ChatGPT

AI Sprite Sheet Generation Pipeline Blueprint for 16BitFit

A) Model & Method Decision Matrix

The table below compares candidate open-source/API-friendly models and techniques on key criteria. Each cell gives a 1–5 score (1=poor, 5=excellent) with justification and evidence:

Model/Technique	**Identity Lock Strength** (consistently preserves character identity)	**Pose Control Strength** (precisely controls pose/ motion)	**Pixel/16-bit Fidelity** (outputs crisp “retro” sprite style)	**Automation / API Friendliness** (easy to integrate & script)	**Cost + Throughput** (compute requirements, speed, scalability)
ComfyUI (node workflow)	**4** – High. Enables combining identity-conditioning modules (e.g. IP-Adapter, LoRA) for near-deterministic character consistency [1]. Identity lock depends on plugged-in models, but the flexible pipeline can reliably enforce a character’s look.	**5** – Excellent. Supports multi-ControlNet (e.g. DWPose skeletal + additional cues) to strictly impose target poses [1]. Complex pose/motion constraints can be built into the workflow with ease.	**4** – Strong. Can integrate style LoRAs or vector filters to achieve clean 2D lines [2]. Must handle resolution carefully – e.g. avoid improper downscaling that blurs pixel art [3] – but with proper nodes, outputs match the 16-bit aesthetic.	**5** – Excellent. ComfyUI is scriptable (JSON graph API) and runs locally. It supports headless execution and custom nodes [4], making it highly reproducible and suitable for batch automation.	**4** – Good. No licensing costs (open-source); runtime simila to base Stable Diffusion. Pipeline complexity ad some overhea (multiple models per generation) bu still within rea time for single frames. Scales linearly for batch jobs (parallel instances on GPU).

Model/Technique	**Identity Lock Strength** (consistently preserves character identity)	**Pose Control Strength** (precisely controls pose/ motion)	**Pixel/16-bit Fidelity** (outputs crisp "retro" sprite style)	**Automation / API Friendliness** (easy to integrate & script)	**Cost + Throughput** (compute requirements, speed, scalability)
IP-Adapter (zero-shot ref.)	**4** – High. Injects an anchor image's features to guide generation, yielding ~90% identity consistency [6]. Strong at preserving overall look (face, outfit) with no training. Fine details can still fluctuate frame to frame (e.g. a logo or accessory may subtly change [7]). The improved "IP-Adapter Plus" further decouples identity from pose, locking character texture while pose changes [8] [6].	**2** – Low by itself. IP-Adapter alone does not control pose (it treats pose as part of the image content). Without an external pose guide, generated poses are unpredictable. It's best used *with* a pose controller (like ControlNet) to achieve both identity *and* pose targets [9].	**3** – Fair. Tends to preserve the character's color scheme and style present in the reference, but minor artifacts or off-style details can appear. The base model's style biases still apply; for true pixel-art fidelity, pairing with a cell-shading model or post-process is needed.	**5** – Excellent. It's a lightweight add-on to diffusion models and available in open pipelines. No training step – just plug and play during inference [10]. This makes it easy to script and ideal for batch processing many characters in one session [11].	**5** – Excellent. Minimal compute overhead (just an extra image encoder pass and some attention injection). No per-character model swaps or training mean high throughput – e.g. one pipeline instance can sequentially handle 100+ characters efficiently [11].

LoRA fine-tune	**5** – Excellent. Training a LoRA on a character locks in their features permanently in the model [13]. The character's face, proportions, and outfit are learned so well that even extreme new poses still resemble the character. A well-trained LoRA "knows" the character's unique details (scars, emblems) that zero-shot methods might miss [14].	**2** – Low intrinsically. LoRA is about identity/style; it doesn't provide pose control by itself. You rely on text prompts or ControlNet for posing even when using a LoRA. (LoRA will ensure the *character* appears doing whatever pose you prompt, but doesn't guarantee the pose without an external pose guide.)	**4** – High. Because the LoRA can be trained on the game's art style (or even on actual sprite images), it reproduces clean lines and flat colors faithfully [14]. Using a "vector-style" or pixel-art training dataset yields outputs with distinct color zones and minimal noise [2]. Visual fidelity to 16-bit style is very strong, limited mainly by the base model's capabilities.	**3** – Medium. Applying a LoRA at inference is easy (supported in most SD pipelines via API). However, the **creation** of each LoRA is a manual/training step that must be automated or performed per character. It's scriptable (there are LoRA training scripts [15]), but integration adds complexity (managing datasets, training jobs, etc.). Once trained, though, using the LoRA in generation pipelines is straightforward.	**2** – Poor to Medium. **Upfront cost**: each character requires 800–1000 training steps [15] (20–60 minutes on GPU [16]) to ge a quality LoRA. This is a one-time cost per character, acceptable for small roster (e.g. 6 main champions [17] but **not scalable** to hundreds of characters. **Throughput** a inference is decent (slight overhead to load the LoRA file), but the need to retrain for new identities is th bottleneck [16]

ControlNet + Pose Estimation *OpenPose / DWPose*	**1** – Very Low. By itself, a pose control method doesn't ensure character identity at all; it only dictates the body arrangement. In fact, if overused, it can *override identity features*: e.g. a high-weight pose ControlNet can cause face/body distortion or interference with the character's look [19]. (This is why pipelines often **combine** pose control with an identity module and carefully balance their influence [20].)	**5** – Excellent. Delivers precise control over pose and motion. OpenPose (or better, DWPose) keypoints provide a skeletal blueprint that the generation will follow exactly [21] [22]. DWPose in particular offers dense and accurate keypoints (including hands/fingers and face orientation) [23], leading to highly faithful recreation of complex fighting stances. Multi-ControlNet setups can even add *depth* or *edge* maps to enforce perspective and outline consistency [21].	**3** – Neutral. Pose control alone does not dictate art style, but it **helps maintain pixel consistency in movement** (no limbs suddenly off-scale or off-perspective frame-to-frame). For true 16-bit *look*, one must pair it with appropriate base models or do palette post-processing. ControlNet *can* use edge detectors (HED) to preserve line art style in generation [24], so combining skeletal and outline controls yields crisper results. Overall, pose control neither adds nor removes pixel-art quality by itself – it ensures structural fidelity more than color fidelity.	**5** – Excellent. ControlNet and pose-estimation models are open-source and can run locally. OpenPose has been widely integrated (though its license is non-commercial for some implementations), and DWPose is Apache 2.0 licensed [25]. Both can be called via Python or as part of a diffusion pipeline. Automating pose extraction (from reference videos or design docs) and applying it in batch generation is straightforward.	**4** – Good. Usin a pose controller add an extra inference step per frame (running the pose detector on a reference or using a prepared pose image) and an extra condition in the diffusion UNet. This increases compute per frame ~20-30% Still, real-time generation is feasible, and there's no training cost. DWPose is optimized (distilled mode and can be faster and mo robust than older OpenPos [25]. Overall throughput remains high, though stackin multiple ControlNets (pose + edges depth) can demand more VRAM/time.

