ComicCrafter AI - Project Report

Self Overview

Hello all, ourself Kavin Aarya A R, Jeevitha S and Hephzibah A, **Computer Science andEngineering (CSE)** students from **Sri Sairam Institute of Technology**, embarked on an impactful **Intel internship**, contributing to the **ComicCrafter AI project.**This initative focused on building a web-based tool that leverages AI to generate comic strips from user-provided text descriptions.

We collaboratively developed a comic generator project aimed at creating custom comic strips using AI and user inputs. This project allows users to generate custom comic strips based on text input using Python, LLMs and text to image generation models. Each member contributed to different aspects including design, development and content generation. This project strengthened our skills in creativity, teamwork and technical implementation, while giving us hands-on experience in blending technology with storytelling.

1. Problem Statement

Creating comic strips or short visual stories traditionally requires significant artistic skill (drawing, panel layout) and writing ability (storytelling, dialogue). Many individuals have creative ideas but lack the time or specific skills to bring them to life visually. There is a need for a tool that can automate or significantly assist in the comic creation process, starting from a simple text-based concept.

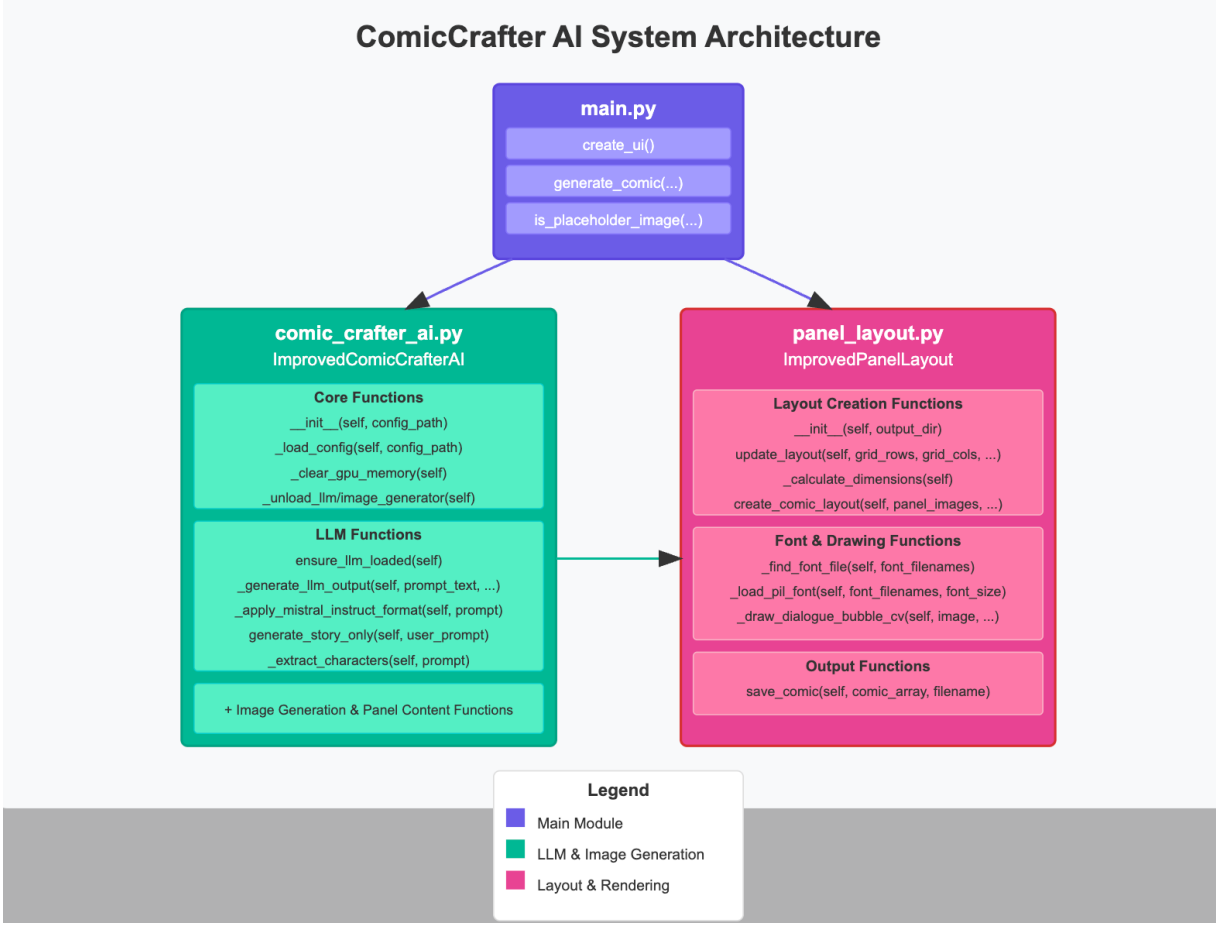
2. Solution: ComicCrafter AI

ComicCrafter AI addresses this problem by leveraging generative Artificial Intelligence (AI). It provides a user-friendly interface where a user inputs a story idea. The application then uses:

1. A Large Language Model (LLM) to generate a narrative, title, moral, per-panel descriptions, and dialogues based on the user's prompt and internal templates.
2. A Text-to-Image Diffusion Model to create visuals for each panel based on the generated descriptions and a selected art style.
3. A Layout Engine (using OpenCV and PIL) to optionally arrange the generated panels, title, panel numbers, and dialogues into a final comic page image.

This allows users to quickly generate multi-panel comics from text prompts with minimal manual effort, bridging the gap between concept and visual creation.

3. System Architecture: ComicCrafter AI



4. Features of ComicCrafter AI

- **End-to-End Comic Generation:** It takes a simple user prompt (story idea) and handles the entire workflow – generating a narrative, title, moral, panel descriptions, dialogue, panel images, and finally assembling them into a complete visual comic page.
- **Integrated Multi-Modal AI:** Seamlessly combines a Large Language Model (LLM) for text-based tasks (story, dialogue, prompts) with an Image Generation Model (like Stable Diffusion XL) for creating the visual panels, all orchestrated within a single application.
- **Structured Story Generation:** Goes beyond simple image generation by first creating a cohesive narrative structure, including a title and a moral, which then guides the generation of individual panel content and dialogue.
- **Automated Visual Layout and Annotation:** The `PanelLayout` module automatically arranges the generated images into a standard comic grid and crucially, overlays generated elements like the title, panel numbers, and dialogue bubbles directly onto the final image using libraries like PIL and OpenCV.
- **Prompt Template Configuration:** Utilizes a `config.json` file to manage and template the various prompts sent to the LLM. This allows for easier customization and fine-tuning of the AI's output for different story elements (narrative, panel description, dialogue, etc.) without modifying the core code.

- **Explicit Art Style Control:** The user interface allows the selection of a specific art style, which is then incorporated into the prompts used for the image generation model, providing direct control over the visual aesthetic of the comic.
- **Built-in Resource Management:** The ComicCrafterAI class includes logic for loading and unloading AI models, potentially helping manage GPU memory usage, which is important when dealing with large models.

5. Libraries Used

The project utilizes several Python libraries:

- **Core AI:**
  - torch: Foundational deep learning framework.
  - transformers: For loading and interacting with the LLM (Mistral).
  - diffusers: For loading and interacting with the text-to-image model (Stable Diffusion XL).
  - accelerate: Assists with efficient model loading and execution across hardware.
  - bitsandbytes: Enables model quantization (like 4-bit) for reduced memory usage.
- **Image Processing & Layout:**
  - Pillow (PIL): For basic image manipulation, text rendering (title, panel numbers), and handling image objects.
  - numpy: For numerical operations, particularly with image data arrays.
  - opencv-python (cv2): Used in the layout engine for drawing dialogue bubbles and text onto the final comic page.
- **User Interface:**
  - gradio: To create the web-based user interface for user input and displaying results.
- **Standard Libraries:**
  - os, json, pathlib, time, uuid, traceback, re, gc, math, textwrap, typing.

6. Code File Explanation

The project is primarily structured into three Python files and a configuration file:

a) config.json

- **Purpose:** Stores configuration settings, model identifiers, and prompt templates.
- **Key Contents:**
  - story_model_path: Specifies the Hugging Face identifier for the LLM (e.g., "mistralai/Mistral-7B-Instruct-v0.2").
  - image_model_path: Specifies the identifier for the text-to-image model (e.g., "stabilityai/stable-diffusion-xl-base-1.0").
  - disable_safety_checker: Boolean to toggle the built-in safety checker in the image model.

- default_settings: Default parameters for generation (LLM temperature, image inference steps, guidance scale, image dimensions, negative prompts).
- prompt_template: Contains structured text templates fed to the LLM for generating the story, title, moral, panel descriptions, and dialogues, ensuring consistent output formatting.

Code:

```
{
"story_model_path": "mistralai/Mistral-7B-Instruct-v0.2",
"image_model_path": "stabilityai/stable-diffusion-xl-base-1.0",
"disable_safety_checker": true,
"default_settings": {
"temperature": 0.7,
"num_inference_steps": 30,
"guidance_scale": 7.0,
"image_width": 1024,
"image_height": 768,
"negative_prompt": "text, words, letters, signature, watermark, blurry,
deformed, bad anatomy, disfigured, poorly drawn face, mutation, mutated,
extra limb, ugly, poorly drawn hands, missing limb, blurry, floating
limbs, disconnected limbs, malformed hands, out of focus, long neck, long
body, duplicate, cropped, low quality, jpeg artifacts, multiple panels,
frame, border, username, artist name, error, glitch, noise, tiling,
stereotype, caricature, biased depiction, duplicate characters, multiple
instances of same character, clones, multiple views, multiple views of the
same character, grid, multiple images, multiple scenes"
    },
"prompt_template": {
"narrative_prompt_template": "Write a short comic book story narrative
(200-400 words) based on the user prompt: \"{user_prompt}\". Feature the
character(s): {character_str}. The story must be neutral, unbiased, and
avoid stereotypes. Output ONLY the story narrative itself, covering a
beginning, middle, and end. Do not add a title or moral here.",
"title_prompt_template": "Based on the following story narrative, generate
ONLY a concise and fitting title (5 words maximum):\n\n---
\n{story_narrative}\n---\n\nTitle:",
"moral_prompt_template": "Based on the following story narrative, generate
ONLY a brief moral or theme (1-2 sentences maximum):\n\n---
\n{story_narrative}\n---\n\nMoral:",
"panel_prompt_template": "Given the overall story context, generate ONLY a
**VERY concise visual description (MAXIMUM 60 words)** for a comic book
style image for Panel {panel_num} of {num_panels}. This description is for
an AI image generator. Describe the scene naturally, avoiding unnecessary
duplication of characters unless context demands it.\n\n**Overall User
```

```
Prompt:** \"{user_prompt}\"\n**Key Characters:**
{character_str}\n**Relevant Story Context (Focus on the {stage} stage):**
\"{story_context}\"\n\n**Instructions for Panel {panel_num} Visual
Description:**\n1. **Visual Focus:** Describe the scene VISUALLY -
character appearance/action/emotion, environment, lighting, composition
(e.g., wide shot, close-up).\n2. **EXTREME CONCISENESS:** 60 words
MAXIMUM. Be descriptive but EXTREMELY brief.\n3. **Output ONLY
Description:** Output *only* the visual description text. NO intro phrases
like 'Panel X:', NO lists, NO explanations, NO panel numbers, NO brackets,
NO markdown.\n4. **NO Text/Sound:** Do NOT include dialogue, sound
effects, captions, words, speech bubbles.\n5. **Anti-Bias:** AVOID
STEREOTYPES. Be neutral.\n\nPanel {panel_num} Visual Description:",
"dialogue_prompt_template": "You are writing dialogue for a comic book
panel. Given the visual description of the panel and the overall story
context, write a single, short line of dialogue (max 25 words) that ONE
character might be saying. Ensure the dialogue is specific to this panel's
context and distinct from other panels. Output ONLY the dialogue text. If
no dialogue fits, output 'NONE'.\n\n**Characters Present:**
{character_str}\n**Panel Visual Description:**
\"{panel_description}\"\n**Story Context:**
\"{story_context}\"\n\nDialogue:"
    }
}
```

b) **comic_crafter_ai.py**

- **Purpose:** Encapsulates the core AI logic for generating story components and panel images.
- **Key Functionality:**
    - Loads configuration from config.json.
    - Manages loading/unloading of the LLM and Image Generation models to conserve GPU memory (ensure_llm_loaded, ensure_image_generator_loaded, _unload_llm, _unload_image_generator). Uses 4-bit quantization for the LLM if CUDA is available (BitsAndBytesConfig).
    - Provides methods to generate each story element using the LLM and configured prompts (_generate_llm_output, generate_storyline, generate_title, generate_moral, generate_panel_prompts, generate_dialogues). Includes basic output cleaning (regex) and retry logic.
    - Extracts potential characters from the user prompt (_extract_characters).
    - Generates panel images using the diffusion model (generate_panel_image), incorporating selected art styles and handling potential failures by returning

placeholder images with specific colors indicating the error type (e.g., prompt failure, generator unavailable, NSFW).

