ChatGPT

Integration Spec: Short-Form Video Reverse-Engineering Pipeline (VideoSpec v1)

Overview

This specification describes a **modular reverse-engineering pipeline** for short-form videos (TikTok, Reels, Shorts) that extracts rich structured metadata from input videos. The pipeline analyzes five key aspects – **timeline structure**, **editing techniques**, **audio structure**, **characters/entities**, and **narrative arc** – and outputs a consolidated JSON report (`VideoSpec.v1.json`). The design emphasizes:

- **Comprehensive Metadata Extraction:** Identify shots/cuts, visual effects (overlays, zooms, captions), audio elements (dialogue, music, SFX, beat), on-screen entities (faces, objects, speakers), and narrative flow (hook → escalation → payoff).
- **Canonical JSON Schema:** A well-defined, versioned schema (v1) for all extracted data, enabling easy integration with content-machine's existing tools.
- **Modularity & Fault Tolerance:** Decoupled analysis modules with caching and error-handling, allowing independent development and robust retries without losing intermediate results.
- **Local-First with Extensible APIs:** Default to local inference (open-source libraries/models) for offline use [1] [2], but allow swapping in external AI services (e.g. Google Gemini multi-modal APIs, Azure Video Indexer) for enhanced accuracy or cloud scalability [3] [4].
- **Two-Pass Processing:** A fast first pass for quick insights (using lightweight models or downsampled data), followed by an optional refinement pass for high-precision results.
- **Timebase Normalization:** All modules share a unified timeline reference (e.g. seconds from video start) to ensure outputs from different analyses align correctly.
- **Provenance & Reproducibility:** Every piece of metadata is traceable to its source tool/model version with confidence scores where applicable, facilitating licensing compliance and result replication across runs.
- **Downstream Integration:** The extracted `VideoSpec` can feed into content-machine's **templating** systems (to auto-generate new videos mimicking analyzed structure) and **analytics** modules (to correlate creative patterns with performance).

Architecture & Design

Modular Pipeline and Orchestration

The pipeline is composed of discrete stages that can be orchestrated via content-machine's workflow infrastructure (e.g. Temporal workflows or BullMQ job queues) [5] [6]. Each stage focuses on a specific analysis task, reading from the video and producing structured data. A controller (workflow or CLI

command) invokes the stages in sequence, or in parallel where possible, and finally aggregates their outputs into the unified JSON spec. Key stages include:

1. **Video Ingestion & Preprocessing:** Load the video file/URL, retrieve metadata (duration, resolution, frame rate) and ensure consistent decoding. Possibly generate a lower-resolution or lower-FPS proxy for quick scanning in pass 1.
2. **Shot Boundary Detection:** Detect shot changes and transitions (populates timeline structure). This stage provides a backbone temporal segmentation for other analyses.
3. **Visual Analysis:** Examine each segment/frame for editing effects and on-screen elements (overlays, text, camera motions, faces, objects).
4. **Audio Analysis:** Process the audio track for speech transcription, music detection, sound events, and rhythmic analysis.
5. **Entity Linking:** Cross-link the audio and visual findings – e.g. associate speaker identities from audio with detected faces on video – to build holistic "characters."
6. **Narrative Analysis:** Using outputs from earlier stages (dialogue transcript, pacing, etc.), infer the narrative arc and engagement tactics.
7. **Result Synthesis:** Compile all data into the `VideoSpec.v1.json` structure, including metadata and provenance info.

This modular design allows each component to be developed/tested in isolation and even replaced with alternative implementations. The orchestration layer will manage dependencies (e.g. ensure shot detection runs before face tracking, since shot boundaries help reset tracking) and can run independent modules concurrently when feasible (for example, audio transcription can run in parallel with visual overlay detection to shorten total runtime). We leverage content-machine's robust job handling – e.g. **Temporal** for long-running durable workflows and **BullMQ** for scalable task queues – to ensure reliability (automatic retries, timeouts, etc.) [6] [7]. For instance, a **Temporal workflow** could coordinate the end-to-end analysis, calling each module as an activity with automatic retries on failure [8] [9], or a Node-based **BullMQ** queue could manage each stage as a job with dependency chaining [10] [11]. This ensures the pipeline can recover from transient errors (e.g. an OCR crash or model loading failure) and continue where it left off, rather than restarting the whole process.