AnimateDiff (temporal SD)	**3** – Medium. AnimateDiff adds a temporal layer so frames flow smoothly, but it doesn't inherently "know" the character – identity can still drift without auxiliary constraints. In practice, it's paired with reference conditioning (IP-Adapter) or face restoration to keep the character on-model [24] [26]. On its own, expect moderate consistency: better than totally independent frame generations (due to temporal coherence), but not as tight as a LoRA/identity prior.	**4** – Good. It excels at *temporal* consistency: ensuring a continuous movement without sudden jumps. With the ability to condition each frame on a ControlNet sequence [27], it can follow a predefined pose per frame while maintaining inter-frame smoothness. However, it's limited in sequence length (typically 16–32 frames max per run) [28] and may struggle with very fast or complex motions (some motions might result in repeated or "skipped" frames).	**2** – Low. By default, AnimateDiff outputs more **smooth, interpolated visuals** (good for video, but could be counterproductive for pixel-art crispness). It can introduce slight motion blur or interpolation artifacts that are undesirable in 16-bit style. It also demands high resolution/VRAM to avoid quality loss [28]. For sharp pixel sprites, one might still need to post-process each frame (e.g. edge enhance or quantize).	**4** – Good. AnimateDiff is available as open-source extensions (e.g. ComfyUI AnimateDiff node [29]). It can be integrated into pipelines and triggered via scripts. The caution is the significant hardware needs and complexity (managing a 3D latent video cube instead of single images). Automation is achievable (especially on a headless server with a high-memory GPU), but the pipeline is heavier to manage.	**2** – Poor. **Cost**: Running e.g. 16-frame animation in one go require ~24 GB VRAM [28] and takes substantially longer than single-image generation. Fo many animations or long sequence it's a bottlenec **Throughput**: Not suited to high-volume frame output; better for sho critical sequences where smoothness is worth the cost

Model/Technique	**Identity Lock Strength** (consistently preserves character identity)	**Pose Control Strength** (precisely controls pose/ motion)	**Pixel/16-bit Fidelity** (outputs crisp “retro” sprite style)	**Automation / API Friendliness** (easy to integrate & script)	**Cost + Throughput** (compute requirements, speed, scalability)

Flux Model (e.g. “SD 3.5”)	**2** – Low. Flux (a next-gen diffusion model akin to SDXL) doesn’t inherently solve consistency. Out-of-the-box, generating the same character in multiple poses via text prompts alone remains hit-or-miss (like SDXL, it’s still probabilistic [30]). To use Flux for a specific character, you’d still apply an identity lock (e.g. fine-tune or reference prompt). So by itself: minimal identity preservation beyond what a prompt description can enforce.	**2** – Low. Flux is a general image model; it follows pose instructions in prompts reasonably well, but for precise control, you’d use ControlNet with it (which is already accounted separately). There’s no inherent pose mechanism in Flux – it relies on the same external tools for pose as SDXL or SD1.x.	**3** – Medium. Flux is noted for high-quality outputs and possibly an architecture optimized for sharper details (mentions of a “cell shaded” style LoRA used with Flux suggest it can produce clean lines [2]). Still, the base model likely leans towards general art or photorealism. To get true 16-bit sprite fidelity, one would use Flux in combination with a pixel-art LoRA or a fine-tuned checkpoint [2]. On its own, it provides clarity and resolution, but not a pixel-art palette.	**5** – Excellent. (Assuming Flux is open or permissively licensed as indicated [31] [1].) It can be run on local hardware like any SD model. Importantly, a “Flux Schnell” variant is noted for speed [32], implying it’s optimized for fast inference (good for automation throughput). Integration into existing diffusion pipelines is straightforward (likely supported by ComfyUI and Diffusers).	**5** – Excellent. Flux is posited as an **efficient** model. A “Turbo” or “Schnell” mode suggests generating images faster than SDXL [32]. Thus, per-fram cost could be lower. If quality holds at lower steps or small model size, tha means more frames per second. In a pipeline, using Flux as the bas could improve generation speed without additional cost (beyond initial model download).

Stable Diffusion XL	**2** – Low. SDXL (without fine-tune) cannot guarantee the *same* unique character appears in every image – it wasn't designed for continuity [30]. For generic archetypes (like "a knight character"), prompting can yield similar outputs, but for a specific look, it drifts. So identity lock is weak unless augmented by reference images or model fine-tuning.	**2** – Low. Like its predecessors, SDXL needs external control for exact poses. It responds to pose descriptions in text loosely, but not enough for frame-by-frame consistency in a game. We'd pair it with ControlNet for reliable results. (Thus, inherently pose control is not a strength of SDXL except via those add-ons.)	**2** – Low/Medium. SDXL excels at high-fidelity detail – which can be a drawback for retro style. It often introduces complex shading or extra elements that make an image look more 3D or photographic. To get 16-bit crispness, one must constrain it (e.g. apply a *"cell shading" LoRA* to reduce photoreal detail [2]). In absence of such constraints, SDXL's richness can actually harm the cohesive pixel-art feel (e.g. inconsistent highlights or too many colors). Therefore, while it can produce beautiful art, its fidelity to a *pixel-art style* out-of-the-box is limited.	**5** – Excellent. SDXL is available via Stability's API and open-source releases (under the CreativeML OpenRAIL license). It's widely supported in tooling. Automation is as straightforward as with SD1.5, just requiring more powerful hardware. From an integration standpoint, it's very friendly (lots of documentation and examples for using it programmatically).	**4** – Good. SDXL main cost is **computationa** it's a larger model (2+ GB and heavier or VRAM). Frame generation wil be slower than SD1.5 (perhap 2× slower if using high-res capabilities). That said, it ca produce highe quality in fewe diffusion steps sometimes, so one might use lower steps to compensate. With a high-en GPU, generating dozens of frames is still practical; but o modest hardware it's pushing limits. For a pipeline that runs man iterations, SDX might reduce throughput unless optimized.