- Orchestrates the full comic generation process (generate_comic_elements).

Code:

```
import os
import json
import torch
import numpy as np
from typing import List, Dict, Any, Optional, Callable
from pathlib import Path
from diffusers import AutoPipelineForText2Image, StableDiffusionXLPipeline
from transformers import AutoModelForCausalLM, AutoTokenizer,
BitsAndBytesConfig
import gc
import re
import math
import time
import traceback
from PIL import ImageDraw


# --- Constants (match main.py) ---
PROMPT_FAILURE_PLACEHOLDER_COLOR = (200, 150, 150)
GENERATOR_UNAVAILABLE_PLACEHOLDER_COLOR = (150, 150, 180)
NSFW_PLACEHOLDER_COLOR = (255, 150, 150)
GENERAL_ERROR_PLACEHOLDER_COLOR = (220, 200, 200)
OTHER_FAILURE_PLACEHOLDER_COLOR = (200, 200, 220)
LLM_LOAD_ERROR_PLACEHOLDER_COLOR = (180, 180, 220)
# --- End Constants ---

class ImprovedComicCrafterAI:
def__init__(self, config_path: str = "config.json"):
self.config = self._load_config(config_path)
self.default_settings = self.config.get("default_settings", {})
self.prompt_templates = self.config.get("prompt_template", {})
self.device = "cuda"if torch.cuda.is_available() else"cpu"
self.compute_dtype = torch.float16 ifself.device == "cuda"else
torch.float32
self.llm_model_path = self.config.get('story_model_path',
'mistralai/Mistral-7B-Instruct-v0.2')
self.image_model_path = self.config.get('image_model_path',
'stabilityai/stable-diffusion-xl-base-1.0')
self.tokenizer = None
```

```
self.llm_model = None
self.image_generator = None
self.is_llm_loaded = False
self.is_image_generator_loaded = False
self.story_characters = []
if torch.cuda.is_available(): self._clear_gpu_memory()
self.ensure_llm_loaded()
ifnotself.is_llm_loaded: print("CRITICAL WARNING: LLM failed to load
during initialization.")

def _clear_gpu_memory(self):
ifself.device == "cuda": print("Clearing CUDA cache..."); gc.collect();
torch.cuda.empty_cache(); torch.cuda.synchronize(); print("CUDA memory
cleared.")

def _unload_llm(self):
ifself.llm_model isnotNone: print("Unloading LLM..."); delself.llm_model;
self.llm_model = None
ifself.tokenizer isnotNone: delself.tokenizer; self.tokenizer = None
self.is_llm_loaded = False; print("LLM unloaded.");
self._clear_gpu_memory()

def _unload_image_generator(self):
ifself.image_generator isnotNone: print("Unloading Img Gen...");
delself.image_generator; self.image_generator = None
self.is_image_generator_loaded = False; print("Img Gen unloaded.");
self._clear_gpu_memory()

def ensure_llm_loaded(self):
ifnotself.is_llm_loaded andself.llm_model isnotNone: print("WARN: LLM flag
inconsistent. Unloading."); self._unload_llm()
ifself.is_llm_loaded andself.llm_model isnotNoneandself.tokenizer
isnotNone: returnTrue
print("--- Ensuring LLM is loaded ---");
ifself.is_image_generator_loaded: print("Unloading image generator
first..."); self._unload_image_generator()
ifself.is_llm_loaded orself.llm_model isnotNoneorself.tokenizer isnotNone:
print("ERROR: LLM state inconsistent."); self.is_llm_loaded = False;
returnFalse
try:
print(f"Loading LLM: {self.llm_model_path} (dtype: {self.compute_dtype})")
self.tokenizer = AutoTokenizer.from_pretrained(self.llm_model_path);
ifself.tokenizer.pad_token isNone: self.tokenizer.pad_token =
self.tokenizer.eos_token
            quant_cfg = None
```

```
ifself.device == "cuda": print("Applying 4-bit quantization...");
quant_cfg = BitsAndBytesConfig(load_in_4bit=True,
bnb_4bit_compute_dtype=self.compute_dtype, bnb_4bit_quant_type="nf4",
bnb_4bit_use_double_quant=True)
self.llm_model = AutoModelForCausalLM.from_pretrained(self.llm_model_path,
torch_dtype=self.compute_dtype if quant_cfg isNoneelseNone,
device_map="auto", quantization_config=quant_cfg, low_cpu_mem_usage=True)
ifself.llm_model.config.pad_token_id isNone:
self.llm_model.config.pad_token_id = self.tokenizer.eos_token_id
self.is_llm_loaded = True; print("LLM loaded successfully.");
self._clear_gpu_memory(); returnTrue
except Exception as e: print(f"FATAL ERROR loading LLM:
{e}\n{traceback.format_exc()}"); self._unload_llm(); self.is_llm_loaded =
False; returnFalse

def ensure_image_generator_loaded(self):
ifnotself.is_image_generator_loaded andself.image_generator isnotNone:
print("WARN: Img Gen flag inconsistent. Unloading.");
self._unload_image_generator()
ifself.is_image_generator_loaded andself.image_generator isnotNone:
returnTrue
print("--- Ensuring Image Generator is loaded ---")
ifself.is_llm_loaded: print("Unloading LLM first..."); self._unload_llm()
ifself.is_image_generator_loaded orself.image_generator isnotNone:
print("ERROR: Img Gen state inconsistent.");
self.is_image_generator_loaded = False; returnFalse
try:
print(f"Loading Image Gen: {self.image_model_path} (dtype:
{self.compute_dtype})")
            safety_args = {"safety_checker": None}
ifself.config.get("disable_safety_checker", True) else {}; print(f"Safety
checker {'DIS' if safety_args else 'EN'}ABLED.")
self.image_generator =
StableDiffusionXLPipeline.from_pretrained(self.image_model_path,
torch_dtype=self.compute_dtype, use_safetensors=True,
variant="fp16"ifself.compute_dtype == torch.float16 elseNone,
**safety_args)
ifself.device == "cuda": print("Attempting VRAM optimizations...");
self.image_generator.enable_model_cpu_offload(); print("Enabled model CPU
offload.") # Simplification for common case
elifself.device == 'cpu': self.image_generator =
self.image_generator.to(self.device)
self.is_image_generator_loaded = True; print(f"Image generator loaded on
{self.device}."); self._clear_gpu_memory(); returnTrue
```

```
except Exception as e: print(f"FATAL ERROR loading image generator:
{e}\n{traceback.format_exc()}"); self._unload_image_generator();
self.is_image_generator_loaded = False; returnFalse

def _apply_mistral_instruct_format(self, prompt: str) ->str:
return f"[INST] {prompt.strip()} [/INST]"

def _generate_llm_output(self, prompt_text: str, max_new_tokens: int,
temperature: float, top_k: int, top_p: float, retries: int = 2) ->
Optional[str]:
ifnotself.ensure_llm_loaded(): print("LLM load failed for generation.");
returnNone
        formatted_prompt =
self._apply_mistral_instruct_format(prompt_text); print(f"\n--- LLM Gen
(MaxTok: {max_new_tokens}, T: {temperature}) ---")
try: inputs = self.tokenizer(formatted_prompt, return_tensors="pt",
max_length=2048, padding=True, truncation=True,
return_attention_mask=True).to(self.device)
except Exception as tok_err: print(f"FATAL TOKENIZATION ERROR:
{tok_err}"); returnNone
        gen_text = None; last_err = None
for attempt inrange(retries):
print(f"LLM Gen Attempt {attempt + 1}/{retries}...")
try:
with torch.no_grad(): outputs =
self.llm_model.generate(input_ids=inputs["input_ids"],
attention_mask=inputs["attention_mask"], max_new_tokens=max_new_tokens,
num_return_sequences=1, temperature=temperature, do_sample=True,
top_k=top_k, top_p=top_p, pad_token_id=self.tokenizer.eos_token_id,
no_repeat_ngram_size=3)
                gen_ids = outputs[0][inputs.input_ids.shape[-1]:]; cur_out
= self.tokenizer.decode(gen_ids, skip_special_tokens=True).strip()
                cur_out =
re.sub(r'^(Okay.*?dialogue:|Here.*?dialogue:|Dialogue:|Answer:|\[/INST\])\
s*', '', cur_out, flags=re.I).strip('"`')
if cur_out andlen(cur_out.split()) >= 1and"sorry"notin cur_out.lower()
and"cannot fulfill"notin cur_out.lower(): gen_text = cur_out;
print(f"Valid output: '{gen_text[:50]}...'"); break
else: print(f"Warn: Attempt {attempt+1} invalid output ('{cur_out}').");
last_err = f"Invalid output: {cur_out}"
except Exception as e: last_err = e; print(f"ERROR LLM Gen (Attempt
{attempt+1}): {e}"); self._clear_gpu_memory(); time.sleep(1.5if attempt <
retries-1else0)
if gen_text: break
if gen_text: print("--- LLM Gen OK ---")
```

```
else: print(f"--- LLM Gen FAILED ({retries} attempts, Last Err:
{last_err}) ---")
return gen_text

def generate_image(self,
                        prompt: str,
                        num_inference_steps: int,
                        guidance_scale: float,
                        image_width: int,
                        image_height: int,
                        art_style: Optional[str] = None
                       ) -> Optional[np.ndarray]:
        ph_shape = (image_height, image_width, 3)
ifnotself.ensure_image_generator_loaded(): print("Img Gen load failed.");
return np.full(ph_shape, GENERATOR_UNAVAILABLE_PLACEHOLDER_COLOR,
dtype=np.uint8)
        is_failed_prompt = isinstance(prompt, str) and
prompt.startswith("F_") # Adjusted check
if is_failed_prompt:
print(f"Placeholder for failed prompt: {prompt}"); ph_color =
LLM_LOAD_ERROR_PLACEHOLDER_COLOR if"LLM Load Error"in prompt else
PROMPT_FAILURE_PLACEHOLDER_COLOR; return np.full(ph_shape, ph_color,
dtype=np.uint8)
ifnotisinstance(prompt, str) ornot prompt orlen(prompt.split()) <3:
print(f"Invalid/short prompt ('{prompt[:50]}...'). Placeholder."); return
np.full(ph_shape, GENERAL_ERROR_PLACEHOLDER_COLOR, dtype=np.uint8)

# Use selected art style or default if none provided
        style_suffix = art_style if art_style else", vibrant comic book
art style illustration, detailed lines, dynamic composition, cinematic
lighting"
        enhanced_prompt = prompt + style_suffix

print(f"\n--- Gen Img (Style: '{style_suffix}'): '{prompt[:100]}...' ---
"); start = time.time()
try:
            neg_prompt = self.default_settings.get("negative_prompt", "");
seed = int(time.time() * 1000) % (2**32); generator =
torch.Generator(device="cuda"ifself.device=="cuda"else"cpu").manual_seed(s
eed)
            img_out = self.image_generator(prompt=enhanced_prompt,
negative_prompt=neg_prompt, width=image_width, height=image_height,
num_inference_steps=num_inference_steps, guidance_scale=guidance_scale,
output_type="np", generator=generator).images
```

```
if img_out isNoneorlen(img_out) == 0: print("ERROR: No images returned.");
return np.full(ph_shape, GENERAL_ERROR_PLACEHOLDER_COLOR, dtype=np.uint8)
            img_arr = img_out[0]
ifisinstance(img_arr, np.ndarray):
if img_arr.dtype in [np.float32, np.float16]: img_arr = (np.clip(img_arr,
0.0, 1.0) * 255).astype(np.uint8)
elif img_arr.dtype != np.uint8: img_arr = np.clip(img_arr, 0,
255).astype(np.uint8)
if img_arr.shape != ph_shape: print(f"ERROR: Img shape mismatch
({img_arr.shape} vs {ph_shape})."); return np.full(ph_shape,
GENERAL_ERROR_PLACEHOLDER_COLOR, dtype=np.uint8)
else: print("ERROR: Output not numpy array."); return np.full(ph_shape,
GENERAL_ERROR_PLACEHOLDER_COLOR, dtype=np.uint8)
            end = time.time(); print(f"Image generated OK ({end -
start:.2f}s)."); return img_arr
except Exception as e: print(f"Error during image gen:
{e}\n{traceback.format_exc()}"); self._clear_gpu_memory(); return
np.full(ph_shape, GENERAL_ERROR_PLACEHOLDER_COLOR, dtype=np.uint8)

def _extract_characters(self, prompt: str) -> List[str]:
print("\n--- Extracting Characters ---")
ifnotself.ensure_llm_loaded():
returnself._extract_characters_fallback(prompt)
        extract_prompt = f'From: "{prompt}", list primary character
names/types (max 3). Rules: ONLY comma-separated names/types. If none,
output NONE.'
        chars = []
try:
            response = self._generate_llm_output(extract_prompt, 60, 0.2,
10, 0.9, 1)
