ChatGPT

Agentic AI Cinematic Video Generation System Design

Overview and Goals

The goal is to build an **agentic, multi-stage AI pipeline** that converts a long-form story (text) into a high-quality cinematic video of **5–60 minutes**. The system will use only **open-source models** (Plan A) for all AI tasks. It will accept a story as input and produce a cohesive video composed of multiple AI-generated clips (5–10 seconds each), stitched together with subtitles, background music, and text-to-speech audio (in English and Hindi). The design emphasizes **modularity** (multiple specialized agents), **configurability** (swappable models and prompts), and **efficiency** (capable of running on a 16GB RAM CPU for debugging, and leveraging GPUs for production on RunPod or similar). The system will be exposed as a **FastAPI web service** for triggering video generation jobs and tracking their status.

Key Requirements and Constraints:
- **Open-Source Models Only:** All AI models (LLMs for text, generative image/video, TTS, music) must be open-source. (Plan B could integrate proprietary models, but here we focus on Plan A). For example, we will use the open **Mistral-7B** series for language tasks (via APIs like Together.ai or Groq) [1], open diffusion models for video (e.g. Stable Diffusion-based text-to-video), and open TTS models.
- **Agentic Pipeline:** The system is broken into multiple **agents**, each responsible for a specific stage (story parsing, scene planning, video generation, audio generation, etc.). This follows the multi-agent design pattern where specialized agents collaborate in a pipeline [2] [3]. Each agent will adhere to SOLID principles (single responsibility, decoupled via clear interfaces, etc.).
- **Configuration & Modularity:** All configuration (model file paths, API endpoints, prompt templates, etc.) will be provided via environment variables (`.env` file), enabling easy swapping of models or services without code changes. Hardcoded strings in logic are avoided; we use **enums/constants** for repeatable keys or labels to ensure maintainability.
- **Data Modeling with Pydantic:** Every agent's input and output is defined with **Pydantic** models for validation and clarity. For example, a `Scene` model contains fields like `id`, `text`, `shots: List[Shot]`, etc., and a `Shot` model might include `description`, `duration`, `video_file_path`, etc. Using Pydantic enforces a structured JSON schema for communication between agents and for the final output format (e.g. a shot list or video manifest) [4].
- **Performance Considerations:** The pipeline design considers **speed and cost**. Wherever possible, heavy computations can be offloaded to GPU (e.g. RunPod cloud) while the orchestrator and lighter logic run on CPU. The system will allow running smaller/faster models or reduced settings for quick iterations (MVP or debugging), and higher-quality models for final production. We assume a production environment with at least one **GPU (e.g. 24 GB VRAM)** available for diffusion models and possibly high-end LLMs, while local debugging may use CPU-friendly models or stubs.
- **Output Quality:** Despite focusing on speed, we aim for high-quality outputs. The design includes **retry and re-roll mechanisms** – if an agent produces unsatisfactory output (e.g. an LLM returns malformed JSON or a video frame is off-style), the system can trigger a retry or an alternate approach (with adjusted prompts

or seeds). We will note assumptions like maximum retry counts (e.g. 2 attempts per agent by default) and quality-check criteria.

With these goals, the following sections detail the **agent pipeline**, the **system architecture**, the **data models**, and the **implementation plan**.

Agent Pipeline Design

The video generation process is divided into stages handled by different agents or modules. Below is an **end-to-end pipeline** outline, with each agent's responsibilities, inputs, and outputs (modeled via Pydantic):

1. **Story Ingestion & Preparation:**
2. *Responsibility:* Accept the raw story text (which could be several pages long) and prepare it for scene splitting. If needed, pre-process the text (e.g. remove unwanted formatting).
3. *Input:* Full story text (plain string, possibly with markdown or screenplay format).
4. *Output:* A sanitized or standardized text format (Pydantic model: `StoryInput` with fields like `text`, `title`, etc.).

5. *Notes:* This is a simple step, possibly just handled within the API layer or orchestrator. It ensures the story is ready for LLM processing (e.g. truncated if extremely long, or chunked if needed due to LLM context limits).

6. **Scene Breakdown Agent (LLM-based):**

7. *Responsibility:* Split the story into **scenes**. Each scene would be a logical segment of the narrative (e.g. a change of location, time, or major event) suitable for a distinct video segment. The agent should produce a summary or description for each scene.
8. *Input:* The full story text (or `StoryInput` from previous step). Possibly a prompt template guiding the LLM to output a JSON list of scenes.
9. *Output:* A list of scenes with descriptions. For example, a Pydantic `Scene` model with fields: `id` (scene number), `text` (the story segment for that scene), `summary` (LLM-generated brief summary or setting), and possibly `shots: List[Shot]` (initially empty, to be filled by next agent). This could be wrapped in a `SceneList` model.
10. *Implementation:* Use an open LLM (e.g. **Mistral-7B** or a similar capable model) via an API. We might use the Together.ai API to call a Mistral model (which is cost-effective and fast) [1]. The prompt will instruct the model to break the story into numbered scenes, perhaps with a brief title or summary for each.
11. *Example:* If the story involves multiple chapters or acts, the LLM might output JSON like `{"scenes": [ {"id": 1, "summary": "Hero meets mentor in village...", "text": "...original text..."}, ... ]}`.

12. *Retry Logic:* If the LLM output is not valid JSON or misses the format, the orchestrator will detect it (via Pydantic parsing). It can then **re-prompt** the LLM (e.g. adding a system message: "Output only valid JSON.") and retry. We assume up to 2 retries for well-formed output. If still failing, the job can be aborted with an error status reported.

