ChatGPT

Autonomous Sprite Sheet Generator Implementation Plan

API capability table

Model/Service	Image-to-image editing	Use reference images?	Pose conditioning?
Stable Diffusion (SDXL / Flux base)	Yes. The **StableDiffusionImg2ImgPipeline** accepts a starting image and produces an edited version conditioned on a text prompt. The documentation notes that image-to-image generation is enabled by passing the `image` argument; the `strength` parameter controls the amount of change [1] [2]. Flux models expose analogous pipelines via `FluxPipeline` and `FluxImg2ImgPipeline` (similar signatures).	Yes. You can pass a list of PIL images or numpy arrays via the `image` parameter; the pipeline encodes and denoises them [2].	Not inherently. SDXL and Flux do no pose control; they rely on external a such as **ControlNet** to impose struc
IP-Adapter / IP-Adapter Plus	Yes. IP-Adapter can be used with text-to-image and **image-to-image pipelines** by passing an initial image via the `image` parameter and a reference image via `ip_adapter_image` [9]. Diffusers provides `AutoPipelineForImage2Image` examples that load IP-Adapter weights and perform image-to-image edits [10].	Yes. A reference image (or list of style and subject images) is passed to the `ip_adapter_image` argument; the adapter extracts features from the reference and injects them into the model [11] [12].	No. IP-Adapter alone does not cont The documentation suggests comb with **ControlNet** when structural gu such as depth or pose is required [1]

Model/Service	Image-to-image editing	Use reference images?	Pose conditioning?
ControlNet (e.g., DWPose, Canny, Depth)	Yes. ControlNet pipelines support text-to-image, image-to-image and inpainting; they accept a control image (pose, depth, edge map) via the `image` argument [16] [17] .	The control image itself is the reference; multiple controls can be composed in a **Multi-ControlNet** by passing a list of images [18] .	Yes. DWPose and other control chec drive the generation to match the s pose; an example shows computing skeleton from a pose image and pa `StableDiffusionXLControlNet` [17] .
LoRA (Low-Rank Adaptation) fine-tunes	Indirect. LoRA adapts the base model by adding trainable rank-decomposed matrices; once loaded into a pipeline, the image-to-image and inpainting capabilities of the base model remain available [20] [2] .	Not directly; LoRA doesn't accept reference images itself. During generation you still pass reference images via the base pipeline's `image` or `ip_adapter_image` parameters.	Not directly; pose conditioning mus provided via ControlNet.
ComfyUI API (SaladTechnologies/ comfyui-api)	Depends on workflow. ComfyUI is a node-graph system; it can run any Stable Diffusion/ControlNet/ IP-Adapter/LoRA workflows, including image-to-image edits, by combining nodes.	Yes. Workflows can accept reference images via IP-Adapter nodes or image inputs.	Yes, through ControlNet nodes (e.g. pre-processor) inserted into the gra

Proposed technical architecture

Overview

The generator should be implemented as a headless service that orchestrates local diffusion models and audit logic. The recommended stack from the research pairs a base diffusion model (SDXL or Flux) with IP-Adapter for identity injection, multi-ControlNet (DWPose for pose) for structural guidance and a per-character LoRA to achieve near-perfect consistency. The pipeline must produce deterministic,

repeatable frames and ensure they match the anchor sprites exactly. The architecture separates **generation**, **audit**, and **packing** stages and allows retrying any failed frame with a different seed.

Components

1. **Model hub** – maintain local copies of the base model (SDXL or Flux), IP-Adapter weights, ControlNet checkpoints (DWPose and optionally a second control such as HED for outlines) and LoRA weights for each character. Use the **diffusers** library for inference and training; ComfyUI may be used as a graphical front-end or for complex multi-control workflows.

2. **LoRA trainer** – a script that automates dataset generation and LoRA fine-tuning for each character:

3. Input: anchor sprite(s) for a character.
4. Uses IP-Adapter + DWPose ControlNet to generate ~50–100 variations in random poses; filters them using SSIM/LPIPS to keep only on-model outputs.

5. Trains a LoRA on the filtered set with the base model. The result is a lightweight LoRA file (~100–200 MB) that locks the character's identity.

6. **Pose scheduler** – defines the keyframes for each animation (idle, walk, jump, attacks, etc.). Each frame corresponds to a pose skeleton generated via DWPose or manually drawn. For the MVP, use a minimal Street-Fighter–style set (e.g., idle: ~4 frames, walk: 6 frames forward/backward, jump: 6 frames, block: 4, light/medium/heavy punches and kicks: 5–7 each, hurt: 4, KO: 6, victory: 6). The counts can be refined based on gameplay feel.