if response isNoneor response.upper() == "NONE"orlen(response) <2: chars =
self._extract_characters_fallback(prompt)
else:
                potentials = re.split(r'[,\n]+', response);
ignore={"prompt","character","output","list",":","name","nan","none","here
","are","the","based","on","story","moral","title","narrative","no","speci
fic","spaces","not","specified","unnamed","n/a","answer","step","only","na
mes","types","max","entries"}
                f_names=[p.title() for n in potentials if
(p:=n.strip('.').strip()) and2<=len(p)<=35and p.lower() notin ignore
andnot re.search(r'\d',p) and (re.match(r'^[A-Za-z][A-Za-z\s\'\-]*[A-Za-
z]?$',p) iflen(p)>0elseFalse)]
                u_names = sorted(list(set(f_names))); g_terms =
{'People','Man','Woman','Boy','Girl','Person'}
```

```
                chars = [c for c in u_names if c notin g_terms]
iflen(u_names)>1else u_names
ifnot chars and u_names: chars = u_names # Keep generics if only option
ifnot chars: chars = self._extract_characters_fallback(prompt)
except Exception as e: print(f"LLM Char Extract Err: {e}. Fallback.");
chars = self._extract_characters_fallback(prompt)
self.story_characters = chars[:3]; print(f"Final characters:
{self.story_characters}"); returnself.story_characters

def _extract_characters_fallback(self, prompt: str) -> List[str]:
print("Executing fallback char extract...");
        potential = re.findall(r'\b(?:[A-Z][a-z]+|[A-Z]{2,})\b', prompt);
phrases = re.findall(r'\b(?:a|an|the)\s+([A-Z][a-z]+)\b', prompt, re.I);
potential.extend(phrases)

common={'City','Metropolis','Street','Building','Sky','World','Danger','St
orm','Field','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday'
,'Sunday','Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov
','Dec','Farm','Castle','Forest','Mountain','River','Ocean','Book','Portal
','Based','Prompt','Write','Short','Story','Comic','Featuring','Must','Neu
tral','Unbiased','Avoid','Stereotypes','Include','Panel','Descriptions','I
mage','Instructions','Title','Generate','Concise','Moral','Theme','Narrati
ve','Beginning','Middle','End','Only','Output','Words'}

ignore={"prompt","character","output","list",":","name","nan","none","here
","are","the","based","on","story","moral","title","narrative","no","speci
fic","spaces","not","specified","unnamed","n/a","answer","step","only","na
mes","types","max","entries"}
        potential=[p.title() for p in potential if p notin common
andlen(p)>1and p.lower() notin ignore]
        final_chars = sorted(list(set(potential)), key=len,
reverse=True)[:3] or ["Character"]; print(f"Fallback chars:
{final_chars}"); return final_chars

def generate_story_only(self, user_prompt: str) -> Optional[Dict[str,
Any]]:
ifnotself.ensure_llm_loaded(): return {"title": "[F: LLM Load]",
"storyline": "[F: LLM Load]", "moral": "[F: LLM Load]", "image_prompts":
[], "dialogues": [], "characters": []}
        res = {"title": "[F: Gen]", "storyline": "[F: Gen]", "moral": "[F:
Gen]", "image_prompts": [], "dialogues": [], "characters": []}
self.story_characters = self._extract_characters(user_prompt); char_str =
", ".join(self.story_characters) ifself.story_characters
else"character(s)"; res['characters'] = self.story_characters
```

```
        tmpl = self.prompt_templates.get("narrative_prompt_template");
narr_prompt = tmpl.format(user_prompt=user_prompt, character_str=char_str)
if tmpl elseNone
        story = self._generate_llm_output(narr_prompt, 800, 0.7, 50, 0.9,
2) if narr_prompt elseNone
if story: res["storyline"] =
re.sub(r'^\s*(Narrative:|Storyline:|Story:)\s*','',story,
flags=re.I).strip(); print("Narrative generated.")
else: res["storyline"] = "[F: LLM Narr Err]"if narr_prompt else"[F: Narr
Template Missing]"; return res
        tmpl_t = self.prompt_templates.get("title_prompt_template");
title_prompt = tmpl_t.format(story_narrative=res["storyline"][:1000]) if
tmpl_t elseNone
        title = self._generate_llm_output(title_prompt, 20, 0.5, 30, 0.9,
1) if title_prompt elseNone
if title:
res["title"]=re.sub(r'^\s*(Title:)\s*','',title,flags=re.I).strip('"`')
or"[F: Empty Title]"
else: res["title"] = "[F: LLM Title Err]"if title_prompt else"[F: Title
Template Missing]"
        tmpl_m = self.prompt_templates.get("moral_prompt_template");
moral_prompt = tmpl_m.format(story_narrative=res["storyline"][:1000]) if
tmpl_m elseNone
        moral = self._generate_llm_output(moral_prompt, 60, 0.6, 40, 0.9,
1) if moral_prompt elseNone
if moral:
res["moral"]=re.sub(r'^\s*(Moral:)\s*','',moral,flags=re.I).strip('"`')
or"[F: Empty Moral]"
else: res["moral"] = "[F: LLM Moral Err]"if moral_prompt else"[F: Moral
Template Missing]"
print("\n--- Final Story ---");
print(json.dumps({k:(v[:100]+'...'ifisinstance(v,str) andlen(v)>100else v)
for k,v in res.items() if k notin ['image_prompts', 'dialogues']},
indent=2)); print("--- End Story ---\n")
return res

def _generate_single_panel_prompt(self, panel_num: int, num_panels: int,
story_context: str, user_prompt: str, characters: List[str]) ->str:
        fail_prefix = f"F_PANEL_{panel_num}"
        tmpl = self.prompt_templates.get("panel_prompt_template")
ifnot tmpl: return f"{fail_prefix}: Template missing."
        char_str = ", ".join(characters) if characters else"character(s)";
stage = "beginning"
if num_panels >1: stage = "end"if panel_num / num_panels >0.75else
("middle"if panel_num / num_panels >0.3else"beginning")
```

```
try: p_text = tmpl.format(panel_num=panel_num, num_panels=num_panels,
user_prompt=user_prompt, character_str=char_str,
story_context=story_context, stage=stage)
except KeyError as ke: return f"{fail_prefix}: Template key err '{ke}'"
except Exception as e: return f"{fail_prefix}: Format err {e}"
        desc = self._generate_llm_output(p_text, 80, 0.6, 40, 0.9, 2)
if desc isNone: return f"{fail_prefix}: LLM call failed."
        patterns =
[r'^\s*(Panel\s*\d+\s*Vis.*?Desc.*?:|Panel\s*\d+:|Desc.*?:|Scene:|Here.*?d
esc.*?:)\s*', r'^\s*\[INST\].*?\[/INST\]\s*', r'^\s*Okay, here.*?:?\s*',
r'^\s*[\d.\-\*]+\s+(?=\w)', r'\n?Remember.*?stereotypes.*$', r'\n?---
\s*$', r'\s*Feel free.*?']
        cleaned = desc; [cleaned := re.sub(p, '', cleaned, flags=re.I |
re.S).strip() for p in patterns]; cleaned =
cleaned.replace('**','').replace('*','').strip('"`[](){}')
ifnot cleaned orlen(cleaned.split()) <4: print(f"Warn: Panel {panel_num}
prompt short after clean: '{desc}' -> '{cleaned}'"); return
f"{fail_prefix}: Invalid LLM output (short)."
return cleaned

def generate_dialogue_for_panel(self, panel_description: str, characters:
List[str], story_context: str) -> Optional[str]:
ifnotself.ensure_llm_loaded(): return"[Dial Gen F: LLM Load Err]"
if panel_description.startswith("F_"): returnNone
        tmpl = self.prompt_templates.get("dialogue_prompt_template")
ifnot tmpl: return"[Dial Gen F: Template Missing]"
        char_str = ", ".join(characters) if characters else"the
character(s)"
try: dial_prompt_txt = tmpl.format(character_str=char_str,
panel_description=panel_description, story_context=story_context)
except KeyError as ke: return f"[Dial Gen F: Template Key Err '{ke}']"
except Exception as e: return f"[Dial Gen F: Format Err {e}]"
        dialogue = self._generate_llm_output(dial_prompt_txt, 50, 0.65,
45, 0.9, 2)
if dialogue isNone: return"[Dial Gen F: LLM Call Failed]"
if dialogue.strip().upper() == "NONE": returnNone
        dialogue = re.sub(r'^\s*(Dialogue:|Line:)\s*', '', dialogue,
flags=re.I).strip('"`')
ifnot dialogue orlen(dialogue.split()) == 0: returnNone
print(f"--- Generated Dialogue: '{dialogue}' ---"); return dialogue

def populate_panel_prompts(self, story_info: Dict[str, Any], user_prompt:
str, num_panels: int, progress_callback: Optional[Callable[[str], None]] =
None) -> Optional[List[str]]:
```

```
ifnotself.ensure_llm_loaded(): print("ERR: Cannot gen prompts - LLM load
failed."); return [f"F_PANEL_{i+1}: LLM Load Error"for i
inrange(num_panels)]
        prompts = []; story = story_info.get("storyline",""); title =
story_info.get("title", user_prompt[:50]+"...")
self.story_characters = story_info.get('characters')
ifnotself.story_characters: print("Warn: Chars missing, re-extracting.");
self.story_characters = self._extract_characters(user_prompt + ("
"+story[:300] if story else"")); story_info['characters'] =
self.story_characters
ifnot story or story.startswith("[F:"): print("ERR: Cannot gen prompts -
Story missing/failed."); return [f"F_PANEL_{i+1}: Story Missing/Failed"for
i inrange(num_panels)]
print(f"\n>>> Populating {num_panels} panel prompts."); ctx = f"Title:
{title}. Story: {story}"; ctx_snip = ctx[:1500] +
("..."iflen(ctx)>1500else"")
for i inrange(num_panels):
            pn = i+1
if progress_callback: progress_callback(desc=f"Gen prompt
{pn}/{num_panels}...")
print(f"--- Gen Prompt Panel {pn}/{num_panels} ---");
            desc = self._generate_single_panel_prompt(pn, num_panels,
ctx_snip, user_prompt, self.story_characters); prompts.append(desc)
print(f">>> Finished generating {len(prompts)} prompts.");
story_info['image_prompts'] = prompts; return prompts

def _load_config(self, config_path: str) -> Dict[str, Any]:
print(f"Loading config: {config_path}");
default={"story_model_path":"mistralai/Mistral-7B-Instruct-
v0.2","image_model_path":"stabilityai/stable-diffusion-xl-base-
1.0","disable_safety_checker":True,"default_settings":{"temperature":0.7,"
num_inference_steps":30,"guidance_scale":7.0,"image_width":1024,"image_hei
ght":768,"negative_prompt":"..."},"prompt_template":{"narrative_prompt_tem
plate":"[DEF]","title_prompt_template":"[DEF]","moral_prompt_template":"[D
EF]","panel_prompt_template":"[DEF]","dialogue_prompt_template":"[DEF]"}}
        default["default_settings"]["negative_prompt"] = "text, words,
letters, signature, watermark, blurry, deformed, bad anatomy, disfigured,
poorly drawn face, mutation, mutated, extra limb, ugly, poorly drawn
hands, missing limb, blurry, floating limbs, disconnected limbs, malformed
hands, out of focus, long neck, long body, duplicate, cropped, low
quality, jpeg artifacts, multiple panels, frame, border, username, artist
name, error, glitch, noise, tiling, stereotype, caricature, biased
depiction, duplicate characters, multiple instances of same character,
clones, multiple views, multiple views of the same character, grid,
multiple images, multiple scenes"
```

```
def merge(d, u): m=d.copy(); [m.update({k: merge(m[k],v)})
ifisinstance(v,dict) and k in m andisinstance(m[k],dict) else
m.update({k:v}) for k,v in u.items() if v isnotNone]; return m
try:
if Path(config_path).is_file():
withopen(config_path,'r',encoding='utf-8') as f: user_cfg=json.load(f)
ifisinstance(user_cfg, dict): print("Merging user config...");
merged_cfg=merge(default, user_cfg); [merged_cfg.setdefault(k,v) for k,v
in default.items()]; [merged_cfg[sk].setdefault(k,v) for sk in
["default_settings","prompt_template"] for k,v in default[sk].items()];
print("Config loaded/merged."); return merged_cfg
else: print("Warn: Config invalid JSON dict. Defaults used.")
else: print(f"Warn: Config not found '{config_path}'. Defaults used.")
except Exception as e: print(f"Error loading config '{config_path}': {e}.
Defaults used.")
print("Returning default config."); return default
```

