ChatGPT

# Updated Plan A AI Video Generation Blueprint (Open-Source & Local Compute)

In this updated blueprint, we enhance the original Plan A design with new features for flexibility, efficiency, and maintainability. The core architecture remains a multi-agent pipeline that converts a long-form story into a cinematic video using only open-source models [1] [2]. We preserve the original design principles – modular agents, environment-based configurability, and Pydantic data models – while adding advanced capabilities for debug vs. production modes, parallelism, and smooth video/audio continuity.

## Modular Folder Structure

To keep the project organized and scalable, we structure the repository into clear sub-packages, separating the FastAPI app, core pipeline, agents, service integrations, configuration, mock assets, and deployment setup. The **proposed folder layout** is:

- `app/` – FastAPI application package. Contains the web API layer:
- `main.py` – Initializes the FastAPI app and defines endpoints (e.g. `/generate`, `/status`), including any startup logic like loading env vars and configuring logging [3].
- `schemas.py` – Pydantic models for API requests/responses (e.g. `GenerateRequest` with story text and options, `StatusResponse` for job progress), possibly reusing core models [3].

- `dependencies.py` – (Optional) FastAPI dependency functions (e.g. providing a singleton orchestrator or DB session) [4].

- `core/` – Core pipeline logic and data models. This is the heart of the system:

- `models.py` – Pydantic models for internal data structures like `StoryInput`, `Scene`, `Shot`, `VideoJob`, etc., plus enums (e.g. `ModelProvider`, `VoiceType`) [5]. These define the schema for data passed between agents.

- `orchestrator.py` – The pipeline orchestrator (e.g. `VideoGenerationPipeline` class) that implements the high-level workflow. It coordinates all agent stages (either via separate methods per stage or a single `execute()` method) and updates job state [6]. This could run tasks synchronously or dispatch background jobs (e.g. via a thread or Celery, see `workers/` below).

- `core/agents/` – Specialized agent classes for each stage of the pipeline [7]. Each agent has a clear responsibility and a `run(input) -> output` method. For example:

- `scene_splitter.py` with `SceneSplitterAgent` (calls an LLM to split the story into scenes) [7].
- `shot_planner.py` with `ShotPlannerAgent` (LLM to break each scene into shots).
- `video_generator.py` with `VideoGenerationAgent` (generates visuals for each shot; could be further subdivided into image vs. motion if needed [8]).

- `audio_tts.py` with `TTSAgent` (generates narration/dialogue audio).
- `music_generator.py` with `MusicAgent` (generates background music).
- `video_compositor.py` with `VideoEditingAgent` (stitches video clips, audio, subtitles, transitions) [9] .

Each agent may internally use lower-level services (for model inference or external API calls) via clear interfaces. This separation follows SOLID principles (each agent has a single responsibility and depends on abstract interfaces) [2] .

- `core/services/` – Low-level integrations with AI models and tools [10] . These are utility modules or classes that agents rely on for actual data generation/manipulation:
- `llm_client.py` – Client for language models (could support multiple backends like local HuggingFace, Together.ai API, etc.) [11] .
- `vision.py` – Image/video generation functions or classes, e.g. a `DiffuseVideoGenerator` that loads a Stable Diffusion pipeline and provides `generate(prompt, init_image=None)` for text-to-image or image-to-video generation [12] . If using an external API (e.g. Stability AI), this service would call that API.
- `tts.py` – Text-to-speech generation, possibly wrapping an open-source TTS library (e.g. Coqui TTS) or calling a local TTS model [13] .
- `music.py` – Music generation wrapper, for example using Meta's MusicGen model to create background music [14] .
- `video_editing.py` – Video editing/compositing utilities, using FFmpeg or MoviePy to concatenate clips, overlay audio, and add subtitles [15] . For instance, a function `compose_video(shots: List[Shot], audio_tracks: List[str], subtitles: str) -> Path` would handle merging all shots and audio tracks into the final video file [15] .

*Dependency Injection:* We inject these services into agents (e.g. via agent constructors or a service locator) configured according to environment variables [16] . This ensures high-level agent logic isn't tied to low-level implementations, following the Dependency Inversion principle [16] . For example, the `VideoGenerationAgent` can accept an `IVideoGenerator` interface; in debug mode this might be a dummy generator, while in production it's a diffusion-based generator (see **Execution Profiles** below).

- `workers/` – (Optional) Background worker setup for async job processing. If using Celery or background threads for long-running video generation tasks, this could contain a `worker.py` or task queue configuration [17] . (For simple deployments, the orchestrator might just run synchronously or spawn a thread, and on RunPod we might run one job per container, making a full task queue optional [17] .)

- `config/` – Configuration files and templates for easy customization:

- `.env.example` – A template environment file listing all configurable settings (model paths, API keys, flags like `MAX_RETRIES`, default shot duration, etc.) [18] . Developers copy this to `.env` and fill in actual values for their setup.
- `prompts/` – Directory for prompt templates (e.g. `scene_prompt.txt`, `shot_prompt.txt`) [19] . Storing prompt text in files allows tweaking the prompt wording or style without changing code – a YAML or text file can define the LLM prompt structure for each agent stage.

- `logging.conf` – (Optional) Config for Python logging, if needed for more complex logging setup [20].

