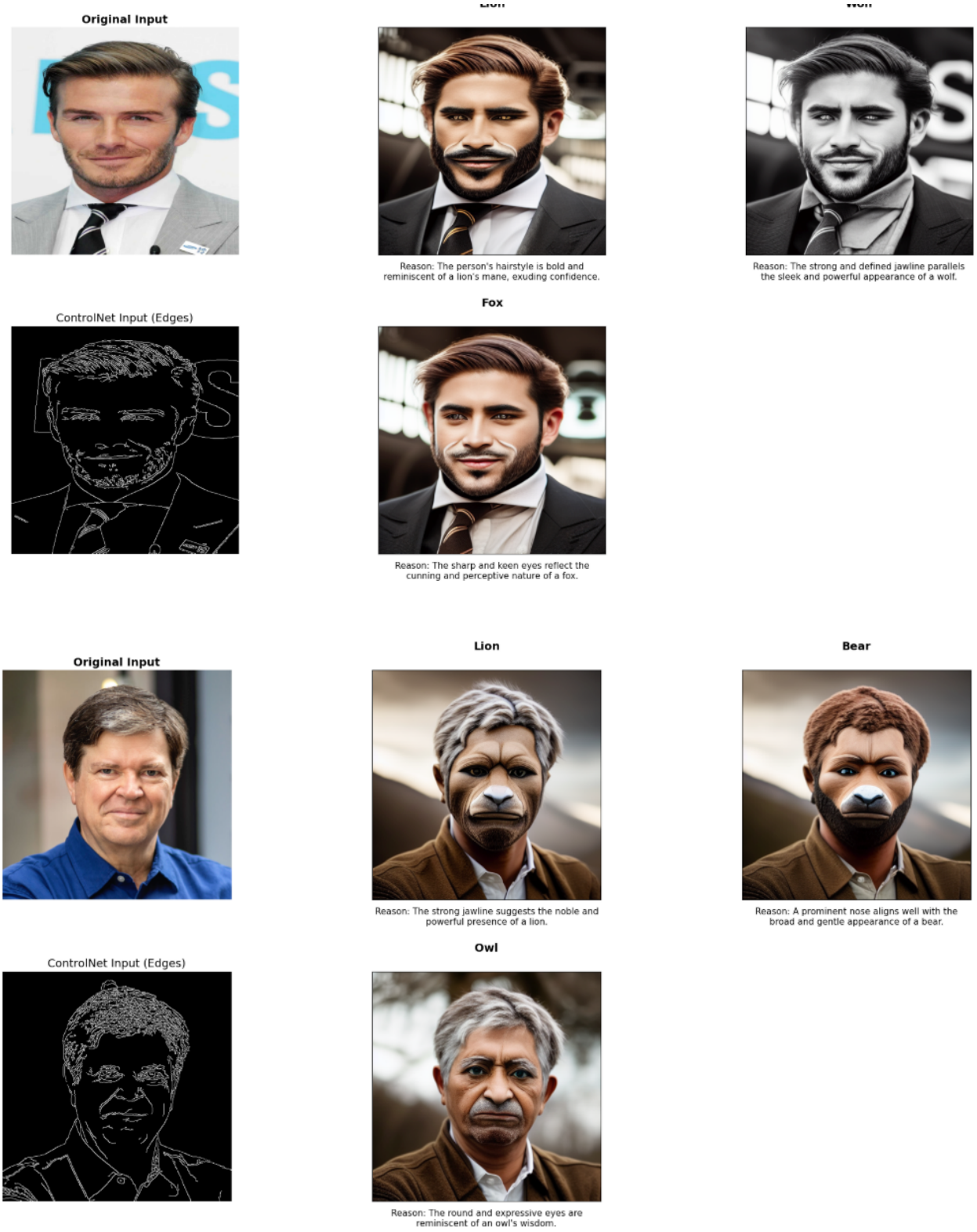
- λ = 1.0 → overly strict adherence (human face painted with animal texture).
- λ = 0.3 → geometry ignored (generic animal).
- λ = 0.65 → sweet spot: preserves eye/jawline geometry while allowing animal texture and topology to emerge.