c) panel_layout.py

- **Purpose:** Takes the generated panel images and metadata (title, dialogues) and creates the final comic page layout image.
- **Key Functionality:**
  - Defines layout parameters (padding, colors, border widths, title area height).
  - Loads fonts using PIL (_load_pil_font), searching common system paths and providing a basic fallback. Used for title and panel numbers.
  - Calculates the overall comic page dimensions based on the number of panels, panel size, and padding (_calculate_dimensions).
  - Uses OpenCV (cv2) to draw dialogue bubbles (currently simple rectangles with tails at fixed positions) and wrap/render dialogue text within them (_draw_dialogue_bubble_cv).
  - Uses PIL (ImageDraw) to render the main title and panel numbers onto the final image.
  - Combines individual panel images onto a background canvas according to a grid layout (e.g., 2x2 for 4 panels, 2x3 for 5-6 panels, etc.).
  - Saves the final composed image (create_comic_page).

Code:

```
import os
import numpy as np
import cv2
```

```
from PIL import Image, ImageFont
from typing import List, Dict, Any, Optional, Tuple
import textwrap
import traceback
from PIL import ImageDraw


class ImprovedPanelLayout:
"""
    Handles the layout and creation of the final comic page image from
individual panels,
    using OpenCV for drawing dialogues and bubbles (at fixed positions).
    """

def __init__(self, output_dir: str = "./outputs"):
# Layout parameters
self.grid_rows = 2
self.grid_cols = 3
self.panel_width = 1024
self.panel_height = 768
self.padding = 25
self.title_height = 100
self.background_color = (255, 255, 255) # White (BGR for OpenCV later)
self.border_color = (0, 0, 0) # Black
self.border_width = 3
self.number_background_color = (255, 255, 255, 190) # Keep for PIL number
bg
self.number_text_color = (0, 0, 0) # Keep for PIL number text

# --- Font loading (Use PIL for title, OpenCV has limited built-in fonts)
---
self.font_search_paths = [".", "/usr/share/fonts/truetype/msttcorefonts/",
"/usr/share/fonts/truetype/dejavu/", "/Library/Fonts/",
"C:/Windows/Fonts/"]
self.title_font = self._load_pil_font(["arialbd.ttf", "DejaVuSans-
Bold.ttf"], 60) # PIL font for title
self.panel_num_font = self._load_pil_font(["arial.ttf", "DejaVuSans.ttf"],
36) # PIL font for panel numbers

# --- OpenCV Font Settings ---
self.cv_font_face = cv2.FONT_HERSHEY_SIMPLEX
self.cv_font_scale = 0.8# Adjust as needed
self.cv_font_thickness = 2
self.cv_text_color = (0, 0, 0) # Black (BGR)
self.cv_bubble_color = (255, 255, 255) # White (BGR)
```

```
self.cv_bubble_outline_color = (0, 0, 0) # Black (BGR)
self.cv_bubble_padding = 10

self._calculate_dimensions()
self.output_dir = output_dir
        os.makedirs(self.output_dir, exist_ok=True)
ifself.title_font isNone: print("CRITICAL WARNING: Could not load any PIL
title font!")
ifself.panel_num_font isNone: print("CRITICAL WARNING: Could not load any
PIL panel number font!")

def _find_font_file(self, font_filenames: List[str]) -> Optional[str]:
for filename in font_filenames:
if os.path.exists(filename): return filename
for path_dir inself.font_search_paths:
                full_path = os.path.join(path_dir, filename)
if os.path.exists(full_path): print(f"Font found: {full_path}"); return
full_path
print(f"Font files not found: {font_filenames}"); returnNone

def _load_pil_font(self, font_filenames: List[str], font_size: int) ->
Optional[ImageFont.FreeTypeFont]:
        font_path = self._find_font_file(font_filenames)
if font_path:
try: return ImageFont.truetype(font_path, font_size)
except Exception as e: print(f"Warning: Error loading PIL font
'{font_path}'. Err: {e}")
print(f"Warning: Could not load PIL fonts: {font_filenames}. Trying
default.");
try: return ImageFont.load_default() # Basic fallback
except Exception as e_def: print(f"CRITICAL ERROR: Could not load default
PIL font. Error: {e_def}"); returnNone

def _calculate_dimensions(self):
self.comic_width = self.grid_cols * (self.panel_width + self.padding) +
self.padding
        total_panel_height = self.grid_rows * (self.panel_height +
self.padding) + self.padding
self.comic_height = total_panel_height + self.title_height
print(f"Calculated comic dimensions:
{self.comic_width}x{self.comic_height}")

def update_layout(self, grid_rows: int, grid_cols: int, panel_width: int,
panel_height: int):
```

```
print(f"Updating layout: R={grid_rows}, C={grid_cols},
Panel={panel_width}x{panel_height}")
self.grid_rows = max(1, grid_rows); self.grid_cols = max(1, grid_cols);
self.panel_width = panel_width; self.panel_height = panel_height;
self.padding = max(5, self.padding); self.title_height = max(30,
self.title_height); self._calculate_dimensions()

def _draw_dialogue_bubble_cv(self, image: np.ndarray, dialogue: str,
panel_x: int, panel_y: int):
"""Draws dialogue bubble and text using OpenCV at a FIXED position."""
ifnot dialogue ornotisinstance(dialogue, str): return# Skip if no dialogue

# --- Configuration for fixed position (Top Center) ---
        bubble_max_width_ratio = 0.7# Max width relative to panel width
        target_y_offset = 30# Fixed distance from top panel border
        line_height_factor = 1.5# Multiplier for line spacing based on
font size

# --- Text Wrapping (Manual for OpenCV) ---
        max_bubble_width_px = int(self.panel_width *
bubble_max_width_ratio)
        wrapped_lines = []
        words = dialogue.split(' ')
        current_line = ""
for word in words:
            test_line = f"{current_line} {word}".strip()
            (text_width, text_height), _ = cv2.getTextSize(test_line,
self.cv_font_face, self.cv_font_scale, self.cv_font_thickness)
if text_width <= max_bubble_width_px:
                current_line = test_line
else:
# Word doesn't fit, push current line and start new one
if current_line: wrapped_lines.append(current_line)
# Handle very long words that might exceed max width alone
if cv2.getTextSize(word, self.cv_font_face, self.cv_font_scale,
self.cv_font_thickness)[0][0] > max_bubble_width_px:
# Simple truncation for very long words (can be improved)
                    wrapped_lines.append(word[:int(len(word)*0.8)] +
"...")
                    current_line = ""
else:
                    current_line = word
if current_line: wrapped_lines.append(current_line) # Add last line

ifnot wrapped_lines: return# No lines to draw
```

```
# --- Calculate Bubble Dimensions ---
        max_line_width = 0
        total_text_height = 0
        (base_width, base_height), baseline = cv2.getTextSize("Tg",
self.cv_font_face, self.cv_font_scale, self.cv_font_thickness) # Approx
height
        line_spacing = int(base_height * line_height_factor)

for line in wrapped_lines:
            (w, h), _ = cv2.getTextSize(line, self.cv_font_face,
self.cv_font_scale, self.cv_font_thickness)
            max_line_width = max(max_line_width, w)
        total_text_height = len(wrapped_lines) * line_spacing -
(line_spacing - base_height) # Adjust for last line

        bubble_width = max_line_width + 2 * self.cv_bubble_padding
        bubble_height = total_text_height + 2 * self.cv_bubble_padding

# --- Calculate Bubble Position (Fixed Top-Center) ---
        bubble_x = (self.panel_width - bubble_width) // 2
        bubble_y = target_y_offset
        bubble_x += panel_x # Adjust relative to panel origin
        bubble_y += panel_y # Adjust relative to panel origin

# Bubble tail (simple triangle pointing down from middle-bottom) - Fixed
position
        tail_base_x = bubble_x + bubble_width // 2
        tail_base_y = bubble_y + bubble_height
        tail_tip_x = tail_base_x
        tail_tip_y = tail_base_y + 20# Length of tail
        tail_width = 15# Width of tail base

# --- Draw Bubble Background & Outline ---
        cv2.rectangle(image, (bubble_x, bubble_y), (bubble_x +
bubble_width, bubble_y + bubble_height), self.cv_bubble_color, -1) #
Filled background
        cv2.rectangle(image, (bubble_x, bubble_y), (bubble_x +
bubble_width, bubble_y + bubble_height), self.cv_bubble_outline_color,
self.cv_font_thickness) # Outline

# --- Draw Bubble Tail ---
        pts = np.array([[tail_base_x - tail_width // 2, tail_base_y],
                        [tail_base_x + tail_width // 2, tail_base_y],
                        [tail_tip_x, tail_tip_y]], np.int32)
```

```
        pts = pts.reshape((-1, 1, 2))
        cv2.drawContours(image, [pts], 0, self.cv_bubble_color, -1) # Fill
tail
        cv2.line(image, (tail_base_x - tail_width//2, tail_base_y),
(tail_tip_x, tail_tip_y), self.cv_bubble_outline_color,
self.cv_font_thickness)
        cv2.line(image, (tail_base_x + tail_width//2, tail_base_y),
(tail_tip_x, tail_tip_y), self.cv_bubble_outline_color,
self.cv_font_thickness)


# --- Draw Text Lines ---
        current_y = bubble_y + self.cv_bubble_padding + base_height #
Start y for first line (approx baseline)
for line in wrapped_lines:
# Center each line horizontally within the bubble
            (w, h), _ = cv2.getTextSize(line, self.cv_font_face,
self.cv_font_scale, self.cv_font_thickness)
            line_x = bubble_x + (bubble_width - w) // 2
            cv2.putText(image, line, (line_x, current_y),
self.cv_font_face, self.cv_font_scale, self.cv_text_color,
self.cv_font_thickness, cv2.LINE_AA)
            current_y += line_spacing


def create_comic_layout(self,
                            panel_images: List[np.ndarray],
                            dialogues: List[Optional[str]], # Add dialogues
parameter
                            title: Optional[str] = "Comic Story") ->
Optional[np.ndarray]:
"""
        Creates comic layout using PIL for base and title, then OpenCV for
panels & dialogue bubbles.
        """
ifnot panel_images: print("Error: No panel images provided."); returnNone
ifnotall(isinstance(img, np.ndarray) for img in panel_images):
print("Error: panel_images list contains non-numpy elements."); returnNone
iflen(panel_images) != len(dialogues): print(f"Warning: Mismatch between
images ({len(panel_images)}) and dialogues ({len(dialogues)}). Some
dialogues may be ignored.");

        ref_h, ref_w, ref_c = panel_images[0].shape
if ref_c != 3: print(f"Error: Panel image channel count ({ref_c}) != 3.");
returnNone
```

```
if ref_h != self.panel_height or ref_w != self.panel_width:
print(f"Warning: Panel img dims ({ref_w}x{ref_h}) != layout
({self.panel_width}x{self.panel_height}). Adjusting layout.");
self.panel_width = ref_w; self.panel_height = ref_h;
self._calculate_dimensions()

# --- Create Base Canvas with PIL (Easier for Title Font Handling) ---
try:
            comic_pil = Image.new('RGB', (self.comic_width,
self.comic_height), color=self.background_color)
            draw_pil = ImageDraw.Draw(comic_pil)
except Exception as e: print(f"Error creating PIL canvas: {e}");
returnNone

# --- Add Title using PIL ---
        display_title = title or"Generated Comic"
ifself.title_font:
try:
                bbox = draw_pil.textbbox((0, 0), display_title,
font=self.title_font)
                tw = bbox[2] - bbox[0]; th = bbox[3] - bbox[1]
                tx = (self.comic_width - tw) // 2; ty = max(5,
(self.title_height - th) // 2)
                draw_pil.text((tx, ty), display_title,
font=self.title_font, fill=(0, 0, 0))
except Exception as e_title: print(f"Error drawing PIL title: {e_title}.
Skipping.")
else: print("Warning: Title font not loaded. Skipping title.")

# --- Convert PIL canvas to OpenCV format (NumPy array BGR) ---
try:
# Convert PIL RGB to NumPy array
            comic_cv = np.array(comic_pil)
# Convert RGB to BGR for OpenCV functions
            comic_cv = cv2.cvtColor(comic_cv, cv2.COLOR_RGB2BGR)
except Exception as e_conv:
print(f"Error converting PIL canvas to OpenCV format: {e_conv}")
returnNone

# --- Add Panels and Dialogue using OpenCV ---
        num_panels_to_draw = min(len(panel_images), self.grid_rows *
self.grid_cols)
print(f"Layout grid: {self.grid_rows}x{self.grid_cols}. Drawing
{num_panels_to_draw} panels with OpenCV.")
```

```
for i inrange(num_panels_to_draw):
            panel_array_rgb = panel_images[i] # Assume RGB from generator
            dialogue = dialogues[i] if i <len(dialogues) elseNone

# Convert panel to BGR for OpenCV processing
try:
                panel_array_bgr = cv2.cvtColor(panel_array_rgb,
cv2.COLOR_RGB2BGR)
except Exception as e_panel_conv:
print(f"Error converting panel {i+1} to BGR: {e_panel_conv}. Skipping.")
continue

            row = i // self.grid_cols; col = i % self.grid_cols
            x = col * (self.panel_width + self.padding) + self.padding
            y = row * (self.panel_height + self.padding) + self.padding +
self.title_height

# Draw border (OpenCV rectangle)
ifself.border_width >0:
                cv2.rectangle(comic_cv, (x - self.border_width, y -
self.border_width),
                              (x + self.panel_width + self.border_width,
y + self.panel_height + self.border_width),
self.border_color, self.border_width)

# Paste panel (roi assignment)
try:
# Ensure panel fits exactly, resize if necessary (shouldn't be needed
ideally)
if panel_array_bgr.shape[0] != self.panel_height or
panel_array_bgr.shape[1] != self.panel_width:
print(f"Warning: Resizing panel {i+1} in OpenCV paste.")
                    panel_array_bgr = cv2.resize(panel_array_bgr,
(self.panel_width, self.panel_height), interpolation=cv2.INTER_LANCZOS4)

                comic_cv[y:y + self.panel_height, x:x + self.panel_width]
= panel_array_bgr
except Exception as paste_err:
print(f"Error pasting panel {i+1} with OpenCV: {paste_err}")
continue# Skip dialogue/number if paste fails

# --- Draw Dialogue Bubble (using OpenCV) ---
# IMPORTANT: This draws at a FIXED position due to lack of object
detection
try:
```

```
self._draw_dialogue_bubble_cv(comic_cv, dialogue, x, y)
except Exception as e_bubble:
print(f"Error drawing CV dialogue bubble for panel {i+1}: {e_bubble}")


# --- Add Panel Number (using PIL on the final image - easier font
handling) ---

# --- Convert final OpenCV BGR image back to RGB PIL Image for numbering &
return ---
try:
            final_comic_bgr = comic_cv
            final_comic_rgb = cv2.cvtColor(final_comic_bgr,
cv2.COLOR_BGR2RGB)
            final_comic_pil = Image.fromarray(final_comic_rgb)
            draw_final = ImageDraw.Draw(final_comic_pil) # Draw object for
final PIL image

# --- Add Panel Numbers using PIL (easier fonts/background) ---
for i inrange(num_panels_to_draw):
ifself.panel_num_font:
                    panel_num_text = str(i + 1)
                    row = i // self.grid_cols; col = i % self.grid_cols
                    x = col * (self.panel_width + self.padding) +
self.padding
                    y = row * (self.panel_height + self.padding) +
self.padding + self.title_height
                    num_padding = 8
try:
                         num_bbox = draw_final.textbbox((0,0),
panel_num_text, font=self.panel_num_font)
                         num_w = num_bbox[2]-num_bbox[0]; num_h =
num_bbox[3]-num_bbox[1]
                         rect_x0=x+num_padding//2;
rect_y0=y+num_padding//2
                         rect_x1=rect_x0+num_w+num_padding;
rect_y1=rect_y0+num_h+num_padding
# Need alpha channel for transparency - create temp layer
                         num_bg_layer = Image.new('RGBA',
final_comic_pil.size, (0,0,0,0))
                         draw_num_bg = ImageDraw.Draw(num_bg_layer)

draw_num_bg.rectangle([(rect_x0,rect_y0),(rect_x1,rect_y1)],
fill=self.number_background_color)
```

```
                            final_comic_pil.paste(num_bg_layer, (0,0),
num_bg_layer) # Paste with alpha

# Redraw draw object on updated image
                            draw_final = ImageDraw.Draw(final_comic_pil)
                            text_x = rect_x0+num_padding//2; text_y =
rect_y0+num_padding//2 - num_bbox[1] # Adjust y based on bbox top
                            draw_final.text((text_x, text_y),
panel_num_text, font=self.panel_num_font, fill=self.number_text_color)
except Exception as e_num: print(f"Error drawing panel number {i+1} with
PIL: {e_num}")
else: print(f"Skipping panel number {i+1} - PIL font failed.")

# Convert final PIL image back to NumPy array for Gradio output
            final_comic_array = np.array(final_comic_pil)
print("Comic layout created successfully using OpenCV for bubbles.")
return final_comic_array
except Exception as e:
print(f"Error during final conversion or numbering: {e}")
returnNone


def save_comic(self, comic_array: np.ndarray, filename: str) ->
Optional[str]:
""" Saves the comic layout numpy array (RGB) to an image file."""
ifnotisinstance(comic_array, np.ndarray): print("Error: Invalid data to
save_comic."); returnNone
if'.'notin filename: filename += ".png"
        file_path = os.path.join(self.output_dir, filename)
try:
# Assuming comic_array is RGB from the final conversion step
            comic_img = Image.fromarray(comic_array)
            comic_img.save(file_path)
print(f"Comic layout saved to: {file_path}")
return file_path
except Exception as e: print(f"Error saving comic image to {file_path}:
{e}\n{traceback.format_exc()}"); returnNone
```

d) main.py

- **Purpose:** Sets up and runs the Gradio web interface, acting as the bridge between the user and the backend AI/layout logic.
- **Key Functionality:**

- Initializes ComicCrafterAI and PanelLayout instances, handling potential initialization errors (e.g., model loading failure, OpenCV not found).
- Defines the Gradio UI layout: input text box, art style dropdown, advanced settings sliders (panel count, image size, quality), generate button, and output tabs (layout, story details, downloads).
- Defines the main generate_comic function triggered by the button press. This function:
  - Takes user inputs from the UI.
  - Calls comic_crafter.generate_comic_elements to get story data and panel images.
  - Calls panel_layout.create_comic_page (if enabled and available) to create the final layout.
  - Handles displaying results (images, text) and progress updates in the Gradio interface.
  - Saves output files (layout image, story details markdown) to the ./outputs directory.
  - Includes logic to check if generated images are placeholders (is_placeholder_image) and adjust layout accordingly.