Model/Technique	**Identity Lock Strength** (consistently preserves character identity)	**Pose Control Strength** (precisely controls pose/ motion)	**Pixel/16-bit Fidelity** (outputs crisp “retro” sprite style)	**Automation / API Friendliness** (easy to integrate & script)	**Cost + Throughput** (compute requirements, speed, scalability)

Post-Process Quantization *Palette & Downscale*	**5** – *N/A (no generation)*. This is an image processing step that doesn't create content, it refines it. It will **not alter the character's identity** except in extremely minor ways (e.g. a subtle shading detail might be lost, but the character's design remains). So, identity is effectively fully preserved when applying proper quantization.	**1** – *N/A*. Has no effect on pose control. It's applied after frames are already generated. (No scoring impact here beyond noting it doesn't help with posing at all.)	**5** – Excellent. Palette quantization is key to achieving a retro 16-bit look. By forcing each frame's colors into a limited palette (e.g. 32 or 64 colors) [33] , it eliminates stray gradients and enforces flat shading. Using K-means or similar clustering yields a **uniform palette** across the sprite [33] , ensuring consistency. Also, downscaling with nearest-neighbor or other resolution-aware methods gives crisp pixel edges. When done correctly, the result is **authentic pixel art fidelity** that pure diffusion often misses.	**5** – Excellent. This step uses standard libraries (PIL, OpenCV, scikit-image) [34] and is trivial to automate. It can be inserted into the pipeline as a script stage after image generation. It's deterministic and doesn't require ML inference, so it's perfectly reproducible and fast.	**5** – Excellent. **Compute cost** is minimal – clustering colors or resizing image is milliseconds of CPU time pe frame. **Throughput**: one can batch- process hundreds of frames almost instantaneous compared to diffusion. There's no per- frame cost concern here; it's negligible relative to generation.

Model/Technique	**Identity Lock Strength** (consistently preserves character identity)	**Pose Control Strength** (precisely controls pose/ motion)	**Pixel/16-bit Fidelity** (outputs crisp “retro” sprite style)	**Automation / API Friendliness** (easy to integrate & script)	**Cost + Throughput** (compute requirements, speed, scalability)

Key: *Identity-locking and deterministic editing are weighted as higher importance than raw visual fidelity.* From the above, **LoRA fine-tuning** and **IP-Adapter** score highest on preserving character identity (5 and 4 respectively), while **ControlNet with DWPose** scores highest on pose accuracy (5). Techniques like **post-process quantization** score high on ensuring the final **pixel aesthetic**, which is crucial for 16BitFit’s style. These will be combined in the recommended pipeline.

(Evidence sources: The decision matrix draws on the attached context and research: e.g. IP-Adapter’s ~90% identity retention [6] vs. LoRA’s near-perfect consistency [13], the value of multi-ControlNet for structure [21], and the importance of palette post-processing for uniform colors [33].)

Appendix (Commercial Tools): *See Appendix section at the end for a comparison of commercial/SaaS sprite-generation options.*

B) Recommended “Best Bet” Stack

Stack Overview: Based on the above, the top solution is a **hybrid pipeline** that combines the strengths of **reference-based generation and fine-tuning**. In practice, we recommend using *Stable Diffusion (SDXL or Flux)* as the base, **IP-Adapter Plus for anchor-based identity injection, multi-ControlNet (with DWPose) for pose control, and a LoRA fine-tuned per character for ultimate consistency**, followed by palette quantization. This stack explicitly supports anchor-locked editing, batch generation, repeatability, and an audit loop:

- **Hybrid Anchor-First Workflow:** Generate initial poses with IP-Adapter, then lock in identity with LoRA. This was identified as the most robust approach in research [35]. Starting from a single anchor image, use IP-Adapter + DWPose ControlNet to create a set of on-model images in various poses (expanding the dataset). After filtering out any off-model results, train a lightweight LoRA on these (or on provided concept art if available). Finally, generate the full animation frames using the LoRA + ControlNet for pose. This yields *deterministic reproducibility* – once a character’s LoRA is trained and a pose input fixed, the output is highly consistent. It solves the “cold start” problem by not requiring a large hand-curated dataset upfront [35]. For 16BitFit’s 6 champions (a small fixed roster), this approach is very feasible (one LoRA per character is manageable and ensures principal characters never drift in appearance). Evidence: The hybrid method achieved the stability of fine-tuning with minimal data by leveraging the anchor in an automated way [35].

- **Zero-Shot Dual-Pass Workflow (backup option):** As an alternative (or complement) for rapid iteration, an all-zero-shot pipeline can be used with careful prompt conditioning and an automated

fix pass. This would involve ComfyUI orchestrating **IP-Adapter Plus + two ControlNets (DWPose for pose, and e.g. HED for outlines)** in the first pass to generate a frame [24]. Because ControlNet can sometimes "fight" the identity (notably causing slight face changes [19]), a second pass inpaints the face or critical regions with the anchor image re-applied at high weight [20]. This dual-pass technique has been shown to fix consistency issues (especially in faces) introduced by pose constraints [20]. The end result is a frame that matches the anchor identity closely without any model fine-tuning. This stack is **fast to deploy (no training)** and suitable for batch generation of many variants or NPCs. It is, however, slightly less deterministic than the LoRA method – it may require a couple of tries for a perfect frame if the initial generation had small off-model details (the audit loop will catch these). Still, with a fixed random seed and deterministic nodes, even this approach is repeatable once tuned. It's recommended as a secondary stack for scenarios where creating a new LoRA is impractical.

Recommended Choice: For 16BitFit's needs (hero characters with *absolute* consistency), the **Anchor+LoRA hybrid stack** is the best bet. It maximizes identity lock (the LoRA baked-in knowledge means even difficult angles are rendered correctly [13]) and integrates well with an audit loop – any frame that fails a check can be regenerated with a different seed without losing identity, since the identity is in the model. Integration-wise, all components are self-hosted: Stable Diffusion, LoRA, ControlNet, etc., which avoids external dependencies. This stack is **auditable and reproducible** – given the same anchor and same random seed, it will produce the same sprite frame every time (diffusion is inherently deterministic when seeded). The pipeline can be implemented in ComfyUI or a Python script using Diffusers + LoRA loading, making it suitable for automation in a WebView/Phaser toolchain.

Moreover, this hybrid approach has an **audit-loop readiness** advantage: because identity deviations are rare (LoRA keeps things on-model), the number of rejections will be low, but when re-renders are needed (due to say a minor glitch), they won't introduce new drift. This aligns with the requirement of iterative refinement until pass [36]. The research conclusion supports this strategy: *"By combining the identity-locking power of IP-Adapters/LoRAs with the structural guidance of DWPose and rigorous post-processing, developers can build automated factories for game assets."* [1] – in other words, the combination of these tools, rather than any single model, is what yields a scalable, reliable pipeline.