- `assets/` – (New) Assets directory for static and mock media. In **debug mode**, this holds placeholder images, video clips, or audio files used as stand-ins for generated content. For example, `assets/mock/` might contain a few pre-made dummy video clips or images (like a black screen or a title card) that can be reused for fast debugging. The pipeline can copy or cache generated mock assets here on first use, then reuse them in subsequent runs.

- **Project root files:**

- `Dockerfile` – Container setup for production (GPU-enabled environment).
- `requirements.txt` and `requirements-dev.txt` – Separate dependency lists for production vs. development (detailed in *Dual Requirements & Docker* below). If using Poetry, the `pyproject.toml` can define profiles or extras for GPU vs CPU environments.
- `README.md` – Documentation on installation and usage, covering how to run locally (CPU debug) vs. on a GPU server [21], etc.

This structure cleanly separates the API layer, core pipeline, model-specific services, and configuration. It improves maintainability and testability, since each agent/service can be unit-tested in isolation with mocks [21]. It also sets the stage for our new debug/production duality and other features.

## Execution Profiles: Debug vs. Production

We introduce an `ExecutionProfile` enum to switch the system's behavior between a **CPU-friendly debug mode** and a **full-featured GPU production mode**. This aligns with the original requirement that the pipeline run on a 16GB RAM CPU for testing, and leverage GPUs for final output [22] [23].

```python
from enum import Enum

class ExecutionProfile(Enum):
    DEBUG_CPU = "debug_cpu"
    PROD_GPU = "prod_gpu"
```

**Profile Behavior:** Many core services implement an interface (e.g. `IVideoGenerator`, `IMusicGenerator`, `ITTSGenerator`). At startup, the orchestrator or a factory will detect the desired profile and inject the appropriate implementations of these services. For example:

- **Video Generation**:
- **DEBUG_CPU** – Use a lightweight stub implementation, e.g. `MockVideoGenerator`, which might not invoke any heavy model. It could return a placeholder video clip (or even just generate a single frame image) for each shot [24]. This stub can be very fast and avoids loading large diffusion models when in debug mode.

- **PROD_GPU** – Use the real generative model, e.g. `DiffuseVideoGenerator` that wraps a Stable Diffusion or text-to-video pipeline [25]. This class would load the model weights (possibly at initialization) and generate actual video frames for each shot using the GPU.

- **Music Generation**:

- **DEBUG_CPU** – Use a `LoopedMusicGenerator` stub that perhaps loads a short, license-free music clip from `assets/mock/` and loops it to the needed duration. This avoids running a model like MusicGen when debugging.

- **PROD_GPU** – Use `MusicGenGenerator` that loads an open-source music generation model (like Facebook's MusicGen) with GPU acceleration to create an original soundtrack.

- **Text-to-Speech (TTS)**:

- **DEBUG_CPU** – Use a simple or offline TTS (or even a no-op). For instance, a `MockTTSGenerator` could just produce silence or a beep, or at most use a quick CPU TTS with a robotic voice. In extreme debug, it might just log the text instead of generating audio.
- **PROD_GPU** – Use a full TTS pipeline, e.g. Coqui TTS or another state-of-the-art model, possibly with multi-lingual support as needed (English/Hindi per original spec).

The blueprint already envisioned using environment flags to disable heavy components on CPU (e.g. `GPU_ENABLED=false` to skip real diffusion generation) [26]. By formalizing this into an `ExecutionProfile`, we make the selection explicit and type-safe. The `.env` file can include a setting like `EXECUTION_PROFILE=DEBUG_CPU` or the startup script can call a `detect_gpu()` (next section) to choose a default.

**Service Injection Example:** At startup, we detect the profile and instantiate services accordingly:

```
profile = ExecutionProfile.DEBUG_CPU  # or PROD_GPU, set via env or detection
if profile is ExecutionProfile.DEBUG_CPU:
    video_gen = MockVideoGenerator()
    tts_gen = MockTTSGenerator()
else:
    video_gen = DiffuseVideoGenerator(model_path=CFG.DIFFUSION_MODEL_PATH)
    tts_gen = CoquiTTSGenerator(model=CFG.TTS_MODEL_NAME)
# Pass these into agents or orchestrator
video_agent = VideoGenerationAgent(video_generator=video_gen)
tts_agent = TTSAgent(tts_generator=tts_gen)
```

In this way, the high-level pipeline code does not need many `if/else` checks; it simply uses the injected service. Each service class can also internally check the profile if needed (though we prefer handling it at composition time). This design ensures that in **DEBUG_CPU** mode, we don't accidentally load large models – heavy dependencies can even be omitted from `requirements-dev.txt` if they're never invoked in debug (see *Dual Requirements* section).

The two profiles enable the system to meet both goals: fast iteration on a CPU (possibly sacrificing quality) and full-quality output on a GPU [24] [27].

## Mock Asset Generation & Caching (Debug Mode)

In debug mode, we want to avoid any external downloads or long model generations. Instead, the system will **generate placeholder assets on first use and cache them** for reuse. This approach lets developers run the entire pipeline quickly on a laptop to verify the flow, without requiring actual AI model outputs [24].

