ChatGPT

AI Cinematic Video Generation Pipeline: Feasibility, Models, and Design

1. Feasibility of Low-Cost AI Video Generation Today

Generating a **5–60 minute cinematic video using AI** is *partially feasible* today with free or low-cost tools, but there are significant practical limits to overcome. Current open-source text-to-video models can create short clips (typically **5–10 seconds each**) with reasonable quality [1] [2]. These clips can be stitched to form a longer video, but achieving a coherent 5–60 minute film is challenging. Key feasibility considerations include:

- **Clip Length Limits:** State-of-the-art generative models (open or commercial) are generally limited to producing only a few seconds of video per generation. For example, the open **CogVideoX** model generates up to ~6–10 second videos at 8 fps [1], and **Stable Video Diffusion** (Stability AI's model) is trained for 14–25 frames (≈2–4 sec) per run [3] [4]. Commercial services like Runway and Luma similarly cap generation at ~5–10 seconds per clip [2]. This means a 5-minute video might require ~30–60 AI-generated shots, while a 60-minute film could require **hundreds of shots**, each rendered separately and then concatenated.

- **Computational Requirements:** High-quality video synthesis is **VRAM-intensive and slow**. Diffusion-based video models essentially generate multiple frames in one go, akin to a high-batch image generation [5]. Running these on consumer GPUs is possible with optimizations (e.g. model offloading and chunked decoding [5]), but it is still heavy. For instance, CogVideoX's large 5B model in full precision consumes ~26 GB VRAM (bfloat16) for a 6-second clip, though with optimizations it can run in ~5 GB VRAM at slower speeds [6] [7]. In practice, using open models locally may demand a **16–24 GB GPU** (or better) for decent resolution and frame rate. Lighter methods like AnimateDiff (which adds a motion module to Stable Diffusion) can run on ~8–12 GB VRAM by processing frames sequentially, at the cost of speed. **Generation time** is significant: e.g. CogVideoX ~3 minutes for a 6s clip on an A100 [7], so creating many minutes of footage would take many GPU-hours (though parallelizing across multiple GPUs or machines can help).

- **Consistency and Coherence:** Ensuring a long video **looks consistent and tells a coherent story** is the biggest challenge. Current text-to-video models have no long-term memory of previous shots – each clip is generated independently from its prompt. This can lead to **character and scene inconsistencies** (appearance changes, layout resets) when stitching shots. Maintaining the same character appearances across dozens of clips may require workarounds, such as using the last frame of one shot as the first frame of the next or fine-tuning models on specific characters. Even then, subtle differences accumulate, causing continuity errors [8] [9]. Narrative coherence (e.g. logical camera movement from scene to scene) must be enforced by the pipeline – the AI won't automatically remember the story arc. In practice, achieving coherence might involve using **consistent prompts, reference images, or control signals** between shots, and possibly manual intervention for key transitions.

- **Quality vs. Cost Trade-offs:** Free tools can produce impressive short clips, but achieving *film-quality* 60-minute output is currently **impractical without significant resources**. An hour of video at 24 fps is ~86,400 frames – far beyond what current diffusion models can render efficiently one-by-one. Strategies like re-using backgrounds or looping segments can reduce new content generation. Nonetheless, expecting a fully automated, high-resolution hour-long film with today's free models is unrealistic. Most likely, one would settle for a shorter proof-of-concept film (1–5 minutes) or a stylized animation instead of a photoreal epic, unless substantial compute is available. Longer videos are **possible** – e.g. by continuously extending clips using image-to-video extensions [10] – but quality often degrades over long durations, introducing flicker or artifact build-up [11].

Despite these hurdles, **it *is* possible** to craft a pipeline today that automates much of the process with open-source tools. Short AI-generated videos (under ~5 minutes) with coherent style have been demonstrated using clever techniques (consistent prompting, leveraging image-to-video to "extend" scenes, etc. [10]). The pipeline design below addresses these issues (clip length limits, consistency, compute cost) with a modular approach. In summary: low-cost AI video generation is **feasible for short films or prototypes** using open models, but pushing to 60 minutes will test the limits of current tech, likely requiring a hybrid strategy and significant compute time. Key challenges remain consistency of characters and settings, achieving truly cinematic quality, and controlling costs – but ongoing model improvements are rapidly extending what's achievable.

2. Survey of Open-Source vs Commercial Video Models

A variety of AI video generation models exist. Below we **compare leading open-source (self-hostable) models and commercial services** relevant to this pipeline, noting each model's type, capabilities, and requirements:

Open-Source / Self-Hostable Models:

- **CogVideoX** – *Text-to-video diffusion model.* CogVideoX (from Tsinghua/Zhipu AI) is a state-of-the-art open model that generates up to ~10-second videos from text prompts [12] [1]. It uses a large diffusion transformer (2 billion or 5 billion parameters) and a specialized 3D VAE for video [13]. Quality is good: it can produce **768×1360 resolution** clips with basic narrative coherence and significant motion [12]. The model weights are openly available [14] under the CogVideoX License (per model card) [15]. **Hardware:** CogVideoX is heavy; the 5B version needs ~26 GB GPU memory for FP16 inference, though with optimizations it can run on ~5–6 GB (at reduced speed) [6] [7]. Inference on a single A100 (80GB) takes ~180 seconds for a 6-second video (50 diffusion steps) [7]. Python integration is easy via HuggingFace Diffusers – a `CogVideoXPipeline` is provided for generating videos in Python [16]. *Cost per minute:* Running locally is free aside from compute costs. If using cloud GPUs, note that ~6 sec takes 3 minutes on A100 [7]; generating 1 minute could cost a few dollars of GPU time. Overall, CogVideoX offers **high-quality free generation** if you have sufficient VRAM or are willing to use optimizations/offloading.

- **Stable Video Diffusion (SVD)** – *Image-to-video (and text-to-video) latent diffusion.* Stable Video Diffusion is an open model released by Stability AI (2023) for high-res video generation [17]. It extends Stable Diffusion into time: one variant generates **14 frames**, another (SVD-XT) generates **25 frames** (at ~1024×576 resolution) per run [3] [4]. By design it's an **image-to-video** model – you provide a starting keyframe image and it predicts subsequent frames with motion. However, the

research paper also trained a text-to-video version that is competitive with closed-source models [18]. *Quality:* SVD yields sharp frames and learned motions; it's considered state-of-the-art among open models, approaching closed Gen-2 quality on short clips [18]. The model is open for research (CreativeML OpenRAIL M license on weights) [19] [20]. **Hardware:** SVD is memory-intensive – effectively doing diffusion in a 3D (HxWxT) latent space. By default it generates all frames simultaneously, which "is very memory intensive" [5] (equivalent to a large batch of images). The authors suggest strategies like sequential frame decoding and model offloading to reduce VRAM needs at the cost of speed [5] [21]. In practice, a high-end GPU (≥16 GB) is recommended for 2–4 second clips; the diffusers example shows how to offload to CPU and chunk decoding frame-by-frame [5] [22]. Python integration is supported via the `StableVideoDiffusionPipeline` in HuggingFace Diffusers [23]. *Cost:* It's free to run if you have hardware. Generating one 4-second 576p clip might take tens of seconds to minutes depending on hardware (especially with heavy offloading). This model's strength is in **turning a static image into a short video** – useful if you want to animate keyframes with an open tool.

- **AnimateDiff** – *Stable Diffusion extension for short animations.* AnimateDiff is a clever open approach to leverage existing text-to-image models for video [24] [25]. It adds a pretrained "motion module" (often a small network or LoRA) that, when enabled, conditions the diffusion process to produce a sequence of frames with coherent motion [26]. Essentially, you pick any Stable Diffusion 1.5 model (for desired visual style), enable AnimateDiff, and get (for example) 16 or 32 frame videos from a single prompt [27]. *Quality:* The visual detail comes from the underlying SD model (so it can be very high, e.g. photorealistic if using a photoreal model). Motion tends to be **generic and looping** (because the model wasn't trained on a specific story) – e.g. a person might just sway or the camera pans slowly. It can't follow a complex scripted action beyond what the prompt generally describes [28]. Still, for simple character animations or scene ambience, it's quite effective [29]. AnimateDiff is fully open-source (MIT-licensed extension) and can be integrated into UIs like AUTOMATIC1111 or ComfyUI [30] [31]. **Hardware:** Because it can generate frames sequentially or in small batches, AnimateDiff can run on modest GPUs. A typical setup might generate 32 frames at 512×512 on an 8 GB GPU by processing 8 frames at a time and stitching the results. The tradeoff is speed: e.g. ~4 seconds of video might take a few minutes to render on a mid-range GPU. *Python integration:* via extensions or by loading the motion module in a custom script. *Cost:* Free on local compute – a very cost-effective method, though it may require more manual tinkering to get smooth results (users often use tricks like frame interpolation to increase FPS [32] or "prompt travel" to vary the prompt over time [33]). Overall, AnimateDiff is a **cost-efficient solution for short, simple clips**, especially in a consistent style (since you can use the same base model for all shots).

- **Open-Sora** – *High-quality open text-to-video family.* Open-Sora is an open initiative to "democratize" video production by HPC-AI Tech. It has released a series of text-to-video diffusion models with increasing power. Notably, **Open-Sora 1.1** introduced support for *2s to 15s videos up to 720p* resolution, any aspect ratio, and even "infinite time generation" via clip extension [34]. The latest **Open-Sora 2.0 (11B)** model (released Mar 2025) is an 11 billion parameter model that reportedly achieves quality on par with top models like Microsoft's 11B HunyuanVideo and Runway's 30B model [35]. *Quality:* Open-Sora has made steady improvements – v1.3 (1B params) improved VAE and transformer design for better fidelity [35], and v2.0's 11B model is at or near state-of-the-art in open video generation. Samples (at 720p) show coherent motion and decent text alignment, though like others it can struggle with very complex prompts. All code, training pipeline, and checkpoints are open-sourced (Apache-2.0 license) [36]. **Hardware:** The largest model (11B) will require substantial

VRAM (likely 20GB+ for inference). However, Open-Sora emphasizes efficiency: their blog noted using ColossalAI and other optimizations to cut training costs in half [37], and presumably those apply to inference as well (for example, using FP16/BF16 and possibly model sharding across GPUs). They even mention running on an H200 (next-gen GPU) and cost-saving vouchers [37]. For pipeline design, one could run a smaller Sora model (e.g. 1B or 2B) locally for drafts, then use the 11B model on a bigger machine for final renders (this aligns with Plan B hybrid ideas). *Integration:* Open-Sora provides code on GitHub and a HuggingFace demo [38], making Python integration straightforward. *Cost:* Free if self-hosted; needed GPU hours would be comparable to CogVideoX or slightly more due to larger size. Given its openness and quality, Open-Sora (especially v1.x for lower resource use) is a top candidate for a **fully local pipeline**.