Integration Considerations: This stack can be containerized or packaged as a service. ComfyUI pipelines can run headless (triggered via HTTP API or Python) for each character's animation set. The LoRA training step can be automated as a one-time preparatory phase for each character. Once done, generating all required animations becomes a matter of feeding pose controls and using the LoRA'd model, which is very fast and consistent. The audit metrics (discussed next) slot into this pipeline easily – e.g. after each frame render, the system computes SSIM/LPIPS/CSFD and either accepts or issues a re-run. The chosen stack uses only open or locally-hosted components, so it can be integrated into the game build process without external dependencies (crucial for a WebView environment, and avoids any legal/licensing concerns with SaaS).

(Justification sources: Hybrid pipeline described in [35], conclusion on combining IP-Adapter, LoRA, DWPose in an automated architecture [1], and dual-pass correction strategy from Ludo.ai research [20].)

C) Proposed Master Workflow (Agentic Pipeline)

To implement the above stack, we define an **agentic workflow** with clear roles and responsibilities. Four agents (or modules) work in sequence: **Choreographer**, **Editor**, **Auditor**, **Assembler**. Each agent operates

autonomously on its task, passing its outputs (and a file manifest) to the next. The workflow ensures generation, verification, and packaging of sprite sheets with minimal manual intervention. Below is the technical specification:

1. Choreographer (Agent) – *Plan & Prepare*
Responsible for interpreting requirements and preparing generation tasks. - **Input:** *Animation Manifest* – a structured list (could be JSON or YAML) defining each character, each animation, and number of frames. For example:

```
{
  "character": "Aria",
  "animations": [
    {"name": "idle", "frames": 4},
    {"name": "walk_fwd", "frames": 6},
    {"name": "walk_back", "frames": 6},
    {"name": "jump", "frames": 3},
    ...
  ]
}
```

This manifest can be hand-authored or generated from a template of typical fighting game animations. (Typical frame counts are informed by genre standards: e.g. ~4 frames for a basic idle loop [37], ~6 for a walk cycle, ~2-4 for attacks depending on speed, etc. A minimal Street Fighter II-style set often uses 2–3 frames for very quick actions and up to ~6 for smoother motions [38].) - **Anchor & Assets:** The Choreographer loads the *anchor sprite* for the character (the reference image that defines the character's look [39]) and any other reference data (palettes, concept art). It also retrieves pre-trained model components for the character, such as a LoRA file if one exists (filename could follow a convention like `CharacterName_styleLoRA.safetensors`). - **Pose Planning:** For each animation, the Choreographer generates or fetches the target pose sequence. This could be done by: - Using a library of *pose templates*: e.g. an "idle" template pose (perhaps taken from a known sprite sheet or a simple hand-designed stick figure) and similarly for punch, kick, etc. - Or, if motion capture/reference video is available, extracting key frames via DWPose to use as pose guides.
In either case, the output is a series of **pose images** (skeleton keypoints or pose heatmaps) for each frame. These will serve as ControlNet inputs to ensure the Editor places the character in the right positions. - **File Manifest & Naming:** The Choreographer prepares a file manifest listing each frame to generate, with naming conventions. For example: `Aria_idle_000.png, Aria_idle_001.png, ...` `Aria_walk_fwd_000.png, ...` etc. This manifest is essentially a queue of generation jobs for the Editor, including references to: - the pose image for that frame (e.g. `Aria_idle_000_pose.png`), - the anchor or identity reference to use (e.g. `Aria_anchor.png` or a path to the LoRA file for Aria), - any specific seed or variation notes (e.g. if an initial seed is predefined for consistency). - **Passing to Editor:** The Choreographer triggers the next agent by sending the manifest and all required assets. It essentially says "Here are all the frames we need for Character X, along with how they should look (pose, identity, style). Go generate them."

Implementation notes: The Choreographer can be a Python script or a small service. It might also handle the LoRA training step if needed: e.g., if no LoRA exists for a main character, Choreographer will first launch a **LoRA Trainer subroutine** – taking the anchor (and possibly generating additional reference images via IP-

Adapter in various poses) to produce a LoRA. This would integrate the "hybrid" approach pre-step [35]. Once the LoRA is ready, it's stored (for reuse in future runs) and the manifest generation proceeds. The agent also sets global parameters like the **target palette** (perhaps derived from the anchor's colors), which will be used in post-processing.

2. Editor (Agent) – *Image Generation*
Generates sprite frames as specified, using the AI models and techniques determined in our stack. Each frame generation involves a deterministic sequence: - **Load Models:** The Editor loads the necessary models into memory: - Base diffusion model (e.g. SDXL or Flux) and the configured pipeline (Diffusers or ComfyUI graph). - ControlNet models for pose (and possibly outline/edge if using multi-ControlNet). - IP-Adapter module (if doing reference injection in zero-shot mode). - The character's LoRA (for identity and style) – applied to the base model if available [13]. - **For each frame in manifest:** 1. **Conditioning Setup:** Attach the pose ControlNet with the prepared pose image (skeleton) for this frame [22]. Set ControlNet weights—e.g. a high weight (0.8–1.0) for pose to ensure accurate limb placement. If using an outline ControlNet (HED) or depth, load those with their respective condition images as well. 2. **Identity Conditioning:** There are two possible paths: - **LoRA path (preferred):** Ensure the LoRA for the character is active in the model (this already biases the generation strongly towards the character's appearance). Optionally, also use the IP-Adapter with the anchor image at a lower weight just to reinforce facial features (though often the LoRA alone suffices). - **Zero-shot path:** If no LoRA, use IP-Adapter Plus at a high weight with the anchor image [8]. This will feed the character's visual features into the UNet's cross-attention, guiding the style/identity. Additionally, include the character's textual description in the prompt if needed (to cover any traits not obvious in the single image, though typically the image is primary).
In both cases, we might also include a textual prompt for context, e.g. *"Game sprite of [Character] performing [Action]"*, along with a style tag like *"pixel art, 16-bit, clean lines"*, and crucially a fixed **seed** for reproducibility. By using the same seed and model weights, the generation is deterministic each run. 3. **Denoising & Generation:** Run the diffusion process to generate the frame. Use a relatively low diffusion step count (since we will refine if needed rather than waste time on perfection in one go). For example, 20–30 steps with a scheduler known for good geometry (Euler or DDIM) might suffice. The output is an image – e.g. `Aria_idle_000_raw.png`. 4. **Face/Detail Fix Pass (if needed):** Depending on pipeline choice, the Editor can automatically do a second-stage inpainting on critical regions: - If using the dual-pass method: Immediately apply a face mask (predefined or via face-detection on the output). Re-run the diffusion with the LoRA or IP-Adapter focused on that face region, using a very low denoise (e.g. 0.4–0.5) [19] [20] so that the pose and rest of body remain unchanged while the face snaps closer to the anchor. This corrects any slight off-model facial features. - If using the LoRA and the face came out fine, this step can be skipped. The need for it can be learned over time or based on Auditor feedback. - **Output & Logging:** Save the generated image (PNG with transparency). Mark the frame as "generated" in the manifest, including any metadata (e.g. the seed used, model hash, etc. for traceability). Continue until all frames in the batch are generated.