Caching & Fault Tolerance

Each module caches its intermediate results to avoid re-computation and to provide fault tolerance. For example:

- **Transcript Cache:** After ASR (speech-to-text), save the transcript (e.g. as an SRT or JSON) keyed by a video content hash or ID. If the pipeline is re-run on the same video, or if a later stage fails and needs a rerun, the cached transcript can be loaded instead of recomputing expensive ASR.
- **Frame Feature Cache:** Frames or features extracted for one module (e.g. key frame images used for OCR or object detection) are stored so they can be reused by other modules or on refinement pass. For instance, if we sample one frame per second for visual analysis in pass 1, those frames and any detection results can be reused or refined in pass 2.
- **Module Output Versioning:** The cache keys include the module's version (or a hash of its config) to prevent stale data use after upgrades. If we improve the face detector or adjust scene detection thresholds, the pipeline can detect a version mismatch and recompute that stage.

The system is **fault-tolerant**: each stage runs in an isolated manner (e.g. separate process or container if needed), and errors are caught and logged with context. A failure in one module does not crash the entire pipeline – the orchestrator can either **fallback** to an alternate implementation or **skip** that module's data in the final spec with a warning. For example, if local OCR consistently fails on a stylized caption, the pipeline could call a cloud OCR API as a fallback (if configured) or mark the `captions` field as incomplete. Similarly, if the face tracking module raises an out-of-memory error on a large video, the system might retry it on a scaled-down version or simply proceed without face data, flagging the omission. All such decisions are recorded in the provenance notes. This approach ensures we get partial results rather than none – vital for long videos or when certain analyses are non-critical.

Two-Pass Analysis Strategy

To balance speed and detail, the pipeline supports a **two-pass analysis**:

- **Pass 1 (Fast Proxy Analysis):** Quickly analyze the video to obtain essential structure and low-resolution insights. This may involve using faster, lightweight configurations: e.g. run shot detection on a downsampled video or with a high threshold (fewer but likely cuts), run speech transcription with a smaller ASR model (for speed) or even just detect language and keywords, and perform cursory face detection on key frames. The goal is to produce a rough draft of `VideoSpec` rapidly (perhaps in seconds for a short video), which can be used for immediate needs (like indexing or previewing results). This pass could skip computationally heavy steps like full OCR of every frame or detailed object tracking, and instead sample sparsely.

- **Pass 2 (Refinement Analysis):** Using the coarse information from Pass 1, perform deeper analysis focusing where needed. For instance, once we know approximate scene boundaries and maybe segments of high activity, we can allocate more resources there: run a high-accuracy ASR (e.g. larger Whisper model) on the audio to get exact transcripts [12], perform OCR on frames where speech was detected to extract precise captions, and track faces throughout each shot now that we know how many distinct faces exist. The second pass can also use results from the first pass to **inform processing** – e.g. knowing the video's overall pace and narrative structure can help an LLM focus on the right segments for narrative analysis. This pass produces the final high-detail metadata, correcting any mistakes from the first pass (for example, adding a missed short shot or refining the timing of a transition).

This two-tier approach is optional – for high-throughput scenarios or when using powerful cloud APIs, one might directly do a single comprehensive pass. However, it provides flexibility: content-machine can quickly surface preliminary analytics (after pass 1) and then update them when pass 2 finishes. All outputs are in the same schema, with perhaps a flag indicating if some fields are preliminary. The pipeline ensures consistency between passes by normalizing timecodes and merging results (e.g. aligning the rough cut list from pass 1 with the exact cut times from pass 2). Any differences are reconciled (e.g. if pass 2 finds an extra shot boundary, it inserts it). The orchestration can run pass 1 and 2 as part of one workflow or allow an on-demand refinement trigger.

Timebase Normalization

All analysis components use a **common timebase** so that their outputs can be merged and compared seamlessly. We standardize on **video timestamp in seconds** (as float or millisecond precision) from the

start of the video (0 at first frame) for all temporal data. Internally, modules may use frame counts or sample indices, but they must convert to absolute seconds for output. The video's exact frame rate is obtained at ingestion (e.g. via ffprobe) and used to translate frame numbers to timestamps accurately. We also account for any variable frame rate or timestamp discontinuities by referencing presentation timestamps from the video container if needed, so that e.g. 10.000s in transcript matches 10.000s in scene cuts.

By enforcing this normalization, we can **align multimodal events** easily – a word from the transcript at `12.34s` can be correlated with a shot boundary at `12.3s` to see it occurs at the shot start, or a face appearance from `5.0–7.0s` overlaps with the second scene. Each module's output is post-processed to this uniform timeline. For example, if a scene detector outputs frames `[250, 400]` for a shot in a 30fps video, we convert that to `{start: 8.33, end: 13.33}` seconds. If OCR yields a caption appearing from frame 100 to 150, we log it as appearing at `3.33–5.0s`. All timings in the JSON are in seconds (floating-point) relative to video start unless otherwise noted (the spec can include a note that times are in seconds). This **consistent timebase** is critical because different tools have different time references (some ASR tools might give times relative to audio segment start, etc.) – we always convert to absolute video time. We also include the video's FPS and duration in metadata for reference so that frame-accurate mapping can be done if needed.