Experiment E — Geometric Whitelisting

- Allowing unconstrained animal suggestions produced failed mappings (e.g., elephant trunk vs. human nose).
- Applying a whitelist (planar/compatible species: lions, wolves, bears, owls) avoided pathological warping and improved final realism.

Final Results & Performance

- Our pipeline generated convincing triptychs for test subjects (e.g., a "Beckham Triptych" of Wolf, Lion, Panther).



- Inference time on Colab T4 was optimized to ~4 seconds per image for the chosen settings.
- Code, Colab notebook, and example outputs are attached (see Project Source Code link).

6. Timeline & Team Organization

Timeline (summary):

- **Week ~9:** Project Scoping, TA Communication, Colab environment setup, Img2Img baseline, Experiments.
- **Week 9-11:** MediaPipe integration; initial ControlNet experiments; discovered "Ghost Edges" failure.
- **Week 11-12:** Implemented Strict Whitelist and Sparse Masking; tuned ControlNet scales to find $\lambda = 0.65$.
- **Week 12-13:** Integrated full pipeline into single execution loop; tested across several portraits.
- **Week 14:** Report writing, slides/demo, video recording.

Team responsibilities:

- **Liwen He (Pipeline Architect):** MediaPipe extraction, Canny masking, core Python pipeline, Colab dependency debugging.
- **Xinyu Liu (Generative AI Specialist):** Stable Diffusion & ControlNet integration; tuned conditioning scale, guidance scale, and scheduler.
- **Ivy Zhong (NLP & Prompt Engineer):** GPT-4o system prompts, whitelist logic, JSON schema and negative prompting.
- **Project Format:** 2-week sprint with weekly meet-up.

7. Future Work

We list three directions we would pursue with more time (kept as stated):

1. **Geometric Warping (TPS):** to support animals with long snouts (horses, crocodiles) via Thin-Plate Spline warping of the landmark mesh.
2. **3D Texture Mapping:** use MediaPipe 3D Face Mesh to generate textures for video/AR.
3. **LoRA Integration:** train a small LoRA adapter for consistent stylistic control (e.g., oil painting, cyberpunk).

8. Lessons Learned

- **Constraints improve outputs:** The more we restrict input (background removal, animal whitelist), the better the model output.
- **Dependency management matters:** Version conflicts (e.g., NumPy vs. MediaPipe) were time sinks—pin versions early.
- **Prompting is engineering:** Negative prompts (e.g., "no human skin") function as filters for diffusion models.

9. Resources

- **Finalized Colab Notebook / Code:** Link - included with submission (main script orchestrates detection → GPT → sparse Canny → ControlNet generation).
- **Recorded Demo Video:** Link
- **Presentation Slides:** Link

10. Acknowledgments

We would like to express our sincere gratitude to **Professor Shi** for his insightful instruction in CIS5810. His lectures on camera geometry and deep learning provided the essential theoretical framework that made this project possible, particularly in our approach to structural feature extraction.

We also extend our thanks to our Teaching Assistant, **Rajnish Gupta**, for his guidance, constructive feedback during project milestones, and support in navigating the technical challenges of the computer vision pipeline.