- **HunyuanVideo** – *Large-scale video foundation model by Tencent.* HunyuanVideo is an open-source 13B-parameter video model introduced in late 2024, claiming performance **comparable or superior to leading closed models** [39]. It uses a "unified image and video generative architecture" with a transformer that can handle both images and videos in one model [40]. It also employs a *Multimodal Large Language Model (MLLM)* text encoder for better text-video alignment [41] and a causal 3D VAE for spatial-temporal compression [42]. *Quality:* According to its authors, HunyuanVideo outperforms Runway Gen-3, Luma 1.6, and other top-tier models in human evaluations [39], demonstrating excellent visual quality and motion diversity. It's essentially at the cutting edge of video diffusion research. The code and model weights were released (Dec 2024) under an open-source license (likely Apache 2.0) [43] [44]. **Hardware:** As a 13B model, running it is non-trivial – the team provided parallel inference code (e.g. sequence parallelism using multiple GPUs) and even an 8-bit quantized version [43] [45]. Community contributions have enabled running HunyuanVideo on one high-end GPU with FP8 or INT8 quantization [46], and ComfyUI nodes exist for it [47]. Still, one would need a serious GPU setup (at least an A100 80GB or a couple of smaller GPUs in tandem) to use the model effectively. *Integration:* Provided via official PyTorch code and expected to come to diffusers as well [48]. *Cost:* The model is free; the main cost is cloud compute if you don't have the hardware. If renting, one might use multi-GPU instances for speed – fortunately, the code supports multi-GPU for faster inference [48]. HunyuanVideo is a prime choice if aiming for **highest quality open generation** and you can allocate the resources – it could produce the most film-like results among open options (especially for complex scenes).

(Other open models:) There are a few more worth mentioning briefly. **VideoCrafter** (by ModelScope/Alibaba) was one of the first open text-to-video models; it can generate ~2-second 256×256 clips and is available as part of a toolkit [49], but its quality is now behind newer models. **Text2Video-Zero** (by Picsart AI) explored zero-shot video generation by leveraging pre-trained image models, but it's more of a research prototype [50]. **AnimateDiff v3 / SDXL** versions are emerging (which work with SDXL base models for higher resolution) [51]. Also, **Deforum** and other notebook workflows exist to animate images using stable diffusion and depth maps, though these require a lot of manual effort and are less applicable to an automated pipeline. Overall, the models detailed above represent the *leading edge* for open-source video generation as of 2025.

Commercial AI Video Models:

- **Runway Gen-2 (and Gen-3)** – *Text-to-video generative models by RunwayML.* Runway's Gen-2 model is one of the best-known commercial text-to-video systems. It creates ~4–8 second video clips from prompts, with results often **highly creative and on prompt** (it was trained on a large dataset

including many cinematic scenes). Gen-2 does not generate audio. Runway has also previewed Gen-3 (in alpha), which aims at longer and more consistent videos; Gen-3 allows 5 or 10 second generations and improves temporal coherence, but costs more credits [52]. *Quality:* Runway Gen-2 is generally very good at artistic and cinematic shots, though it may produce artifacts on faces or fast motions. It's closed-source and available through Runway's web app (no downloadable model). **Pricing:** Runway uses a credit system. For example, the Standard plan (~$12/month) gives ~2250 credits monthly, which equates to about **450 seconds of Gen-2 video** generation [53]. In other words, Gen-2 uses ~5 credits per second of video. The free tier includes 125 credits one-time [54] (about 25 seconds of video). Gen-3 (when available) is pricier: **10 credits per second** [52]. Under the Unlimited plan (~$28/mo) you get 2250 credits/mo plus an "unlimited" mode for experimentation (with some quality or speed limits) [55] [53]. Practically, Gen-2 costs on the order of **$0.02 per credit**, i.e. **$0.10 per second** of video (or ~$6 per minute) on the standard plan. *Integration:* Runway does not have an open API for Gen-2; it's meant to be used on their platform (web or desktop app). However, for a pipeline, one could use Runway's export options or possibly an unofficial automation (Runway provides an HTTP API for some services, but Gen-2's API is not publicly documented). Overall, Runway Gen-2 is a **strong choice for quality**, but generating many minutes of content could become costly (e.g. ~\$360 for 60 min at standard pricing, ignoring that such volume might require an enterprise plan).

- **Luma AI (Dream Machine, Ray Model)** – *High-fidelity text/image-to-video via API.* **Luma AI** offers a suite of models (Ray 1.6, Ray 2, Ray 3) accessible through their Dream Machine app and API [56] [57]. Luma's focus is on *realistic 3D-aware video generation* – their models often excel at smooth camera motion and consistent scenes, leveraging techniques from NeRF and 3D capture (Luma is known for its 3D scanning tech). You can input text, images, or video, and generate new video outputs ("reimagine" existing video or create from scratch) [58]. *Quality:* Very high – Luma's latest Ray3 model produces **coherent motion, logical event sequences, and ultra-realistic details** (per their research posts) [59]. It can do things like dynamic camera moves, subject interactions, etc., better than many competitors. Luma even supports features like *reference images for characters* and *keyframe control* in Ray3 Modify [60], which is great for consistency (you can tell it to use the same character in different shots). **Pricing:** Luma uses a credit subscription model. The **Plus plan ($23.99/mo)** provides 10,000 credits [61] with full quality access (4K up-res, no watermark). Credit consumption depends on model, resolution, and duration [62]. For example, using the Ray2 model at 720p: a **10s video costs ~320 credits** [63]; with Ray3 at 720p: **10s costs 640 credits** [64]. At plus plan rates, 640 credits ≈ $1.54 (since $23.99/10000 = $0.002399 per credit). That is about **$9.24 per minute** for Ray3 720p. Lower resolutions or the Ray1.6 model are cheaper (e.g. Ray1.6 5s = 80 credits [63]). The Lite plan ($7.99) has fewer credits and only "draft" outputs (watermarked, lower res) [65]. For serious use, the Plus or Unlimited ($75.99/mo) plans are needed [66] [67]. *Integration:* Luma **provides a REST API and Python SDK** for developers [57], so our pipeline could programmatically generate clips on Luma's servers. This makes Luma ideal for a hybrid approach: use local compute for some tasks, and call Luma's API for certain shots that need top quality or complex 3D movements. In summary, Luma AI offers **industry-leading quality with a flexible API**, at a cost that is moderate per clip (on the order of $1–$2 for a 10-second HD clip on paid plans).

- **Pika Labs** – *Idea-to-video platform with fast stylized generation.* **Pika** (pika.art) is a popular web-based AI video generator known for its speed and ease of use. It specializes in **short (5 second) videos**, often with a stylized or music-video-like aesthetic. Pika provides various modes and templates: "Pikascenes" (text-to-video), "Pikadditions" and "Pikaswaps" (editing or integrating images),

"Pikatwists" (style transfers) – these correspond to different model versions (Turbo or Pro engines at different quality levels) [68] [69]. *Quality:* Pika's outputs are generally lower resolution (up to 480p on basic models [70], possibly higher on Pro) and geared towards creative effects rather than photorealism. It's great for quick concept videos, with vibrant colors or anime styles, etc., but not as detailed as Runway or Luma. **Pricing:** Pika uses a credit subscription system. For example, the **Standard plan ($28/mo)** gives 700 credits [71]. Different generation types consume different credits – e.g. using the Turbo model for a basic text-to-video costs 10 credits per 5s video, whereas the highest quality Pro model with "Pikatwist" style costs 80 credits per 5s [72] [73]. At Standard plan rates, that means one 5s clip can cost as little as 10 credits ($0.40, using Turbo) or as much as 80 credits ($3.20, using Pro). That translates to **$8 per minute (Turbo)** up to **~$38 per minute (Pro)**. There is also a free tier (80 credits/month, watermark, limited features) [74]. Pika's **strength is cost-efficiency for drafts** – e.g. for ~$8 you could generate 1 minute of rough video (Turbo mode) to storyboard your film, then later invest in higher quality renders for select shots. *Integration:* Pika currently is primarily via their web UI; an API exists but is in closed beta (the site's API docs require login). In a pipeline, one might use their web interface manually or use browser automation for batch processing shots. Pika is best suited for **fast prototyping and stylized content** on a budget.

- **Kuaishou Kling** – *Advanced text/image-to-video with audio (Chinese market).* **Kling AI** is a suite of cutting-edge generative models developed by Kuaishou (a major Chinese tech company). Kling has evolved through versions 1.0 up to **2.6** and a new **O1** series [75]. It's offered via apps and the Scenario.com platform for international users [76]. *Capabilities:* Kling stands out for **character animation, motion consistency, and camera realism** [77]. The latest **Kling 2.6** models can generate video **with native audio** – including voices, sound effects and ambiance in sync [78] – which is unique. (For example, Kling 2.6 T2V will not only show a singer performing, but also generate the singing voice and background noise in one pass [78] [79].) There are also image-to-video modes and even a "Motion Control" mode allowing precise control of camera paths and character movement [80]. Kling O1 models provide specialized tools for structured animation, video editing, and reference-guided generation [81]. *Quality:* Arguably among the **best in the world** as of 2025. Kling 2.6 offers smooth, jitter-free motion and excellent temporal coherence [77] [82]. It handles complex multi-shot sequences well and maintains consistent character looks (especially when using reference image conditioning in Pro modes [83]). Users praise it for avoiding the flicker and artifacts common in other systems [77] [84]. **Pricing:** Kling is accessed through credits on platforms like Scenario or via Kuaishou's own app. According to user reports, **a 5-second video might cost ~10–35 credits** depending on quality settings [85]. Scenario's credit pricing for Kling isn't publicly listed, but one Reddit user summarizes that **1 minute of video can cost \$2–\$15 in credits** on Kling, assuming minimal re-generations [86]. In practice, many users spend around \$50 per final minute after iterations [87]. For example, one data point: *"10-second 720p clip = 200 credits (~\$2) on a \$30/3000 credits plan"* [88]. This implies \$0.20 per second, i.e. \$12 per minute for high quality – quite cost-effective given the quality (much cheaper than hiring animators or live-action footage). *Integration:* Currently through Scenario's web interface or API. Scenario provides an API/SDK where you can choose Kling 2.x models for generation, which a Python pipeline could utilize (with an API key). The process involves sending a prompt (and optional images or audio for some modes) to the endpoint and receiving the video. With Kling's advanced features (like providing a **JSON prompt with detailed scene description**, a technique some use to control Kling outputs [89]), one can really script out each shot. In summary, Kling AI is a **premium solution for top-quality cinematic video**, offering capabilities like built-in audio that can reduce the need for separate narration for certain

scenes. The downside is it's proprietary and requires paid credits, but for a few critical shots it might be worth the cost in a hybrid pipeline.