Licensing, Reproducibility & Provenance

Licensing: We prioritize tools and models with permissive licenses (MIT, BSD, Apache) to avoid legal constraints on content-machine's distribution. For instance, we use **PySceneDetect** for shot detection (BSD-3 license) [1], **OpenAI Whisper** for transcription (MIT license) [12], and various MIT/Apache licensed models for OCR and detection [13] [14]. All default components are open-source. If a module with a restrictive license (e.g. GPL) is the only available option for a particular task, it will be isolated (run as an external process or microservice) so that its license does not infect the rest of the codebase. The spec's provenance data will clearly note any such component. Optional cloud services (Gemini, Azure, etc.) have their own terms of use but since they run externally via API, they don't affect the pipeline's code license – they're used only when configured by the end user.

Reproducibility: To ensure results can be replicated, we **pin versions** of models and tools. The pipeline will use specific versions of dependencies (e.g. a particular PySceneDetect version and detection threshold, a specific Whisper model checkpoint) and record these in the output. The `VideoSpec` includes a `provenance` section (detailed later) listing module versions and model identifiers. This means that given the same video file and the same pipeline version, one should obtain the same JSON output (subject to nondeterministic factors like floating-point variations or if an ML model has randomness – in which case we set seeds where possible). If using LLMs for narrative analysis, we mitigate their nondeterminism by using fixed prompts and zero-temperature settings to make responses as consistent as possible. The pipeline itself is versioned (this spec is v1); future improvements will bump the spec version and ideally provide backward compatibility or conversion tools for older outputs.

Provenance Tracking: Each data element in the JSON can be traced back to its source. We achieve this by both **module-level attribution** and **item-level confidences**:

- At module level, the spec lists what tool or model was used for that module, including version or model signature. For example: `"scene_detection": "PySceneDetect v0.6`

`(ContentDetector)"` or `"speech_recognition": "Whisper-large v2"`. For external APIs, we include the service name and any model/preset info (e.g. `"face_detection": "Azure Video Indexer (API ver.2025-11)"`).

- At item level, when applicable, we attach **confidence scores** or other quality metrics. For instance, each transcribed speech segment might include a confidence from the ASR model; each detected face or object comes with a detection confidence score from the model. These give a sense of reliability for each piece of data. If a module does not naturally provide a confidence, we may compute a proxy (e.g. for scene cuts from PySceneDetect, we could output the difference metric that triggered the cut if available).
- The provenance also notes any **post-processing** or manual steps. If an analyst or a secondary process had to adjust something (for example, if an LLM was used to label the narrative segments), we mention that (e.g. `"narrative_analysis": "OpenAI GPT-4 via prompt X.Y, temperature 0"`). All timestamps and results remain machine-generated unless explicitly flagged.

This metadata is invaluable for auditing and updating the pipeline. If a user questions a result (say a face was mis-identified), one can see which model was responsible and update or retrain it. Provenance also helps with **provenance of content**: if we incorporate any reference data (for example, if we used a database of known faces for recognition, we would list that dataset source to comply with attribution). In summary, the pipeline treats the *analysis process* itself as a first-class output, documenting how it arrived at the given metadata.

Module Implementation Options & Outputs

For each functional module, multiple implementation options are available. Below we outline each module's responsibilities, the possible tools/algorithms (local and external) for implementation, and how the module's output is structured in the `VideoSpec.v1.json`. **Recommended defaults** are indicated for the typical content-machine setup (favoring local, open-source solutions). Each module's output schema is later consolidated in the **VideoSpec Schema** section.

Timeline Structure: Shots, Transitions & Pacing

This module detects the video's **shot boundaries** (cut points between clips or scenes), identifies the type of transition between shots, and computes pacing metrics. It defines the temporal skeleton of the video.

- **Shot Detection:** We detect all cuts or scene changes in the video. The recommended default tool is **PySceneDetect** (Python) using a content-aware detector [1]. PySceneDetect offers multiple algorithms – **ContentDetector** for detecting abrupt cuts via histogram differences, **AdaptiveDetector** for detecting cuts with camera motion (two-pass analysis), and **ThresholdDetector** for gradual fade-in/out [15] [16]. By combining these (e.g. run ContentDetector first, then Adaptive or Threshold for missed edits), we achieve high recall of cuts. PySceneDetect is fast and production-proven [17]. Alternative options include:
- *OpenCV/FFmpeg-based:* Using FFmpeg's scene detection filter (e.g. `-vf "select='gt(scene, 0.4)',showinfo"` to detect cuts by frame difference) or OpenCV manually computing differences between frames. These can find hard cuts quickly but may miss subtle transitions and lack built-in timestamp logic.
- *Cloud API:* Using **Azure Video Indexer** or **Google Cloud Video Intelligence**, which have shot segmentation in their video analysis presets. For example, Azure's Video Indexer "basic" preset

automatically provides *scene and shot detection* metadata [3]. This can be very accurate and also sometimes differentiates "scenes" (groups of shots by location) vs. raw cuts. The trade-off is API cost and latency, so this is optional unless cloud integration is desired.