11. Project Code

```
# ==========================================
# PART 1: SYSTEM SETUP & DRIVE INTEGRATION
# ==========================================

# Import the os module to handle file paths and directory creation
import os

# Import the Google Drive module to access your cloud storage
from google.colab import drive

# Mount Google Drive to the local Colab environment
# force_remount=True ensures a fresh connection if connection was lost
drive.mount('/content/drive', force_remount=True)

# Define the master project path in Google Drive
# Optimization: Storing this in a variable allows for easy changes later
project_path = '/content/drive/MyDrive/CIS5810 Fa25/project_final'

# Check if the project directory exists
if not os.path.exists(project_path):
    # If not, create the directory structure recursively (creates parents if missing)
    os.makedirs(project_path)
    print(f"Created new directory: {project_path}")
else:
    # Confirm the directory was found to reassure the user
    print(f"Found existing directory: {project_path}")

# Change the current working directory to the project folder
# This ensures all saved images automatically go to your Drive
os.chdir(project_path)
print(f"✅ Active Directory set to: {os.getcwd()}")
```

```
# ==========================================
# PART 2: LIBRARY INSTALLATION & IMPORTS
# ==========================================

# Install required Deep Learning libraries
# Optimization: -q flag creates 'quiet' mode to reduce log clutter
# &> /dev/null hides standard output to keep the notebook clean
!pip install -q diffusers transformers accelerate controlnet_aux mediapipe &>
/dev/null
!pip install -q opencv-python-headless &> /dev/null

# Import Computer Vision library (OpenCV) for image processing
import cv2
# Import NumPy for matrix/array operations (image data handling)
import numpy as np
# Import PyTorch, the core Deep Learning framework
import torch
# Import PIL for easy image loading and resizing
from PIL import Image
# Import Matplotlib for visualizing the results in a grid
import matplotlib.pyplot as plt

# Import specific modules from the Diffusers library (Hugging Face)
# StableDiffusionControlNetPipeline: The main logic that combines SD + ControlNet
# ControlNetModel: The specific neural network that understands edges/structure
# UniPCMultistepScheduler: An optimized scheduler that makes generation faster (fewer
steps)
from diffusers import StableDiffusionControlNetPipeline, ControlNetModel,
UniPCMultistepScheduler

# Import the edge detector specifically for extracting human features
from controlnet_aux import CannyDetector

print("✅ Libraries installed and imported.")

# ==========================================
# PART 3: MODEL LOADING (With Caching)
# ==========================================

# Optimization: Check if 'pipe' exists in global variables.
# This prevents reloading the heavy AI models (2GB+) if you re-run this cell.
if 'pipe' not in globals():
    print(" Loading Generative AI Models (First time run - approx 2 mins)...")

    # Initialize the Canny Detector
    # This tool extracts the "wireframe" of the user's face (edges)
    canny_detector = CannyDetector()

    # Load the ControlNet model trained on Canny edges
    # torch_dtype=torch.float16 reduces memory usage by 50% without losing quality
    controlnet = ControlNetModel.from_pretrained(
        "lllyasviel/sd-controlnet-canny",
        torch_dtype=torch.float16
    )

    # Load the main Stable Diffusion 1.5 model
    # We attach the 'controlnet' we just loaded to this pipeline
    pipe = StableDiffusionControlNetPipeline.from_pretrained(
        "runwayml/stable-diffusion-v1-5",
        controlnet=controlnet,
        torch_dtype=torch.float16,
        safety_checker=None # Optimization: Disabling safety checker saves RAM and
speeds up inference
```

```
    )

    # Set the scheduler to UniPC
    # Optimization: Drastically improves speed. 20 steps with UniPC = 50 steps with
others.
    pipe.scheduler = UniPCMultistepScheduler.from_config(pipe.scheduler.config)

    # Enable CPU Offload
    # Optimization: Automatically moves parts of the AI to RAM when GPU is full.
    # Essential for running on the free tier of Google Colab.
    pipe.enable_model_cpu_offload()

    print("✅ Models successfully loaded into memory.")
else:
    print("⚡ Models already loaded. Skipping setup to save time.")

# ==========================================
# PART 3.5: OPENAI API SETUP
# ==========================================

"""
System Setup for OpenAI Authentication.

This block handles the secure retrieval of the API key needed to communicate with GPT-
4o.
It implements a "Graceful Fallback" mechanism:
1. First, it tries to fetch the key from Google Colab's encrypted 'Secrets' store
(Best Practice).
2. If that fails (key not found), it pauses execution and asks the user to paste the
