Code Documentation

1. load_pipelines_on_startup()

```python
# It Loads all the required pipelines at app startup
def load_pipelines_on_startup():
    global text2img_pipe, img2img_pipe , controlnet_pipe
    print("[INFO] Loading pipelines at startup...")

    # Load the Text-to-Image Stable Diffusion pipeline
    if text2img_pipe is None:
        text2img_pipe = StableDiffusionPipeline.from_pretrained(
            "CompVis/stable-diffusion-v1-4",
            torch_dtype=torch.float16           # Use fp16 for faster inference
        ).to(device)

    # Load the Image-to-Image Stable Diffusion pipeline
    if img2img_pipe is None:
        img2img_pipe = StableDiffusionImg2ImgPipeline.from_pretrained(
            "CompVis/stable-diffusion-v1-4",
            torch_dtype=torch.float16
        ).to(device)

    # Load ControlNet pipeline with Canny edge detection
    if controlnet_pipe is None:
        controlnet = ControlNetModel.from_pretrained(
            "lllyasviel/sd-controlnet-canny", torch_dtype=torch.float16  # Canny ControlNet variant
        ).to(device)

        controlnet_pipe = StableDiffusionControlNetPipeline.from_pretrained(
            "CompVis/stable-diffusion-v1-4",
            controlnet=controlnet,    # Injected ControlNet into the pipeline
            torch_dtype=torch.float16
```

This function is responsible for initializing all the Stable Diffusion pipelines at application startup to avoid redundant loading during runtime. It first checks if the global variables text2img_pipe, img2img_pipe, or controlnet_pipe are None, and loads them if necessary. For the text-to-image and image-to-image pipelines, it loads the CompVis/stable-diffusion-v1-4 model with torch_dtype=torch.float16 for memory efficiency and faster performance. For the ControlNet pipeline, it loads the lllyasviel/sd-controlnet-canny ControlNet model and injects it into the main pipeline, setting a UniPCMultistepScheduler for optimized inference. The output is the global initialization of all three pipelines, with no return value.

2. get_canny_image(image, low_threshold=100, high_threshold=200)

```python
# It convert an image into a canny edge map for use with ControlNet
def get_canny_image(image, low_threshold=100, high_threshold=200):
    image = np.array(image.convert("RGB"))
    image = cv2.Canny(image, low_threshold, high_threshold)
    image = np.stack([image]*3, axis=-1)
    return Image.fromarray(image)
```

This function generates a Canny edge-detected image from the input PIL.Image. It first converts the image to RGB format and then applies the cv2.Canny() function using the provided low and high thresholds. The resulting edge-detected image is stacked into three channels to match input requirements for the ControlNet pipeline. The function returns a new PIL.Image that represents the edge map of the original image. This edge map is crucial for guiding image generation via ControlNet.

3. generate_multiple_images(prompt, num_images=3)

```python
# Function generate multiple images using the Text-to-Image pipeline
def generate_multiple_images(prompt, num_images=3):
    try:
        print(f"[INFO] Generating {num_images} images for prompt: {prompt}")

        # Maintaining pharma context in prompt
        domain_suffix = (
            "in a pharmaceutical setting, possibly with lab equipment, bioreactors, or cleanroom background"
        )

        full_prompt = f"{prompt}, {domain_suffix}"
        pipe = text2img_pipe
        results = [pipe(full_prompt).images[0] for _ in range(num_images)] # it generates 3 images
        return results , results # Return twice (for UI use with gallery + state)

    except Exception as e:
        print(f"[ERROR] Generation failed: {e}")
        # Return blank images in case of error
        return [Image.fromarray(np.zeros((512, 512, 3), dtype=np.uint8)) for _ in range(num_images)]
```

This function generates multiple images from a text prompt using the Text-to-Image Stable Diffusion pipeline. It appends a predefined pharmaceutical domain suffix to the user's prompt to ensure contextual relevance to the pharmaceutical industry. Then it uses the text2img_pipe to generate num_images (default is 3) results. The output is a tuple of two lists of images: one for display in the gallery and the other for internal state tracking. If an error occurs, it returns blank images.

4. regenerate_image(selected_img, index, new_prompt)

```python
# This function regenerate image based on edited prompt and previously selected image
def regenerate_image(selected_img,index, new_prompt):
    try:
        index = int(index)
        selected_img = selected_img[index]

        if selected_img is None:
            raise ValueError("No image selected for regeneration.")

        # Maintaining pharma context in prompt
        domain_suffix = (
            "in a pharmaceutical setting, possibly with lab equipment, bioreactors, or cleanroom background"
        )
        full_prompt = f"{new_prompt}, {domain_suffix}"
        pipe = img2img_pipe  # our global variable pipeline
        img = pipe(prompt=full_prompt, image=selected_img, strength=0.75).images[0]
        return img

    except Exception as e:
        print(f"[ERROR] Regeneration failed: {e}")
        # Return blank image on failure
        return Image.fromarray(np.zeros((512, 512, 3), dtype=np.uint8))
```

This function is used to regenerate an image based on a user-selected image and a modified prompt. It retrieves the selected image by its index from a list of previously generated images and combines the new prompt with the domain-specific suffix. Using the Image-to-Image pipeline (img2img_pipe), it generates a new image with a strength of 0.75, which controls how strongly the generation adheres to the new prompt. The function returns a single image, which is an edited version of the selected one.