- **Storyboard Images:** When the VideoGenerationAgent is asked to produce a shot in debug mode, it can create a simple image with the shot description text overlaid (simulating a storyboard panel). For example, using Python's PIL or MoviePy, it can render the prompt text on a plain background (perhaps color-coded per scene). This image serves as a stand-in "key frame" for the shot. We then save it to `assets/mock/shot_{id}.png`. Subsequent runs of the same shot description can load this cached image instead of regenerating it.

- **Dummy Video Clips:** If the pipeline expects an actual video file per shot (even in debug), we can programmatically create a short video (e.g. 1-second black screen or a simple slideshow of the storyboard image). Libraries like MoviePy can turn an image into a video clip easily. The first time a given scene/shot is processed in debug mode, we generate a tiny MP4 file (perhaps with the image and some text) and store it in the mock assets cache. Later, if the same content is requested, the cached file path is returned. Alternatively, the agent can reuse one generic clip for all shots if differentiation isn't important for the test.

- **Audio Stubs:** For narration or dialogue in debug mode, the TTSAgent could output a silent audio file or a tone of the correct duration. We might generate a silent WAV of appropriate length (based on reading speed for the text) and save to cache. Similarly, a single music loop could be stored and reused for all background music needs in debug.

Crucially, these mock assets need only be generated once. The orchestrator can check if a placeholder for a given type exists, and if not, create it. This is more efficient than downloading stock media and complies with the open-source/offline ethos (no external calls). It also aligns with the original design which suggested that in absence of models, the system should "return a dummy output (like a pre-generated test clip or a black screen)" rather than erroring [28]. By caching these dummy outputs, we ensure the debug pipeline is fast and deterministic.

Additionally, the blueprint noted we could simulate video by just cross-fading still frames as an extreme simplification [29]. Our approach realizes that by using either a single frame or a very short clip for each shot. The goal is to **enable quick end-to-end tests**: as the design document stated, a developer should be able to run through the pipeline on a 16GB RAM laptop, verifying each agent's output format, without needing any heavy model [26] [30].

## Automatic GPU Detection and Fallback

To streamline configuration, we implement a `detect_gpu()` utility that checks for GPU availability at runtime. On startup, the application will:

- **Detect GPU:** For example, attempt to import torch and call `torch.cuda.is_available()`, or use NVIDIA utilities to see if a CUDA device is present. If a GPU is found, we assume the environment can handle the heavy models.

- **Set Default Profile:** If GPU is available, default to `ExecutionProfile.PROD_GPU`. If no GPU is found, log a warning and automatically fall back to `ExecutionProfile.DEBUG_CPU` mode. This ensures that a user who launches the app on a machine without a GPU doesn't need to manually edit the .env; the system won't try to load CUDA-specific models by default [26]. (This complements the `.env` flag like `GPU_ENABLED` mentioned in the original design [26].)

- **Allow Override:** The `.env` or a command-line option can still override the profile. For instance, a developer with a GPU might still run in debug mode intentionally (to test the pipeline with stubs), or vice versa for consistency. An environment variable (e.g. `FORCE_PROFILE=DEBUG_CPU`) can force a mode regardless of auto-detection. The code will prioritize an explicit setting over auto-detect.

This automatic fallback increases robustness: the app "just works" on CPU-only machines by skipping GPU-only features. In production (with GPUs), it will by default run full-quality. The design goal of running on both local CPU and cloud GPU is thus met without manual toggles each time [22] [27].

*Implementation detail:* The detect function can reside in an init script or the FastAPI startup event. It should run early, before models are loaded. If falling back to CPU mode, we may also adjust some config (e.g. reduce number of diffusion inference steps or lower resolution in debug to further lighten load).

## Parallel LLM Calls (Async Agents with LLM_PARALLEL)

To improve speed, we add support for running certain LLM-based agents in parallel. In the .env configuration, a flag `LLM_PARALLEL=true` will enable parallel processing for scene and shot planning stages (and any other suitable stage), using Python concurrency (async IO or thread pools).

**Scene Breakdown Agent:** Typically, splitting the story into scenes is a single LLM call that processes the whole story [31]. This step usually cannot be parallelized *after* the fact, since one call produces the entire scene list. However, if the story is extremely long, we could **pre-split the story text** (e.g. by chapters or logical segments) and call the LLM on each segment concurrently. This would effectively treat each segment as a sub-story and merge the resulting scene lists. This approach might reduce context size per call [32], but it risks losing some global coherence. By default, we treat scene breakdown as one call (parallelism doesn't apply), but for advanced use-cases, the pipeline could be extended to parallelize it after careful NLP preprocessing (see **Story Ingestion** below).

**Shot Planning Agent:** This is a clear opportunity for parallelism. Once we have *N* scenes identified, the ShotPlannerAgent needs to generate shots for each scene. Each scene's shot plan is independent – one scene's breakdown doesn't affect another's. Therefore, if `LLM_PARALLEL=true`, the orchestrator will dispatch **concurrent LLM calls** for the *N* scenes in parallel (subject to available resources and API rate limits). For example, if there are 5 scenes, we can fire off 5 requests to the LLM (or spawn 5 asyncio tasks) to get shots for each scene at the same time. This can dramatically speed up the planning phase on multi-core CPUs or when using an LLM API that allows parallel requests. If `LLM_PARALLEL=false` (default), the scenes are processed sequentially (scene 1 → scene 2 → ...), which is simpler but slower.