Output: The pipeline produces a list of **shots**, each with start and end time (seconds). In `VideoSpec.json`, this is an array under `timeline.shots`. Each shot may also include an ID and possibly descriptive tags. We also capture the **transition type** leading into the shot (for the first shot, no transition). If using PySceneDetect's ThresholdDetector, we can label a transition as `"fade"` or `"dissolve"` when a gradual change is detected, otherwise `"cut"` for abrupt cuts [16]. If more exotic transitions (wipes, swipes) are detected (not inherently identified by PySceneDetect), we could attempt to classify them by analyzing intermediate frames, but initially we categorize basic types (cut or fade). Transitions can either be listed as a separate array `timeline.transitions` (with each entry like `{time: 12.34, type: "cut"}`) or as properties of the shot that begins at that transition. In v1, we opt to include them in the shot list for clarity – e.g. each shot entry might look like `{ "id": 3, "start": 10.0, "end": 15.2, "transition_in": "fade" }`.

- **Pacing Metrics:** With shot timings known, we compute summary statistics to characterize pacing. These include: total number of shots, average shot duration, median and min/max shot length, and perhaps the cut frequency (cuts per minute). We may also flag rapid-cut sequences (e.g. "montage from 0:30–0:35 with 5 cuts in 5 seconds"). The pipeline can classify the overall pace as `"fast"` / `"moderate"` / `"slow"` by comparing metrics to typical values (for example, an average shot length <2s would be extremely fast-paced). These metrics are stored under `timeline.pacing`. For example: `{ "shot_count": 20, "avg_shot_duration": 1.5, "median_shot_duration": 1.0, "fastest_cut": 0.3, "classification": "fast" }`. Such data is useful for analytics – e.g. to correlate with engagement, or to drive editing style in content generation.

Why PySceneDetect: It's open-source (BSD) and content-machine has already vetted it as the "champion" tool for scene detection [1]. It handles different cut types and even integrates with splitting video files if needed. Using PySceneDetect in Python within content-machine (which is Node/TS-based) can be done via a Python subprocess or a small service. Its accuracy and speed on short videos are excellent (a 60s video is processed in a few hundred ms to seconds). The alternative, Azure Video Indexer, is powerful but introduces external dependency and cost; we keep it as a backup or optional module for users who want cloud-augmented accuracy. Both approaches are version-consistent and will yield nearly the same shot timestamps (differences typically within a frame or two). The pipeline can even cross-verify (e.g. run PySceneDetect and compare with Azure's output if both available, to validate or catch any misses).

Visual Editing Techniques: Overlays, Zooms & Captions

This module inspects the visual frames to identify post-production editing elements: on-screen graphics/text overlays (especially captions), camera motions like zooms or pans, and editing styles like jump cuts. These are key to understanding how the video was crafted to engage viewers.

- **Text Overlays & Captions:** Many short videos use on-screen text – subtitles for dialogue, title cards, or stickers. We use **Optical Character Recognition (OCR)** to extract any visible text in the video frames. The pipeline focuses especially on **captions** (text corresponding to spoken dialogue) and distinguishes them from other text overlays.

OCR Implementation: Our default approach is to use a combination of **Tesseract OCR** (via the Node.js `tesseract.js` library) and more advanced OCR as needed. Tesseract.js (Apache 2.0 license) runs entirely locally in Node and is efficient for high-contrast, large-font text like TikTok captions [18]. In content-machine's research, TikTok-style subtitles (typically white text with black outline) were found to be handled "sufficiently accurately" by Tesseract with proper preprocessing [19]. We apply optimizations such as: crop to the likely caption region (e.g. bottom part of frame where subtitles usually appear) and preprocess (grayscale, threshold to black-and-white, maybe upscaling small text) before OCR [20]. This yields faster and more accurate results (often <1s per frame for 1080p after cropping) [21] [22].