5. generate_from_image(prompt, image, num_images=3)

```python
# This function generate multiple images from a user-provided image and prompt using Img2Img
def generate_from_image(prompt, image , num_images=3):
    try:
        # Maintaining pharma context in prompt
        domain_suffix = (
            "in a pharmaceutical setting, possibly with lab equipment, bioreactors, or cleanroom background"
        )
        full_prompt = f"{prompt}, {domain_suffix}"
        pipe = img2img_pipe
        image = image.resize((512, 512)) # Resize input to 512x512 for stable diffusion
        img = [pipe(prompt=full_prompt, image=image, strength=0.75).images[0] for _ in range(num_images)]
        return img ,img # Output for gallery and state

    except Exception as e:
        print(f"[ERROR] Img2Img failed: {e}")
        return Image.fromarray(np.zeros((512, 512, 3), dtype=np.uint8))
```

This function allows users to generate multiple variations of an image based on a prompt and an input image. It first resizes the input image to 512x512 pixels to match Stable Diffusion's expected input size. Then, it appends the pharmaceutical domain suffix to the prompt and passes it along with the image to the img2img_pipe, generating the specified number of output

images. The function returns a tuple: a list of images for display and the same list for internal state tracking.

6. store_selected(index, gallery_images)

```python
# This function store selected image from Gradio gallery
def store_selected(index, gallery_images):
    if index is None or gallery_images is None:
        raise gr.Error("No image was selected.")
    selected_img = gallery_images[index]  # Select image from gallery based on user click
    return selected_img, selected_img
```

This function is used to store the image that a user selects from a Gradio gallery. It takes an index and the list of gallery images, and returns the selected image twice—once for display and once for downstream logic like regeneration. If no image is selected, it raises a Gradio error to alert the user. This is mainly used in the image editing workflow.

7. generate_controlnet_image(prompt, input_image)

```python
# Generate an image using ControlNet (Canny) based on prompt and input image
def generate_controlnet_image(prompt, input_image):
    try:
        canny_image = get_canny_image(input_image)   # Getting Canny edge map from input image
        pipe = controlnet_pipe
        result = pipe(prompt=prompt, image=canny_image, num_inference_steps=20).images[0]  # It generate image using ControlNet
        return result
    except Exception as e:
        print(f"[ERROR] ControlNet generation failed: {e}")
        return Image.fromarray(np.zeros((512, 512, 3), dtype=np.uint8))
```

This function creates an image using the ControlNet pipeline, guided by edge detection. It starts by applying the get_canny_image() function to convert the input image into an edge map. Then, it uses the controlnet_pipe to generate an output image from the edge map and user-provided prompt in 20 inference steps. The result is a single PIL.Image that reflects the structure of the input while incorporating the creative elements of the prompt. If there's an error, it returns a blank image.

Approach Taken

Architecture & Modularity

The app is built using a modular and extensible architecture, organized into 3 main tabs—each targeting a different generation modality. These modules share global pipelines but use separate logic and UI components to ensure clean separation of concerns:

Text-to-Image (T2I)
Image-to-Image (I2I)
ControlNet-based Image Enhancement

These modules are loaded and controlled using Gradio Blocks, which makes it easy to develop a reactive, event-driven UI that supports live interaction.

Pipelines Used

The application leverages the diffusers library from Hugging Face, specifically:

StableDiffusionPipeline for Text-to-Image
StableDiffusionImg2ImgPipeline for Image-to-Image
StableDiffusionControlNetPipeline (with Canny ControlNet) for structure-preserving enhancement

Each pipeline is loaded once at startup using the load_pipelines_on_startup() function, minimizing runtime delays and enabling fast switching between workflows. These pipelines run on GPU (if available) using torch_dtype=torch.float16 for faster and memory-efficient inference.

Prompt Engineering Strategy

A key detail in the approach is the automatic enrichment of prompts with a fixed suffix:

"in a pharmaceutical setting, possibly with lab equipment, bioreactors, or cleanroom background"

This ensures domain relevance across all image generations and removes the burden on the user to consistently describe pharmaceutical context. The suffix is dynamically added at runtime in every generation or regeneration function.

Editable Workflows

Users can:

- Click on any generated image to select it.
- Modify the prompt in a textbox.
- Generate an "edited" version using that selected image as a base.

This is done using regenerate_image(), which takes the selected image and a new prompt and regenerates the image using the img2img pipeline.

Memory Management

A helper function unload_pipeline() is provided to manually free up GPU memory if needed. It uses both garbage collection (gc.collect()) and CUDA memory cleanup (torch.cuda.empty_cache()). This keeps the app running smoothly, even with multiple large models loaded.