Code:

```
# main.py
import os
import json
import numpy as np
import gradio as gr
from typing import List, Dict, Any, Tuple, Optional
from pathlib import Path
import time
import uuid
from PIL import Image
import traceback
import re
from PIL import ImageDraw

from comic_crafter_ai import ImprovedComicCrafterAI
from panel_layout import ImprovedPanelLayout

# --- Constants (Ensure these match comic_crafter_ai.py) ---
PROMPT_FAILURE_PLACEHOLDER_COLOR = (200, 150, 150)
GENERATOR_UNAVAILABLE_PLACEHOLDER_COLOR = (150, 150, 180)
NSFW_PLACEHOLDER_COLOR = (255, 150, 150)
GENERAL_ERROR_PLACEHOLDER_COLOR = (220, 200, 200)
OTHER_FAILURE_PLACEHOLDER_COLOR = (200, 200, 220)
LLM_LOAD_ERROR_PLACEHOLDER_COLOR = (180, 180, 220)
```

```
# --- End Constants ---

OUTPUT_DIR = Path("./outputs")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

config_path = "config.json"
comic_crafter = None
panel_layout = None
initialization_errors = []

# --- Art Styles ---
# Define available art styles (key: display name, value: prompt suffix)
ART_STYLES = {
"Comic Book (Default)": ", vibrant comic book art style illustration,
detailed lines, dynamic composition, cinematic lighting",
"Anime / Manga": ", anime art style, vibrant colors, manga illustration",
"Fantasy Art": ", digital fantasy art, epic composition, detailed
painting, cinematic lighting",
"Sci-Fi Concept Art": ", sci-fi concept art, futuristic design, detailed
digital painting",
"Pixel Art": ", pixel art style, 16-bit, retro game art",
"Minimalist Line Art": ", minimalist line art, simple, clean lines, black
and white",
"Watercolor": ", watercolor painting style, soft edges, blended colors",
"No Specific Style": ""# Allows using only the panel description
}

# --- Initialization ---
try:
print("Initializing ComicCrafterAI..."); start_init = time.time()
    comic_crafter = ImprovedComicCrafterAI(config_path); end_init =
time.time()
print(f"ComicCrafterAI initialized ({end_init - start_init:.2f}s).")
if comic_crafter isNoneornot comic_crafter.is_llm_loaded:
initialization_errors.append("WARN: LLM failed initial load.")
except Exception as e: error_msg = f"FATAL ERROR initializing
ComicCrafterAI: {e}\n{traceback.format_exc()}"; print(error_msg);
initialization_errors.append(error_msg); comic_crafter = None
try:
print("Initializing PanelLayout (with OpenCV)...")
    panel_layout = ImprovedPanelLayout(output_dir=str(OUTPUT_DIR)) # Now
uses OpenCV internally
print("PanelLayout initialized.")
```

```
except ImportError: initialization_errors.append("ERROR: Failed to import
OpenCV (cv2). Panel Layout disabled. Install with 'pip install opencv-
python'"); panel_layout = None
except Exception as e: error_msg = f"ERROR initializing PanelLayout: {e}";
print(error_msg); initialization_errors.append(error_msg); panel_layout =
None


# --- Placeholder Check (Same as before) ---
def is_placeholder_image(img_array: Optional[np.ndarray]) -> Tuple[bool,
str]:
ifnotisinstance(img_array, np.ndarray) or img_array.ndim != 3or
img_array.shape[2] != 3:
returnFalse, "Invalid data"
try:
        h, w, _ = img_array.shape; points = [(h // 2, w // 2), (h // 4, w
// 4), (3 * h // 4, 3 * w // 4)]
        first_clr = None; all_match = True
for r, c in points:
          clr = tuple(img_array[r, c, :]);
if first_clr isNone: first_clr = clr
elif clr != first_clr: all_match = False; break
ifnot all_match or first_clr isNone: returnFalse, "Not solid color"
        ph_colors = {tuple(PROMPT_FAILURE_PLACEHOLDER_COLOR),
tuple(GENERATOR_UNAVAILABLE_PLACEHOLDER_COLOR),
tuple(NSFW_PLACEHOLDER_COLOR), tuple(GENERAL_ERROR_PLACEHOLDER_COLOR),
tuple(OTHER_FAILURE_PLACEHOLDER_COLOR),
tuple(LLM_LOAD_ERROR_PLACEHOLDER_COLOR)}
if first_clr == tuple(PROMPT_FAILURE_PLACEHOLDER_COLOR): returnTrue,
"Prompt Failed"
if first_clr == tuple(GENERATOR_UNAVAILABLE_PLACEHOLDER_COLOR):
returnTrue, "Generator Unavailable"
if first_clr == tuple(NSFW_PLACEHOLDER_COLOR): returnTrue, "NSFW Detected"
if first_clr == tuple(GENERAL_ERROR_PLACEHOLDER_COLOR): returnTrue,
"General Error"
if first_clr == tuple(OTHER_FAILURE_PLACEHOLDER_COLOR): returnTrue, "Other
Failure"
if first_clr == tuple(LLM_LOAD_ERROR_PLACEHOLDER_COLOR): returnTrue, "LLM
Load Error"
returnFalse, "Solid color (unknown)"
except Exception: returnFalse, "Check error"


# --- Main Generation Function (Adjusted to return PIL Image) ---
def generate_comic(
```

```
    prompt: str,
    art_style_key: str,
    num_panels: int = 6,
    panel_width: int = 1024, panel_height: int = 768,
    num_steps: int = 30, guidance_scale: float = 7.0,
    create_layout: bool = True,
    progress=gr.Progress(track_tqdm=True)
) -> Tuple[Optional[Image.Image], Optional[str], str]: # Return type
changed to Optional[Image.Image]
if comic_crafter isNone: returnNone, None, "FATAL ERROR: Comic Crafter AI
not initialized."
ifnot comic_crafter.is_llm_loaded:
print("LLM not loaded. Reloading..."); comic_crafter.ensure_llm_loaded();
ifnot comic_crafter.is_llm_loaded: returnNone, None, "ERROR: LLM failed to
load."
    layout_available = panel_layout isnotNone
if create_layout andnot layout_available: print("Warn: Panel Layout
unavailable."); create_layout = False

# Get the art style suffix from the selected key
    selected_art_style = ART_STYLES.get(art_style_key, ART_STYLES["Comic
Book (Default)"])
print(f"Selected Art Style Suffix: '{selected_art_style}'")

    progress(0, desc="Initializing..."); start_time = time.time();
unique_id = uuid.uuid4().hex[:8]; timestamp = time.strftime("%Y%m%d-
%H%M%S"); output_prefix = f"{timestamp}_{unique_id}"
# Initialize comic_output_image as None
    comic_output_image: Optional[Image.Image] = None
    story_path, story_display_text = None, "Generation started..."
# Progress stages adjusted for dialogue
    p_story_end=0.20; p_prompt_start=p_story_end; p_prompt_end=0.50;
p_dialogue_start=p_prompt_end; p_dialogue_end=0.65;
p_image_start=p_dialogue_end; p_image_end=0.90; p_save_start=p_image_end;
p_layout_start=0.95; p_finish=1.0
try:
        progress(0.05, desc="Generating story elements..."); story_info =
comic_crafter.generate_story_only(prompt)
if story_info isNoneor story_info.get("storyline", "").startswith("[F:"):
          error_msg = story_info.get("storyline", "[Story Gen F:
Unknown]") if story_info else"[Story Gen F: None]";
print(f"Crit Fail: Story failed: {error_msg}");
          story_display_text = f"## Story Gen Failed\n**Reason:**
{error_msg}";
```

```
if comic_crafter: comic_crafter._clear_gpu_memory(); returnNone, None,
story_display_text
        progress(p_story_end, desc="Story elements generated.");
story_info['dialogues'] = []
        progress(p_prompt_start, desc="Generating panel prompts...")
def panel_prog_cb(desc=""):
          prog=p_prompt_start; m=re.search(r'(\d+)/(\d+)',desc);
if m:
try:
              c,t=int(m.group(1)),int(m.group(2));
prog=p_prompt_start+(p_prompt_end-p_prompt_start)*(c/t) if t>0else
p_prompt_start;
except ValueError: pass;
          progress(max(p_prompt_start, min(prog,p_prompt_end)), desc=desc)
        image_prompts = comic_crafter.populate_panel_prompts(story_info,
prompt, num_panels, progress_callback=panel_prog_cb)
if image_prompts isNone: print("Crit Err: populate_panel_prompts returned
None."); image_prompts = [f"F_PANEL_{i+1}: Crit LLM Err"for i
inrange(num_panels)]
        story_info['image_prompts'] = image_prompts; num_prompt_failed =
sum(1for p in image_prompts ifisinstance(p,str) and p.startswith("F_"));
print(f"Prompt gen finished. Total: {len(image_prompts)}, Failed:
{num_prompt_failed}"); progress(p_prompt_end, desc=f"Prompts finished
({num_prompt_failed} failed).")
        progress(p_dialogue_start, desc="Generating dialogue...")
        dialogues = []
if num_panels >0:
             story_ctx = f"Title: {story_info.get('title','')}. Story:
{story_info.get('storyline','')}"; ctx_snip = story_ctx[:1500] +
("..."iflen(story_ctx)>1500else"")
for i inrange(num_panels):
                 cur_prompt = image_prompts[i]; prog = p_dialogue_start +
(p_dialogue_end-p_dialogue_start)*((i+1)/num_panels); progress(min(prog,
p_dialogue_end), desc=f"Gen dialogue {i+1}/{num_panels}...")
                 dialogue = None;
ifisinstance(cur_prompt, str) andnot cur_prompt.startswith("F_"): dialogue
= comic_crafter.generate_dialogue_for_panel(cur_prompt,
story_info.get('characters',[]), ctx_snip)
else: print(f"Skip dialogue for failed panel {i+1}")
                 dialogues.append(dialogue)
        story_info['dialogues'] = dialogues; num_dial_failed = sum(1for d
in dialogues ifisinstance(d,str) and d.startswith("[Dial Gen F"));
num_no_dial = sum(1for d in dialogues if d isNone); print(f"Dialogue gen
finished. Fails: {num_dial_failed}, None: {num_no_dial}");
progress(p_dialogue_end, desc="Dialogue finished.")
```

```
# --- MODIFIED: Simplified Story Display Text ---
        story_display_text = f"# {story_info.get('title', '[Title
N/A]')}\n\n"
for k, t in [("storyline", "Storyline"), ("moral", "Moral")]:
            cont = story_info.get(k)
            disp = f"[{t} N/A]"
if cont andisinstance(cont, str):
                disp = cont ifnot cont.startswith("[F:") else
f"**{cont}**"
            story_display_text += f"## {t}\n{disp}\n\n"

        progress(p_image_start, desc="Starting image generation...")
        panel_images: List[Optional[np.ndarray]] = []
        num_prompts = len(image_prompts)
if num_prompts >0:
for i inrange(num_prompts):
                panel_prompt = image_prompts[i]; prog = p_image_start +
(p_image_end - p_image_start)*((i+1)/num_prompts); desc = f"Gen image
{i+1}/{num_prompts}"
ifisinstance(panel_prompt, str) and panel_prompt.startswith("F_"): desc +=
" (Placeholder)";
                progress(min(prog, p_image_end), desc=desc)
# Pass the selected art style to the image generator
                panel_img_arr = comic_crafter.generate_image(
                    panel_prompt, num_steps, guidance_scale, panel_width,