key manually.
"""

# Import 'base64' to encode binary image files into text strings (required for the
OpenAI Vision API)
import base64
# Import 'json' to parse the structured text response (lists/dictionaries) returned by
GPT-4o
import json

# Import the main client class from the official OpenAI Python library
from openai import OpenAI

# --- OPENAI AUTHENTICATION LOGIC ---

# Start a 'try' block to attempt the most secure method first
try:
    # Attempt to retrieve the value associated with 'OPENAI_API_KEY' from Colab's
Secrets
    # Note: 'userdata' is a secure module from google.colab defined in Part 1
    api_key = userdata.get('OPENAI_API_KEY')

    # Validation check: If the key name exists but has no value, or doesn't exist
(returns None)
    if api_key is None:
        # Manually trigger an error to force the code to jump to the 'except' block
        raise ValueError("Key not found")

    # If we get here, the key was found successfully; print a confirmation
    print("✅ OpenAI API Key retrieved from Secrets.")

# The 'except' block catches any errors from the 'try' block (missing key, missing
userdata module, etc.)
```

```
except:
    # Fallback Mechanism: If automatic retrieval fails, ask the user to input it
interactively
    # .strip() removes any accidental whitespace (spaces/newlines) from the copy-paste
    api_key = input("Please enter your OpenAI API Key: ").strip()

# Initialize the OpenAI client connection using the valid API key
# This 'client' object will be used to send the image analysis requests later
client = OpenAI(api_key=api_key)

# ==========================================
# PART 4: THE AI ANALYST (GPT-4o)
# ==========================================

def encode_image(image_path):
    """Encodes image to Base64 for OpenAI API."""
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode('utf-8')

def analyze_image_with_gpt(image_path):
    """
    Uses OpenAI GPT-4o to analyze a human face and suggest spirit animals.

    UPDATES:
    - strict_whitelist: Forces GPT to pick animals that align with human geometry.
    - feature_scope: Restricts 'feature_focus' to specific anatomical parts.
    """
    print(f"🤔 Sending image to OpenAI (GPT-4o) for analysis...")

    base64_image = encode_image(image_path)

    # REFINED PROMPT with STRICT GEOMETRIC & FEATURE CONSTRAINTS
    system_prompt = """
    You are a creative visual analyst. You must output JSON only.
    Analyze the input image (facial expression, vibe) and suggest 3 distinct spirit
animals.

    CRITICAL INSTRUCTION: The user is merging these animals onto a HUMAN FACE using
ControlNet.
    To ensure the final image is recognizable as the human, you MUST follow these
constraints:

    1. ALLOWED ANIMALS (The "Safe List"):
       - Felines (Lion, Panther, Cheetah, Cat) -> Best alignment.
       - Canines (Wolf, Fox, Husky, Coyote) -> Good snout mapping.
       - Ursidae (Bear, Panda, Polar Bear) -> Round faces match human skulls.
       - Strigiformes (Owl) -> Flat face maps well to human eyes.

    2. BANNED ANIMALS (Do NOT Suggest):
       - Elephant (Trunk distorts nose).
       - Crocodile/Horse (Snout is too long).
       - Birds with long beaks (Toucan, Stork).
       - Insects/Fish (No facial alignment).

    The output must be a JSON object containing a key "suggestions" which is a list of
3 items.
    Each item must have:
    - "animal": Name of the animal.
    - "feature_focus": EXACTLY ONE of the following strings: "eyes", "nose", "mouth",
"hair", "jawline".
       (Choose the most distinct feature of the person to emphasize in the
generation).
    - "reason": A short sentence explaining the match.
```

```
    """

    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": [
                {"type": "text", "text": "Analyze this person and provide 3
geometrically compatible spirit animal suggestions in JSON."},
                {"type": "image_url", "image_url": {"url":
f"{base64_image}"}}
            ]}
        ],
        response_format={"type": "json_object"}
    )

    try:
        content = response.choices[0].message.content
        data = json.loads(content)
        return data['suggestions']
    except Exception as e:
        print(f"❌ Error parsing OpenAI response:\n{content}")
        raise e

# ==========================================
# PART 6: MAIN EXECUTION LOOP
# ==========================================

import textwrap

def run_dynamic_project(human_image_path):
    """
    Orchestrates the full 'Animorphic Triptych' pipeline.

    This function performs four main steps:
    1. VALIDATION: Checks if the input image exists.
    2. ANALYSIS: Sends the image to OpenAI GPT-4o to analyze the facial expression
       and generate 3 distinct 'spirit animal' suggestions (returned as JSON).
    3. GENERATION: Iterates through the suggestions and uses Stable Diffusion +
ControlNet