**Implementation:** We can use an `asyncio` event loop or a thread pool. If using an external API for LLM (like Together.ai), making asynchronous HTTP calls is feasible. We would gather the results and then combine them back into the ordered scene list. The orchestrator ensures that even if done in parallel, the final ordering of scenes and shots remains correct.

**Other Parallel Opportunities:** While not explicitly asked, this flag could also enable parallel execution in other stages if safe. For instance, if multiple shots need to be rendered into video and the hardware has multiple GPUs or a GPU with enough memory, we could generate multiple shots concurrently. However, in our single-GPU Plan A scenario, it's safer to do video generation sequentially to avoid VRAM exhaustion [33]. The LLM stages (being CPU-bound or API-bound) are the primary beneficiaries of parallelism. The design remains mostly linear [34], but with this controlled parallelism we optimize the longest sequential part (LLM text planning).

**Concurrency Considerations:** We need to ensure thread-safety for shared resources (if any) when enabling parallel calls. Each LLM request is independent, so the main consideration is avoiding race conditions in assembling results. We also must handle API limits: e.g., if an LLM service only allows a few requests at once, we might cap the parallelism or use a semaphore. The `.env` flag gives flexibility to turn this on only when it's beneficial. By default, it might be off to keep things simple and avoid unforeseen issues, but advanced users can toggle it to accelerate the planning phase.

*(In summary, scene breakdown is generally a single-call process (parallelism not applicable unless story is chunked), whereas shot planning across scenes can be parallelized. Both agents are LLM-based, so this flag is specifically aimed at those to reduce overall processing time.)*

## Dual Requirements & Docker Strategy

Since we have two execution profiles, we also maintain two sets of dependencies and environment configurations:

- **Development (CPU/Debug) Requirements:** In `requirements-dev.txt` (or a dev section of `pyproject.toml`), we pin packages that allow the pipeline to run in debug mode without GPU. This includes core libraries (FastAPI, Pydantic, MoviePy, etc.) and **lightweight** AI model versions or none at all. For example, we might include a CPU-friendly version of PyTorch or even skip installing heavy diffusers/transformers libraries if our debug stubs don't use them. We can rely on small models or dummy outputs as discussed, so the dev environment doesn't need the full weight of diffusion or large TTS models. The blueprint suggested using "CPU-friendly models or stubs" for local debugging [35], and even skipping actual video generation if needed [36] – our dev requirements reflect that by not forcing huge downloads. We might include smaller alternatives (e.g. a tiny text-generation model for LLM agents if we want some actual output, or no model at all and stub the response).

- **Production (GPU) Requirements:** The `Dockerfile` (and possibly a `requirements.txt`) will specify the full set of heavy dependencies with GPU support. For instance, we will install PyTorch with CUDA (matching the GPU on RunPod or the target environment), the diffusers library (for Stable Diffusion models), and other model-specific libraries (HuggingFace Transformers, xFormers for efficiency, etc.). We also include the actual model weights either in the Docker image or mounted at runtime [27]. For example, the stable diffusion model checkpoint, the text-to-video model, the TTS model, etc., could be downloaded in the Docker build so that the container is self-contained [27]. The Docker image will be based on a CUDA-compatible base (like `nvidia/cuda`).

- **Conditional Installs:** If using Poetry, we can define two groups of dependencies: `[tool.poetry.dependencies]` for core and perhaps an optional group `[tool.poetry.extras]` or separate dev dependencies for GPU features. Alternatively, we maintain two requirement files and use the appropriate one. The project README can instruct developers to use `requirements-dev.txt` for local setup, and use the Docker for production (or `pip install -r requirements.txt` for full install). We ensure that the naming and versions of packages align between the two to avoid inconsistency. Critical packages like Pydantic, FastAPI, etc., will be the same version in both; the difference is mainly in model libraries and hardware-specific builds.

- **Docker Configuration:** We create a single Dockerfile (possibly multi-stage). In the build, we copy the application code and install production requirements. We set environment variables like `EXECUTION_PROFILE=PROD_GPU` and ensure `GPU_ENABLED=true` in the container's env so that all features are on [37]. The Dockerfile might also optimize for size by excluding dev/test files and using specific base images for machine learning. We pin library versions that are known to work with our selected models (e.g. diffusers version compatible with our video model, a specific transformers version for the LLM if needed, etc.). This ensures reproducibility in production.

By splitting dependencies, we satisfy both the **lightweight dev environment** and the **optimized production environment**. The design already emphasized using smaller models for quick iterations and full models for final output [23], which this strategy enables. It also prevents installing unused heavy packages in development, saving time and disk space.

Finally, we document this in the README (with perhaps a table of what's included in each profile) [21]. A developer knows that if they run locally, they should stick to debug mode, and to get real outputs they should use the Docker on a GPU machine. This dual approach ensures the project remains accessible and easy to test, while still being capable of high-quality results in its target environment.