panel_height, art_style=selected_art_style
                );
                panel_images.append(panel_img_arr)
else: print("No prompts for image gen.")
        num_imgs = len(panel_images); num_phs = sum(1for img in
panel_images if is_placeholder_image(img)[0]); print(f"Image gen finished.
Imgs: {num_imgs}, Placeholders: {num_phs}"); progress(p_image_end,
desc="Images finished. Saving...")

        progress(p_save_start, desc="Saving story details...");
story_fname = f"{output_prefix}_story_details.md"; story_path_obj =
OUTPUT_DIR / story_fname
# Save the *simplified* story details to the markdown file as well
try:
withopen(story_path_obj, 'w', encoding='utf-8') as f:
            f.write(story_display_text); story_path = str(story_path_obj);
print(f"Story saved: {story_path}")
except Exception as e:
print(f"Error saving story MD: {e}"); story_path = None
```

```
        comic_path = None; valid_layout_images = [];
corresponding_dialogues = []
for idx, img inenumerate(panel_images):
if img isnotNoneandnot is_placeholder_image(img)[0]:
valid_layout_images.append(img);
corresponding_dialogues.append(story_info['dialogues'][idx] if idx
<len(story_info['dialogues']) elseNone)
if create_layout and layout_available and valid_layout_images:
            progress(p_layout_start, desc="Creating layout image...");
num_valid = len(valid_layout_images); print(f"Layout with {num_valid}
valid images.")
if num_valid <= 0: rows, cols = 0, 0
elif num_valid == 1: rows, cols = 1, 1
elif num_valid == 2: rows, cols = 1, 2
elif num_valid == 3: rows, cols = 1, 3
elif num_valid == 4: rows, cols = 2, 2
elif num_valid <= 6: rows, cols = 2, (num_valid+1)//2
elif num_valid <= 9: rows, cols = 3, (num_valid+2)//3
else: cols=4; rows=(num_valid+3)//4
if rows >0and cols >0:
                actual_h, actual_w, _ = valid_layout_images[0].shape;
panel_layout.update_layout(rows, cols, actual_w, actual_h);
                layout_title = story_info.get('title', 'Generated
Comic');
if layout_title.startswith('['): layout_title = 'Generated Comic'
# Pass dialogues to OpenCV layout function
                comic_array =
panel_layout.create_comic_layout(panel_images=valid_layout_images,
dialogues=corresponding_dialogues, title=layout_title)
if comic_array isnotNone:
# --- CHANGE: Load image from array into PIL instead of saving/reloading -
--
try:
                        comic_output_image = Image.fromarray(comic_array)
print("Comic layout generated as PIL Image.")
# Optional: Still save a copy to disk
try:
                            comic_fname =
f"{output_prefix}_comic_layout.png"
                            save_path =
panel_layout.save_comic(comic_array, comic_fname)
if save_path: print(f"Layout also saved to: {save_path}")
else: print("Failed to save layout copy.")
```

```
except Exception as save_err: print(f"Error saving layout copy:
{save_err}")
except Exception as pil_err:
print(f"Error converting layout array to PIL Image: {pil_err}")
                        comic_output_image = None
else: print("Failed create layout array.")
else: print("Skip layout: No valid images or zero rows/cols.")
elif create_layout: print(f"Skip layout: {'Component unavailable' if not
layout_available else 'No valid images'}.")
        progress(p_finish, desc="Finished!"); end_time = time.time();
print(f"Total time: {end_time - start_time:.2f}s")
print("Run finished. Cleanup...")
if comic_crafter: comic_crafter._clear_gpu_memory()
# --- CHANGE: Return the PIL image object ---
return comic_output_image, story_path, story_display_text
except Exception as e:
      error_message = f"FATAL Error during comic gen:
{e}\n\n{traceback.format_exc()}"; print(error_message); print("Error
occurred. Cleanup...");
if comic_crafter: comic_crafter._clear_gpu_memory(); returnNone, None,
f"## Gen Failed\n\n**Error:**\n```\n{error_message}\n```"

# --- UI Function (Updated Image component type) ---
def create_ui():
with gr.Blocks(theme=gr.themes.Soft(primary_hue=gr.themes.colors.blue,
secondary_hue=gr.themes.colors.sky), title="Comic Crafter AI") as
interface:
        gr.Markdown("# Comic Crafter AI)\nGenerate stories & visuals!")

        gr.Markdown(
"""
            **How to Use:**
            1.  **Enter Your Story Idea:** Type a short concept or theme
for your comic in the text box (e.g., "A cat detective solves the mystery
of the missing yarn", "Two robots explore a jungle planet").
            2.  **Choose an Art Style:** Select a visual style for the
comic panels from the dropdown.
            3.  **(Optional) Adjust Advanced Settings:** Expand the
section below to change the number of panels, image size, quality steps,
or how strictly the images follow the text prompts.
            4.  **Generate:** Click the "Generate Comic" button.
Generation can take several minutes depending on settings and hardware.
            5.  **View Results:**
                * The final comic page layout (if enabled) will appear in
the "Comic Page Layout" tab.
```

```
                * The generated story (Title, Storyline, Moral) will be in
the "Story & Details" tab.
                * Download the story details file from the "Downloads"
tab. The comic image can be downloaded directly from its display.

            **Example Prompts:**
            * A superhero squirrel saves the city park from a litterbug
monster.
            * A lonely astronaut finds a friendly alien on Mars.
            * Medieval knights discover a time-traveling phone booth.
            * A group of vegetables starts a rock band.
            """
        )

with gr.Row():
with gr.Column(scale=2):
                prompt_input = gr.Textbox(label="Enter Your Story Idea",
placeholder="e.g., A brave knight confronts a friendly dragon.", lines=4)

# Art Style Dropdown
                art_style_input = gr.Dropdown(
                    label="Select Art Style",
                    choices=list(ART_STYLES.keys()),
                    value="Comic Book (Default)", # Default selection
                    info="Choose the visual style for the comic panels."
                )

                cfg_defaults = comic_crafter.default_settings if
comic_crafter else {}
                default_panels=6;
default_width=cfg_defaults.get("image_width", 1024);
default_height=cfg_defaults.get("image_height", 768);
default_steps=cfg_defaults.get("num_inference_steps", 30);
default_guidance=cfg_defaults.get("guidance_scale", 7.0)

with gr.Accordion("Advanced Settings", open=False):
                     gr.Markdown("Adjust generation parameters
(optional):")
with gr.Row():
                        num_panels = gr.Slider(label="Number of Panels",
minimum=1, maximum=9, value=default_panels, step=1, info="How many
distinct images/panels for the comic?")
with gr.Row():
```

```
                        panel_width = gr.Slider(label="Panel Width (px)",
minimum=512, maximum=1024, value=default_width, step=64, info="Width of
each generated image panel.")
                        panel_height = gr.Slider(label="Panel Height
(px)", minimum=512, maximum=1024, value=default_height, step=64,
info="Height of each generated image panel.")
with gr.Row():
                        num_steps = gr.Slider(label="Image Quality
Steps", minimum=20, maximum=50, value=default_steps, step=1, info="More
steps = better image quality, but slower generation.")
                        guidance_scale = gr.Slider(label="Prompt
Adherence (Guidance)", minimum=1.0, maximum=12.0, value=default_guidance,
step=0.5, info="Higher value = image follows prompt more strictly.")
                    create_layout = gr.Checkbox(label="Generate Final
Comic Page Image", value=True, info="Combine panels into a single layout
image (using OpenCV)?")

                generate_btn = gr.Button("Generate Comic",
variant="primary")

with gr.Column(scale=3):
with gr.Tabs():
with gr.TabItem("Comic Page Layout"):
# --- CHANGE: Set type to "pil" ---
                        comic_output = gr.Image(label="Generated Comic
Page", type="pil", show_label=True, show_download_button=True,
interactive=False, height=600)
with gr.TabItem("Story & Details"):
# Output only Title, Storyline, Moral
                        story_output = gr.Markdown(label="Generated Story
Details")
with gr.TabItem("Downloads"):
                        gr.Markdown("Download generated files:")
                        story_path_output = gr.File(label="Download Story
Details (.md)", interactive=False)
# Comic download is available directly from the Image component

        generate_btn.click(
             fn=generate_comic,
             inputs=[
                 prompt_input,
                 art_style_input,
                 num_panels,
                 panel_width,
                 panel_height,
```

```
                num_steps,
                guidance_scale,
                create_layout
            ],
            outputs=[
                comic_output, # Mapped to the first return value (PIL
Image)
                story_path_output, # Mapped to the second return value
(story file path)
                story_output # Mapped to the third return value (story
markdown text)
            ],
            api_name="generate_comic"
        )
        gr.Markdown("--- \n *Powered by AI. Results vary. May take several
minutes.*")
return interface

# --- Main Launch Block ---
if __name__ == "__main__":
print("\n" + "="*50)
ifnot initialization_errors: print("Initialization check passed. Launching
Gradio UI..."); print("Allow time for interface availability.");
print("="*50 + "\n"); ui = create_ui(); ui.launch(share=True, debug=False)
# share=True for external access if needed
else: print("ERROR: Cannot launch UI due to critical init failures:");
[print(f"- Err {i+1}: {' '.join(msg.splitlines()[:2])}...") for i, msg
inenumerate(initialization_errors)]; print("\nCheck model paths,
dependencies (incl. opencv-python), CUDA, VRAM. Review logs.");
print("="*50 + "\n")
```