However, Tesseract can struggle with stylized or animated text. For cases where it fails (e.g. fancy fonts, curved text, low contrast), we integrate a **fallback**: Python-based **EasyOCR** or **PaddleOCR** (both Apache 2.0) can be called to handle tougher cases [23] [24]. EasyOCR, for instance, has higher accuracy on styled text and supports GPU for speed [25]. Content-machine's plan is to call a Python script for EasyOCR only when needed (to avoid always incurring the Python startup cost) [26]. This two-tier OCR strategy ensures we cover most text. (Alternatively, one could use **Azure Video Indexer's OCR** capabilities: it automatically extracts text that appears in video frames [27], consolidating it into searchable strings. This is a good external option if cloud is enabled, providing language detection and possibly better multi-language support out-of-the-box. But it may not catch rapid or stylized text as reliably in short meme-style videos, and it requires a paid account and internet connection.)

Caption vs Overlay: Once we have text instances and their timestamps, we differentiate **captions (subtitles)** from other text. We do this by comparing the detected text to the spoken dialogue transcript. The pipeline will align OCR text timings with the ASR transcript timings: if an OCR text string closely matches the words being spoken at that moment (allowing minor phrasing differences), we label it as a **caption**. For example, if at 10.0–12.0s the OCR finds "This is awesome" and the transcript has the speaker saying "This is awesome" at that time, that overlay is a caption. Captions often appear in sync with speech and in sequence. We group consecutive caption text into subtitle lines if needed (some editors break one sentence into multiple on-screen lines; we'll keep it aggregated per timestamp range). Text that does not align to any dialogue (e.g. a username watermark, or a label like "Step 1") is classified as a **non-dialogue overlay**. It might be a title, hashtag, or other on-screen annotation.

Output: In `VideoSpec.v1.json`, we will have an `editing.captions` array and an `editing.text_overlays` array. - **Captions:** Each entry includes the text, the time interval on screen, and possibly a reference to the speaker (if we know who said it). For example: `{ "text": "This is awesome!", "start": 10.0, "end": 11.5, "speaker": "Person1" }`. We retain capitalization/punctuation as detected, and we can also include a confidence for the OCR if available. The caption entries should appear in chronological order.
- **Text Overlays:** For non-speech text, each entry will have the text and its time interval. We may also include positional information (e.g. region on screen like "top-left" or bounding box coordinates) especially if needed for downstream templating (to place new overlay text in the same spot). For example: `{ "text": "#DIYTips", "start": 0.0, "end": 3.0, "position": "top_center" }`. If the overlay is an image or sticker rather than text, we might describe it, e.g. `{ "image": "heart_sticker.png", "start":5.2, "end":7.0, "position": "bottom_right" }` (for v1, we focus on text, but the schema can accommodate an "image overlay" with perhaps an identifier if recognized).

Both captions and overlays are versioned with OCR engine info in provenance (e.g. it might note "captions extracted with Tesseract.js, verified by EasyOCR for 2 frames"). We also store multi-language text if found (TikTok captions could be bilingual; we'd capture both lines).

- **Visual Overlays & Filters:** Beyond text, short videos often have graphical elements: e.g. progress bars, logos, or AR effects (like emojis or face filters). Detecting arbitrary overlays is challenging, but we make some attempts:
- We include a check for known platform watermarks (for instance, a TikTok logo that bounces at corners). Using template matching, we can spot the TikTok or Instagram logo and note it as an overlay (this can inform where the video was likely sourced from originally). This is a simple image recognition task and if found, we'll mark in the spec under `editing.overlays` something like `{ "type": "watermark", "text": "TikTok", "start": 0.0, "end": 60.0 }`.

- For other graphics, we rely on object detection (addressed below in Entities) to some extent: e.g. if a big emoji or sticker is present, a general object detector might label it ("smiley face" or "text box"). However, to keep things focused, the v1 pipeline does not deeply analyze filter effects (like color filters or AR masks). Those could be inferred by comparing frames to detect unnatural augmentations (e.g. if a dog ears filter is on a face, an object detector could see "dog" features on a face – but that's complex). We note it as a possible extension. For now, overlays are mainly text and obvious graphics.

- **Camera Movements (Zooms/Pans):** We analyze each shot for **editing-induced camera motions** like zooms, pans, or rotations. In short-form content, creators often add a slow zoom-in on a static shot to add dynamism, or quickly punch-in (jump zoom) for emphasis. To detect a **zoom**, we can compare the scale of the image over time. One method: use feature tracking or optical flow – e.g. track several visual keypoints in a shot; if they all move radially outward from the center consistently over many frames, that indicates a zoom-in (the frame is enlarging) as opposed to a physical camera motion. We can also use the motion vectors from video encoding (if accessible via ffmpeg) or apply a global motion estimation algorithm (like computing an affine transform between consecutive frames). If the affine transform has a scaling component >1 (for zoom-in) or <1 (zoom-out) consistently, we log that. Another simpler heuristic: measure if the borders of the scene are cropping – e.g. if objects at edges gradually disappear, likely a digital zoom-in.