To recap the models, the table below compares key characteristics:

Model	Type	Quality	License	Max Clip	Hardware (VRAM)	Python Support	Cost per Minute
CogVideoX (5B)	T2V diffusion	High (768×1360, ~16fps)	Open (CogVideoX)	~6–10 sec [1]	~5–26 GB (opt. vs full) [90]	Yes (HF Diffusers) [16]	Free (local; ~3 min/6s clip) [7]
Stable Video Diff.	I2V (T2V finetuned)	High (576p, sharp frames)	Open (Research)	2–4 sec (14–25 fr) [3] [4]	>10 GB (needs offload) [5]	Yes (Diffusers Pipeline) [23]	Free; heavy compute (frame-chunking)
AnimateDiff	T2V via SD1.5	Med–High (detail = SD model; motion = simple)	Open (MIT)	~4 sec @ 8 fps (32 fr) [27]	8–12 GB (sequential gen)	Yes (WebUI ext./ ComfyUI) [31]	Free; moderate GPU (minutes per clip)
Open-Sora 2.0	T2V diffusion	High (720p, SOTA-level)	Open (Apache-2.0) [36]	15 sec+ (extensible) [34]	Very high (11B params, multi-GPU) [35]	Yes (Open GitHub/ Spaces) [38]	Free; compute-heavy (efficiency optimized)
HunyuanVideo	T2V diffusion	Very High (13B, top-tier)	Open (Tencent)	~? 5–10 sec (not stated)	Extremely high (13B, multi-GPU FP8) [91]	Yes (Code released, diffusers & ComfyUI) [45]	Free; requires enterprise GPU resources
Runway Gen-2	T2V (closed)	High (creative, 720p max)	Closed (SaaS)	~4–6 sec (Gen-3: 10s) [52]	N/A (cloud service)	Limited (UI, no public API)	~$6/min (std plan: 5 cred/sec) [53]
Luma Ray/ Dream	T2V / I2V (closed)	Very High (cinematic, 3D-consistent)	Closed (API SaaS)	5 sec (Free) – 10 sec (Paid) [2]	N/A (cloud)	Yes (REST API) [57]	~$9/min (720p, Ray3 on Plus) [64]

Model	Type	Quality	License	Max Clip	Hardware (VRAM)	Python Support	Cost per Minute
Pika Labs	T2V (closed)	Medium (stylized 480p)	Closed (Web)	5 sec	N/A (cloud)	No official API (UI only)	~$8–$30/min (plan-dependent) [72] [73]
Kling 2.6	T2V / I2V (closed)	Very High (film-like; +audio) [77] [78]	Closed (Scenario API)	5–10 sec (HD)	N/A (cloud)	Yes (Scenario API)	~$12/min (est. 200 cred/10s) [88]

(Table Notes:) "Max Clip" indicates typical generation durations (often limited by providers). "Hardware" is for self-hosted models; cloud models are run on vendor servers. "Cost per Minute" for closed models is approximate, assuming paid plans and minimal retries (actual costs can be higher with multiple generations per shot). For open models, cost is just compute time (no licensing fee). All open models above support **Python integration**, allowing them to be embedded in our pipeline code. The closed ones either have official APIs (Luma, Scenario for Kling) or can be used via their applications.

3. Cost-Efficient Strategy (Fully Open-Source vs Hybrid)

To minimize costs while achieving acceptable quality, we propose two strategies:

- **Plan A: Fully Open-Source / Local Compute.** This approach uses *only open models and local resources* to avoid any per-minute fees. It is viable if you have access to a decent GPU (or can rent one cheaply) and are willing to accept some limitations in quality/length. All stages of the pipeline – from image generation to video rendering to audio – would use free tools. For example, Plan A could involve using **Stable Diffusion + AnimateDiff** for most video clips (since this can run on a single consumer GPU) and leveraging **CogVideoX or Open-Sora** for any shots that need more complex motion or higher fidelity (running these on a larger GPU when necessary). The narration audio can be generated with an open Text-to-Speech model (e.g. Coqui TTS or an offline Tacotron2 model), and music from free sources or AI music generators like Mubert's free tier. The **practical limits** here are that everything will be slower and potentially lower-quality than state-of-art. You might generate 1–2 minutes of footage as a proof-of-concept on a 8–16 GB GPU, but scaling to 60 minutes would be extremely time-consuming. Also, maintaining character consistency with only open tools may require additional tricks (like fine-tuning a Stable Diffusion model on the main character's face, which itself is time and compute intensive). **However, Plan A ensures minimal monetary cost** – essentially the cost of electricity or cloud GPU time. If using cloud GPUs, one can optimize costs by using spot instances or services like Vast.ai for short periods. For example, renting a 24 GB GPU for a day (~$10–$20) could produce a few minutes of video with careful pipeline optimizations, which is far cheaper than commercial per-minute prices.

- **Plan B: Hybrid Approach (Local + Paid Services).** This strategy **combines local computation with selective use of commercial AI services** for the most demanding tasks, thus balancing quality and cost. The idea is to do as much as possible offline (story processing, basic shot generation, draft

videos, etc.), and **call a paid API only for segments or features where it's truly needed** – essentially *pay by exception*. For instance:

- Use local models to generate **draft versions of each scene** (perhaps lower resolution or simpler style), to validate timing and narrative flow without spending credits.
- Identify shots that are especially complex or important (e.g. a dramatic close-up of a character's face, or a fast action sequence requiring smooth motion). For those, use a service like **Kling or Luma** to regenerate at high quality. This could be, say, 20% of the shots, while the other 80% use local generative content.
- Use open-source tools for tasks like **background creation or environment panoramas**, and then apply minimal calls to commercial models for things like **character animation**. For example, perhaps use Stable Diffusion to create a detailed static background of a forest, and then use a few seconds of a Runway or Luma video generation to animate a character walking in that forest – then composite or blend these.
- **Local control vs. commercial polish:** It might make sense to do initial runs of each shot with local models (which gives more controllability – you can tweak prompts and seeds freely), and only when a shot's composition is satisfactory, use a paid model to upscale or add fidelity. Some services even allow using an input video as a reference – e.g. Luma's "Modify" can take a rough video and re-render it with better consistency. Similarly, if AnimateDiff produces a decent but low-res clip, one could feed those frames to a high-quality image-to-image model (like a Hierarchical Text2Video on Scenario) to re-render each frame at better quality.
- **Division of labor:** We can assign different pipeline stages to local vs cloud. For example, the **StorySplitter and PromptEngineer** are light-weight and can run locally (possibly with a local LLM if needed). **Keyframe generation** can be local (using Stable Diffusion to make storyboard frames). **Video assembly and FFmpeg** mixing is local. The heavy part is **VideoGen** – here we selectively outsource. A sensible plan is to use open-source VideoGen for simpler shots (especially ones that don't involve detailed human faces or that are static camera scenes) and use commercial VideoGen for shots with **complex motion, human characters, or critical aesthetics** (where the closed models currently have an edge). This way, you reserve credits for the hardest visuals. As a concrete example, maybe all wide establishing shots (landscapes, buildings) are done with Open-Sora or AnimateDiff locally (these often look fine since landscapes are easier for diffusion models), but a climactic fight scene or emotional closeup is done with Kling 2.6 to get smooth character movement and facial consistency.
- **Audio and others:** Even in a hybrid model, audio TTS can be done locally to avoid voiceover costs (there are good free TTS models, though possibly not as natural as paid voices). Alternatively, one might use a low-cost service like Amazon Polly or ElevenLabs for a small fee if high-quality narration is needed; since narration is long, a high-quality TTS might be worth the cost (but it's still relatively cheap – e.g. ElevenLabs charges around $0.30 per 1000 characters, so a full movie narration might be only a few dollars).
- **Cost analysis:** By our earlier numbers, a fully commercial generation of a 5-minute video could easily cost dozens to hundreds of dollars (e.g. 5 min on Runway ~ $30). A hybrid approach might cut this drastically. For instance, perhaps 4 minutes of content you do with local models (cost ~$0 direct, maybe $5 of electricity), and 1 minute you do with a mix of Luma/Kling (cost ~$10). This yields a high-quality 5-min video for under $15, versus maybe $100 if all done on commercial. As models improve, the proportion that can be kept local will only increase.
- **When to choose hybrid:** If one's goal is a **long video with acceptable quality**, hybrid is likely the only practical path at present. Pure open-source can struggle with realism and will take a long time

for many shots; pure commercial is costly and might be limited in duration output. Hybrid gives a middle ground: use the **strengths of each tool for what they do best** (local for flexibility and infinite retries, cloud for quality on demand). The pipeline can be designed to make this seamless – e.g., an agent could automatically decide "if shot involves a human face close-up, use external API X; otherwise use local Y model."

Summary of Cost Mitigation: Regardless of plan, we employ strategies to reduce overall compute: - Generate at lower resolution and use AI upscalers (free) for final output rather than generating full 1080p frames natively. - Use lower frame rates where possible (e.g. 12–15 fps for slow scenes, interpolating to 24 fps later, which costs no credits and limited compute). - Re-use assets: If multiple shots occur in the same setting, generate one high-quality background plate and use it in several scenes (perhaps with slight pan/zoom effects via video editing rather than re-generating). - Limit the use of expensive models to only what's necessary for storytelling impact.

In designing the pipeline, we will explicitly mark which stages can remain local and which could optionally call external APIs. The system can have configuration to toggle between "local-only mode" and "hybrid mode." This ensures we can **minimize cost dynamically** – for example, during initial drafts use 100% local (almost free), and for the final version switch some key shots to cloud rendering.

4. Agentic Pipeline Design

We propose an **agent-based pipeline** consisting of modular components ("agents"), each responsible for a specific stage of the video creation process. Breaking the system into agents makes it easier to manage complexity, allows swapping implementations (local vs cloud), and enables adding retry logic and quality checks at each step. Below is the pipeline with each agent, their role, and their I/O design:

1. **StorySplitter** – This agent takes the *long story text* as input and splits it into a structured narrative format (acts → scenes → shots). It performs script or storyboard segmentation. **Input:** raw story text (could be a screenplay or narrative). **Output:** a JSON (or Python dict) with a hierarchical breakdown, for example:

```
{
  "title": "Story Title",
  "scenes": [
    {
      "scene_id": 1,
      "description": "Scene 1 summary ...",
      "shots": [
         {"shot_id": 1, "description": "Shot 1: Opening shot of ...",
"duration_sec": 5},
         {"shot_id": 2, "description": "Shot 2: Closer view of ...",
"duration_sec": 8}
      ]
    },
    {
      "scene_id": 2, "description": "Scene 2 summary ...", "shots":
```

```
[ ... ]
    }
  ]
}
```

The splitter can use simple heuristics or AI: for example, if the story text has paragraphs or scene headings, use those. If not, a GPT-4 prompt could be used to propose a scene-by-scene breakdown (though that might incur cost; alternatively, a smaller local LLM or rule-based algorithm can attempt it). The important thing is each **shot** now has a textual description and an intended duration. The durations might be initially estimated (perhaps based on sentence length or dramatic timing). This structure will guide subsequent agents. *(Controls:* one can adjust the granularity by specifying max shot length, etc. If the user wants manual control, they could provide a script with scene/shot annotations directly, which StorySplitter would simply parse rather than generate.)*

2. **PromptEngineer** – This agent converts each shot's description into a **detailed prompt (or set of prompts)** for image/video generation. **Input:** a shot description (from the JSON) plus context (could include the scene description or any global style settings). **Output:** a **prompt bundle** for that shot, potentially including:

3. A main text prompt describing the scene in cinematic detail.
4. A negative prompt listing aspects to avoid (e.g. "blurry, deformed, text" to reduce artifacts).
5. Metadata like desired camera angle or lighting.