7. Model Selection Justification

a) Large Language Model: mistralai/Mistral-7B-Instruct-v0.2

- **Why Used:**
  - **Performance Efficiency:** Mistral-7B offers strong performance for its size (7.3 billion parameters), outperforming larger models like Llama 2 13B on many benchmarks, especially in reasoning and comprehension tasks. This provides good quality generation without the extreme hardware requirements of much larger models.
  - **Instruction Fine-tuning:** The Instruct version is specifically fine-tuned to follow instructions well, which is crucial for the structured generation tasks in this

project (generating stories, prompts, dialogues based on specific templates defined in config.json).

- **Context Window:** Version 0.2 boasts a 32k token context window, significantly larger than the 8k of v0.1 or older models like Llama 2 (often 4k). This allows it to better handle longer stories and maintain coherence across multiple panel descriptions and dialogues that rely on the initial story context.
- **Efficient Architecture:** It utilizes Grouped-Query Attention (GQA) and Sliding Window Attention (SWA) for faster inference and better handling of longer sequences compared to standard attention mechanisms.
- **License:** Released under the Apache 2.0 license, allowing for flexible and unrestricted use.

- **Comparison with Alternatives:**

LLM Comparison: mistralai/Mistral-7B-Instruct-v0.2 vs. Alternatives

Feature/Aspect	mistralai/Mistral-7B-Instruct-v0.2 (Project Model)	Llama 2 13B (Alternative)	Larger Models (e.g., Llama 34B/70B, GPT-3.5/4) (Alternative)
Performance	Strong, outperforms Llama 2 13B on many benchmarks, approaches Llama 1 34B	Good, but generally lower than Mistral-7B on benchmarks	Potentially higher peak performance, state-of-the-art for some (e.g., GPT-4)
Size	7.3 Billion parameters	13 billion parameters	Significantly larger (34B, 70B, or much more for GPT models)
Resource Needs	Moderate (Runnable on high-end consumer hardware, especially with quantization)	Higher than Mistral-7B	Very High (Often require datacenter-grade GPUs / Cloud APIs)
Context Window	32,000 tokens (v0.2)	Typically 4,096 tokens	Varies (e.g., GPT-4 has large context options, older Llama smaller)
Architecture	Uses GQA & SWA for efficiency	Standard Attention mechanisms	Varies, often standard attention or other advanced methods
Tuning	Instruction fine-tuned	Available in Instruct versions	Available in Instruct/Chat versions
License	Apache 2.0 (Open & Permissive)	Custom Llama 2 License (Free for research/commercial, some restrictions)	Varies (Llama custom, GPT proprietary/closed)

Suitability	Well-suited for structured tasks (like in project) due to instruction tuning & efficiency	Capable, but potentially less efficient or performant than Mistral-7B	High capability but may be overkill, less accessible, or have restrictive licenses

b) Image Generation Model: stabilityai/stable-diffusion-xl-base-1.0

- **Why Used:**
  - **High Quality Output:** SDXL 1.0 represents a significant improvement over previous Stable Diffusion versions (1.5, 2.x), capable of generating higher-resolution images (1024x1024 native) with better detail and prompt adherence.
  - **Improved Architecture:** It typically uses a larger U-Net backbone and often incorporates two text encoders (e.g., OpenCLIP and CLIP) for enhanced understanding of text prompts compared to single-encoder predecessors.
  - **Openness:** As part of the Stable Diffusion family, SDXL 1.0 base model weights are openly available, allowing for local deployment and integration without relying on paid APIs (though Stability AI also offers APIs).
  - **Balance:** It provides results competitive with state-of-the-art models while being more accessible than some newer or closed-source alternatives. The project uses the base model, sufficient for generating diverse panels.
- **Comparison with Alternatives:**