Similarly, a **pan** or tilt can be detected if the global motion is primarily translational (left-right or up-down). If feature points shift in one direction uniformly, that's a pan. **Rotation** (less common in these videos) could be detected via oriented FAST features or so, but likely not needed often. We focus on zooms because they are explicitly mentioned and quite common in edits.

This detection might be done per shot: after segmenting shots, for each shot we estimate if the shot is **static** (no significant camera movement or zoom), **zooming in/out**, or **panning/tilting**. We could utilize existing algorithms (OpenCV's `VideoStab` module does some estimation of global transforms for stabilization – we could repurpose that to classify motion type).

Output: Under `editing.camera_motion` (or as part of each shot entry), we list any detected camera moves. For example: `{ "shot_id": 5, "motion": "zoom_in", "start": 24.0, "end": 30.0, "speed": "slow" }`. If the entire shot has a uniform slow zoom-in, we mark the whole interval. If the zoom happens only in part of the shot (e.g. they cut to a closer scale midway – which is actually a cut not a

continuous zoom, or a ramp where zoom speeds up), we can break it down further or just approximate. Pans/tilts would be similarly noted: e.g. `"motion": "pan_left"`. If no significant movement, we might label it "static". This information is useful for templating – e.g. if a scene is identified as having a slow zoom, a generated video can replicate that for effect.

Also, **jump cuts** (which are an editing technique where consecutive shots have the same subject with only a slight change, creating a jarring jump) can be inferred at this stage. If two adjacent shots from the timeline analysis have very similar visual content (say, same background/person) but were cut (likely some content was removed in between), we flag that as a **jump cut**. Implementation: compare the last frame of shot N and first frame of shot N+1 using a similarity measure (histogram or perceptual hash). If similarity is above a threshold yet a cut was detected, it's a jump cut. We could annotate the transition entry with `"jump_cut": true`. This is common in vlog-style shorts (where a speaker cuts out pauses). For v1, we treat jump cut identification as a nice-to-have: we will mark obvious cases and include a boolean or note in the transition metadata.

- **Other Editing Techniques:** We remain aware of other possible techniques even if not explicitly listed in the prompt, such as **speed changes (slow motion or time-lapse)** or **freeze frames**. These can be inferred by audio or motion mismatch (e.g. if audio goes silent and video repeats a frame – freeze frame – or if audio stays normal but video action slows – slow motion). While v1 might not systematically detect these, the architecture allows adding such detectors. For now, we assume most short videos play at normal speed, except if the narrative analysis or audio analysis flags something (like a music beat mismatch indicating a slow-mo). We include a placeholder in the schema for `editing.speed_changes` if needed (list segments that are 0.5x or 2x, etc., if any). Provenance notes would mention we haven't implemented that in v1 if not.

Audio Structure: Dialogue, Music, SFX & Beat

This module breaks down the audio track into its constituent components: spoken dialogue (and its transcript), music/background audio, sound effects or other audio events, and the rhythmic timing (beat) if music is present. It heavily relies on AI models for speech recognition and audio classification.

- **Dialogue Transcription (ASR):** To get the spoken content, we perform **Automatic Speech Recognition** on the audio. The recommended default is **OpenAI's Whisper** model, running locally if GPU is available. Whisper has state-of-the-art accuracy on many languages and is robust to noisy social media audio [12]. We can use a medium or large model for high accuracy (Whisper-large yields very accurate transcripts on clear speech) [12]. If running Whisper locally is too slow, an alternative is to call an external service: e.g. **Azure Cognitive Services Speech-to-Text** or **Google Cloud Speech API** – these are cloud solutions that can transcribe with high accuracy and are scaled, at the cost of usage fees. Another alternative is **Paraformer (FunASR)** models from Alibaba, which content-machine has noted as "industrial-grade" for Chinese/English [28]. In fact, FunClip (Alibaba's tool) integrates Paraformer ASR with great success [29]. So, for bilingual or Chinese content, we could switch to that model to improve accuracy [28]. The pipeline architecture allows choosing the ASR model based on video language (language detection could be a preliminary step – Whisper itself can auto-detect language).

We run the ASR over the entire audio (preferably in one go to get a global transcript, since short videos are only up to 60–180 seconds). Whisper will produce a time-stamped transcript – a sequence of segments with start/end times and text. We refine these segments by merging or splitting if needed to align with sentence

boundaries or breath pauses. If speaker diarization is enabled (next bullet), we will also assign speaker labels to each segment or even do a joint diarization+ASR approach (Whisper doesn't do diarization by itself). For now, we treat transcription and diarization separately: get the raw transcript first.