13. **Shot Planning Agent (LLM-based):**

14. *Responsibility:* For each scene, break it further into **shots** or camera scenes of ~5–10 seconds. Each shot will have a detailed description to guide video generation. The agent effectively storyboards the scene into a sequence of visual moments.
15. *Input:* One scene's text or summary (from Scene Breakdown), plus any context (like overall style).
16. *Output:* A list of `Shot` models for that scene. Each `Shot` may include: `id` (shot number within scene), `description` (text prompt describing the visuals), `duration_seconds`, and maybe metadata like `type` (e.g. wide shot, close-up, if we want to classify shot types). This could be structured as JSON per scene: e.g. `{"scene_id":1, "shots":[ {"id":1, "description":"A wide shot of the village at sunrise, birds chirping...", "duration":5}, {...} ]}`.
17. *Implementation:* Another LLM prompt (possibly a more creative one) will be used. For example, using a slightly larger or more creative model (if available via API, e.g. **Qwen-14B** via Together for complex reasoning [1], or still Mistral with a higher temperature). The prompt template might say: "You are a film director. Given the scene description: <scene_summary>, break it into a sequence of cinematic shots. Provide 5–10 second shots, each described vividly. Output as JSON with fields: id, description, duration."
18. *Considerations:* This agent ensures the story is translated from narrative text to **visual descriptions**. It might also extract any dialogue lines to be spoken in that shot or note which characters are present (we could extend the Shot model with optional `dialogue` or `subtitle_text`). That way, we know what subtitle or voiceover to generate for that shot. For example, if the story text in a scene includes dialogue, the agent might attach it to the corresponding shot.

19. *Retry & Re-roll:* Similar JSON validation as above. Additionally, if a description is too vague or not visual enough, a heuristic or even a second LLM check could flag it for improvement. In MVP, we likely rely on prompt quality to get good results initially, but later we could have a **review agent** check consistency (this is noted as a potential future improvement).

20. **Visual Asset Generation Agent (Video Generation):**

21. *Responsibility:* Generate the actual video clip for each shot description. Because purely generative video from text is resource-intensive, this might be broken into two sub-steps:
    - **Image Frame Generation:** Use a text-to-image model to generate a key frame or keyframes for the shot.
    - **Animation:** Use a text-to-video or image-to-video model to animate the frame into a 5–10 second clip.
22. *Input:* A `Shot` description (text prompt) and desired duration. Optionally, a reference image if available (for consistency across shots, one could carry over a character's appearance between shots by reusing the last frame of the previous shot as a starting point – this is an advanced feature to consider).
23. *Output:* A video file or sequence of frames for that shot. The `Shot` Pydantic model will be updated with the filename or path to the generated clip.
24. *Implementation (Models):* We leverage **diffusion-based video generation** models that are open-source. For example:
    - Use **Stable Diffusion XL (SDXL)** or similar for high-quality image generation [5], and
    - **Stable Video Diffusion 1.1** (an open text-to-video diffusion model) to animate those images [6]. Stable Video Diffusion (by Stability AI) can generate ~4-second videos from a prompt or initial frame [6], and can be looped or extended by generating multiple segments. Another

option is **ModelScope Text2Video** or newer models like **Mochi 10B** or **Wan 2.2** (Alibaba's open video models) which have shown good consistency [7] [8]. Wan2.2, for instance, offers a balance of quality and efficiency suitable for local testing [8].
    - Given the hardware constraint, **for MVP** we might use a smaller pipeline: e.g. generate a series of still frames (one per shot) with Stable Diffusion and simply apply a Ken Burns effect or cross-fade to simulate motion (this drastically lowers compute needs). In full production, we switch to actual motion synthesis.
    - We will run these models on GPU. On RunPod, we can load the diffusers pipeline for text-to-video. Alternatively, use services like **OctoAI** which host such models via API [9]. The design is flexible: the visual generation agent could either call a local diffuser or make an API call (URL configurable in `.env`) to a model provider.
25. *Performance:* Each 5–10s clip at moderate resolution (e.g. 480p or 720p) can be slow to generate. We assume using a modern GPU (A100 class) for production; e.g. *HunyuanVideo 13B* (Tencent's model) requires multi-GPU for long clips [10], but smaller ones like **Mochi 10B** or **Stable Video Diffusion** can run on a single high VRAM GPU. We aim for each shot to be generated in under ~2 minutes on such hardware (Stable Video Diffusion advertises creating short videos in ~2 min [11]). For a 60-minute film (which might require ~120 shots of ~30s each), generation could be lengthy, but the architecture could support parallel generation if multiple GPUs are available (the orchestrator could farm out shots in parallel tasks).

26. *Re-roll Strategy:* If a generated video is of poor quality (e.g. incoherent frames or not matching prompt), the agent can try a **re-roll**: either regenerate with a different random seed or adjust the prompt (perhaps the LLM could provide an alternate phrasing). This can be automated for issues like extreme artifacts (e.g. black frames) or can be a user-driven re-roll (the API could offer an endpoint to regenerate a particular shot with modifications). We assume a low re-roll rate to keep costs in check – e.g. at most 1 re-roll per shot unless user manually requests more.

27. **Audio Generation Agents:** These run in parallel to the visual generation (or after, depending on design) to produce the audio elements:

28. **Dialogue & Narration TTS Agent:**
    Responsibility: Convert any speech content in the story into audio. This includes narrator voice (narration of story text) and character dialogues. The story might not be in screenplay format, so this agent might need to **extract dialogues** or simply use the story text as narration. If the story is first-person or has an unseen narrator, we generate a single narration track. If there are multiple characters speaking lines, we could generate distinct voices for each (this is an advanced feature requiring speaker identification in text).
    Input: The original story text along with the scene/shot breakdown (to sync with video). We might attach dialogue lines to Shots during the Shot Planning stage, as mentioned.
    Output: Audio files for voice-over. Possibly one audio per scene or per shot. Also, a subtitle file (like SRT) can be generated in parallel, containing timestamps and text for each line.
    Implementation: For **text-to-speech**, we use open-source TTS models. A strong option is **Indic Parler-TTS** by AI4Bharat, which supports English and Hindi with many voices [12]. It provides a unified model for 20 languages (including English and Hindi) and even supports style prompts (controlling tone, pitch, etc.) [13] [14]. Another option is **Meta's TTS models** (e.g. the Orpheus model family) which are multi-lingual and can do voice cloning [15]. For simplicity, we could use one English voice for narration and one Hindi voice for any Hindi lines (if the story has code-mixed language). All model choices are configured via `.env` (e.g. `TTS_MODEL_NAME`). The agent will likely use an API or

local inference: if local, ensure the voice model is small enough for real-time (some open models require GPU for faster synthesis).
Timing & Sync: The agent may need to pace the narration to fit the video. Our pipeline can handle this by either adjusting the video clip length to match speech (e.g. if a narration for a scene is 8 seconds, ensure the visual shots total ~8 seconds), or by allowing slight pauses. We plan to use the shot durations as guidance for how long each TTS clip should be (some TTS APIs allow adjusting speed or inserting pauses to hit a target duration). In MVP, we can manually align them or simply let the narration run and consider it in final editing.