7. **Frame generator** – a headless Python service that:

8. Loads the base model and LoRA for the current character; loads IP-Adapter weights and ControlNet checkpoints.
9. For each pose image, calls the **StableDiffusionXLControlNetPipeline** (or Flux equivalent) with the LoRA and IP-Adapter enabled. Pass the pose skeleton as `image`, the anchor sprite as `ip_adapter_image`, and fix a random seed; generate one or more images with `num_images_per_prompt` for redundancy.
10. Applies palette quantization or pixel-art filters to enforce a 16-bit palette consistent with the game.

11. Returns the candidate frame images.

12. **Audit loop** – after each frame is generated:

13. Compute **identity similarity** (e.g., structural similarity index – SSIM, perceptual loss – LPIPS) between the generated frame and the anchor; enforce a threshold (e.g., SSIM > 0.95).
14. Check **style/palette drift** by comparing color histograms and using a cell-shading classifier.
15. Verify baseline alignment: ensure the character's feet/pivot align with a predefined pixel row. This can be done by analyzing silhouette or by referencing keypoints from the DWPose skeleton.

16. If the frame passes all checks, accept it; otherwise, re-invoke the generator with a different seed or adjust ControlNet weights. Because LoRA locks the identity, retries do not introduce drift.

17. **Sprite sheet packer** – once all frames for an animation pass, crop/resize them to the exact anchor dimensions, add 4 px padding, and pack them into a single PNG sprite sheet. Generate a matching JSON manifest for Phaser 3 (frame names, x/y/width/height, pivot). Tools like **TexturePacker** or a custom Python script using `Pillow` can automate packing.

18. **Manifest and data storage** – store all inputs and outputs under a structured folder hierarchy (see below). A **generation_manifest.json** per animation describes the model settings (base model, LoRA, IP-Adapter strength, ControlNet types, seed, etc.), the list of poses and the resulting frame file names.

Folder structure & manifest schema

```
project/
├── anchors/
│   └── {character}/
│       ├── base.png            # primary anchor sprite
│       └── extra_*.png         # optional extra anchors
├── loras/
│   └── {character}/
│       └── {character}_lora.safetensors
├── controls/
│   └── {animation}/
│       ├── frame_000_pose.png  # pose skeleton images (generated via DWPose)
│       └── …
├── scripts/
│   ├── train_lora.py           # dataset expansion + LoRA fine-tuning
│   ├── generate_frames.py      # frame generator + audit loop
│   ├── pack_spritesheet.py     # packing into PNG + JSON
│   └── audit_metrics.py        # SSIM/LPIPS checks
├── outputs/
│   └── {character}/
│       ├── frames/{animation}/frame_000.png
│       ├── spritesheets/{animation}.png
│       └── metadata/{animation}.json
├── manifests/
│   └── generation_manifest.json
└── README.md
```

`generation_manifest.json` **schema (simplified)**

```
{
  "character": "Kenji",
  "base_model": "sdxl-base-1.0",
  "loras": ["loras/Kenji/Kenji_lora.safetensors"],
  "ip_adapter": true,
```

```
  "controlnets": ["dwpose_sdxl", "hed_sdxl"],
  "animations": [
    {
      "name": "walk_forward",
      "frames": [
        {"pose": "controls/walk_forward/frame_000_pose.png", "seed": 42},
        {"pose": "controls/walk_forward/frame_001_pose.png", "seed": 43},
        …
      ]
    },
    …
  ],
  "output_size": [64, 64],
  "padding": 4,
  "palette": "16bitfit_palette.pal"
}
```

This manifest acts as both a job description and a record of the seeds used, enabling perfect reproducibility.

Step-by-step build plan

1. **Set up infrastructure**
2. Install Python 3.10+, PyTorch (CUDA), `diffusers`, `transformers`, `peft`, `accelerate`, `safetensors`, `Pillow`, and `einops`. If using ComfyUI, deploy a headless ComfyUI instance or the **comfyui-api** server; configure GPU access.
3. Download the base model weights (e.g., `stabilityai/stable-diffusion-xl-base-1.0` or `black-forest-labs/FLUX.1-dev`), IP-Adapter weights (`h94/IP-Adapter`), ControlNet checkpoints (`lllyasviel/control_v11p_sd15_openpose` or `dimitribarbot/controlnet-dwpose-sdxl-1.0`), and set up a directory for LoRA weights.
4. Prepare anchor sprites for each character; crop them to the exact bounding box and record their dimensions and baseline pivot.
5. **Automate LoRA fine-tuning per character**
6. **Dataset expansion:** For a given character, run a script that loads the base model with IP-Adapter and DWPose. Pass the anchor sprite as the `ip_adapter_image` and random pose skeletons as the `image` input to generate ~50–100 on-model variations. Vary seeds and slightly vary pose weights. Use the audit metrics (SSIM/LPIPS) to filter results. Save the approved images and corresponding captions (e.g., “character performing pose X”).
7. **Fine-tune LoRA:** Use the PEFT library to fine-tune the UNet and/or text encoder on the filtered dataset. Train only for a few epochs (e.g., 1–3) with a low learning rate. Save the resulting LoRA weights (`.safetensors`). Register them in `loras/character/`.
8. **Define animation poses**