Output: The transcript is stored as structured data under `audio.transcript`. We do not simply dump a text blob; instead, we keep it as an array of spoken segments (utterances), each with start time, end time, speaker label (if identified), and the text. For example:

```
"transcript": [
    { "start": 1.2, "end": 3.5, "speaker": "Person1", "text": "Hey everyone," },
    { "start": 3.5, "end": 6.0, "speaker": "Person1", "text": "welcome back to
my channel." }
]
```

If we haven't done speaker labeling, `speaker` could be null or omitted, and we treat it as a single speaker by default. We ensure that punctuation and casing are preserved reasonably (Whisper provides punctuated text). If needed, we could also include word-level timing (Whisper large can output word timestamps, or we can align text after the fact). For now, segment-level timing suffices for our purposes (word-level detail could blow up the JSON). The entire transcript array gives the dialogue content, which is vital for narrative analysis and also useful for search/indexing.

- **Speaker Diarization (Who is speaking):** If the video contains multiple speakers (say an interview or a skit with two voices), we enable diarization to attribute segments to different speakers. We can use a tool like **pyannote.audio** (which has pretrained diarization pipelines) or FunClip's integrated **CAM++ speaker recognition** [28] [30]. A practical approach: use the Whisper transcript segments combined with a speaker embedding model – e.g. take each segment's audio, compute a speaker embedding (using a model like ECAPA-TDNN or similar), cluster those embeddings to identify distinct voices. This yields labels like "Speaker0" and "Speaker1". We then assign these speaker IDs to the transcript segments in chronological order.

We also refine segment boundaries if needed to not mix speakers in one segment (if an ASR segment accidentally had two speakers because they overlapped or switched quickly, diarization would tell us and we could split it). Given short-form content often has one main speaker (the creator) and possibly some secondary voices (or none), diarization might usually find just one cluster. In cases of voice-overs or multi-person videos, it's valuable.

Output: The transcript segments get a `speaker` field as shown above. Additionally, in the `entities` section (discussed later), we will create entries for each speaker (Person1, Person2, etc.) and link them to faces if possible. For now, the audio module just ensures the speaker labels exist in the transcript. Optionally, we may include a separate summary in `audio.speakers` like:

```
"speakers": [
  { "id": "Person1", "description": "Main narrator (female voice)", "segments":
[ {start:..., end:...}, ... ] },
```

```
  { "id": "Person2", "description": "Guest (male voice)", "segments": [ ... ] }
]
```

That might be redundant with the transcript, so we might skip a separate list in v1 and rely on the unified `entities.characters` for a cross-modal representation of people (see Entities section). The key is that "who spoke when" is captured. If the pipeline cannot confidently diarize (e.g. voices overlap or audio is poor), it may default all to one speaker and note uncertainty. The provenance will note the diarization method and its cluster output (with maybe a silhouette score or similar as confidence).

- **Music Detection:** Short videos frequently use background music or a trending sound. We detect if **music is present**, and if so, where. This can be treated as an audio segmentation problem: classify segments of the audio as speech, music, or silence/noise. We can leverage existing audio classification models or pipelines:
- **inaSpeechSegmenter** (an open-source toolkit by INA) can segment audio into speech, music, noise, etc., and is optimized for broadcast audio. It would tell us time ranges that are music vs speech with pretty good accuracy.
- **YAMNet or VGGish**: These are general audio classifiers (trained on AudioSet) which can detect the presence of music and even the genre or type of sound, but usually on short windows. We could slide a window (e.g. 1s long) across the audio and see if "music" class probability is high to build segments.
- Simpler: use the ASR transcript – wherever ASR did not detect speech but audio is not silent, could be music. Also, music often coexists under speech (background music under narration). We might need to separate that: we could run a **voice activity detector** to find all regions with voice. The inverse of that (excluding pure silence) can be considered music or SFX.

We aim to produce segments of background music. Many short videos have a continuous backing track from start to finish. In that case, one segment "music: 0 to 60s" with maybe a lower volume during speech is fine. Sometimes music only starts after an intro or stops for a dialogue moment. We reflect those changes.

If feasible, we also identify the **track or melody**. Identifying the exact song (e.g. the name of a pop song used) requires either an audio fingerprinting integration (like using Shazam or Echoprint API) or checking metadata if available (TikTok's API sometimes can tell the track name if we had that context, but with just the video file we don't have it directly). In v1, we likely won't identify the song name (licensing issues as well), but we'll note presence. However, we will store an acoustic fingerprint (like Chromaprint) or a short audio sample in the cache so that if needed, one could later resolve the song via an external service.