29. **Music Generation Agent:**
    Responsibility: Create a background music track suitable for the mood of the story or scene. This adds cinematic quality.
    Input: Could be the entire story or scene descriptions to gauge mood (e.g. an action scene vs. a calm scene). Perhaps the LLM can classify each scene's mood (sad, upbeat, tense, etc.) as part of the scene metadata.
    Output: An audio file (e.g. MP3 or WAV) with instrumental music. Possibly one per scene that can be cross-faded, or a single track for the whole video.
    Implementation: We use an open-source music generation model such as **Meta's MusicGen**. MusicGen is open-sourced and can generate ~12 seconds of audio from a text prompt (or longer by stitching segments) [16] [17]. It requires roughly 16GB GPU memory to run, which fits our production scenario [17]. We can generate music in segments (e.g. generate 30 seconds for a scene with a prompt like "tense orchestral music for a battle scene") and then combine them. Another option is to use **Stable Audio (Stable Diffusion-based audio)** or **Riffusion** for a more lo-fi approach, but MusicGen's quality is higher.
    Alternatives: We also allow the user to **upload external audio or music** instead of AI-generated music, to give flexibility. The system design accounts for an optional music input that, if provided, will be used directly rather than generating new music. This is useful to save time or if the user wants a specific soundtrack.
    Sync: Music can simply play in the background; we will likely loop or trim the generated music to cover the whole video duration. For MVP, a single background track softly looping under the narration might suffice.

30. **Video Composition & Editing Agent:**

31. *Responsibility:* **Stitch together** all the visual and audio pieces into the final video file. This includes concatenating video clips in order, overlaying or mixing audio tracks (narration, dialogue, music), and adding subtitles if needed. It also handles any transitional effects between shots/scenes (e.g. fade to black between scenes).
32. *Input:* The list of `Scene` objects with their `Shot` video files, plus all generated audio files (with knowledge of which audio goes where) and subtitle data.
33. *Output:* The final video file (e.g. MP4 or MKV) with integrated audio and subtitles (we can also output a separate subtitle file if desired).
34. *Implementation:* We can leverage powerful open-source tools like **FFmpeg** (via command-line) or Python libraries like **MoviePy** to assemble the video. For example, MoviePy can read the video clips and audio and composite them on a timeline [18] [19]. The agent will iterate through scenes and shots in sequence, using each shot's duration to know when to cut. Subtitles can be burned in using MoviePy's text overlay or by generating an `.srt` and using FFmpeg to embed it.

35. *Performance:* This step is IO-heavy but not too computational. It should be fine on CPU for even long videos, though memory might be needed for loading large clips. We ensure to handle files one by one (streaming or using temp files) to avoid memory bloat.
36. *Quality Control:* After rendering, the agent can do a quick verification (e.g. ensure the video length matches the sum of shot durations, check that all expected files were included). If something is wrong (missing clip, etc.), it can raise an error for the orchestrator. In future, a preview could be generated for user approval at scene level before final stitching.

Throughout this pipeline, each agent's interactions are **orchestrated** by a central controller, and intermediate results (like the `Scene` and `Shot` data) are stored and passed along as **Pydantic data models** in memory (and optionally saved to disk or a database for persistence). The use of structured models means agents can be swapped out or improved independently as long as they adhere to the input/output schema.

System Architecture and Components

High-Level Architecture: This system follows a **modular microservice-style** architecture but can be deployed as a single Python application. The main components include:

- **FastAPI Web Service:** Exposes endpoints for starting a video generation job and querying status/results. For example, a `POST /generate_video` endpoint accepts the story text and returns a job ID, and a `GET /status/{job_id}` gives progress or the final video URL. FastAPI also serves as the orchestrator entry point, invoking the pipeline (possibly via a background task or worker thread). Pydantic models are used here to validate request and response schemas.
- **Orchestrator (Coordinator):** This is the core controller that manages the workflow across agents. It receives a job request and manages the sequence of agent calls, as outlined in the pipeline above. It could be implemented as a class `VideoGenerationPipeline` with a method `run(job: VideoJob)`. The orchestrator handles **dependency injection** of services (LLM client, model references) based on configuration, so agents do not hardcode which LLM or model to use. It also implements **retry logic**: if any agent fails (exception or invalid output), it can retry or gracefully handle the error (e.g. skip a scene if non-critical, or mark job failed).
- **Agent Classes:** Each agent (SceneSplitter, ShotPlanner, etc.) can be implemented as a separate class with a common interface, say `Agent.run(input) -> output`. These classes encapsulate the logic described earlier. For LLM-based agents, they will internally call an LLM service (abstracted behind an interface, to allow swapping API providers or local models). For generation agents, they might call external processes or libraries. Following SOLID principles, each agent class has a single responsibility and can be extended or modified without affecting others. For example, a `SceneSplitterAgent` might have a reference to an `LLMClient` but knows nothing about how videos are made. This separation also allows easier **unit testing** of each stage.
- **LLM Service / Client:** A small module that handles interactions with language models. We define an interface (e.g. `LLMClient.call(prompt) -> str`) and provide implementations for different providers:
- *TogetherAIClient:* Uses Together AI API to call a specified model (like `mistral-7B` or `qwen-14B`). Together.ai offers open models with reasonable pricing and even chain-of-thought support for complex tasks [1]. We configure the model choice in `.env` (e.g. `LLM_PROVIDER=together`, `LLM_MODEL=mistral-7B-v0.1`).