9. For each required animation (idle, walk forward/backward, jump, block, light/medium/heavy punch/kick, hurt, KO, victory), decide on the number of frames and key poses. Reference classic fighting games for counts (e.g., walk: 6 frames; basic attacks: 5–7 frames). Create pose skeleton images for each frame using DWPose or manual keypoint editing.
10. Store the pose images in the `controls/{animation}/` directory. Record the desired frame order and assign seeds (can be sequential or derived from a base seed).

11. **Implement the frame generator and audit loop**

12. Write `generate_frames.py` to accept a `generation_manifest.json`. For each animation and frame:

    - Load the base model and LoRA weights; call `StableDiffusionXLControlNetPipeline.from_pretrained` with the DWPose ControlNet; load IP-Adapter; set the LoRA via `load_lora_weights`. Pass the pose skeleton (`image`), anchor sprite (`ip_adapter_image`), and the configured seed to the pipeline; request a single image (`num_images_per_prompt=1`).
    - Apply palette quantization or pixel-art filters to the output to enforce the 16-bit aesthetic.
    - Evaluate similarity using SSIM/LPIPS; if below threshold or if baseline alignment fails, increment the seed and retry. Limit the number of retries (e.g., 3–5) to avoid infinite loops.
    - Save the accepted frame in `outputs/{character}/frames/{animation}/` with a naming convention (frame_000.png). Record the actual seed used in the manifest.

13. **Sprite sheet packing**

14. After all frames of an animation pass, run `pack_spritesheet.py` to assemble them into a single PNG with 4-px padding. Use `Pillow` or `texturepack` to generate a spritesheet and export frame coordinates (x/y/width/height) in a JSON file compatible with Phaser 3.

15. Verify that the final sheet has the same frame dimensions as the anchors and that the baseline alignment is maintained.

16. **Integrate with Phaser 3 and audit loop**

17. Load the generated spritesheets into the Phaser project. Ensure the JSON metadata defines the correct pivot for each frame (usually the baseline row). Use Phaser's `this.load.atlas` to load the PNG and JSON.
18. In development builds, implement an in-game toggle that overlays the anchor sprite and generated frame to visually inspect identity match. Use this for manual audits alongside automated metrics.

19. Provide a QA script that runs through all animations and logs any frames that deviate from the anchor spec.

20. **Production hardening**

21. Containerize the generator (e.g., with Docker) to ensure consistent dependencies. Configure GPU memory allocation. Use the **comfyui-api** server if you prefer to orchestrate workflows via HTTP; otherwise rely on the pure Python `diffusers` scripts.
22. Add caching of intermediate latents and skeletons to speed up regeneration. Optionally implement a queue to process multiple characters concurrently using separate GPU workers.
23. Document the pipeline and provide CLI commands for training LoRAs, generating frames, and packing spritesheets. Include a logging mechanism that records seeds, failure reasons, and timestamps for auditability.

Risk list & mitigations

Risk	Impact	Mitigation
Style drift or identity loss across frames	Off-model sprites break game immersion.	Use the hybrid pipeline (IP-Adapter + LoRA + ControlNet) which locks identity. Employ an audit loop with SSIM/LPIPS metrics and baseline alignment checks; regenerate frames with different seeds until pass.
Insufficient training data for LoRA	LoRA may overfit or underperform on uncommon poses.	Generate synthetic data via IP-Adapter + ControlNet to augment the dataset; include varied poses and angles. Fine-tune with a small learning rate and monitor loss.
Model bias toward photorealism / high detail	Outputs may contain shading or details inconsistent with pixel-art style.	Apply pixel-art or cell-shading LoRAs during generation; post-process with palette quantization and dithering. Test with different base models (Flux or SDXL) to find the one that best approximates 16-bit graphics.
Pose control conflicts with identity	ControlNet sometimes distorts facial features when enforcing extreme poses.	Use multi-control techniques and the dual-pass inpainting strategy described in the research: generate a frame with pose control, then re-inpaint critical regions using IP-Adapter with high weight.
Performance & memory constraints	Generating hundreds of frames per character may exceed GPU memory or take too long.	Use efficient schedulers (e.g., Euler or DDIM) and latent consistency models; tune `num_inference_steps` to balance speed and quality; run jobs in batches; consider the Flux "Schnell" variant for faster inference.
License restrictions (Flux)	Flux models may have non-commercial licences that restrict usage.	Review the **Flux Non-Commercial License**; if it conflicts with the game's commercial goals, fall back to SDXL. Keep all models self-hosted to avoid SaaS limitations.