During this process, the Editor follows **retry rules** as instructed by the Auditor: for instance, if the Auditor flags a frame as a failure, the Editor will re-generate it (possibly with a new random seed or adjusted conditioning weight). The Editor agent can hold a queue of "to-regenerate" frames separate from the main sequence, giving priority to completing at least one version of each frame, then revisiting those that need improvement.

Technical specifics: In a ComfyUI setup, this agent might actually be a persistent server running the diffusion graph, and the Choreographer's manifest triggers node executions via API. Alternatively, as a Python module, it could use Diffusers library with loaded pipeline and call `pipeline(prompt,`

`controlnet_pose=..., image=anchor, negative_prompt=..., generator=seed)`. The key is to keep the environment loaded to avoid re-loading models for each frame (which would slow it down). The agent should also obey **stop conditions**: for example, if a particular frame has been retried N times (say 3) and still fails audit, the Editor could either escalate (flag for manual review) or attempt an alternative strategy (e.g. increase IP-Adapter strength or apply a different scheduler) and then stop if still failing. These conditions prevent infinite loops on one stubborn frame.

3. Auditor (Agent) – *Quality Assurance & Consistency Checks*
This agent automatically evaluates each generated frame to ensure it meets quality and consistency criteria before inclusion in the final sprite sheet. It implements the **Sprite Fidelity Suite (SFS)** checks as gates [40] [41]: - **Structural Similarity (SSIM):** The Auditor compares the generated frame to the original anchor sprite (and/or to the previous frame in the sequence, depending on the animation type) using SSIM. This metric catches major off-model structures (e.g. if a limb is wildly out of proportion compared to anchor). A threshold might be set (e.g. SSIM >= 0.85 vs. the anchor for key features) [42]. For instance, if a frame's SSIM falls below 0.85, it likely means the shape or outline of the character deviated (perhaps a problem with pose or a missing feature). - **LPIPS (Learned Perceptual Similarity):** The Auditor computes LPIPS between the frame and the anchor (or a reference correct frame). This detects **texture or style drift** – e.g. if the color shading changed from matte to shiny [43] or the detailing became too noisy. A low LPIPS difference implies the style remains consistent. We might require LPIPS below a certain threshold (lower is better) for a pass. - **CSFD (Cross-Scene Face Distance):** For identity, especially facial identity, the Auditor uses a pretrained face recognition model to compare the frame's face to the anchor's face. It calculates the CSFD score [44] – essentially the distance in feature space. If the distance is above a threshold (meaning the face doesn't match the same identity), that's an auto-fail. This catches the dreaded "identity swap" issue where the character's face subtly turns into someone else between frames [44]. In practice, a threshold might be set such that if face similarity < 0.95, reject frame (this ensures even subtle changes in eye shape or nose are caught, given the sensitivity of face embeddings). - **Other Visual Checks:** The Auditor can also enforce: - **No Background / Clean Alpha:** Ensure the background is fully transparent (no unwanted artifacts). This can be done by checking the alpha channel or if a known background color was used, verifying its absence. - **Anatomy & Artifacts:** Optionally, use a vision-LLM or heuristic to catch obvious errors (like missing limbs, extra arm) [45]. E.g., run a quick object detection to ensure two arms, two legs are present and properly attached. While ControlNet should prevent gross errors, this adds an extra safety net. - **Pixel Consistency:** After quantization, verify palette consistency across frames (the palette set should remain the same, or at least ensure no frame introduces a wildly new color unless intended). This could be as simple as checking that each frame's color set is a subset of the approved palette. - **Decision & Feedback:** For each frame: - If all checks pass (PASS), mark the frame as approved. - If any check fails (REJECT), log which test failed and update the manifest status. The Auditor then requests a **retry** for that frame: it can send back instructions to the Editor, such as "Frame 3 failed identity – try again with a different seed" or "increase IP-Adapter weight for face on frame 3" depending on the failure mode. These rules can be codified: e.g., *if CSFD failure (identity issue), then on retry increase anchor conditioning strength or do a face inpaint pass; if SSIM structure failure, perhaps the pose was off – ensure the pose input is correct or increase ControlNet weight; if LPIPS style failure, maybe apply the LoRA more strongly or reduce steps to avoid over-drawing details.* The Editor will then regenerate as instructed. - The Auditor also implements a **retry limit**: e.g. allow up to 2 or 3 attempts per frame. If after 3 tries a frame still doesn't pass, the Auditor flags it as a hard failure and stops further retries for that frame. (Stop condition triggers: could log this for human review. However, if our pipeline is tuned well, such cases should be rare.) - **Batch vs Individual:** The Auditor can run checks after each frame is generated, or batch them after an entire animation is generated. A practical approach is to check on the fly – as soon as the Editor produces a frame, run the audit. This way, a failure can be corrected immediately

before moving on, which is efficient and avoids compounding errors. It also means the Editor could even run two threads: one generating, one auditing, to maximize GPU usage (generate next frame while auditing the last on CPU).

The Auditor keeps track of metrics for each frame and overall. These stats (like average SSIM, number of retries) will feed into the final report or debugging info.

Implementation: The Auditor is essentially a set of evaluation functions. Libraries like OpenCV or scikit-image give SSIM; LPIPS can be computed via a pretrained network (there are public models for LPIPS that can be used in Python). For CSFD, one can use a face recognition model like ArcFace or Facenet – feed both images and get an embedding distance [46]. These computations are fast (milliseconds per image on CPU). The Auditor could be implemented as part of the main script (just functions called after generation) or as a separate process that monitors an output folder for new images and then analyzes them. In either case, it communicates back to the Editor regarding retries. In a sequential script it might just loop until pass; in an asynchronous setup, it could use a message queue or shared manifest file status.