- *GroqClient:* Similarly, an implementation that calls Groq API if we choose (Groq provides ultra-fast inference for certain open models like "Gemma-7B" or a Mistral variant at low cost [1]). This might be used for speed-critical agents (Groq's models return results in single-digit milliseconds for short prompts [1]).
- *LocalLLMClient:* Optionally, for debugging, a local model runner (using HuggingFace Transformers with a smaller model on CPU). For instance, a 7B model with quantization might run on CPU for testing logic (albeit slowly). This could be gated by an env flag (e.g. `USE_LOCAL_LLM=True` for debug).
  The orchestrator can instantiate the appropriate LLM client based on config and pass it to LLM-based agents. Ensuring this **swappability** satisfies the requirement that LLMs are configurable and not hard-coded.
- **AI Model Wrappers:** Similar to LLM client, we create service classes for other AI models if needed:
- *Text2VideoService:* Provides an interface `generate_clip(prompt, duration) -> filepath`. One implementation might wrap the **Diffusers** library to run Stable Diffusion and Video Diffusion pipelines (taking care of loading models into memory, etc.). Another implementation might call out to a cloud API (like a hosted ModelScope text2video) if available. The selection can be configured in `.env` (e.g. choose between local vs. remote). We may also include in this service the logic for the two-step image+animate approach described (generate image, then video), abstracting it away from the orchestrator.
- *TTSService:* Provides `synthesize(text, voice) -> audio_file`. Implementation might use **Coqui TTS** or **Parler-TTS** locally, or call a huggingface API. Since TTS may require different models for English and Hindi, the service could internally choose based on language tags or accept a voice parameter that implies a language. We will set voice names in config (e.g. `TTS_VOICE_EN=english_male`, `TTS_VOICE_HI=hindi_female`) and map those to model settings.
- *MusicService:* Provides `generate_music(prompt or mood, duration) -> music_file`. Implementation will likely call the **MusicGen** model. We can run MusicGen via the `audiocraft` library locally on GPU (if available) or use a pre-deployed API. The prompt to MusicGen can be a combination of scene mood and desired length (the model itself may generate only 12 sec at a time [16], so for longer music we might loop or stitch results).

- *VideoEditingService:* Encapsulates video editing functions (perhaps just thin wrappers around MoviePy or FFmpeg commands). For example, a method to concatenate clips, another to overlay audio, another to add subtitles. This service will be used by the Video Composition Agent. Abstracting it allows replacing MoviePy with direct ffmpeg calls if performance dictates.

- **Data Storage & Management:** We need to manage intermediate and final outputs:

- *File Storage:* Video clips and audio files can be stored in a designated directory (configurable path). On RunPod or cloud, this could be a mounted volume or cloud storage (S3, etc.). The system should generate unique filenames (perhaps including job ID and scene/shot IDs). For tracking, we maintain these paths in the Pydantic models (e.g., `Shot.video_path`). After job completion, the final video path is returned or made downloadable.

- *Database:* For MVP, we might not use a database—jobs can be tracked in memory or simple JSON files, given one job at a time. But for robustness, a lightweight database or even a Redis cache could store job states and results (especially if multiple concurrent jobs need to be handled). This is an implementation detail that can evolve; the design keeps it simple initially.

- **Job Queue / Worker:** In production, generating a video is a long process. The API should ideally not block the request. We will likely implement job handling such that the `POST /generate_video` immediately returns a job ID after validating input, and then the actual processing runs asynchronously. We can use Python's background tasks (FastAPI supports this) or a task queue like **Celery/RQ** for more scalability. In either case, the **job state** (queued, in_progress, completed, failed) is tracked. The `GET /status` endpoint reads that state and may provide intermediate progress (e.g. "Scene 2 of 5 is being processed"). The orchestrator can update the status after each major step.

- For example, after scene breakdown, status might update to "scenes identified", after each scene's video is done, it could increment a progress counter. This helps in tracking and possibly in resuming if something fails.

- On RunPod, one could also consider using RunPod's API to spin up a worker pod per job, but a simpler approach is to have a long-running service on a GPU-enabled pod that handles one job at a time from a queue.

- **Configurability and Environment (.env):** All secrets and paths go into a `.env`. For instance:

- `TOGETHER_API_KEY=<key>` for LLM API,
- `MODEL_MISTRAL_PATH=./models/mistral-7b` if running locally,
- `TTS_MODEL_NAME=ai4bharat/indic-tts` (HuggingFace model name),
- `VIDEO_MODEL=StableVideoDiffusion1.1`,
- `GPU_ENABLED=True`, etc.
  We use a library like python-dotenv or Pydantic's `BaseSettings` to load these config values. This allows switching models or providers by editing the env file, without touching code. All string literals that are likely to change (like prompts or special tokens) will either be in the `.env` or in a dedicated configuration file. For example, prompt templates for LLM agents can be stored in a config (possibly multi-line values in the env or a JSON/yaml config file loaded at runtime). Using **enums** for fixed options: e.g., define `class VoiceType(Enum)` for available voices, `class ModelProvider(Enum)` for LLM provider names, etc., to avoid sprinkling raw strings around the code.

- **Multi-Agent Coordination:** Although our implementation can be hand-coded, we considered existing frameworks:

- **LangChain**: Provides chains/agents abstractions but might introduce unnecessary complexity and overhead for our pipeline. Our steps are fairly linear and well-defined, so a custom orchestration may be cleaner. LangChain is more beneficial for dynamic tool usage or conversation-based agents, whereas we have a fixed sequence.
- **LangGraph** and **Pydantic AI**: LangGraph is a framework for orchestrating multi-agent workflows with a graph-like structure and persistent state, and Pydantic AI uses Pydantic models for agent I/O [20]. This aligns conceptually with what we do (we already plan to use Pydantic schemas). Indeed, one community approach combined LangGraph and Pydantic AI to build scalable agent systems, leveraging Pydantic for type-safe agent outputs and LangGraph for workflow management [21] [22]. In our design, since the workflow is mostly linear (scene -> shots -> generation -> assembly) and not highly branched, we may not need the full power of LangGraph. A simpler orchestrator class with

sequential logic and error handling could suffice (making the system easier to debug and with less overhead, per the requirement to only use such frameworks if they offer measurable benefit).

- **AutoGen (Microsoft)**: Another framework for multi-agent collaboration. AutoGen can simplify agent conversations and tool usage. In our case, the agents aren't exactly chatting with each other; they pass data along a pipeline. If we introduced a feedback loop where, say, the Video Generation agent could request clarification from the LLM if a prompt was unclear, then a framework like AutoGen might help manage that dialogue. For now, we note it as a potential extension but not necessary for MVP.
- **Conclusion on Frameworks:** We will implement the orchestration manually with Pydantic models and straightforward control flow, given it's a defined pipeline (this ensures we maintain control over each step for optimization). However, the design doesn't preclude integrating these frameworks in the future if the system grows more complex (e.g., adding branching logic or a richer tool-using agent). The focus is on **modularity and clarity** rather than heavy framework usage. (As one Reddit user noted, sometimes LangChain/LangGraph can be "overkill" for simpler flows [23] [24], and a lighter approach can work just as well.)