## Continuity in Video Stitching

One challenge in multi-shot video generation is ensuring **visual and auditory continuity** between shots and scenes. We incorporate several strategies to make the final video feel cohesive rather than a series of disjointed clips:

- **Last-Frame as Init Image:** To maintain visual consistency between consecutive shots (especially within the same scene), the VideoGenerationAgent can take the last frame of shot *N* as an input (initial frame) for generating shot *N+1*. This technique helps carry over characters, scenery, or lighting from one shot to the next [38]. For example, if shot 1 ends with an image of the hero's face, shot 2's generation can start from that frame (with a new prompt) so the hero's appearance remains stable. The design document noted this as an advanced feature to consider, and we plan to implement it when using image-to-image or video continuation models [38]. In practice, for production we can use Stable Diffusion's img2img feature or a video diffusion model that accepts an initial frame to guide the next clip.

- **Consistent Styles and Characters:** We ensure that prompts carry persistent descriptors for important elements (main characters, settings) throughout a scene. An idea is to have the ShotPlannerAgent include some "memory" of the scene's key details in each shot description, so the generator doesn't drastically change e.g. the character's clothing or the time of day between shots. This isn't a direct stitching technique, but it's part of continuity – the LLM could be prompted to maintain consistency (possibly by providing a brief context of previous shot summary to the next prompt). We also consider using the same seed or model settings for shots that need the same style.

- **Smooth Transitions:** The VideoEditingAgent will add transitional effects between shots and scenes as needed [39]. For instance, between major scenes we might do a **fade to black** or a cross-fade to signal a shift [39]. Between quick consecutive shots in the same scene, a simple cut or a short cross-dissolve can make the transition less jarring. We can programmatically insert a few overlapping frames or use MoviePy/FFmpeg to create a cross-fade of, say, 0.5 seconds between clips.

- **Background Music Continuity:** Instead of generating completely separate music for each scene (which could lead to jarring audio cuts), we often generate one longer music track or a thematically consistent loop that runs under the whole video [40]. The MusicAgent could generate, for example, a 2-minute music piece and the VideoEditingAgent will loop or trim it to fit the full video length [41]. If separate music pieces are needed per scene (for mood changes), we will cross-fade audio during scene transitions so the music doesn't abruptly stop [40]. The audio mixing can also duck (lower) the music when narration/dialogue is present and raise it during pauses, for a more professional sound.

- **Frame Interpolation:** As an advanced measure, if there is a slight visual jump between the end of one clip and the start of the next (despite using the last frame init, differences can occur), we could use frame interpolation tools to generate intermediate frames. There are open-source frame interpolation models that could create a few in-between frames to blend two shots smoothly. This can be integrated into the VideoEditingAgent pipeline for scene transitions or even between any two clips that need smoothing.

- **Ken Burns Effects for Still Shots:** In cases where actual animation is limited (especially in MVP debug mode), we can simulate motion between key frames by panning and zooming on still images (Ken Burns effect) and cross-fading [29]. For example, if we only have two static images for a shot, slowly zooming in on the first and cross-fading to the second can imply movement. While this is more of a placeholder technique than a final solution, it maintains viewer interest and continuity without high compute cost [29].

All these techniques aim to make the stitched video feel like a continuous story. The blueprint explicitly mentioned adding subtle transitions like fade-to-black between scenes [39] – we extend that with smarter continuity like frame reuse [38] and consistent audio. The VideoEditingAgent is responsible for these touches: it knows the sequence of shots/scenes and can insert transitions and ensure audio flows across cuts [42]. The result should be a more **professional and cohesive video**, even though each clip is individually generated.

## Human-Friendly & Maintainable Code

We place a strong emphasis on code quality and configurability so that the project is **human-friendly** for developers and easy to adapt. Several practices and patterns are reinforced:

- **Centralized Configuration:** All important strings, file paths, model names, and magic numbers are configurable via environment variables or config files. The blueprint required that *"all configuration (model file paths, API endpoints, prompt templates, etc.) will be provided via environment variables (.env file)"* [43]. We abide by this: for example, model checkpoint paths, API keys, default durations, parallelism flags – all are read from `.env` (with sane defaults in `.env.example` [18]). Hard-coded values are minimized. This makes it easy to swap models or adjust parameters without code changes.

- **Pydantic Models for Data Schemas:** Each agent's input/output is defined with Pydantic models, enforcing a clear schema [44]. We continue this approach and ensure every data structure (Story, Scene list, Shots, etc.) has a corresponding model class. This provides validation (catching errors if an LLM returns malformed data) and self-documenting code. The Pydantic models include field descriptions and default values where appropriate, acting as an always-updated documentation of the pipeline's data flow. The use of enums (e.g. for `ShotType`, `VoiceType`) makes the code self-explanatory and avoids stringly-typed mistakes [45].

- **SOLID and Clean Architecture:** We maintain the separation of concerns between agents and services (as described in the structure) [16]. Each agent does one thing (e.g. planning shots, generating video, etc.), and each service handles one type of model. This modular design means a developer can "swap out" an implementation by modifying one class or one config entry. For example, to try a new diffusion model, one could create a new `AlternativeVideoGenerator` service and plug it in via config, without changing the orchestrator or agents. We include docstrings and type hints on all public classes and functions, describing their purpose and usage. This makes the codebase easier to read and navigate. Comments are added where non-obvious decisions are made (especially around advanced features like parallelism or continuity tricks).