4. Assembler (Agent) – *Final Assembly & Export*
Once all frames for an animation are approved (PASS), the Assembler collects them and builds the sprite sheet assets for the game. - **Alignment & Pivot:** It reads the pose data (from DWPose) to determine a consistent **pivot point** (anchor point for the sprite in engine). For example, it might use the character's ankle or feet position across frames [47] [48]. It adjusts each image's positioning (add padding) so that this pivot is at the same y-coordinate in every frame. This prevents the sprite from "jittering" or floating up and down in the game when animating. All frames are thus registered to a baseline. - **Sheet Packing:** The Assembler then arranges frames into a grid or strip as required. According to requirements, we use a 4px padding between frames [49] (unless our tools determined a different padding). It can create a single horizontal strip per animation, or a tiled grid if more compact. Each frame's filename and position is recorded. - **Meta-data (JSON):** The Assembler generates a JSON (or Atlas data) with frame coordinates and possibly pivot points for Phaser. For example:

```
{
  "frames": {
     "Aria_idle_0": { "frame": { "x":0,"y":0,"w":64,"h":64 }, "pivot": { "x":
32, "y":60 } },
     "Aria_idle_1": { "frame": { "x":68,"y":0,"w":64,"h":64 }, "pivot": { "x":
32, "y":60 } },
     ...
  }
}
```

This provides the game engine the means to cut and position frames correctly. The pivot (32,60 in this example) might indicate the point 4 pixels from bottom center (footing). - **Transparent Background Check:** The Assembler ensures the final PNGs have transparency. If any background remained (it shouldn't, since generation was done either on transparency or a flat color keyed out), it will remove it (using an automated background remover if needed, e.g. removebg or an internal algorithm, though our pipeline likely already handled this by generating on alpha). - **Palette Consistency:** As part of final polish, the Assembler could enforce a **unified palette** across all animations of the character. If each animation was quantized

separately, it's possible the palettes differ slightly. We can combine all frames and run one more K-means clustering to ensure a single palette for the entire character [33]. This guarantees, for instance, that "Aria's hair blue" is the exact same RGB in every frame. This step is optional if earlier quantization was done with a fixed palette, but it's a good sanity check. - **Output:** The final outputs are: - `CharacterName_SpriteSheet.png` – containing all the character's animations (could also be one sheet per animation depending on preference or engine requirements). - `CharacterName_SpriteSheet.json` (or .atlas) – metadata for Phaser3 with frame names, coords, and pivot/hitbox info. - Additionally, any per-animation sheet (if separated) plus a master manifest listing animations. - **Verification:** The Assembler might do a quick verify by loading the sheet and playing animations (if integrated in a test harness) or simply by counting frames and ensuring everything expected is present. Since all frames were audited individually, the assembly should be straightforward.

Finally, the pipeline ends with the Assembler handing off the ready-to-use sprite sheets. These can then be plugged into the 16BitFit game.

Naming Conventions & File Management: Throughout the workflow, consistent naming is used (as exemplified). Temporary files like pose images or intermediate inpaint masks can be stored in a scratch folder and cleared after assembly to keep things tidy. Each run (for a character) could be in its own directory, e.g. `output/Aria/idle/Aria_idle_0.png`, etc., ultimately consolidated into one sheet. Logging from each agent is also saved (e.g. audit metrics per frame, seeds used) for traceability – useful for the "audit loop" record, or if a human needs to review any failures.

This agentic pipeline ensures that: - **Every frame is pose-accurate and identity-locked** before it ever reaches the game (due to the Auditor gate). - **Failures trigger automatic retries** with known strategies, and nothing gets to final assembly unless it passes all checks [50]. - The output is consistent in art style (thanks to LoRA + quantization) and ready for integration (correct format, sizing, pivots).

(Sources: The structure draws from the need to generate then verify frames [40]. Audit metrics from [43] [44] and the automated rejection loop approach in [50]. Pivot alignment via pose data from [47]. Post-process palette quantization from [33].)

D) Prototype Plan

Before fully implementing the pipeline, we will execute a **small-scale prototype** to validate the chosen stack's effectiveness. The goal is to de-risk the approach by measuring key outcomes like identity consistency, frame quality, and performance. The prototype will focus on one character (or two, to compare methods) and a subset of animations.

Prototype Outline:

- **Scope:** Use one champion (e.g. **Aria**) and generate a simple animation sequence, such as a 4-frame idle and a 3-frame punch. These animations cover both a low-motion loop and a fast action, revealing how the pipeline handles subtle and significant pose changes.
- **Setup:** Prepare Aria's anchor sprite (the reference image) as input. If a LoRA is to be tested, either:
- Manually create a small dummy dataset (e.g. 5 images of Aria in different poses, which could even be generated via IP-Adapter beforehand), then train a quick LoRA on these.

- Or use the single anchor in the zero-shot mode for initial runs.
- **Generation Methods:** We will test **two pipeline variants**:
- **Zero-shot pipeline (IP-Adapter + ControlNet, dual-pass):** Generate the frames with Aria's anchor as reference (no LoRA), using DWPose for target poses. Apply the second-pass face inpainting as described.
- **LoRA pipeline:** Generate the frames using Aria's LoRA (if trained) + ControlNet poses, without needing second-pass corrections (ideally).