Image Generation Model Comparison: stabilityai/stable-diffusion-xl-base-1.0 vs. Alternatives

Feature/Aspect	stabilityai/stable-diffusion-xl-base-1.0 (Project Model)	SD 1.5 / 2.x (Alternative)	Playground v2 (Alternative)	Midjourney / DALL-E 3 (Alternative)
Native Resolution	1024x1024	512x512 (SD 1.5) / 768x768 (SD 2.x)	1024x1024	High Resolution (Varies, often 1024x1024 or higher)
Image Quality	High, improved detail & prompt adherence over SD 1.5/2.x	Good (1.5 often preferred stylistically over 2.x by some users)	High, potentially more consistent style ("AAA game aesthetic")	State-of-the-art, often excels at prompt coherence and specific aesthetics
Architecture	Larger UNet, often dual text encoders (OpenCLIP +	Smaller UNet, single text encoder (CLIP	Similar architecture to SDXL	Proprietary / Closed

	CLIP ViT-L)	for 1.5, OpenCLIP for 2.x)		
Openness	Open Weights (Base model)	Open Weights	Open Weights	Closed Source, API/Platform access only
Variety vs Style	Can produce varied outputs for same prompt	Varies	Can be more stylistically consistent	Varies, Midjourney known for distinct artistic style, DALL-E 3 for prompt adherence
Speed (Base)	Standard diffusion speed, comparable to Playground v2 before optimizations	Standard diffusion speed	Comparable to SDXL before optimizations	Generally fast via optimized cloud infrastructure
Ecosystem	Large community, many tools/optimizations (LoRAs, ControlNet, etc.)	Large community (esp. 1.5), established tools	Newer, fewer specific optimizations available initially	Platform-specific features, limited customization
Suitability	Strong baseline for high-quality, open generation. Good balance of quality/ accessibility	Still viable, 1.5 popular for specific styles. Lower resolution/ detail than SDXL	Strong alternative if consistent style is preferred. Comparable quality/performance to SDXL	Top-tier quality but lacks openness, local deployment, and fine-tuning potential of SDXL

8. Challenges Faced

Developing this project likely involved several challenges:

- **GPU Resource Management:** Running both a large LLM and a large diffusion model is memory-intensive. The code explicitly includes GPU memory clearing (`_clear_gpu_memory`) and logic to load only one model type into VRAM at a time (`ensure_llm_loaded`, `ensure_image_generator_loaded`), indicating a significant challenge in managing limited GPU resources.
- **Prompt Engineering:** Designing effective prompts (stored in `config.json`) to coax the desired structured output (story, title, descriptions, dialogue) from the LLM requires

careful iteration and experimentation. Ensuring the LLM adheres to format constraints (e.g., word limits, outputting only descriptions) is non-trivial.

- **LLM Output Parsing:** LLMs can sometimes include extraneous text or fail to follow formatting perfectly. The use of regex (`re.sub`) after LLM calls suggests challenges in cleaning and extracting the exact desired information from the model's raw output.
- **Cross-Platform Compatibility:** Ensuring features work across different operating systems, particularly font loading (`panel_layout.py` searches multiple common font directories) and hardware availability (CPU vs. GPU detection), can be complex.
- **Maintaining State:** Ensuring the correct model (LLM or Image Gen) is loaded when needed and that the system accurately tracks the loaded state required careful state management logic.
- **Visual Consistency:** Achieving visual consistency across panels (e.g., character appearance, style) is inherently difficult with standard diffusion models without additional techniques (like LoRAs, ControlNet, IP-Adapters, which are not explicitly used here).
- **Text Rendering:** Drawing clean, well-placed, and wrapped text (especially dialogues within bubbles) onto images using libraries like OpenCV requires careful coordinate calculation and handling of different text lengths.

9. Learnings from the Project

- **Synergy of Generative Models:** Combining LLMs for narrative/semantic tasks and Diffusion Models for visual tasks allows for powerful multi-modal content creation.
- **Importance of Prompting:** The quality and structure of prompts given to AI models are critical determinants of the output quality and usability.
- **Resource Optimization:** Techniques like model quantization (4-bit) and dynamic model loading/unloading are essential for running large models on consumer-grade hardware.
- **User Interface Importance:** Using frameworks like Gradio makes complex AI pipelines accessible to non-technical users.
- **Robustness through Fallbacks:** Implementing graceful fallbacks (placeholders, error skipping) is crucial for a usable application when dealing with inherently unpredictable AI models.
- **Modularity:** Separating concerns into different modules (AI logic, layout logic, UI) improves code maintainability and organization.

10. Limitations

- **Dialogue Placement:** Dialogue bubbles are drawn at a fixed top-center position within each panel, which may not align well with character positions or the visual focus of the generated image.
- **Character Consistency:** Standard diffusion models struggle to maintain character appearance consistency across multiple panels without specific fine-tuning or conditioning techniques not implemented here. Characters may look different in each panel.

- **Layout Rigidity:** The comic layout grid (e.g., 2x2, 2x3) is determined automatically based on the number of valid panels, offering limited user control over the final page composition. Panel arrangement is fixed.
- **AI Variability:** The output of generative AI can be unpredictable. Story coherence, image quality, and adherence to the prompt can vary between runs even with the same input.
- **Hardware Dependency:** Performance is heavily dependent on hardware, particularly a CUDA-enabled GPU with sufficient VRAM. CPU execution is possible but likely extremely slow for image generation.

11. Potential Future Enhancements

- **Configurable Layouts:** Allow users to choose different grid arrangements or manually position panels.
- **Improved Dialogue Placement:** Implement object detection or user interaction to place dialogue bubbles more appropriately near speaking characters.
- **Character Consistency:**Improve character consistency to maintain character appearance across panels.
- **Model Flexibility:** Allow users to select from different compatible LLMs or image generation models via the UI or configuration.
- **Refiner Model:** Option to use the SDXL refiner model for potentially enhanced image detail.
- **Feedback Loop:** Allow users to regenerate specific panels or edit generated text/prompts.
- **Negative Prompt UI:** Expose the negative prompt in the UI for user customization.
- **Advanced Font Handling:** More robust font selection and management.

12. Conclusion:

In conclusion, ComicCrafter AI represents a significant step forward in applying generative AI to creative workflows. By effectively integrating Large Language Models for narrative and dialogue generation with sophisticated Image Generation models for visual panels, it successfully automates the complex process of comic creation from a simple idea. The inclusion of automated layout, title, and dialogue bubble placement results in a complete, end-to-end pipeline that lowers the barrier for users to transform their concepts into visually engaging comic pages, showcasing the power of multi-modal AI in enabling new forms of digital storytelling.

13. Video link and Sample Photos generated by ComicCrafter AI:

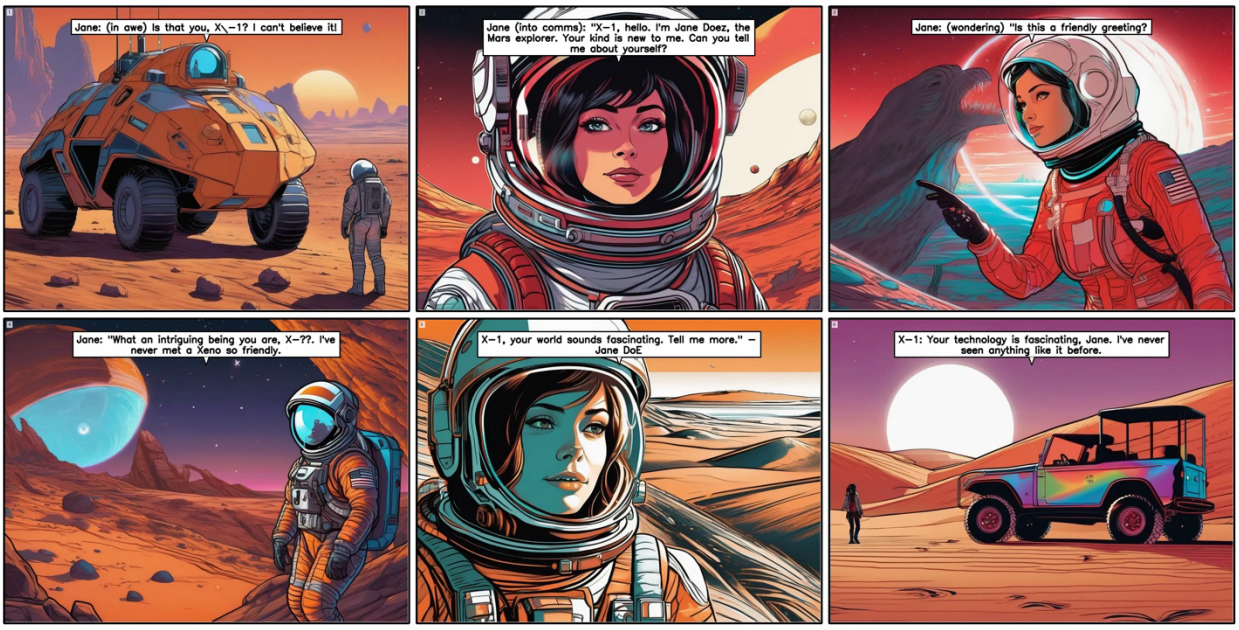
Video Link: https://drive.google.com/file/d/1Hb3J-SfHZ17VQpY8_dNz0J-yfWDkuxuf/view?usp=sharing

Comic strip generated for input: "Medieval Knights Discover Time – Travel Phone Booth", "Mars Encounter: Xenomorph's Greeting", "Merlin Reveals Ancient Prophecy".

Medieval Knights Discover Time-Travel Phone Booth

Mars Encounter: Xenomorph's Greeting"

or



Team Contribution:

Core AI Model Integration (comic_crafter_ai.py) done by Kavin Aarya (Team Lead)

This involves setting up the foundational interaction with the chosen Large Language Model (LLM) and the Image Generation model. Key tasks include loading models (like Mistral and Stable Diffusion XL), managing resources (GPU memory), handling basic text/image generation calls, and implementing initial prompt formatting.

Structured Text Generation Engine (comic_crafter_ai.py, config.json) done by Kavin Aarya and Hephzibah

Building upon the core LLM integration, this part focuses on using the LLM specifically for the creative writing aspects of comic creation. It includes designing and implementing the prompt templates (in config.json) and logic within comic_crafter_ai.py to generate cohesive story elements like the overall narrative, title, moral, distinct panel descriptions, and dialogue based on user input and story context.

Visual Comic Assembly & Annotation (panel_layout.py) done by Kavin Aarya and Jeevitha

This part concentrates on the visual output. It involves taking the raw generated panel images and programmatically arranging them into a comic book style grid. Crucially, it also includes developing the functionality (using libraries like PIL and OpenCV) to overlay

generated text elements like the comic title, panel numbers, and dialogue bubbles onto the final image.

User Interface and Workflow Orchestration (main.py) done by Kavin Aarya

This involves creating the user-facing front-end (using Gradio) to accept user input (story idea, art style, settings) and display results. It also entails writing the main script logic that coordinates the entire process: taking UI input, calling the ComicCrafterAI module for text and image generation, passing results to the PanelLayout module for assembly, and finally, presenting the completed comic back to the user via the UI.