6. A seed value (if we want deterministic outputs for reproducibility). For example, if the shot description is "Wide shot: A dense forest in early morning light. The hero stands near a fallen log, looking determined," the PromptEngineer might produce:

```
{
  "shot_id": 1,
  "prompt": "Wide angle shot of a dense forest at dawn, mist hovering
between tall trees. Soft golden sunlight filters through the canopy. A
rugged male hero in leather armor stands near a fallen log in the
foreground, looking determinedly into the distance. Cinematic, 4K detail,
dramatic lighting, movie scene.",
  "negative_prompt": "low resolution, blurry, cartoon, text",
  "seed": 123456789,
  "style": "cinematic realistic"
}
```

It enriches the description with cinematic language (camera lens, atmosphere) and ensures key elements (hero, environment) are mentioned explicitly. It also tags the shot with a **style** (here "cinematic realistic") that might be used to route to a particular model or add style-specific terms. The PromptEngineer could be implemented with templating plus a bit of ML: e.g., have predefined templates for "cinematic realism" vs "stylized animation," and fill in the blanks using the shot description. Another approach is using an LLM (like GPT-4) to rephrase descriptions into prompts (some community workflows use ChatGPT to generate detailed prompts from plain descriptions). We

should ensure consistency – e.g., if the story has recurring characters, the prompt should include the **same descriptors or code-name for that character** each time (this ties in with the next agent). *Controls:* The user or config can provide prompt templates (for instance, a template for outdoor scenes vs indoor dialogues). The PromptEngineer can also incorporate **global tags** – e.g., if the whole film should be "anime style," it appends that to every prompt.

7. **Style/CharacterManager** – This agent maintains **consistent style and characters across shots**. It can be seen as a database of the film's look and cast. **Inputs:** could be the list of characters with their descriptions or reference images, and style guidelines (e.g. color grading, aspect ratio preferences). **Function:** It intercepts the prompts from PromptEngineer and enhances them for consistency. For example:

8. If the main character is described as "the hero, a tall woman with red hair named Alice," ensure every prompt that includes Alice contains that description (and possibly a special token if we fine-tuned an embedding). This agent might even generate a **custom embedding or LoRA** by taking an image of the actress or a concept art of the character and training a tiny model, then using a token like `<Alice>` in prompts. However, doing that on the fly is expensive – an alternative is to simply enforce the text descriptors and hope the generative model yields a similar look (not guaranteed, but with strong descriptors and same seed across shots featuring that character, it might be somewhat consistent).
9. If a certain style (e.g. "film noir, black and white") is desired scene-wide, ensure each shot's prompt contains those terms.

10. **Output:** possibly updated prompt structures with references. It could also output additional control info: e.g., which Stable Diffusion model checkpoint to use for this shot. For instance, maybe we have one model fine-tuned for our character's face – the StyleManager would say "for shots with Alice's close-up, use model `model_alice.ckpt` instead of the default model." It could also supply **reference images**: e.g., attach the hero's concept art image to the prompt (some video models allow image input as a conditioning). The output might be something like:

```
{
   "shot_id": 1,
   "final_prompt": "<style:cinematic> A tall red-haired woman (Alice) in
leather armor... [rest of prompt]",
   "model": "stableDiffusion_v1.5.ckpt",
   "reference_image": "assets/hero_face.png"
}
```

In practice, Style/CharacterManager might be more of a **data store** than a process – e.g., we define a JSON of characters at the start. But making it an agent allows dynamic adjustments (say if a character's depiction is coming out wrong, this agent could tweak the descriptor or switch to a different LoRA). *Controls:* JSON schemas here could define character attributes. There can also be a **Shot-to-shot memory**: e.g., store the last frame of each shot in a dictionary so that if the next shot is a direct continuation, we use that image as a starting point (to maintain continuity of position/costume). This agent thus ensures coherence: same costume, same face, same overall art style throughout.

11. **KeyframeGenerator** – This agent generates **key frame images or guiding frames** for each shot. The idea is to produce one or more still images that will serve as reference or starting frames for video generation. **Input:** the detailed prompt (from PromptEngineer, post-style adjustments) for the shot. **Output:** one or a sequence of images (keyframes) and possibly associated metadata like depth maps or segmentation maps for those images. There are two main modes:

12. *Single keyframe mode:* Generate one representative image of the shot. This image can be fed into an image-to-video model (like SVD or Gen-1/Gen-2 type models) to animate it slightly, producing motion. It can also be used for continuity – e.g., the last frame of one shot might be used as the keyframe of the next.

13. *Multiple keyframes mode:* If the shot has a clear change (e.g., camera trucking or character moving), we might generate a **start frame and end frame**, and later use interpolation or controllable video gen to fill the middle. For example, shot description "camera zooms from wide to close-up" – we could generate a wide view image and a close-up image of the scene. In general, generating images is easier and faster than generating video, so this agent leverages the maturity of text-to-image models to set visual targets. For example, the KeyframeGenerator might call Stable Diffusion 1.5 with a realism model to produce a 512×512 image of the described scene. If not satisfied, it could retry with different seeds until a suitable image is obtained (this is a spot for QA feedback – see Re-roll agent below). The resulting keyframe is saved, and passed on. In some pipelines, this is optional – if we plan to use a pure text-to-video model like CogVideoX, we might not need a keyframe. But having one can help with consistency (CogVideoX also supports image+text to video modes [92]). *Controls:* This agent might allow a "preview" mode where it only generates keyframes for user approval before video gen, or an automatic mode where it proceeds after generating. It will also record the **seed** used for each keyframe in case we need to regenerate it exactly.

14. **VideoGen** – This is the core **Video Generation agent** that creates a moving clip for each shot, ideally 5–10 seconds long as specified. **Input:** It can take multiple forms of conditioning depending on the chosen model stack:

15. The text prompt (and negative prompt).
16. Optionally, the keyframe image from the previous agent (for image-to-video generation).
17. Other condition data: e.g., depth map of the keyframe if using depth2video models, motion hints, etc. It will then invoke the selected video model to produce the frames. This agent is designed to be pluggable with different backends:
18. In Plan A (open-source mode), VideoGen might call an internal function that loads an open model (like AnimateDiff or Open-Sora via diffusers) and runs it locally.
19. In Plan B (hybrid), VideoGen might decide to call an external API. For instance, it could format the prompt and send a request to Luma's API or to Scenario's Kling endpoint, then download the resulting video. We can implement logic to choose the backend per shot. For example:

```
if shot["style"] == "cinematic realistic":
    if shot_contains_face and high_demand:
        backend = "kling_api"
    else:
        backend = "open_sora_local"
```

```
elif shot["style"] == "stylized animation":
    backend = "animate_diff_local"
```

This decision could be configured by the user (which stages to do locally vs remote). The agent might have a fallback sequence as well – e.g., try local model, if output quality fails QA, then automatically try a cloud model.

Output: the video clip (frames or an mp4 file) for that shot, and possibly some metadata (like actual frame rate, any generation info). To keep things manageable, the agent might output the path to a video file, e.g., `outputs/scene1_shot1.mp4`.

Challenges & Solutions in VideoGen: Consistency is crucial here. If using purely generative models, there's randomness. We mitigate it by: - Setting a **seed** for each shot generation (for diffusers pipelines, you can pass a random seed for reproducibility [93]). - Using the keyframe image to anchor the content – e.g., if the last shot showed a castle and next shot is same castle closer, feeding the last frame to the model (using image2image video or controlnet) can preserve details. - Possibly using **ControlNets or conditioning**: e.g., generate a depth map from the keyframe and feed that to a depth-guided video diffusion model so that the structure persists. Another aspect is length control: we will generate the number of frames corresponding to the shot's duration at the desired frame rate (say 8 fps for diffusion models, which we later interpolate to 24fps, or directly 24 if model supports). If the model cannot do the full length in one go (like AnimateDiff doing 32 frames max), we might generate in chunks and concatenate (some UIs allow "extend video" by feeding last frame repeatedly [10] – our pipeline could do similar in a loop for longer than model limit, though quality may degrade).

Retry logic: If VideoGen fails (model crashes or returns nonsense), the agent can adjust parameters (reduce resolution, half the duration, etc.) and retry. It will log any issues for QA.

1. **Re-roll QA** – This agent performs **Quality Assurance and optional re-generation ("re-roll")** of clips. After VideoGen produces a clip, we want to verify it meets a quality bar and the prompt intent. **Input:** the generated video (or frames) and the shot description + prompt info. **Output:** either an acceptance of the clip or a decision to retry/improve it, possibly with an updated prompt/seed. QA can be both automated and manual:
2. *Automated checks:* We can use computer vision to detect obvious problems. For example, use an **image classifier or detector** on each frame to see if faces are distorted (there are AI models that detect face landmarks – if those fail on a frame, it means the face is likely messed up). Or use an OCR tool to ensure no stray text appears in frames (common diffusion artifact). Another idea is to use CLIP or BLIP to caption the frames and see if the captions match the intended scene – if not, the content might be off-script.
3. *Temporal checks:* Ensure consecutive frames are not too inconsistent. A simple metric: compute difference between adjacent frames – if a frame is drastically different (flashes to a completely different scene due to model glitch), mark as fail. Consistency for color/lighting can also be checked by histogram comparison.
4. Each shot also has an expected duration; QA can ensure the video file has the correct length and frame rate. If a model returned fewer frames, the agent could decide to generate extra frames by interpolation, or request the model for more frames if possible.
5. *Re-roll mechanism:* If a clip fails, the agent can try **regenerating with a different seed or model**. For example, "Shot 5 face looks bad – re-run with seed=42 or try a face-enhancement model." This could

loop a few times (with a limit to avoid infinite attempts). We could maintain a log of which seeds were tried to avoid repeats. If after N tries the clip is still poor, the agent might escalate (e.g. switch to a more powerful model or flag for human review).

6. Another approach for improvement is to do **post-processing** instead of full re-gen: e.g., if everything is fine except the hero's face, run an **AI face restoration** (like GFPGAN or CodeFormer) on each frame [94]. Or if the motion is janky, use a frame interpolation on the clip to smooth it out. The QA agent can automatically apply some of these fixes.

7. **Output handling:** If the clip is accepted, it passes along to assembly. If not, after fixes, the final accepted clip is output. If no acceptable output can be made (e.g., model just can't do that prompt well), the agent could mark the shot as "problematic." In a robust pipeline, we might then either substitute a static image or shorten the shot to something we can generate. (For example, if "a crowd of 1000 soldiers" is too hard to render, maybe the agent changes it to "suggest a crowd in distance with blur." This kind of prompt fallback could be done via a ruleset or AI prompt rewriter.)

8. **Assembler** – This agent **merges all the validated clips into a coherent full video**. It handles video editing tasks like concatenation, adding transitions, and ensuring the visuals align with the narrative timeline. **Input:** the list of video clips for all shots (with their intended order), plus any directives on scene transitions (e.g. cuts or fades). It may also take the **story's narration text** segmented by shot or scene (to align with audio later). **Function:** The simplest assembly is just a cut: put clips back-to-back in the sequence. Using a tool like FFmpeg, the agent will concatenate the shot MP4 files in order. We'll want to make sure they have consistent format (resolution, frame rate) which we ensure in VideoGen. If durations are slightly off, the assembler can trim or pad (e.g., if shot was supposed to be 5s but came out 5.2s, perhaps cut a few frames or slow it by 0.96x to fit – slight adjustments that aren't noticeable). The Assembler can also insert **transitions**: e.g., a short crossfade between scenes, or fade to black if indicated in the scene description. We could allow the StorySplitter or prompt to specify transitions (like "Scene 2 starts with a fade-in"). The Assembler would then generate a few black frames or use an FFmpeg filter for the crossfade.