For both, use identical poses and seeds where possible, to compare outputs. - **Evaluation Metrics:** After generating the test frames, compute: - **Identity Drift Rate:** What percentage of frames were flagged by the Auditor for identity issues? (Ideally 0% for LoRA pipeline, and low for IP-Adapter pipeline). This can be measured via the CSFD score – e.g. we expect the face embedding similarity between each generated frame and the anchor to be above 0.9 for all frames. We define drift rate as the fraction of frames falling below that threshold. - **Audit Pass Rate:** How many frames passed all checks on the *first* try vs after retries? For example, perhaps out of 7 frames (4 idle + 3 punch), the IP-Adapter method might pass 5/7 first try and needed 2 re-renders, whereas the LoRA method passes 7/7 first try. This metric tells us the efficiency of the pipeline. - **Frame Generation Time:** Measure average time per frame for each method. Include both initial generation and any second-pass or retries. E.g., "LoRA pipeline averaged 8 seconds/frame on an RTX 3090, while zero-shot with dual-pass averaged 12 seconds/frame due to the inpaint step." This helps ensure we meet performance needs (with ~50 frames per character, an ideal target might be <1 minute total per character on a good GPU). - **Visual Inspection:** Although quantitative metrics are primary, we will also manually review the frames to catch anything the metrics might miss (e.g. if a pose looks awkward but still passed metrics). This is just for prototype feedback – the final system will rely on metrics, but early human eye checking is valuable. - **Go/No-Go Criteria:** We will consider the prototype successful (Go to full implementation) if: - **Identity Consistency:** 100% of frames generated have no noticeable identity drift. In metrics, this could mean **CSFD scores indicating same identity for all frames**, and perhaps a threshold like *at least 90–95% of frames pass identity checks on first or second attempt*. Practically, we want to see the character's face and key features remain identical across the sequence [6] [13]. A no-go would be if, say, some frames look like a different person or require extensive manual correction – that would mean the pipeline isn't reliable enough yet. - **Low Rejection/Retry Rate:** The audit loop should not be overloaded. If more than, say, 20% of frames require retries, or any frame cannot pass after 3 tries, that's a warning sign. A Go condition is if the majority of frames sail through, and any that fail pass on a retry with an automated adjustment. For example, we set a target: *at least 90% of frames pass all checks within 2 generations*. If the prototype shows we're doing far worse (lots of iterations needed), we'd refine the model or thresholds before scaling up. - **Frame Quality & Fidelity:** The final assembled mini-sheet should look game-ready: consistent art style, no glitches, proper transparency. Specifically, we expect the quantized frames to have a uniform palette and crisp lines. If we observe blurriness or inconsistent coloring between frames, that's a fail. One quantitative proxy is **LPIPS between consecutive frames** – for a stable idle, LPIPS should be extremely low (since the character essentially doesn't change much) indicating the model didn't introduce random texture changes. We can set a threshold like *LPIPS < 0.05 between frames in idle*. If style consistency metrics are higher than expected, we may need to adjust the model (e.g. enforce a stronger pixel-art LoRA). - **Performance:** The pipeline must be fast enough to be practical. We'll define a threshold like *<= 10 seconds per 512x512 frame on available hardware*. So generating ~50 frames would take ~8–9 minutes or less, which is reasonable. If the prototype reveals each frame taking 1 minute+, we might consider optimizations (or that AnimateDiff approach, but that's likely slower). A no-go would be if the pipeline is so slow or resource-heavy that generating the needed frames is not feasible in a reasonable time (this seems unlikely given our use of efficient models, but it's part of the check).

If the prototype meets these criteria, we proceed to implement the full pipeline for all characters and animations. If not, we iterate: for instance, if identity drift is found in the zero-shot method beyond acceptable levels, we know the LoRA approach (or additional training data) is mandatory – thus we'd commit to using LoRAs for all characters. Or if performance is an issue, we might try using the "Flux Schnell" model or reducing resolution earlier in the pipeline.

Additional Considerations: During the prototype we will also refine our **audit thresholds**. We might start with the suggested values (SSIM 0.85, etc.) and see if they correlate well with perceived quality. We'll adjust them so that the automated checks are neither too lenient nor too strict (initial values drawn from literature [42] [43] will be fine-tuned with actual outputs).

By the end of this experiment, we expect to have evidence (screenshots of frames, metric logs) demonstrating that our chosen stack can produce, say, Aria's idle and punch animation looking as if an artist drew them – same character every frame, smooth motion, proper retro aesthetics. Once confirmed, the blueprint can be rolled out to the remaining animations (walk, jump, block, etc.) and other characters with high confidence.

(Source references: Identity consistency targets from IP-Adapter (~90%) vs LoRA (100%) [6] [13]; the rejection loop with different seed on fail [50]; and typical performance considerations from diffusion model comparisons [51]. Frame count rationale for testing drawn from context of minimal animations in SF2 style [37] [38].)

Appendix: Commercial Tool Viability for Automated Pipeline

(A brief survey of commercial/SaaS AI image tools for sprite generation, noting whether they meet the needs of an autonomous, repeatable pipeline. Only those that support image-to-image from an anchor, batch automation, and output consistency are scored; others are noted as not viable.)

- **Leonardo.ai** – *Viable (with caveats).* Leonardo is a cloud AI art generator based on Stable Diffusion, which offers an **image-to-image "character pose" feature** and even allows training custom models (similar to LoRA/DreamBooth). It has an API for automation and supports batch jobs. **Identity Lock:** High – it recently introduced a "character reference" input that significantly improves consistency across images (users report it outperforms Midjourney's attempt at the same [52]). **Pose Control:** Medium – it doesn't natively integrate ControlNet in the UI yet, but you can guide generation with an uploaded pose image or prompt. **16-bit Fidelity:** Medium – you can apply pixel-art models/filters on Leonardo, but the outputs may require additional polishing (the tool is geared towards high-res art). **Repeatability:** Fairly good – you can set seeds and use the same fine-tuned model to get repeatable results, but the service may update models over time which could introduce slight changes. **Cost/Throughput:** Leonardo uses a credit system (paid tiers for heavy use). It supports batch generation via API, but large-scale automation would incur costs and depends on cloud inference speed. *Overall:* Leonardo can be used for consistent character generation (especially with its character reference feature) [52], and it has an API, so it's one of the few SaaS that could fit an automated pipeline. Score: **(Offers required features; viable, but cost and reliance on external service are downsides).**

- **Midjourney** – *Not viable for autonomous pipeline.* Midjourney produces stunning images but fails key criteria: **no direct image-to-image anchor** control (aside from using an image as a loose inspiration,

it cannot reliably replicate a specific character across different poses), **no official API or batch automation** (it runs via Discord bot, and while some unofficial workarounds exist, it's not made for large-scale programmatic use), and **lack of repeatability** (even with the same prompt and seed, outputs have variance, and Midjourney model versions update periodically, breaking consistency). Many users note Midjourney struggles with maintaining a consistent character in multiple images [53] – it might change clothing or facial features unless heavily guided, and even then, it's not guaranteed. Therefore, Midjourney cannot be integrated as a deterministic frame-by-frame generator. Score: **Not viable** – it's a creative tool for concept art, not for generating sprite sheets with exact consistency.

- **Ludo.ai (Pose Animator)** – *Partially viable, but closed ecosystem.* Ludo offers a specialized "Pose Editor" for turning one sprite into full animations [54] [55]. It can maintain consistency because it uses the *same source image* to generate all poses of that character [56]. It has built-in pose presets and even auto-animates between them. **Anchor support:** Yes (the "First Frame" you create effectively acts as an anchor that all subsequent frames derive from) – however, it's *internally* using that image; you might not be able to upload your own drawn anchor, it generates one via AI from text initially [57]. **Automation:** Low – Ludo is a web platform with a GUI. There is no public API documented for fully automated batch processing; it's meant for one-by-one usage via their interface. **Repeatability:** Moderate – since it's AI-based, results can vary, but because it's a guided pipeline, it likely yields similar results given the same initial sprite. **Quality:** Ludo's outputs are reportedly decent for quick prototyping, and it even has a pixel art filter [56]. But it's a black-box: you can't tweak the model much. **Cost:** It's a subscription model and presumably charges for generation. *Overall:* Ludo showcases the kind of pipeline we want (one-image to many animations), but without an API or the ability to deeply configure it, it's not suitable for *autonomous* integration. We mark it as **"Not viable for autonomous pipeline"** – it's fine for manual use by an artist, but we cannot script it to produce 100s of frames hands-free.