       to generate the corresponding images.
    4. VISUALIZATION: Creates a Matplotlib grid displaying the original human,
       the computer vision edge map, and the three generated spirit animals
       along with their reasoning.

    Args:
        human_image_path (str): The file path to the input human photograph.

    Returns:
        None: This function saves images to disk and displays a plot inline.
    """

    # --- INPUT VALIDATION ---
    # Check if the file exists at the specified path to prevent crashing later
    if not os.path.exists(human_image_path):
        print(f"❌ Error: {human_image_path} not found.")
        return

    # --- STEP 1: Get Analysis from OpenAI ---
    try:
        # Call the GPT-4o wrapper function defined in Part 4.
        # This returns a list of dictionaries, e.g., [{'animal': 'Lion', ...}, ...]
```

```
        targets = analyze_image_with_gpt(human_image_path)
    except Exception as e:
        # Catch network errors or API key issues so the notebook doesn't crash
entirely
        print(f"Aborting: {e}")
        return

    # --- STEP 2: Print the JSON Output ---
    # We print the raw JSON to the console for debugging and transparency.
    # This confirms exactly what GPT "saw" and suggested before we generate images.
    print("\n" + "="*40)
    print("OPENAI GPT-4o ANALYSIS (JSON OUTPUT)")
    print("="*40)
    print(json.dumps(targets, indent=4)) # 'indent=4' makes the JSON readable
    print("="*40 + "\n")

    # --- STEP 3: Generate Images ---
    results = [] # Initialize an empty list to store the generated image objects

    # Load and resize the original image once for the final display comparison
    orig_img = Image.open(human_image_path).resize((512, 512))

    # Set a fixed seed for the visualization loop.
    # This ensures that all 3 animals share the same underlying random noise patterns.
    display_seed = 999

    print(f"🎨 Starting Generation based on GPT suggestions...")

    # Loop through the list of 3 animals provided by GPT
    for target in targets:
        # Extract data from the dictionary
        animal = target['animal']
        focus = target['feature_focus']
        reason = target['reason']

        print(f"   ✨ Generating {animal}...")

        # Call the main generator function (Part 5)
        # result_img = The final AI art; edge_map = The Canny lines used to guide it
        result_img, edge_map = generate_spirit_animal(human_image_path, animal, focus,
seed=display_seed)

        # Construct a filename and save the result to Google Drive immediately
        save_name = f"gpt_generated_{animal}.png"
        result_img.save(save_name)

        # Add the result to our list for the plotting step
        results.append({"img": result_img, "title": animal, "reason": reason, "edges":
edge_map})

    # --- STEP 4: Visualization ---
    # Create a large figure (canvas) size (20x12 inches) to fit all images and text
clearly
    plt.figure(figsize=(20, 12))

    # Plot 1: The Original Human Input (Top Left)
    plt.subplot(2, 3, 1) # (Rows, Columns, Index)
    plt.imshow(orig_img)
    plt.title("Original Input", fontsize=14, fontweight='bold')
    plt.axis('off') # Hide X/Y axis numbers

    # Plot 2: The ControlNet Edges (Bottom Left)
    # This shows the user what the AI "saw" (the structural wireframe)
```

```
    plt.subplot(2, 3, 4)
    plt.imshow(results[0]['edges'], cmap='gray') # Display in grayscale
    plt.title("ControlNet Input (Edges)", fontsize=14)
    plt.axis('off')

    # Plot 3, 4, 5: The Generated Results
    # We map the results to specific grid positions: Top-Mid, Top-Right, Bottom-Mid
    plot_indices = [2, 3, 5]

    for i, res in enumerate(results):
        plt.subplot(2, 3, plot_indices[i])
        plt.imshow(res['img'])

        # Create a bold title for the Animal Name
        full_title = f"{res['title']}\n"

        # Wrap the 'Reason' text so it doesn't run off the screen.
        # breaks the long sentence into lines of max 50 characters.
        wrapped_reason = "\n".join(textwrap.wrap(f"Reason: {res['reason']}",
width=50))

        # Set the title (Animal Name)
        plt.title(full_title, fontsize=14, fontweight='bold', pad=10)

        # Set the xlabel (The Reason). We use xlabel because it sits naturally *under*
the image.
        plt.xlabel(wrapped_reason, fontsize=11, labelpad=10)

        # Hide axis ticks but keep the label visible
        plt.xticks([])
        plt.yticks([])

    # Adjust layout to prevent images and text from overlapping
    plt.tight_layout(pad=3.0)
    # Add extra white space at the bottom to ensure the text reasons aren't cut off
    plt.subplots_adjust(bottom=0.15)

    # Finally, render the plot to the screen
    plt.show()

# ==========================================
# PART 7: RUN
# ==========================================

run_dynamic_project("your_image_file.jpg")
```