Risk	Impact	Mitigation
API downtime or SaaS dependence	Reliance on external APIs may introduce latency or availability issues.	Prefer locally hosted models via diffusers and ComfyUI; only use external APIs for experimentation. For ComfyUI API deployments, monitor health probes and scale horizontally [23].
Inconsistent palette / color shifts	Differences in color palette between frames cause flickering.	After generation, quantize images to a fixed 16-bit palette using a tool like ImageMagick or a custom script. Include the palette in the manifest.
Audit metrics false positives/ negatives	Automated checks may wrongly accept/reject frames.	Tune thresholds empirically using manual inspection. Combine multiple metrics (SSIM, LPIPS, color histograms). Provide a manual override in the QA tool.
Version drift of models / dependencies	Different model versions produce inconsistent results over time.	Pin model versions and dependencies in a `requirements.txt`. Store model files and LoRA weights under version control. Record the model commit hashes in the manifest.

Minimal viable pipeline vs production pipeline

Minimal Viable Pipeline (MVP)

The MVP aims to produce usable sprites quickly without any training. It leverages zero-shot generation via IP-Adapter and ControlNet and runs a simple audit loop.

1. **Base model**: use SDXL base model or Flux with no LoRA fine-tuning.
2. **Anchor injection**: pass the anchor sprite to IP-Adapter Plus to approximate the character's look.
3. **Pose control**: use DWPose ControlNet; optionally combine with an edge detector like HED for cleaner outlines.
4. **Generation**: call `StableDiffusionXLControlNetPipeline` with `image=pose_skeleton` and `ip_adapter_image=anchor`; fix the seed and generate one or two candidate images.
5. **Audit**: compute SSIM/LPIPS against the anchor; reject if below threshold; try a new seed; repeat up to a small number of retries.
6. **Post-process**: palette quantization to a 16-bit palette; crop to anchor size.
7. **Packing**: assemble frames into a spritesheet and output JSON metadata.

This pipeline can be implemented entirely in ComfyUI or using a simple diffusers script. It is fast to set up and suitable for NPCs or prototyping, but identity accuracy is lower and may require manual cleanup.

Production Pipeline (Anchor + LoRA)

The production pipeline follows the research recommendation and delivers deterministic, high-quality sprites with minimal drift.

1. **Prepare anchors** and run the dataset expansion using IP-Adapter + ControlNet to generate pose variations.
2. **Train per-character LoRA** on the expanded dataset to encode the character's identity. Training is a one-time cost and yields a small LoRA file.
3. **Pose definition**: design keyframes for each animation; generate or draw pose skeleton images.
4. **Generation**: load the base model, LoRA, IP-Adapter, and ControlNet; generate frames for each pose using fixed seeds. Because the LoRA encodes identity, the IP-Adapter weight can be reduced or disabled after training.
5. **Audit loop**: run SSIM/LPIPS and baseline checks; retry until frames pass. Failures are rare because the LoRA fixes identity.
6. **Post-process & packing**: palette quantization, cropping, spritesheet assembly, JSON metadata.
7. **Integration**: update Phaser assets, maintain manifest with seeds and model versions for reproducibility.

This pipeline requires the extra step of LoRA training but results in stable characters across all frames and is recommended for the main heroes of 16BitFit.

---

[1] [2] Image-to-image
https://huggingface.co/docs/diffusers/en/api/pipelines/stable_diffusion/img2img

[3] [16] [18] ControlNet
https://huggingface.co/docs/diffusers/en/using-diffusers/controlnet

[4] [5] [6] Stable diffusion pipelines
https://huggingface.co/docs/diffusers/v0.6.0/en/api/pipelines/stable_diffusion

[7] [8] black-forest-labs/FLUX.1-dev · Hugging Face
https://huggingface.co/black-forest-labs/FLUX.1-dev

[9] [10] [11] [12] [13] [14] [15] IP-Adapter
https://huggingface.co/docs/diffusers/en/using-diffusers/ip_adapter

[17] [19] dimitribarbot/controlnet-dwpose-sdxl-1.0 · Hugging Face
https://huggingface.co/dimitribarbot/controlnet-dwpose-sdxl-1.0

[20] [21] LoRA
https://huggingface.co/docs/diffusers/en/tutorials/using_peft_for_inference

[22] [23] [24] GitHub - SaladTechnologies/comfyui-api: A simple API server to make ComfyUI easy to scale horizontally. Get outputs directly in the response, or asynchronously via a variety of storage providers.
https://github.com/SaladTechnologies/comfyui-api