- **Prompt Templates & YAML Configs:** We externalize prompt text into template files (under `config/prompts/`) [19]. These could be simple text files or a YAML defining the structure of system/user prompts for the LLM. Using external prompt templates has two benefits: (1) Non-developers (or future us) can fine-tune the wording or style without touching Python code, and (2) It encourages thinking of prompts as data, which can be versioned and experimented with systematically. For instance, a `scene_prompt.txt` might contain something like: *"You are a screenwriter. Split the following story into scenes... output as JSON."* – which we can tweak or even have multiple versions for A/B testing. The system can load these at runtime (perhaps caching them).

- **Logging and Monitoring:** We use Python's logging library to trace what the system is doing (e.g. logging when each agent starts/finishes, any fallbacks or retries, etc.). The `logging.conf` (or equivalent code config) sets log levels and formats, possibly making it easy to switch to debug-level logs when needed. In debug mode, we might log more verbosely (since performance is not critical), whereas in production we log high-level milestones and warnings/errors. This helps in debugging issues and understanding the pipeline's behavior on real runs.

- **Example-rich Documentation:** The README will contain example usage, including how to run a quick test (maybe a short story provided as sample) and what output to expect. It also covers how to deploy to production (covering environment variables needed for GPU, etc.) [21]. We will also provide an architecture overview (similar to this write-up) in the docs so new contributors or users can grasp the system quickly.

In summary, the codebase is structured to be **approachable and modifiable**. By following clean code conventions (naming, structuring, and documentation), we ensure that this complex pipeline of agents remains understandable. This is crucial for an open-source project where others may join to use or improve the code.

## Enhanced Story Ingestion & NLP Optimization

Upfront processing of the story text can significantly improve the pipeline's efficiency and the quality of LLM outputs. We add NLP-based steps to the **Story Ingestion & Preparation** stage to ensure the LLM context is as small and relevant as possible:

- **Story Pre-Processing and Truncation:** The story ingestion component will sanitize and normalize the input text [32]. Beyond simple cleaning, we will **truncate or split the story if it's extremely long** to respect LLM context limits [32]. For example, if using a 8k-token context model and the story is a novel, we might break it into chapters and later handle each sequentially. The blueprint already noted chunking if needed [32]; we formalize that by potentially detecting natural breakpoints (chapters, scene headings in a screenplay, etc.) in advance.

- **Summarization for Context Reduction:** To handle very long stories, we may incorporate an NLP summarization step. For instance, use an open-source summarizer model (or even the same LLM in a summarization mode) to condense each chapter or the overall story into a shorter summary that still contains key plot points. This summary can then be fed to the Scene Breakdown Agent to determine scenes. By doing this, we ensure the LLM isn't overwhelmed with details that might not affect scene splitting. There is a trade-off (we might lose minor details), so this could be configurable – for a 60-minute video it might be necessary, but for a 5-minute short story maybe not. The context window management is important to keep LLM calls efficient and within limits [32].

- **Entity and Theme Extraction:** Another NLP enhancement is to extract a list of main characters, locations, and themes from the story at ingestion time. This could be done with simple NER (named entity recognition) or by prompting an LLM quickly. These extracted elements can then be provided to the Scene Breakdown prompt to help it organize scenes more intelligently ("the main characters are X, Y in Z setting"). It can also help later stages (e.g. ensuring consistent naming or visuals for characters).

By **doing more NLP work upfront**, we reduce the load on the Scene Breakdown LLM and guide it better. The result is a smaller context for the LLM and potentially more accurate scene boundaries.

## TOON Format for LLM Communication (JSON Alternative)

We explore using **TOON (Token-Oriented Object Notation)** as a more LLM-friendly output format instead of raw JSON. TOON is "a compact, human-readable encoding of the JSON data model for LLM prompts" [46]. In other words, it represents the same structures as JSON but with a syntax designed to use fewer tokens and be easier for language models to produce and follow reliably (it uses indentation and minimal quoting, among other features [47] [48]).

**Why TOON?** JSON, while structured, can be verbose with braces and quotes, which consumes token space and sometimes confuses LLMs (they might omit a quote or brace, yielding invalid JSON). TOON format, by contrast, has shown ~40% fewer tokens needed and slightly higher accuracy in staying valid [49]. For a pipeline like ours that relies on the LLM to output a structured scene list or shot list, using TOON could mean we get more information through the limited context window and with fewer formatting errors.

**Integration:** We would still define our Pydantic models (scenes, shots) in Python, but when prompting the LLM, instead of saying "output JSON with fields X, Y, Z", we say "output in TOON format" and perhaps give a TOON schema hint. For example, a TOON output for scenes might look like:

```
scenes [3]:
1:
  summary: Hero meets mentor in village...
  text: "...original text..."
2:
  summary: Villain attacks the village...
  text: "...original text..."
3:
  ...
```

This is equivalent to a JSON array of 3 scene objects, but without curly braces and quotes around keys, etc. The LLM might find this easier to produce correctly.

On the parsing side, we can use a TOON parser (there are libraries in Python) to convert the LLM's response back into a Python dict, which Pydantic can then validate as usual. If the LLM accidentally returns JSON anyway or makes small mistakes, our system should handle that gracefully (perhaps try to parse JSON if TOON parse fails, or vice versa).