Another role is to overlay any **text visuals** if needed (like chapter titles or credits), but in our case we will do subtitles in a later stage, so Assembler might leave the video visuals raw at first. However, if there's any **B-roll or static images** to insert (maybe a static title card at the start), Assembler can integrate those by converting them to a video segment (a still frame for X seconds).

Essentially, this agent produces the initial cut of the *picture lock* – i.e., the video track without audio. It outputs a single video file (say `rough_cut.mp4`). We call it "rough" because audio isn't in yet, but visually it should be in final order. If during assembly we find the total runtime doesn't match expectations (maybe our shots when concatenated make a 4 min film instead of 5 because some were shorter), we can decide to adjust pace or note this for possibly stretching some shots slightly with minor slow-motion. But likely we'll accept a small variance.

1. **AudioNarrator** – This agent generates and integrates **audio elements**: narration voice-over in English and Hindi, plus background music, and prepares subtitles. It works closely with the final story script. **Inputs:** the segmented story text (each scene or shot's narration/dialogue), possibly the entire script for context, and any user settings for voices (e.g. male vs female voice, Hindi voice preference). **Outputs:** audio files for each language narration, a music track, and a subtitles file. Let's break it down:

2. **Text-to-Speech (TTS) for Narration:** We have the story text, which might be like a screenplay with dialogue or a descriptive narration. We likely want an **English narration track** that reads either the whole story or at least provides scene descriptions/dialogue. We also need a **Hindi narration** (either a translation or a separate provided script). If the story text is only in English, the agent can first translate it to Hindi (using a translation model or API). For TTS, there are open solutions like Coqui-AI TTS which can produce reasonably natural speech given a trained model (for English there are pretrained voices; for Hindi, voices exist too, though quality may vary). Alternatively, if using a hybrid approach, one could plug into a service like Google Cloud TTS or Azure (which have Hindi voices as well). But staying low-cost, we might use open models or even Festival/Mimic for Hindi if needed (they might sound robotic though).

    - The agent will iterate through the script segments and synthesize audio. It should maintain correct timing with the video: since we know how long each shot is, the narration for that shot must fit (or we adjust the shot length or narration speed). We can control TTS speed by splitting sentences or adjusting the model's pacing. If a narration line is too long to fit in the shot, we either have to speak it faster or consider extending the clip (maybe easier to slightly extend a clip by slow-mo than to shorten dialogue unnaturally). This requires a bit of alignment logic: e.g., we know Shot 1 is 5s, and its text is "In a faraway land, a hero begins her journey." The TTS agent synthesizes that, measures it (maybe it's 4 seconds long audio; that's fine we can leave 1s silence, or if it was 6s long, we might compress it or see if we can cut a pause).
    - We will produce two sets of narration: English (default) and Hindi. Possibly our final deliverable could be a dual-language video (one could embed both tracks or produce separate video files per language).

3. **Background Music:** Music can greatly enhance cinematic quality. The agent will either select or generate a background score. For low cost, we can use *royalty-free music* (like from YouTube Audio Library or FreeSound, etc.) matching the mood of each scene. We can store a few tracks (tense, uplifting, melancholy, etc.) and the agent picks which to use per scene based on scene description (e.g. battle scene -> dramatic percussion track). Simpler: pick one long music track that fits overall tone and loop it softly. Alternatively, use an AI music generator. There are some open models (e.g. Diffwave for music or Riffusion, but they are not great). There are also services like Mubert that can generate music by mood (they have an API with a free tier or inexpensive license for non-commercial use). Another option: not use music at all (some short films rely just on narration). But assuming we want it, the agent will produce or fetch an MP3 music track.

    - The agent may also do **basic sound design**: e.g., if using Kling for some scenes, those scenes might come with generated SFX/voices we want to keep. Or if we have time, we could generate some ambient sounds (like forest birds chirping using an audio generative model or library). However, that can get complex; a simpler approach is to let the background music carry the atmosphere and keep the pipeline simpler.

4. **Subtitles:** The narration text can double as subtitle captions. The agent will create a subtitle file (e.g. SRT or WebVTT) with timecodes matching the narration. Since we know exactly when each line is spoken (we control the timeline), we can set subtitle entries for each sentence or scene. For dual language, we might make separate SRT files (one for English, one for Hindi) or burn one language into the video and provide the other as optional.

The AudioNarrator essentially syncs the story text with the assembled video. It might need to adjust pacing: e.g., if a scene has no narration for a few seconds (perhaps just music and visuals), it can insert an appropriate silent gap or just let music play. If a scene has dialogue, it might attempt multi-speaker TTS (which is possible if we have voice models for different characters). For now, we assume a single narrator voice reading the story as a voiceover, for simplicity.

Output: - `narration_en.wav` (the full English narration audio). - `narration_hi.wav` (the Hindi narration audio). - `music.mp3` (background music track, or multiple tracks combined). - `captions_en.srt` and `captions_hi.srt` (subtitle files).

These are ready to be merged with the video.

1. **ExportAgent** – The final agent that **muxes video and audio, adds subtitles, and exports the final film file**. It takes the assembled video (from step 7) and the audio tracks (from step 8) and combines them. **Input:** `rough_cut.mp4`, `narration_en.wav`, `narration_hi.wav`, `music.mp3`, subtitle files. **Process:** Using FFmpeg (or similar), it will:
2. Mix the narration and music into a single audio track per language. Typically, background music should be lower volume than narration. We can use FFmpeg's `amix` filter with volume adjustments [62] [95]. For example, if music should be at 30% volume under the voice, and voice at 100%, the ExportAgent constructs a filter graph `[voice][music]amix=inputs=2:dropout_transition=2, volume=...` etc. The agent can duck the music when voice is present (some advanced audio mixers can automatically lower music volume when voice is active).
3. Embed the English audio track into the video. If we want a bilingual output, we could embed Hindi as a second audio stream (MP4 container allows multiple audio tracks). Alternatively, produce two versions of the video.
4. Add subtitles: Using FFmpeg, we can either burn subtitles onto the video (hardcoded) or attach them as soft subtitles. If the goal is a single deliverable with both languages, perhaps soft subs is better. We might generate the film with English audio and English subtitles for accessibility, and separately provide the Hindi-dubbed version.
5. Export format: likely MP4 (H.264 video, AAC audio) for compatibility. We'll ensure we use appropriate encoding settings for size/quality.
6. The ExportAgent is also responsible for final touches like ensuring the video starts and ends cleanly (no abrupt cuts). If needed, it can add a **fade-in/out** on the first/last second (FFmpeg has fade filters).
7. The output file might be named `Final_Film.mp4`.

Note: This agent essentially orchestrates FFmpeg commands. For example, to mux narration and music:

```
ffmpeg -i rough_cut.mp4 -i narration_en.wav -i music.mp3 -filter_complex "[1:a]
[2:a]amix=inputs=2:weights=1 0.3[aout]" -map 0:v -map "[aout]" -c:v copy -c:a
aac -b:a 192k output_with_audio.mp4
```

Then add subtitles:

```
ffmpeg -i output_with_audio.mp4 -i captions_en.srt -c copy -c:s mov_text
Final_Film.mp4
```

This would attach the subtitles track for players that support it, or one could burn them in by using `-vf subtitles=captions_en.srt` filter (but that's irreversible).

The ExportAgent ensures the final deliverable meets specifications (resolution, file size if needed, etc.). It then outputs the final file(s) and perhaps some metadata (like a log or a list of all assets used).

Agent Controls & Retry Logic: Each agent's input/output can be defined by a JSON schema to facilitate passing data between them. For instance, after StorySplitter, the scenes/shots JSON goes to PromptEngineer which appends prompts and passes to VideoGen, etc. If an agent fails or its output is unsatisfactory, the pipeline can route the task back for a retry. We mentioned some of this, like Re-roll QA triggering VideoGen to retry. Similarly, if AudioNarrator finds the narration is longer than the video, it could either trim the audio or instruct Assembler to extend the visual a bit (maybe by slowing down a clip slightly, which is easier if a clip is a few frames short). This interplay can be handled by either a central orchestrator or by agents feeding back into previous ones.

Agents can also log their outputs into a unified structured log (for reproducibility and debugging). For example, each agent could output a JSON snippet with what it did (like the prompt used, the seed, file names of outputs, time taken). This can be saved as `storyboard.json` or similar, providing a "paper trail" of the production.

In an *agentic implementation*, you might even have the agents as independent services or functions that communicate, which opens up the possibility of parallelizing some work (e.g., generate different shots in parallel if you have multiple GPUs or threads). However, since later steps depend on earlier ones (and for narrative coherence we often need sequential), we'd likely run mostly sequentially per scene.

Overall, this design isolates each concern (text processing, visual generation, audio, assembly), making it easier to swap modules. For example, if a better text-to-video model comes out, we just update VideoGen's backend for relevant shots. Or if we want to add a "Director AI" agent to adjust camera angles, we could insert that between PromptEngineer and VideoGen in the future.

5. Python Project Blueprint

We now outline a **Python project structure** and key libraries for implementing this pipeline. The focus is on creating a maintainable codebase that can be run via command-line (CLI) or as an API (FastAPI server), and that supports caching/reproducibility for iterative development.

Project Directory Layout:

```
ai_film_maker/
├── agents/
│   ├── story_splitter.py       # code for StorySplitter agent
```

```
│   ├── prompt_engineer.py      # code for PromptEngineer agent
│   ├── style_manager.py        # code for Style/CharacterManager
│   ├── keyframe_generator.py   # code for KeyframeGenerator
│   ├── video_generator.py      # code for VideoGen agent
│   ├── qa_reroll.py            # code for Re-roll QA agent
│   ├── assembler.py            # code for Assembler agent
│   ├── audio_narrator.py       # code for AudioNarrator agent
│   └── export_agent.py         # code for ExportAgent
├── models/                     # directory for model weights or references
│   ├── stable_diffusion/...
│   ├── animatediff/...
│   └── TTS_models/...
├── assets/
│   ├── characters/             # images or data for characters
│   └── music/                  # music tracks files
├── outputs/
│   ├── frames/                 # maybe store keyframe images
│   ├── videos/                 # interim shot videos
│   └── final/                  # final output videos
├── data/
│   ├── example_story.txt       # sample input story
│   └── story_config.json       # optional configuration (like casting info)
├── main.py                     # CLI entry point to run the pipeline
├── api.py                      # FastAPI app exposing an HTTP interface
├── requirements.txt            # Python dependencies
└── README.md                   # instructions and documentation
```

In this structure, each agent is implemented in a separate module for clarity, but they will interact through a orchestrator (which can be `main.py`). The `models/` folder can hold any local weight files or config (though we might rely on HuggingFace Hub for models to avoid large files in repo). The `assets/` folder holds static assets like character reference images (if provided) or music files. The `outputs/` is organized to store intermediate outputs for caching and debugging: e.g. keep keyframe images and shot videos, which is useful if the pipeline crashes in the middle – you don't want to regenerate everything from scratch.