- **Scenario.gg** – *Viable (for trained character models).* Scenario provides a service to train custom diffusion models on your assets and generate variations. You can upload a character's images to get a tailored model. **Anchor:** Yes, via model training (the anchor concept is baked into the model, similar to a LoRA). It doesn't have a one-shot reference like IP-Adapter, but after training, you can use a token to invoke the character. **Automation:** Scenario has an API for generation, so you can integrate it into a pipeline. **Pose Control:** Currently limited – Scenario's focus is on static asset generation. It doesn't natively support ControlNet pose conditioning in its interface (as of recent info). You'd have to prompt for poses, which is not reliable for exact frames. This is a major limitation for sprite workflow. **Repeatability:** High – since you can fix seeds and use the custom model, you'll get repeatable outputs. **Cost:** It's a paid service (you pay for training and generation credits). It could become expensive for large frame batches. *Overall:* If Scenario adds pose control or if one is willing to generate many images and manually pick frames that resemble desired poses, it could be used, but that's not truly autonomous. For now, Scenario is great for creating consistent *static* images or a few key frames, but **not a turnkey solution for sprite sheet animation**. Score: **Not viable for full pipeline** (viable for character concept art or model generation, but lacking precise pose tools for automation).

- **PixelVibe by Rosebud** – *Not viable (no guaranteed consistency).* PixelVibe is aimed at generating 2D game assets quickly from text. It can output sprite-like images, but as of current capabilities, it's more about generating random characters or items in a style, not about maintaining a specific input

character across multiple outputs. There's mention of uploading a concept image on PixelVibe's new iteration or a related tool (PixelLab has a feature to upload a reference sprite for consistency [58]). **If** such a feature works, it might allow anchor-based gen. However, even if anchor is supported, **automation** is a question – these are web tools with UIs, no evidence of an API for bulk generation. **Repeatability:** unknown/likely low (these services often fine-tune a model behind the scenes but not exposed for user reuse in a controlled way). *Overall:* Without strong evidence of an API and proven consistent-character output, we consider it not suitable for our needs. Score: **Not viable** – geared towards one-off asset creation.

- **Adobe Firefly** – *Not viable for this use-case.* Adobe's Firefly (and related Photoshop generative fill) can do image-to-image and even frame interpolation, but it doesn't allow a user-provided model or full character lock (and it has content restrictions). There's an API, but Firefly's terms and unpredictability, especially with exact character replication, make it unsuitable. It's also not oriented to pixel art. **No fine control**, **no guarantee on consistency** beyond maybe similar prompt = similar style. So we exclude it for an autonomous sprite pipeline.

- **Stable Diffusion Web APIs (e.g. Stability AI's API, Replicate)** – *Viable (with custom code).* These are essentially ways to run Stable Diffusion in the cloud. They do support image-to-image and allow specifying seeds. With effort, one could use them along with open-source models (including ControlNet) if the API exposes those features. For example, Stability's API might not directly support ControlNet yet, but Replicate has community models where you could call a ControlNet-enabled diffusion model. **Automation:** yes (they are APIs). **Repeatability:** yes (seeded diffusion). **Identity lock:** only as good as the model you use – you'd likely have to fine-tune a model or use textual inversion via these APIs, which is possible but requires uploading training data. The downside is cost per image and potentially slower iteration compared to local GPU. *Overall:* If one doesn't have a GPU, this is an alternative. It's basically "roll your own" pipeline on rented hardware via API. Viability is high technically, but economically you'd need to watch costs for large frame counts. Score: **Viable** (but since it's essentially the same models we chose, just hosted, it's an alternative deployment method rather than a different tool).

In summary, **none of the off-the-shelf commercial SaaS tools perfectly meet the autonomous, repeatable pipeline requirement out-of-the-box**. Leonardo.ai comes closest with its reference-based generation and API (one could script Leonardo to generate poses of a character in batch, but one must still solve pose input, possibly by generating poses separately). Most others either lack an API (Midjourney, Ludo) or lack the pose control needed (Scenario, PixelVibe). Additionally, relying on a SaaS introduces uncertainties (model changes, service downtime) and licensing concerns (for example, sprites generated might have ambiguous copyright depending on the service's training data, whereas our open-source pipeline can be constrained to properly licensed models [59]).

Conclusion (Appendix): We prioritize the open-source pipeline detailed in sections A–C. Commercial tools were examined for completeness: they may assist in prototyping or inspiration, but for a fully autonomous sprite generation factory, the in-house stack is favored for determinism and control. The few viable services could be fallback options (e.g. Leonardo's consistent generation if our pipeline needed a quick cloud boost), but otherwise will not be core to the implementation.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 21 22 23 25 30 31 32 33 34 35 41 42 43 44 45 46 47 48 50 51 59 Gemini-Deep-Research_Refining AI Sprite Generation Pipeline.pdf
file://file_000000002504722f814a7f9d12689bb4

15 18 19 20 24 26 27 28 29 54 55 56 57 Perplexity-AI-Assisted 2D Sprite Sheet Generation_ Research.pdf
file://file_00000000127471f5ab56b71fb0ec6819

17 36 39 40 49 context_packet.md
file://file_000000004758722fa92f687e795d3906

37 How many frames are in an idle animation? : r/animation
https://www.reddit.com/r/animation/comments/16poi90/how_many_frames_are_in_an_idle_animation/

38 HTML5 Game Devs Forum - HTML5GameDevs.com
https://www.html5gamedevs.com/topic/2168-sprite-animation-how-many-frames-for-a-sprite-animations-is-too-many-frames/

52 Leonardo’s Character Reference vs. Midjourney’s : r/leonardoai
https://www.reddit.com/r/leonardoai/comments/1czrcyr/leonardos_character_reference_vs_midjourneys/

53 Midjourney vs Leonardo AI: Best Character Art Generator in 2025
https://demodazzle.com/blog/midjourney-vs-leonardo-ai

58 PixelLab - AI Generator for Pixel Art Game Assets
https://www.pixellab.ai/