**Adoption:** Since TOON is a relatively new format (designed for AI communication), we will make this an optional advanced feature. We can have a config flag like `USE_TOON_FORMAT=true/false`. If true, prompts will be adjusted to request TOON output, and the parser will expect TOON. If false (default), we stick to JSON (since it's more familiar). We will **experiment** with TOON to see if it improves the reliability of the LLM stages. The goal is to avoid chasing down missing commas or quotes in JSON by giving the model a format that's both concise and unambiguous in structure [50]. Early benchmarks suggest TOON can improve parsing reliability (e.g., fewer errors in structure) [47], which could reduce the need for retries in our pipeline.

By incorporating TOON, we stay on the cutting edge of prompt formatting for LLMs, potentially making the pipeline more token-efficient and robust. This complements the above NLP steps: we minimize input size and also minimize output size from the LLM, ensuring we can handle longer stories and more complex outputs within context limits.

## Debug Mode Execution Flow Example

To illustrate the interplay of these features, consider an example **MVP debug run** on a local CPU, using all the mock and debug facilities:

1. **Story Input:** The user provides a story (say a 5-page short story). The FastAPI endpoint receives it, and the orchestrator creates a `StoryInput` model. The text is preprocessed – perhaps truncated or lightly summarized – to fit the LLM context if needed [32]. Key entities are noted for later use.

2. **Scene Breakdown (LLM, debug):** The `SceneSplitterAgent` is invoked with the processed story. In debug mode, we might not call an actual large LLM. If a small local model is available (e.g. a 7B parameter model running in int4), we could use it; otherwise we could stub this by loading a pre-written scene breakdown (for repeatable testing). Let's assume we call a local LLM API in this case, but with a **reduced prompt** (maybe the story was summarized to a paragraph). The agent asks for, say, JSON/TOON output of scenes. The model returns a scene list (perhaps not very good in detail, but structured). Pydantic validates it. We now have e.g. 3 scenes with summaries.

3. **Shot Planning (LLM, debug, parallel):** The orchestrator now calls `ShotPlannerAgent` for each scene. Since we enabled `LLM_PARALLEL=true` in .env, it will dispatch these in parallel. Suppose each scene is just a few sentences; we again use either a small local model or a stub that returns a fixed set of shots (for deterministic tests). In our debug run, let's say we stub it: for each scene, rather than truly generating, the agent might load a template "shot list" from a file or generate 2 generic shots with placeholder descriptions ("Shot of characters talking", "Wide shot of setting"). This happens quickly and concurrently for the 3 scenes. We get back shot lists, each with a couple of shots, and assemble them into our full plan.

4. **Video Generation (mock):** Now for each shot, the `VideoGenerationAgent` should produce a video clip. In debug mode, our `MockVideoGenerator` kicks in. It checks `assets/mock/` for a file corresponding to this shot. Since this is the first run, none exist. So for each shot, it performs the placeholder generation:

5. It creates an image with the shot description text: e.g., for "Wide shot of village", we get an image with the words "Wide shot of village" over a gray background.
6. It then uses MoviePy to turn that image into a 1-second video clip (perhaps just a still frame video). It saves it as `assets/mock/scene1_shot1.mp4`, etc.

7. It returns the path of this clip. These steps repeat for each shot. If the same description appears again (unlikely in one run), it would reuse the cached asset. Now we have a list of video file paths, one per shot.

8. **Audio Generation (mock):**

9. For narration/dialogue: The `TTSAgent` sees we are in debug mode and uses `MockTTSGenerator`. Perhaps it simply writes the subtitle text to a `.txt` file instead of audio, or it generates a silent audio file of the correct duration. Let's assume it creates a silent WAV of 2 seconds for each scene (or maybe one per scene since our design might treat each scene's text as a block of narration). It saves these as `scene1_narration.wav`, etc.

10. For music: The `MusicAgent` in debug might just copy a royalty-free music loop from `assets/mock/` (say we have a file `assets/mock/bg_music.mp3`). It could also generate a trivial tone or just skip music. But we want to test the integration, so assume it picks a short music file and notes that it should loop.

11. **Video Composition:** The `VideoEditingAgent` now takes the list of shot videos, the narration audio files, and the background music. It stitches them in order:

12. Concatenate the 1-second shot clips in sequence (so the total video is a few seconds long).
13. Overlay the silent narration audio (essentially no audible sound, but it tests that the timing alignment works).
14. Add the background music loop underneath; if the video is longer than the music clip, it loops it quietly.
15. Add subtitles if any (in debug, we might not have generated a subtitle track, but if the TTSAgent produced a text, the editing agent could burn that in as subtitles for demonstration).
16. Apply a simple transition between scene boundaries (maybe fade to black between scene 1 and 2, etc., which in our short clip might just be a quick dip since each scene only had one shot).

17. The result is a composite video file, e.g. `output_video.mp4`.

18. **Output:** The orchestrator marks the job complete and returns the path or URL to `output_video.mp4`. The video will be very rough (just text cards with music), but the pipeline logic is validated end-to-end.