Key Tools and Libraries:

- **PyTorch** – the backbone for all model inference (most AI models run via PyTorch). We'll use PyTorch with CUDA for local acceleration.
- **Hugging Face Diffusers** – a convenient library for diffusion models. It provides ready pipelines for text-to-image, and also specific ones like `StableDiffusionPipeline`, `StableVideoDiffusionPipeline`, `CogVideoXPipeline` [16], etc. We can leverage diffusers to load models from the Hub (e.g. `CogVideoX-5B` or AnimateDiff's motion module) and run inference with a few lines of code. Diffusers also has utility to save video (`export_to_video`) and functions to enable memory optimizations (like model.offload to CPU as we saw [96]).

- **Transformers** – if using any text-related ML like an LLM for StorySplitter or translation, the HuggingFace Transformers library will be used. For example, we could load a small GPT-2 or T5 model for splitting if needed, or a translation model for Hindi (like `opus-mt-en-hi`).
- **OpenCV or PIL** – for image processing (e.g. resizing images, reading/writing frames). If we need to manipulate frames or overlays, OpenCV is handy.
- **FFmpeg** – the workhorse for video/audio processing. We won't reinvent video editing in Python – instead, after generating content, we'll call FFmpeg for concatenation, audio mixing, etc. Python can call FFmpeg via `subprocess` or use wrappers like `moviepy` or `ffmpeg-python`. However, direct `subprocess` calls with carefully constructed command strings might be simplest for full control. For example, the assembler will write a file `shots_list.txt` listing all shot video filenames in order, then call:

  ```
  ffmpeg -y -f concat -safe 0 -i shots_list.txt -c:v libx264 -c:a copy
  outputs/videos/assembled.mp4
  ```

  to concatenate all shots (using `-c:a copy` here if shots have silent audio or no audio, which is fine). Then ExportAgent will use ffmpeg filters as discussed for audio mixing and subtitles.
- **TTS (Text-to-Speech) libraries:** Likely **Coqui TTS** for offline speech synthesis. Coqui has models like `tts_models/en/ljspeech/glow-tts` for English and some multi-lingual ones. We might use their Python API to generate narration. Alternatively, **PyTTSx3** (which uses system TTS) could be an option on Windows or Mac, but cross-platform open solution is better. If quality is insufficient, one could integrate Azure or Google TTS using their SDKs (requiring API keys, etc.), but that introduces cost – perhaps as an optional config.
- **Upscalers:** If our videos are low-res (e.g. 480p), we can use an image upscaling model like **Real-ESRGAN** or **Stable Diffusion upscaler**. There are Python packages for Real-ESRGAN that can upscale frames. We could incorporate this as part of VideoGen or QA (e.g., after generating a 480p clip, upscale it to 720p or 1080p). Another tool is **latent video diffusion upscalers** which are research-y; simpler is per-frame upscaling plus maybe a temporal smoothing (frame-by-frame upscaling can introduce slight flicker because each frame is upscaled independently; however, Real-ESRGAN is relatively consistent, and one can reduce flicker by blending a bit with original). For MVP, we might skip upscaling, but it's a good enhancement for final quality without model re-generation.

- **FastAPI** (for serving): If we want to expose the pipeline as a service, we can wrap it in a FastAPI app (`api.py`). This would allow a user to send a story (and maybe assets) via an HTTP request and get back a URL or file of the video. FastAPI can handle asynchronous tasks, so we might run the pipeline in a background thread/process and stream progress. This is useful if integrating into a web interface. For CLI usage, `main.py` will just parse arguments (like `--story path/to/story.txt --output final.mp4 --plan hybrid`) and call the sequence of agents.

- **Data Storage & Caching:** We aim to cache intermediate results to avoid recomputation on iterative runs. For example, after the first run, we'll have all shot videos saved. If we tweak only one scene's prompt, we shouldn't regenerate the others. We can implement a simple caching: use a hash of the shot's description or prompt to identify its outputs. For instance, after PromptEngineer, each shot can have a unique ID and we generate a hash of (prompt text + seed + model used). Before running VideoGen, check if `outputs/videos/{shot_id}_{hash}.mp4` exists. If yes, skip generation. This way, if you re-run the pipeline with the same input story and seeds, it will just reuse the prior outputs

(ensuring reproducibility as well). If you change something, the hash changes and it will render anew.

We also store the seeds and model versions in the output metadata (maybe compile everything into a JSON or log file). This allows someone to exactly reproduce the film later if needed (provided they have the same models installed). Reproducibility matters if this is part of a research or production pipeline where deterministic output is valued. Setting seeds at all stochastic points (diffusion noise seeds, TTS randomness if any, etc.) will give consistent results run-to-run [93] .

Additionally, intermediate assets like keyframes could be cached to allow manual inspection or replacement (e.g., a user could swap a keyframe image with a custom drawn one to influence the outcome, then the pipeline uses that custom image for final video gen on re-run).

- **Configuration & Templates:** To make the system flexible, we can use JSON or YAML config files for various aspects:
- A config file to specify which models to use (like choose CogVideoX vs StableVideoDiff for VideoGen, choose which TTS voice for each language, etc.).
- Prompt templates and style definitions can be stored in a JSON (like `prompt_templates.json` mapping style names to prompt patterns).
- Character definitions (with optional image paths or textual descriptions) in a file, so that non-developers can edit the "casting" without touching code.

This blueprint ensures that the project is not a monolithic script but a set of cooperating components with clear boundaries. This makes it easier to test each part (e.g. test the PromptEngineer on some input to see the prompt output, test VideoGen separately by feeding it a prompt). It also aids **MVP development**: we can implement a minimal version of each agent and get a rough end-to-end flow, then improve each in turn.

MVP (Minimum Viable Product) plan: The MVP goal is to produce a **1–2 minute coherent test film** to validate the pipeline. To achieve this: - First, create a very short story (maybe a paragraph that splits into 2 scenes, ~4 shots total). This will be our test input. - Implement a stub or basic version of each agent: 1. *StorySplitter:* maybe start with a manual JSON (hardcode the scenes for MVP) or do a simple split by sentences. 2. *PromptEngineer:* use a basic template like `"{description}, cinematic, HD"` just to test. 3. *StyleManager:* initially, if not handling multiple characters, this could be a no-op (or just enforce a default style). 4. *KeyframeGenerator:* for MVP, possibly skip if using text-to-video directly, or generate 1 key image with SD. 5. *VideoGen:* integrate one open-source model (e.g. Animatediff or ModelScope's text2video) to produce a few seconds video for each prompt. Keep resolution low (256p or 512p) to ensure it runs. 6. *QA:* for MVP, just log that generation happened; maybe not actually doing automated QA yet. 7. *Assembler:* use FFmpeg concat to join MP4s; test that we get a combined video. 8. *AudioNarrator:* use any TTS (even OS built-in or dummy audio) to create one voice line, or just use a placeholder music. 9. *Export:* mux whatever is available. - Run this end-to-end to produce a rough video. This tests the plumbing (passing data, calling FFmpeg, etc.). - Then iterate: improve the prompt detail, add actual TTS voices, add the QA checks.

Libraries for specific tasks: - *ComfyUI or Automatic1111 integration:* For development and testing of complex pipelines (like using AnimateDiff with ControlNets), one can prototype in these UIs. But in code, we will rely on direct model calls. However, we might borrow some **workflow logic from ComfyUI** (they have a graph for things like chaining depth maps to video models [97]). It's possible to run ComfyUI headless by

feeding it a JSON graph – but that adds complexity. Instead, incorporate similar logic directly using diffusers or model-specific code. - *Voice Alignment:* If we want subtitles timed perfectly, we might use an aligner. For example, use an ASR (automatic speech recognition) on the generated narration to get timestamps for each word, then align subtitles. But since we control the TTS input text, a simpler way is to split narration by sentence and assign each to the shot's duration. We can fine-tune timing by adding short pauses (most TTS engines let you insert commas or SSML `<break>` tags). This might be sufficient for MVP.

Data management: Store all dynamic data under `outputs/` so the project dir remains clean. Possibly provide a clear naming scheme for outputs (like prefix by scene and shot number). Example: `outputs/videos/scene1_shot2_v2.mp4` (maybe v2 for second version if re-generated).

Reproducibility: Besides seeding, another aspect is recording the exact model version used. If models are loaded from HuggingFace, we should note the model IDs and versions (perhaps locking them in `requirements.txt` or downloading them to `models/`). If any model updates or is re-trained, results could differ, so for a truly fixed output we might snapshot the models. This is more important if the pipeline is used in production.

In summary, the project blueprint emphasizes **modularity, configurability, and keeping things as simple as possible** for a first working system. All heavy-lifting is done by established libraries (diffusers for AI, FFmpeg for video assembly, TTS libraries for audio). The orchestration code mainly transforms data between these and handles the logic of when to call what.

6. Implementation Samples and Configurations

To illustrate the design, here are some **example implementations and formats** for key parts of the pipeline:

Shot JSON Format (example): This JSON represents a single shot after the PromptEngineering and Style stages, ready for video generation. It includes all info needed for that shot:

```
{
  "scene_id": 2,
  "shot_id": 5,
  "description":
"The hero stands atop a hill, overlooking the valley as the sun sets.",
  "duration_sec": 6,
  "characters": ["hero"],
  "style": "cinematic_realism",
  "prompt": "Drone shot of a lone female hero standing on a hill at sunset,
overlooking a vast valley. The sky is ablaze with orange and pink hues. The
hero's red hair flows in the wind. Cinematic, 4K resolution, high dynamic range
lighting, epic composition.",
  "negative_prompt": "blur, low-detail, no-face, text",
  "seed": 90817236,
  "use_initial_image": true,
```

```
  "initial_image": "outputs/frames/scene2_shot4_lastframe.png"
}
```

Explanation: This shot (scene 2, shot 5) shows the hero at sunset. We specify it's 6 seconds long. The `prompt` has been carefully constructed to include the hero and setting, plus cinematic keywords (4K, HDR, epic). A `negative_prompt` is provided to avoid certain artifacts. The `seed` is set for reproducibility. `use_initial_image:true` with an `initial_image` path means we plan to initialize the video generation using an image – in this case, perhaps the last frame of shot 4 (maybe shot 4 was a close-up of the hero's face, and now shot 5 starts wide but we want continuity of her position? This could also be a continuity mechanism). If no initial image, that could be false or omitted. This JSON could be fed to the VideoGen agent, which would then decide which model to use based on style. For example, since style is cinematic_realism, it might use Open-Sora or StableVideoDiff, feeding in the initial image for consistency.

Prompt Templates for Styles:

We define templates that the PromptEngineer or StyleManager use to ensure prompts have a consistent style. Example templates:

- *Cinematic Realism:*
- Description: intended for live-action film look. Emphasize camera specifics, realistic lighting, and high resolution.
- **Template:** `"{scene_description}. Cinematic, ultra high detail, {camera_lens} lens, {lighting} lighting, {resolution}."`
- We might have a few variants internally for camera shots:
  - Wide shot: use a wide-angle lens (e.g. 24mm), mention sweeping view.
  - Close-up: use a telephoto lens (85mm), shallow depth of field.