Hardware Split (CPU vs GPU):
The architecture considers running in two modes: local (CPU) and production (GPU). We design the code to detect available resources or modes: - In **local debugging mode** (e.g. an environment flag `DEBUG=True` or absence of GPU), the pipeline might skip or stub out heavy steps. For instance, instead of actually generating videos, the Visual Generation agent could retrieve placeholder images or use a very low-end model to generate a single frame, just to allow testing the stitching logic. TTS could use a fast but low-quality voice or simply print text. This allows a developer on a 16GB RAM laptop to run through the pipeline quickly, verifying that each agent produces the right format, without requiring a beefy GPU. We assume 16GB CPU can handle running the Mistral-7B via API (the API call is light for the client) and maybe small diffusions at tiny resolution if needed.
- In **production mode** (with GPU), the full fidelity models are used. If using RunPod, we would deploy this entire service in a container on a GPU instance. Alternatively, we might run the orchestrator on a CPU instance and have it call a separate GPU worker (but that complicates deployment). A simpler deployment is a single container with access to GPU that can do everything. The `.env` can specify different model paths if needed (like a high-res model path for production vs. a small model path for local). Also, in production we might enable parallel processing (e.g. generate multiple shots concurrently if multiple GPU cores or multiple GPUs are present).

Architectural Diagram:
The system can be visualized as follows: the **FastAPI** layer receives a request and creates a **Job** object (with a unique ID and initial status). The **Orchestrator** then sequentially invokes: (1) SceneSplitterAgent (using LLM Service) -> yields Scene list (Pydantic). (2) For each scene, ShotPlannerAgent (LLM) -> yields Shots for that scene. (3) For each shot, VisualGenerator (VideoService on GPU) -> yields video file; also parallelly, DialogueTTS (TTSService) -> yields audio file for speech, and MusicGen (MusicService) -> yields background music segment. (4) After all shots, VideoEditor (VideoEditingService) compiles the final video. During each step, status is updated in a **Job Status Registry** (could be in-memory or DB). Finally, the job is marked complete and the final video path is returned or stored. Logging and error handling wrap each agent call.

(In text form, not an actual diagram):
User/API → submits story → **Orchestrator** → [calls] **SceneSplitter (LLM)** → Scenes JSON [4] → **ShotPlanner (LLM)** → Shots JSON per scene → loop scenes/shots: **VideoGen (Diffusion)** → clip.mp4, **TTS**

(Synth) → narration.wav, **MusicGen** → music.wav → **VideoComposer** (ffmpeg/moviepy) → final_video.mp4 with audio+subtitles [18] → output back to **User**.
Throughout, **Pydantic models** ensure each stage's output is validated and structured, and config from **.env** is used (for model choices, API keys, file paths) at each step.

Repository Structure

We will organize the code repository in a **modular, scalable** way. Below is a proposed structure of the Python project (each bullet is a directory or file, with sub-points as its contents):

- `app/` – FastAPI application package
- `main.py` – Initializes FastAPI app, defines API endpoints (`/generate`, `/status`, etc.), and includes any startup configuration (loading env, setting up logging).
- `schemas.py` – Pydantic models for API requests and responses (e.g. `GenerateRequest` with story text and options, `StatusResponse` with job progress). It may reuse models from the core pipeline (scenes, shots) by importing from core.
- `dependencies.py` – If using FastAPI dependencies (e.g. to provide orchestrator instance or DB session), defined here.
- `core/` – Core pipeline logic
- `models.py` – Pydantic models for the internal data structures: `StoryInput`, `Scene`, `Shot`, `VideoJob`, etc. Also define enums like `ModelProvider`, `VoiceType`, etc. These models map to the data passed between agents.
- `orchestrator.py` – Contains the `VideoGenerationPipeline` class or similar, which implements the high-level run logic orchestrating agents. It will likely have methods for each stage, or one big `execute()` that calls submethods in order. This module handles job state updates. Possibly it could be designed to run in a background thread or Celery task.
- `agents/` – a sub-package containing each agent class:
  - `scene_splitter.py` with `SceneSplitterAgent` class (LLM calls to split scenes).
  - `shot_planner.py` with `ShotPlannerAgent`.
  - `video_generator.py` with `VideoGenerationAgent` (could further be split into image_gen and motion modules, but one class can handle it based on config: e.g. use `self.mode = 'text2video' or 'img2img'`).
  - `audio_tts.py` with `TTSAgent` (could handle both narration and dialogue by selecting voices).
  - `music_generator.py` with `MusicAgent`.
  - `video_compositor.py` with `VideoEditingAgent`.
  - Each agent class will have a `run(input) -> output` method. They may internally use services from the next folder.
- `services/` – integration with external models/APIs:
  - `llm_client.py` – classes for LLM clients (TogetherAI, Groq, LocalHF). This might use external SDKs or `requests` to call APIs.

    - `vision.py` – functions or classes for image/video generation. For example `DiffuseVideoGenerator` that loads a diffusers pipeline (perhaps at init) and has `generate(prompt, init_image=None)`. If using an API like Stability's, this class would call that API.
    - `tts.py` – functions to call TTS models. Possibly wrappers around libraries like Coqui or an HTTP call to a service if one is set up.
    - `music.py` – wrapper for MusicGen or other music model.
    - `video_editing.py` – uses MoviePy or ffmpeg to implement concatenate and overlay. e.g. a function `compose_video(shots: List[Shot], audio_tracks: List[str], subtitle_file: str) -> str`.
    - These services are used by agents; we might inject them via the agent's constructor or have them as singleton utilities. Using dependency injection (passing in the service instances, configured with .env values) adheres to the Dependency Inversion principle (the high-level agent logic is not tied to low-level implementation of model calling).

- `workers/` – (Optional if using a task queue) could contain Celery worker setup or background thread management. For instance, a `worker.py` that the process uses to run jobs asynchronously. On RunPod, maybe not needed if one job per container instance is used.

- `config/` – Configuration files or templates:

- `.env.example` – Sample environment file listing all configurable keys (model names, API keys, any special settings like `MAX_RETRIES`, `DEFAULT_DURATION` for shots, etc.). This helps users/developers know what to set.
- `prompts/` – possibly store prompt templates as text files (e.g. `scene_prompt.txt`, `shot_prompt.txt`) that the code can load. This way, prompt tuning can be done without changing code.