Throughout this flow, the system logs each step (e.g. "Generating shot 1 of scene 1 with MockVideoGenerator – saved to scene1_shot1.mp4"). If any step failed (exception or validation error), the orchestrator would catch it and report an error status, but in debug mode with controlled stubs, that's unlikely.

This MVP debug run accomplishes two things: **(a)** it confirms that the orchestration of agents works (data passes correctly from scene list to shots to video, etc.), and **(b)** it allows developers to see a tangible (if low-quality) output quickly. As the design intended, this can be done on a CPU-only machine in minutes [26], before committing to long GPU runs.

## Production Deployment Considerations

When switching to production mode (ExecutionProfile=PROD_GPU), we need to prepare the system to handle heavy workloads on a GPU host, such as a RunPod instance or any cloud VM with a suitable GPU. Key points for deployment:

- **Containerization:** We package the application as a Docker container [27]. This container includes all necessary libraries (CUDA, PyTorch, model inference code) and ideally the model weights. If the model files (for diffusion, TTS, etc.) make the image too large, we can mount a volume with those weights at runtime, but RunPod also allows large images, so bundling is fine for convenience [27]. We use a base image with NVIDIA drivers (e.g. `nvidia/cuda:11.8-cudnn8-runtime-ubuntu20.04` plus Python). The Dockerfile installs the production requirements and copies the code.

- **Environment Setup:** In the container, we set environment variables for production. For instance, `.env` inside the container would have `EXECUTION_PROFILE=PROD_GPU`, `GPU_ENABLED=true`, `LLM_PROVIDER=together` (if using an external API for LLM) and actual paths to model weights [51]. Secrets like API keys can be passed in securely via environment as well. We ensure that the container can see the GPU (using the `--gpus` flag when running Docker or the equivalent in RunPod).

- **Resource Management:** On a single GPU, the pipeline will likely run shots sequentially to avoid VRAM overload [33]. For example, generating one 5-second video might use 8 GB VRAM; doing two concurrently could exceed a 24 GB card. We plan the orchestrator to be mindful of memory: it could, for instance, generate each scene's shots one by one and free GPU memory in between (by clearing models from memory if not needed further). The blueprint suggested not running multiple heavy steps simultaneously on one GPU [33]. We can also configure the diffusion pipelines to run in lower precision (fp16) to save VRAM, and unload models when done (e.g., load the TTS model only when doing TTS, then unload).

- **Job Scheduling:** If deploying on a service like RunPod, one common strategy is **one job per container** – i.e., each container handles one video generation task at a time [17]. This avoids contention. If we need to handle multiple user requests concurrently, we would scale out horizontally (multiple containers) rather than running them in parallel on one container. This is aligned with the design's suggestion to use multiple pods for concurrency if needed [33]. We will likely run the FastAPI app inside the container, and each request triggers the orchestrator. We might integrate a simple queue if we expect overlapping requests, or just rely on FastAPI's request handling (maybe with `workers/` threads as backup [17]). For an MVP, one container = one request at a time is acceptable.

- **RunPod Specifics:** RunPod allows specifying machine type (GPU model) and then you can upload your image or Dockerfile. We would test on an RTX 3090 or A100 (as mentioned in original design) [37] to ensure our models fit in memory and that performance is acceptable. We should also handle GPU initialization time (some models take a while to load into VRAM). Possibly, we keep the models loaded for the duration of the container's life to reuse them if multiple jobs are run sequentially on the same pod.

- **Monitoring and Logging:** In production, we should monitor GPU usage and memory. We might integrate something like an API to report status (the `/status` endpoint can tell if a job is in progress and maybe progress percentage). Logging should be set to info level, and important events (start of generation, start of each agent, etc.) can be logged. If using RunPod, their platform might provide some logging and we can also push logs to a volume or stdout for capture.

- **Failure Handling:** If a model fails mid-generation (e.g., diffusion pipeline runs OOM), the orchestrator should catch this, possibly try to reduce load (like generate a lower-resolution video as a fallback), or mark the job failed gracefully. We can also set timeouts on LLM calls etc. to avoid hanging. These resilience measures are important for a long-running job in production.

- **Security:** Since this is an open-source project deployed on a server, we ensure that the FastAPI endpoints are secured (at least with a simple API key or token, to prevent misuse). Also, since we allow arbitrary story input which could be large, we might put a limit on input size to prevent someone from overloading the service.

Deploying on a service like RunPod essentially means our updated pipeline can run “in the cloud” with full GPU power, generating the high-quality videos as intended. All the advanced features – ExecutionProfile, GPU detection, etc. – make it easy to deploy the same codebase in both environments. Locally, it will detect no GPU and run in debug mode. In RunPod, it finds the GPU and runs production mode, or we explicitly set the env to PROD. This duality was a core requirement and now is fully realized [22] [37].

---

**Sources:**

- AI Cinematic Video Generation Pipeline Design (Plan A) [1] [2] [43] [23] [26] [24] [27] [33] [39] [38] [29] [5] [7] [10] [12] [15] [16] [18] [21]
- TOON Format – Token-Oriented Object Notation (for LLM prompts) [46] [47] [48]

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [51] research-1.pdf
file://file_00000000946c71f48d5cb426d6bcdb2d

[46] [47] [48] [49] [50] TOON | Token-Oriented Object Notation
https://toonformat.dev/