- For example, plugging into template: scene_description = "a lone female hero on a hill at sunset overlooking a valley"; camera_lens = "wide-angle"; lighting = "golden hour sunlight, high dynamic range"; resolution = "4K". The resulting prompt: *"A lone female hero on a hill at sunset overlooking a vast valley. Cinematic, ultra high detail, wide-angle lens, golden hour sunlight, high dynamic range lighting, 4K resolution."* – which is similar to the above prompt. We also often add things like "masterpiece, award-winning photography" in community prompts, but those can introduce style biases and might not be needed if the model knows "Cinematic".

- *Stylized Animation:*

- Description: for an animated/cartoon or artistic style (could be 2D anime or 3D Pixar-like).
- **Template example (anime style):** `"{scene_description}. Art by Studio Ghibli, vibrant colors, hand-drawn 2D animation style, sharp lines, expressive characters."`
- For a 3D Pixar style: `"Highly detailed 3D animation, Pixar style, {scene_description}, smooth motion, bright cinematic lighting."`
- The user can select a style tag (like "anime" vs "pixar" vs "comic"). The PromptEngineer would append a corresponding block. For consistency, we'd use the *same style keywords each time*. If the film is supposed to mimic a specific animation style, we fix those terms in every prompt.

- Example outcome: *"The hero runs through the forest. Art by Studio Ghibli, vibrant watercolor backgrounds, expressive anime style, dynamic movement."*

- *Consistent Characters:*

- We handle this by including the same descriptors or tokens. If we have a custom token (say we trained an embedding for the hero named `<hero123>`), our template might be `"{scene_description} featuring <hero123>."` plus whatever style. If not, we use textual descriptors: e.g., always refer to her as "red-haired female hero in leather armor". This phrase should repeat. The PromptEngineer can ensure that whenever `hero` is in the shot's characters list, it injects that phrase in the prompt. Over multiple shots, this (hopefully) guides the model to generate similar looking person.
- Another tip is to use **image prompts**: some systems allow giving a reference image of the character along with the text prompt to guide generation (like ControlNet Reference). If we have a consistent portrait of the hero, we could use that in each shot generation that involves her. That is not exactly a text template, but the pipeline could attach that reference automatically.

The agent will also manage seeds for consistency – e.g., using the same seed for the hero's face close-ups across scenes might ironically keep some features similar (though in diffusion it doesn't guarantee same face, it might just keep overall composition if prompt is similar).

FFmpeg Stitching Config:

We'll present an example of FFmpeg commands/config used in the final assembly:

To **concatenate video clips** (assuming clips have no audio yet or we don't care about their placeholder audio):

```
# Create a file "inputs.txt" with:
# file 'outputs/videos/scene1_shot1.mp4'
# file 'outputs/videos/scene1_shot2.mp4'
# file 'outputs/videos/scene2_shot1.mp4'
# ...
ffmpeg -f concat -safe 0 -i inputs.txt -c:v libx264 -preset medium -crf 18 -c:a
aac -b:a 128k outputs/videos/assembled.mp4
```

This will stitch all shots in order into `assembled.mp4`. We choose H.264 encoding with a high quality (CRF 18) for final. If the clips are already in a final codec, we could use `-c:v copy`, but since we might have different sources (some from diffusers as images to video at 8 fps, others at 24 fps), re-encoding ensures uniformity (we would set a target frame rate, e.g., add `-r 24` to enforce 24 fps output).

To **mix audio tracks** (English narration + music) and attach to video:

```
ffmpeg -i assembled.mp4 -i narration_en.wav -i music.mp3 \
  -filter_complex "[1:a]volume=1.0[aud1]; [2:a]volume=0.3[aud2]; [aud1]
```

```
[aud2]amix=inputs=2:dropout_transition=2:duration=longest[aout]" \
  -map 0:v -map "[aout]" -c:v copy -c:a aac -b:a 192k outputs/final/
film_en_temp.mp4
```

This uses two `volume` filters to set narration at 1.0 (100%) and music at 0.3 (30%), then mixes them. `dropout_transition=2` helps avoid clicks when one input ends earlier. We map the original video (0:v) and the mixed audio to output. We copy the video stream (no re-encode) since it's already encoded above, and encode audio to AAC at 192 kbps stereo. The result `film_en_temp.mp4` has English audio.

To **add subtitles** (English):

```
ffmpeg -i film_en_temp.mp4 -vf
"subtitles=captions_en.srt:force_style='FontSize=24,PrimaryColour=&HFFFFFF&'" \
  -c:v libx264 -c:a copy outputs/final/film_en_sub.mp4
```

This burns in subtitles from an SRT, styling them with white color and font size 24. If we prefer soft subs (not burned in), we could do:

```
ffmpeg -i film_en_temp.mp4 -i captions_en.srt -c:v copy -c:a copy -c:s mov_text
outputs/final/film_en.mp4
```

This will include the SRT as a mov_text subtitle stream that many players can toggle. (Note: MP4 supports only certain sub formats; mov_text is one.)

We would repeat a similar process for Hindi: mix Hindi narration with same music, mux, and either burn Hindi subtitles or provide separately.

We can automate these FFmpeg calls inside the ExportAgent using Python's `subprocess.run([...])` with the necessary arguments.

These sample commands show how we handle concatenation, audio mixing, and subtitles using widely available tools (FFmpeg), ensuring our pipeline doesn't need to manually handle binary audio/video data beyond feeding into FFmpeg.

7. Recommendations and Roadmap

Finally, based on our research, here are **our top recommendations for model/tool stacks** and an execution roadmap:

Best Model/Tool Stacks:

1. **Fully Open-Source Stack – "Local Diffusion Studio":** Utilize *Stable Diffusion-based tools end-to-end.* For instance, use **Stable Video Diffusion** (image-to-video) or **AnimateDiff** for generating each clip, combined with Stable Diffusion (txt2img) for any keyframes or higher-res stills. This stack can be run

on a single machine. For audio, use **Coqui TTS** models for narration and open music. This stack is essentially free, and with recent open models like Open-Sora or CogVideoX, you can achieve decent 480p to 720p footage. The trade-off is speed and some quality limits, but it's ideal for experimentation and internal drafts. As open models improve (HunyuanVideo, etc.), you can plug them in to push quality higher. **When to use:** if budget is zero or you require full offline control (e.g., for privacy or on-premise usage). *Example:* A 2-minute short film could be produced on a mid-range GPU machine in a few hours using this stack, at virtually no cost. [7] [98]

2. **Hybrid Stack – "Mix and Match":** Combine local generation for simpler tasks with **cloud APIs for complex scenes.** This might involve using **AnimateDiff or Stable Diffusion** locally to get rough cuts or backgrounds, then using **Runway Gen-2 or Kling** for key hero shots. Also use local tools for final assembly. For example, you could generate the majority of scenes with an open model at lower resolution, and only use, say, **Luma AI's Ray API** to regenerate a few establishing shots in full HD glory (for eye-candy moments). Another hybrid approach is to use **AI upscale and enhancements on local outputs** rather than calling a service for initial generation: e.g., generate all clips with Open-Sora at 512px, then use a small number of credits on **Deepfake/face refinement services** or an AI editor to clean up faces or increase resolution. This stack aims to minimize credit usage: perhaps ~80% of frames are from cheap local generation, 20% from paid high-quality generation. **When to use:** if you want higher quality than pure open source can afford, but you need to manage a limited budget carefully. This is likely the most practical approach for a longer film given today's landscape. You might schedule cloud generation for the final production step after validating everything locally as storyboards (so you don't pay for iterations).

3. **Cloud-Only Stack – "Premium Commercial":** If budget allows, use *all-in-one commercial platforms* for speed and quality. For example, you could use **Kling 2.6 Pro** for generating every single shot (which ensures a consistent style and possibly even consistent characters if you use their image conditioning features [83]). Narration could be done with a service like **Azure Cognitive Services** (for lifelike multi-language voices) and music from a platform like **Audiio or Artlist** (stock music). Runway or Luma could be used similarly to do every shot. This would yield top-notch quality and require minimal engineering on the model side – your pipeline mostly orchestrates API calls and assembly. However, cost will be high (tens of dollars per minute of video) [86] , and some platforms impose limits (like Runway might not easily generate 60 min continuous content). **When to use:** if the project is commercial or you have a budget for it, or if time-to-deliver is critical (since these services often generate faster than local diffusion can, thanks to powerful servers). In such a scenario, our pipeline's value is in the narrative orchestration and multi-tool integration, rather than in cost savings. It ensures even with different services, everything comes together in one final product.

Given the rapid advances, one might start with Stack 2 (Hybrid) – get the MVP done – and keep an eye on new open models to gradually replace cloud parts, or vice versa if a new service offers a dramatically cheap option (competition might drive down per-minute costs by 2026).

Comparison Table of Key Models/Tools: *(We include both open and commercial ones relevant to our pipeline use case, summarizing their attributes)*

Model/Tool	Quality & Output	Licensing	Clip Duration	VRAM / HW Needs	Python Integration	Cost (Approx per min)
CogVideoX 5B	Good quality; 720×480 at 8fps [99] (coherent 6s scenes, some artifacts)	Open (free, CogVideoX License)	~6 sec per gen [1]	High – 18GB+ GPU (5GB with offload) [90]	Yes – Diffusers pipeline [16]	~$0 (free); compute ~3min/clip on A100 [7]
Stable VideoDiff	High detail (576p); strong short clips [3]	Open (research; CC BY-NC)	2–4 sec (14–25 frames) [4]	High – ~>12GB (25-frame mode) [5]	Yes – HuggingFace diffusers [23]	$0 free; slow on consumer GPU (frame-chunk to fit) [5]
AnimateDiff	Medium motion; inherits SD image quality (up to 512p). Great for small character actions.	Open (MIT)	~4 sec (32 frames @8fps) [27]	Moderate – e.g. 8GB for 16 frames at 512^2	Yes – A1111 ext or custom script [31]	$0 free; runs ~minutes per clip on RTX3060 (example)
Open-Sora 2.0	Very high – near Runway/ Luma level at 720p [35]. Smooth motion, good alignment.	Open (Apache-2.0) [36]	Up to 15 sec (and supports extending) [34]	Very high – 11B params (multi-GPU or ~40GB VRAM est.)	Yes – open code & HF demo [38]	$0 free; needs heavy compute (though optimized training cost)

Model/Tool	Quality & Output	Licensing	Clip Duration	VRAM / HW Needs	Python Integration	Cost (Approx per min)
HunyuanVideo	Very high – beats Runway Gen-3 per eval [39], diverse motion, up to 1080p possible.	Open (Tencent Source)	(Not stated; likely ~5–8 sec default)	Extreme – 13B; multi-GPU or FP8 on high-end GPU [91]	Yes – Code released, diffusers/ ComfyUI support [45]	$0 free; requires enterprise GPU setup (designed to scale)
Runway Gen-2	High – 720p max, great creativity; some artifacts on complex inputs.	Closed (SaaS)	~5 sec (Gen-3 up to 10s) [52]	N/A (cloud)	Limited – no public API (UI/CLI only)	~$6 per min (std plan) [53]; higher on free tier (credits)
Luma (Ray3)	Very high – cinematic, smooth camera, 720p/ 1080p, HDR options.	Closed (API SaaS)	5s (free) / 10s (paid) [2]	N/A (cloud)	Yes – REST API available [57]	~$9 per min (720p on Plus) [64]; needs subscription
Pika Labs	Medium – 480p default, stylized/ artistic outputs. Fast gen.	Closed (Web)	5 sec	N/A (cloud)	No official API (web only)	~$8–$38 per min (depending on model quality) [72] [73]
Kling 2.6	Very high – film-like video *with audio*, superb motion & consistency [77] [78].	Closed (Scenario API)	5–10 sec	N/A (cloud)	Yes – via Scenario SDK [76]	~$12 per min (est. $2 per 10s at 720p) [88]; can vary by settings [86]

Model/Tool	Quality & Output	Licensing	Clip Duration	VRAM / HW Needs	Python Integration	Cost (Approx per min)
Coqui TTS (voices)	High quality speech for some voices; entirely offline.	Open (MPL)	N/A (generates audio)	Low – CPU okay for TTS	Yes – Python TTS API	$0 free; slower than real-time for long narration
Azure/ Cognitive TTS	Very high (near-human), many languages.	Closed (API)	N/A	N/A (cloud)	Yes – SDK/ REST	~$4 per hour of speech (e.g. ~$0.5 per 7.5 min at std rate)
FFmpeg (video)	Lossless editing, industry-standard results.	Open (LGPL)	N/A	N/A (CPU tool)	Yes – via subprocess or ffmpeg-python	$0 free; just CPU usage

(Table notes: VRAM is given for models run locally. Cost per minute for AI video assumes relatively few retries; actual may be higher as discussed. Quality descriptions are relative as of early 2026.)