- `logging.conf` – if a custom logging configuration is needed (or just configure in code).

- `README.md` – Documentation for how to install and run the project, covering local vs production usage, etc.

This structure cleanly separates the API layer, the core pipeline, and the model-specific integration. It also makes testing easier (each agent and service can be tested in isolation with mocked inputs).

Sample Pydantic Models and .env Configuration

Below are examples of critical data models (simplified for illustration):

```
from pydantic import BaseModel, Field
from enum import Enum

class ShotType(str, Enum):
    WIDE = "WIDE"
    CLOSEUP = "CLOSEUP"
```

```
    # etc, optional classification

class Shot(BaseModel):
    id: int
    description: str  # visual description/prompt
    duration: int = Field(..., description="Duration in seconds")
    dialogue: str = None  # any dialogue or narration text in this shot
    video_path: str = None  # path to generated video file

class Scene(BaseModel):
    id: int
    summary: str  # short description of scene
    text: str = None  # original story text for this scene (optional)
    shots: list[Shot]

class VideoJob(BaseModel):
    job_id: str
    status: str  # e.g. "pending", "running", "completed", "error"
    story: str
    scenes: list[Scene] = []
    output_path: str = None  # final video path when done
```

These models will be used throughout. For example, `SceneSplitterAgent` will create `Scene` objects. `ShotPlannerAgent` will fill in `Scene.shots` with `Shot` objects. The orchestrator encapsulates everything in a `VideoJob` model that tracks the whole job state. Using Pydantic ensures if an LLM returns something unexpected, we catch it early via validation.

Environment Variables (.env) example:

```
# .env example content
# API Keys and endpoints
TOGETHER_API_KEY=<your_key_here>
LLM_PROVIDER=together      # could be "together", "groq", "openrouter", or
"local"
LLM_MODEL= mistri-7B-v0.1  # model name for LLM (if API, use their naming; if
local, path or name)
# Model and resource configs
DIFFUSION_MODEL_PATH=./models/stablevid1.1   # path or name for text2video model
SD_MODEL_PATH=./models/sdxl_base             # if using SD for image generation
TTS_MODEL_NAME=ai4bharat/indic-parler-tts    # HuggingFace model for multi-
lingual TTS
MUSIC_MODEL=facebook/musicgen-melody         # example identifier for MusicGen
GPU_ENABLED=true        # flag to indicate if GPU is available
MAX_SCENES=10           # any limits on scenes/shots to prevent overloading
# Prompt customization
SCENE_PROMPT_TEMPLATE="Break the story into scenes... (template text)"
```

```
SHOT_PROMPT_TEMPLATE="For the scene: {scene_summary}, describe shots...
(template text)"
```

This shows how one might configure the system. The `.env` values are loaded into a config object, which is passed around or accessed by agents.

MVP Execution Plan (1–2 min Video Prototype)

For the MVP, we will implement a simplified version of the pipeline to generate a **short 1–2 minute video** from a given story. The purpose of the MVP is to prove the end-to-end flow works and to identify performance bottlenecks. We'll likely use a shorter story (or a single scene from a long story) to limit scope. Key steps for the MVP:

1. **Setup Development Environment:** Ensure we have a machine (local or small cloud VM) with at least 16GB RAM. Install required libraries: FastAPI, Pydantic, transformers (for LLM if local), diffusers, moviepy, TTS libraries, etc. Also set up model files: download a smaller text-to-video model (e.g. Stable Diffusion 1.5 + AnimateDiff or ModelScope text2video which can generate a few seconds at low res). If GPU is not available locally, we may skip actual generation and focus on orchestration.
2. **Implement Scene and Shot Agents (LLM):** For MVP, these could be stubbed or use a small model. We might skip directly to using an available open API with an instruction-tuned model (for example, TogetherAI's endpoint for a 7B model). Test these with a sample story to ensure we get a reasonable scene/shot breakdown. If the story is short, it might only produce 1–2 scenes and a handful of shots – perfect for MVP. Verify the JSON outputs parse into Pydantic models.
3. **Implement a Basic Visual Generation:** Instead of full video generation which could be slow, the MVP might generate a single representative image per shot using Stable Diffusion (which can run decently on a 16GB CPU if using a small model and low resolution). We can then use a simple pan/zoom effect or just display it statically for a couple of seconds as the "video". This can be done by saving the image and then using moviepy to create a video clip from it (or even just treat it as 2-second video via duplication of frames). This simplification means we test integration with diffusion without needing long sequences. If a GPU is accessible (say on RunPod for testing), we can try a short actual text-to-video (like 2 seconds of animation) just to see it works.
4. **TTS and Audio:** Use a readily available TTS voice for MVP. For example, the Coqui TTS library has pre-trained English models that can run on CPU. We can generate a short narration of one or two lines from the scene. For Hindi, if needed, test with a small Hindi sentence. If installation of a Hindi model is complex, we might skip Hindi for the initial test, but ensure our design supports adding it. We can also simply use an online service for quick prototyping (though for open-source purity, maybe not). The key is to have at least one voice audio to overlay.
5. **Music:** For MVP, we might not train or run MusicGen due to complexity. Instead, use a royalty-free music clip (just as a placeholder) or generate a simple tone with a Python library. The goal is to include the concept of background music in the final mix, even if it's a static file. (Alternatively, if time permits, try out the smaller version of MusicGen or an available HuggingFace Space to get a short music clip, then use it locally.)
6. **Stitching:** Use MoviePy to compose the final video. For MVP, we will have maybe 2–3 shots (images) each 5 seconds => ~15 seconds of video, which is fine. We'll overlay the TTS audio (which might also be ~15 seconds) and the background music (trimmed to length). Add simple text subtitles (MoviePy TextClip) at the bottom for what the narration says. Render to an MP4 file.

7. **Test End-to-End:** Run the pipeline with the orchestrator on the sample story. Monitor the console or logs to see each stage. Ensure errors are handled (if LLM fails, etc., but with a small prompt it should be okay). We should get an output `final_video.mp4`. Manually review it – does it roughly match the story scene? Are audio and video aligned? Note any improvements.
8. **Iterate & Refine:** If the video is too disjointed, we might adjust shot descriptions or durations. If audio timing is off, adjust how we time it (maybe by splitting narration by shot and aligning each piece with its image). Ensure all intermediate results are saved for debugging (scene JSON, shot JSON, etc.). This MVP process will reveal if any part is unexpectedly slow or error-prone. For instance, generating an image on CPU might be very slow (maybe 1 minute per image). If that's too slow, we might use a smaller model (Stable Diffusion 1.5 with 256x256 resolution for testing). We note these trade-offs but accept them for MVP.
9. **Success Criteria:** MVP is successful if we can input a short story and get a short video that has at least one scene's visuals, with a voiceover and possibly music. Even if the quality is basic, the fact that all components (LLM breakdown, generation, composition) worked together is a milestone.