From the above, our **suggested stack for an MVP** is: use **AnimateDiff or Stable Video Diffusion locally for initial videos**, and integrate with **Coqui TTS for narration**. This fully open stack can create a short film with no recurring costs. Once the MVP (1–2 min film) is working and if higher quality is desired, scale up by swapping in **Open-Sora or CogVideoX** for improved video quality and by possibly using **Kling or Luma for selected shots** where absolute quality or complex motion is needed. This hybrid ensures the majority of easy content is done cheaply, and only the “hero shots” use paid credits.

MVP Execution Roadmap (Milestones):

1. **Setup Environment & Models:** Install PyTorch, diffusers, and FFmpeg. Download small versions of models (e.g., Stable Diffusion 1.5 and AnimateDiff motion module) to test pipeline [30] [31]. Verify you can generate an image and a short video clip in a notebook or script.
2. **Implement Text Segmentation (StorySplitter):** Start with a simple splitter that splits on blank lines or a delimiter in a test story. Test it outputs a structured JSON with scenes and shots.
3. **Prompt Engineering & Style Definitions:** Create a basic PromptEngineer that takes a shot description and returns a prompt string. Define a couple of style templates (e.g. a default cinematic template). Test with a sample input that the prompt looks reasonable.
4. **Integrate Video Generation (VideoGen):** Write the VideoGen agent to call a chosen model. For MVP, try something straightforward like using the *Text2Video-Zero* pipeline or AnimateDiff for ~2

second video (for speed). Use a fixed seed for determinism. Run it on one of the prompt examples to produce an MP4, verify the video plays (it might be very short or low-res at first).

5. **Basic Assembly:** Once you can generate a few sample clips, implement the Assembler to concat them. Use FFmpeg to join 2–3 short clips into one video. Test that the timing/order is correct.
6. **Text-to-Speech Integration:** Set up Coqui TTS (or even OS TTS as placeholder) to generate a voice line for a sample text. Ensure you can save to WAV. Then implement AudioNarrator to generate narration for each scene or for the whole script. For MVP, you might generate one audio file for the entire narration.
7. **Audio-Video Muxing:** Use FFmpeg to add the narration audio to the assembled video. Test that the voice aligns (for MVP, you might not worry about precise sync, just overall background narration).
8. **End-to-End Test (1 minute):** Put it all together on a short story (~3–5 shots). Run the pipeline via `main.py` and produce an output video with voice. Check the result for coherence. This is the MVP achievement: a pipeline that automatically turns input text into a small narrated video.
9. **Milestone: Coherence Improvements:** Evaluate the output and add improvements:
10. If characters vary, incorporate StyleManager with consistent descriptors or even fine-tune an embedding if possible.
11. If video is choppy, adjust frame rates or try longer generation with a better model.
12. If narration doesn't line up, break the narration by shot and align or use subtitle as guide.
13. Add the Re-roll QA agent to automatically detect a bad generation (you can simulate a failure by intentionally prompting a tricky scenario to see if QA catches it).
14. Test the retry logic by, say, forcing one attempt to use an alternate seed.
15. **Scaling Up to ~2–5 minutes:** Gradually increase the story length and see how pipeline performs. Pay attention to memory and caching – implement caching so if you run the same story twice it doesn't redo everything (this helps during debugging). Also, monitor performance: rendering a 5-minute video with dozens of shots might take hours; consider concurrency (if GPU memory allows, generate two shots at once in separate processes to halve time).
16. **Incorporating External APIs (if hybrid):** Once the local MVP is solid, test one external call. For example, integrate Luma's API for one shot: provide an API key, replace VideoGen for that shot with an API call (mock it if needed to avoid cost in testing). Ensure the pipeline can fetch the result and continue. This is optional for MVP but necessary for final hybrid demonstration.
17. **Finalize Subtitles & Multi-language:** Generate the Hindi narration via translation and TTS, and produce subtitles. Verify the final ExportAgent can produce two outputs or one output with multiple tracks.
18. **Polish & Documentation:** Clean up temporary files (maybe have ExportAgent remove intermediate files unless debug mode is on). Document how to run the pipeline, and how to configure it (for example, how to switch between open vs hybrid via a config flag).

Following this roadmap, you will incrementally build a robust system. By the end, you'll have a flexible pipeline where, for instance, replacing the model in `video_generator.py` from AnimateDiff to CogVideoX is straightforward, or switching the TTS voice is just a config change.

Each milestone ensures a working product at that stage (e.g., after milestone 8 you have a basic but complete pipeline). Subsequent steps then refine quality and add features. Testing at each step with a short story is key – you don't want to attempt a 60-minute run until you're confident the 1-minute version works reliably and all agents handle edge cases.

In conclusion, creating a 5–60 min AI-generated cinematic video on a budget is an **ambitious but increasingly attainable goal**. By leveraging the latest open models (for cost savings) and integrating

commercial services selectively, we can build a pipeline that outputs a narrated, musically-scored film with minimal human post-production. The proposed design addresses current limitations (clip length, consistency) through careful orchestration, and is poised to improve over time as AI models evolve. All tools and models discussed are cited with sources for further reference, and the plan provides a clear path from MVP to a full-featured AI film generation system.

Sources:

- CogVideoX capabilities and VRAM needs [12] [6]
- Stable Video Diffusion high-level description [3] [5]
- AnimateDiff method and limitations [28] [27]
- Open-Sora updates (15s clips, quality improvements) [34] [35]
- HunyuanVideo performance claims [39]
- Runway Gen-3 credit cost (10 credits/s) [52] and Gen-2 plan credits [53]
- Luma pricing (credits for 10s 720p) [64]
- Pika pricing (credits for Turbo vs Pro) [72] [73]
- Kling features (audio generation, smooth motion) [78] [82] and cost estimates [88] [86]
- HuggingFace diffusers pipeline usage for CogVideoX [16]
- Example optimization flags (offload, etc.) for diffusion [100]
- Reddit discussion on extending videos with frame reuse (CogVideo) [10] [11]
- Luma AI API reference [57] and Dream Machine details [62]
- Scenario guide on Kling versions [76] [83]
- Reddit cost discussion (Kling ~$2–$15/min) [86]

---

[1] [6] [7] [15] [16] [90] [93] [96] [99] [100] zai-org/CogVideoX-5b · Hugging Face
https://huggingface.co/zai-org/CogVideoX-5b

[2] [61] [62] [63] [64] [65] [66] [67] [95] Dream Machine Pricing: Plans and Credits | Luma AI
https://lumalabs.ai/pricing

[3] [4] [5] [21] [22] [23] Stable Video Diffusion
https://huggingface.co/docs/diffusers/en/using-diffusers/svd

[8] [9] [10] [11] [94] CogStudio: a 100% open source video generation suite powered by CogVideo : r/StableDiffusion
https://www.reddit.com/r/StableDiffusion/comments/1flfc0a/cogstudio_a_100_open_source_video_generation/

[12] [13] [14] [2408.06072] CogVideoX: Text-to-Video Diffusion Models with An Expert Transformer
https://arxiv.org/abs/2408.06072

[17] [18] [19] [20] [2311.15127] Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets
https://arxiv.org/abs/2311.15127

[24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [51] [98] AnimateDiff: Easy text-to-video - Stable Diffusion Art
https://stable-diffusion-art.com/animatediff/

[34] [35] [36] [37] [38] GitHub - hpcaitech/Open-Sora: Open-Sora: Democratizing Efficient Video Production for All
https://github.com/hpcaitech/Open-Sora

39 40 41 42 43 44 45 46 47 48 91 97 GitHub - Tencent-Hunyuan/HunyuanVideo: HunyuanVideo: A Systematic Framework For Large Video Generation Model
https://github.com/Tencent-Hunyuan/HunyuanVideo

49 AILab-CVC/VideoCrafter: VideoCrafter2: Overcoming Data ... - GitHub
https://github.com/AILab-CVC/VideoCrafter

50 [PDF] Evaluation of Text-to-Video Generation Models - NIPS papers
https://proceedings.neurips.cc/paper_files/paper/2024/file/c6483c8a68083af3383f91ee0dc6db95-Paper-Conference.pdf

52 How many credits does each video generation use? : r/runwayml
https://www.reddit.com/r/runwayml/comments/1g4oz07/how_many_credits_does_each_video_generation_use/

53 Runway Pricing 2025: Plans & Costs Reviewed - Tekpon 2026
https://tekpon.com/software/runway/pricing/

54 AI Image and Video Pricing from $12/month - Runway
https://runwayml.com/pricing

55 Runway AI Pricing: Free vs Paid Plans - Imagine.Art
https://www.imagine.art/blogs/runway-ai-pricing

56 57 58 59 60 Luma AI | AI Video Generation with Ray3 & Dream Machine | Luma AI
https://lumalabs.ai/

68 69 70 71 72 73 74 Pika
https://pika.art/pricing

75 76 77 78 79 80 81 82 83 84 Guide to Kling Video Models: Features & Use Cases | Scenario Help | Scenario
https://help.scenario.com/en/articles/kling-video-models-the-essentials/

85 Kling AI Video Length Limits 2026: Free Vs Paid Maximum Duration ...
https://aitoolanalysis.com/kling-ai-video-length-limits/

86 87 What can I expect to spend on Credits for Videos? : r/KlingAI_Videos
https://www.reddit.com/r/KlingAI_Videos/comments/1nypc2u/what_can_i_expect_to_spend_on_credits_for_videos/

88 Kling 2.0 Video Generation Service Pricing and Realism Issues
https://www.facebook.com/groups/846203050725189/posts/1046648807347278/

89 Prompt Structuring with JSON for AI Creative Control - Glen E. Grant
https://glenegrant.com/blog/prompt-structuring-with-json/

92 CogVideo - AI-Driven Text-to-Video, Image-to-Video, and Video-to ...
https://cogvideo.net/