After MVP, to reach the full 5–60 min range, we will incrementally scale: use more powerful GPUs, improve prompt engineering for LLM to handle longer texts (maybe process story in chunks if over context limit), integrate actual video animation models for better visuals, etc. But the core pipeline will remain as designed.

Deployment Notes (Local vs Production)

Local Debugging (16GB CPU):
- We run the FastAPI app on a local machine without GPU. In `.env`, we set `GPU_ENABLED=false` and perhaps use `LLM_PROVIDER=local` with a small model like `llama2-7b-chat-int4` for quick tests (or route to an external API to not strain local CPU too much). Visual generation might be turned off or simplified (e.g., use a placeholder image for each scene rather than actually generating). We ensure the pipeline can be executed in this mode quickly. This is useful for testing logic, but obviously will not produce the final quality output.
- Developers can step through agents, use test stories, and verify JSON structures. Because we use Pydantic, any mismatches will raise errors that we can catch in logs. We can also run unit tests for each agent using dummy data (e.g., feed ShotPlannerAgent a known scene description and see if it returns the expected format).
- All external dependencies (like model weights) should be optional or stubbed in debug mode. For example, if `DIFFUSION_MODEL_PATH` is not found, the VideoGenerator agent can simply log a warning and return a dummy output (like a pre-generated test clip or a black screen). This ensures the pipeline doesn't crash in dev environment.

Production Deployment (RunPod or Cloud GPU):
- We will containerize the application (Dockerfile including all dependencies and possibly model weights). On RunPod, we can launch this container on a suitable instance (with GPU, e.g., an A100 40GB or RTX 3090 24GB depending on model needs). The `.env` for production will enable all features: e.g., `GPU_ENABLED=true`, `LLM_PROVIDER=together` (with API key set), and paths to actual model weights (the Docker image might include the diffusion model weights or we mount a volume with them).
- **Resource Management:** On a single GPU, generating long videos might still be sequential. We might choose to deploy with one job at a time per pod (ensuring high VRAM usage doesn't overlap). If we need to handle multiple jobs concurrently, we could scale horizontally (multiple pods) and use an external

orchestrator to assign jobs to free pods. That is beyond MVP but worth noting for scalability.
- **Tracking and Monitoring:** In production, we'll integrate logging (possibly to a file or cloud logging). We also ensure any large files are stored in a persistent volume or uploaded to cloud storage after generation (so that the user can download the result). E.g., after final video is ready, the service might upload the MP4 to an S3 bucket and provide the link in the API response, to offload serving large files from the API server.
- **Differences from Local:** The main difference is that in production we do *not* stub anything – all actual models run. We must double-check that the specific versions of models and their memory usage fit the GPU. For example, if using SDXL and a text2video model concurrently, memory might be an issue; we might load and unload models as needed (e.g. load diffusion model, generate all frames, then free it and load MusicGen). The orchestration might schedule these to avoid peak memory usage (this can be part of pipeline optimization). We should document assumptions like needing X GB VRAM for diffusion and Y GB for MusicGen and note if sequential execution is required due to memory.
- **RunPod specifics:** RunPod can be triggered via API as well. One deployment strategy is to have a persistent service on RunPod that listens for requests (the FastAPI itself). Alternatively, one could create on-demand pods for each job. Our design favors a persistent service for simplicity. The user would call our FastAPI (hosted at some endpoint) to start jobs.

Security and Config: All API keys (for LLM services) must be kept secret (not exposed to users). FastAPI could have an authentication layer if needed (especially if deploying publicly). Also, model licenses need checking: e.g., Stable Video Diffusion might be non-commercial research only [6]. Since we aim for open-source, we assume either licenses allow our use-case or that this is for internal/educational use. If commercial use is intended, we must ensure models like MusicGen (CC BY-NC licensed weights) are acceptable or find truly Apache-2.0 models (like Mochi's Apache license [25] or others). This is an assumption to clarify: for Plan A, some open models have non-commercial clauses, which might be okay if this project is open-source non-commercial. Otherwise, an alternative Plan A could be to rely on user-provided models that they have license to use.

Reusable Tools and References

In building this system, we can take advantage of existing open-source projects and tools to avoid reinventing the wheel. Below is a list of some useful resources and libraries that align with our design (with links for reference):

- **ComfyUI Workflows:** *ComfyUI* is a powerful node-based UI for diffusion models. It has community-developed **workflows for text-to-video** (e.g., nodes for ModelScope or Stable Video Diffusion) [26]. While our system doesn't use a UI, we can reuse the underlying workflow logic or even run a ComfyUI server in headless mode to generate video clips. This could simplify handling multi-step diffusion pipelines (e.g., using a ComfyUI workflow that takes an initial image and returns a video, which we trigger programmatically).
- **Hugging Face Diffusers:** The HuggingFace `diffusers` library provides pipelines for text-to-video (e.g., **StableVideoDiffusionPipeline** and others) and is continually updated by the community. Using diffusers would give us a standard interface to many models and handle details like scheduler, frame interpolation, etc. (The Modal article updated in Oct 2025 highlights these pipelines for models like Hunyuan, Mochi, Wan2.2 [27] [28].) We should leverage these rather than writing our own low-level diffusion code.
- **Pydantic AI / LangGraph:** As discussed, if we wanted to formalize the agent orchestration, the **Pydantic AI** framework (by the Pydantic team) could be used to define agents with type-checked

input/outputs, and **LangGraph** to connect them in a flow [29] [22]. Since our manual approach is similar in spirit, we can draw ideas from these frameworks. For example, Pydantic AI suggests patterns for quick agent definition and ensuring one agent's output matches the next's expected schema [22]. Even if not using them directly, their concepts reinforce our design.

- **Temporal.io (Workflow Orchestration):** For a highly reliable production system, one might integrate with a workflow engine like Temporal to handle long-running processes (ensuring they can resume on failure, etc.). In fact, Pydantic AI has integration with Temporal for durable execution [30]. This could be overkill for MVP, but it's an option for scaling (especially if job needs to survive server restarts or be distributed).
- **AI4Bharat Indic TTS** and **Coqui TTS**: The AI4Bharat models [12] provide high-quality voices for Hindi and other Indian languages and are open-source (Apache 2.0). Coqui TTS has a rich repository of voices and tools to train or fine-tune. We might use pre-trained voices from these sources for good results (e.g., a pleasant English narrator voice and a Hindi voice). Reusing these models saves us from training our own TTS.
- **MusicGen / AudioCraft by Meta:** The open-source release of MusicGen [16] (part of the AudioCraft toolkit [31]) can be directly used. There are also community forks that allow longer generation or better quality (some use diffusion for music as mentioned in the TechCrunch article [16] [17] where multi-band diffusion was an upgrade). We should monitor those for improvements. Additionally, **Open Music libraries** like Suno's Bark (which is more for speech but can do music-like sounds) might be interesting, though Bark is more for multi-lingual speech.
- **MoviePy and FFmpeg:** These will be the backbone of video editing. *MoviePy* is pure Python and easy to use, but for final production, directly using FFmpeg commands might be more efficient (MoviePy ultimately uses ffmpeg under the hood). We can reuse lots of community recipes for adding audio streams, subtitles, etc., using ffmpeg. Also, tools like **ffmpeg-python** provide a Pythonic way to compose ffmpeg pipelines.
- **Example Projects:** We have inspiration from Thierry Moreau's recipe video pipeline [4] [18], which chained LLM (Mixtral 8x7B) to JSON formatting, SDXL for images, and Stable Video Diffusion for 4s clips, then MoviePy to stitch [4] [18]. Also Giulio Sistilli's multi-agent video system (used BeautifulSoup, LangChain, OpenCV, MoviePy) [19]. We can reuse patterns from these, such as using OpenCV if we need any frame post-processing, or how they managed memory by switching to local models to cut API costs [32]. Both examples reinforce that our chosen tools are viable in practice.
- **GitHub Repos**: We should identify and bookmark repositories for the models we use: e.g. the official *Mistral* repo (for any prompt format specifics), *ModelScope text2video* GitHub (which might have sample code), *HuggingFace spaces or examples* for MusicGen or TTS usage. These will provide code snippets and help debug any model integration issues. If possible, we'll use stable APIs (like huggingface's inference API for some parts during prototyping) and then swap to local to ensure we remain open-source and cost-effective.

Finally, we will document all assumptions and findings. For instance, if we assume *Shot durations* ~5s, but find out in editing that speech takes 7s, we note to adjust duration dynamically. If a certain open model isn't producing desired quality, we may list a backup (Plan B: use a paid API like ElevenLabs for TTS if open model fails, etc., keeping it optional). The architecture is designed to be **robust, extensible, and transparent** about these choices, enabling continuous improvement towards the goal of full-length AI-generated cinematic videos.

Sources:

- Moreau, T. (2024). *GenAI video generation pipeline (recipe example)* – Used multiple open models (Mistral 7B, SDXL, Stable Video Diffusion) and stitched results with MoviePy [4] [18].
- Giulio Sistilli (2025). *Multi-Agent Video Processing System* – Demonstrated a similar multi-agent approach (crawler, scriptwriter, asset creator, etc.) with LangChain and MoviePy on local GPU [19].
- Reddit AI Agents Community – Discussed using **Groq** for fast open-model inference and **Together.ai** for reasoning (Qwen-14B), and frameworks like LangGraph/AutoGen for orchestration [1]. Also highlighted combining Pydantic for agent I/O with LangGraph for complex workflows [29] [22].
- Stability AI & Modal Blog – Provided insight into state-of-art open video models (HunyuanVideo, Mochi, Wan2.2) and their resource requirements [7] [8], and open TTS models (Higgs, Kokoro, Orpheus with multi-lingual support) [33] [15].
- AI4Bharat Indic TTS – Open-sourced models for Hindi and 20 Indic languages (plus English) under Apache-2.0 [12]. Suitable for our TTS needs.
- Meta AI (MusicGen) – Open-source music generator (code MIT, weights CC BY-NC) requiring ~16GB GPU, capable of conditioning on text and melody for ~12s clips [16] [17].

---

[1] What's the cheapest(good if free) but still useful LLM API in 2025? Also, which one is best for learning agentic AI? : r/AI_Agents
https://www.reddit.com/r/AI_Agents/comments/1m1ag00/whats_the_cheapestgood_if_free_but_still_useful/

[2] [3] [19] [32] Building a Multi-Agent AI System for Video Processing | by Giulio Sistilli | Level Up Coding
https://levelup.gitconnected.com/building-a-multi-agent-ai-system-for-video-processing-ca629a09e210?gi=87811bb79e83

[4] [5] [6] [9] [18] Build your own GenAI video generation pipeline | by Thierry Moreau | Medium
https://medium.com/@thierryjmoreau/build-your-own-genai-video-generation-pipeline-cdc1515d1db9

[7] [8] [10] [25] [26] [27] [28] Top open-source text-to-video AI models
https://modal.com/blog/text-to-video-ai-article

[11] Stable Video — Stability AI
https://stability.ai/stable-video

[12] [13] [14] ai4bharat/indic-parler-tts · Hugging Face
https://huggingface.co/ai4bharat/indic-parler-tts

[15] [33] The Top Open-Source Text to Speech (TTS) Models
https://modal.com/blog/open-source-tts

[16] [17] Meta open sources an AI-powered music generator | TechCrunch
https://techcrunch.com/2023/06/12/meta-open-sources-an-ai-powered-music-generator/

[20] [21] [22] [23] [24] [29] The Most Powerful Way to Build AI Agents: LangGraph + Pydantic AI (Detailed Example) : r/AI_Agents
https://www.reddit.com/r/AI_Agents/comments/1jorllf/the_most_powerful_way_to_build_ai_agents/

[30] Here's how to build durable AI agents with Pydantic and Temporal
https://temporal.io/blog/build-durable-ai-agents-pydantic-ai-and-temporal

[31] AudioCraft - Meta AI
https://ai.meta.com/resources/models-and-libraries/audiocraft/