Output: `audio.music_segments` will list time intervals where music is present. E.g.

```
"music_segments": [
  { "start": 0.0, "end": 60.0, "track": null, "description": "Background music
(pop song)", "confidence": 0.95 }
]
```

In this example, music plays throughout. If there are multiple distinct music pieces (maybe a video switches song at some point, or stops music then resumes), we'd list each segment. The `track` field could be filled with an identifier if we found one (e.g. track ID or name), otherwise null. The description might include

characteristics (if our classifier can guess genre like “rock instrumental” or mood). Confidence is how sure we are it’s music vs not.

We also note if the music is **foreground or background**. Usually, if someone is speaking, the music is background (lower volume). If no speech, the music might be foreground. This could be inferred by relative volume or just by overlap with speech segments. Possibly include a boolean `background: true/false`. This is useful because background music might be treated differently in analysis than a video that is basically a music video.

- **Sound Effects & Audio Events:** Beyond speech and music, short videos often include various sound effects or unique audio cues (e.g. a “whoosh” for a transition, laughter, applause, a car honk, etc.). We aim to capture notable **sound events** that could be relevant to content understanding:
- We can use a classifier model (like YAMNet) that has many classes of sounds (it can recognize things like “Applause”, “Laughter”, “Bark”, “Explosion” with moderate accuracy). The pipeline can scan the audio and whenever a non-speech, non-music sound occurs, attempt classification. For example, if the video includes a crowd cheer sound, the model might label that segment as “Applause”. We would then log an event at that time.
- Another approach: leverage the **transcript** – sometimes ASR or manual transcription will include tags like “[LAUGH]” if it’s obvious, but Whisper doesn’t do that. Alternatively, the content creator might put a caption “*laughs*” which our OCR would catch as text. If such a caption appears, we could infer a laughter sound. But relying on that is inconsistent.
- We could also pick up **peaks** in the waveform that are not speech (like a sudden loud bang). A simple method is to subtract the estimated speech and music signals (if we had them separated via an algorithm like Spleeter or demux from source if we had stems) and analyze residual sounds. However, that might be overkill.

We focus on distinctive, intentional SFX: e.g. the classic “boom” sound effect, or a comedic “record scratch”. Many of these could be part of the background audio track in editing. We attempt to label a few broad categories: **Laughter/Crowd**, **Transition Woosh**, **Explosion/Impact**, etc., if detected by the classifier with high confidence.

Output: `audio.sound_effects` will be an array of sound event objects. Each has a timestamp (or short interval), a label, and maybe a confidence. For example:

```
"sound_effects": [
  { "time": 4.2, "type": "Laughter", "confidence": 0.88 },
  { "time": 18.5, "type": "Slide Whistle (down)", "confidence": 0.70 }
]
```

In this example, at 4.2s there's a laugh (maybe an audience or canned laughter sound), and at 18.5s a slide whistle down (like a cartoon fall sound). The types would come from a fixed set or free-form depending on classifier output. If confidence is low or ambiguous, we might skip or label generically (e.g. “Sound effect” without specific type). This list helps in narrative understanding (e.g. laughter indicates a joke/payoff moment, a “ding” might indicate a tip or next step, etc.). Also, for content re-creation, knowing where SFX occur can help decide where to put them.

- **Beat & Rhythm Analysis:** Music-driven cuts are common – editors sync cuts or caption flashes to the beat of the music. We extract a **beat grid** for the background music. Using a music processing library (like **librosa** in Python or **Essentia**), we can compute the tempo (BPM) and the timestamps of beats (onsets). Librosa's `beat_track` can give an estimated BPM and beat positions in seconds. Alternatively, if the music has a steady rhythm, a simpler onset detector (librosa's onset_detect or an FFT-based approach) will identify peaks in energy. We will produce a list of beat times (probably on the downbeats). We also note the BPM (if it's relatively consistent; many TikTok audios have a fixed BPM).

Output: Under `audio.beat_grid` (or `audio.beats`), we provide the musical timing info. For example:

```
"beat_grid": {
    "bpm": 120,
    "beats": [0.500, 1.000, 1.500, 2.000, ...]
}
```

The first beat at 0.5s and then every 0.5s (120 BPM is 2 beats per second). We align the beats to the actual music; if the track has a lead-in, the first beat might not be at exactly 0.0. These times can be used to see if cuts occur on beats (we can cross-ref shot times with beat times to quantify how rhythm-synced the editing is – a valuable analytic metric). They can also assist narrative analysis; a payoff might coincide with a beat drop, for example.

If the video has no music or a very arrhythmic background, this section can be empty or `bpm: null`. If music is present but too short to establish BPM, we might estimate or skip. We ensure the beats array corresponds only to segments where music is active (e.g. if music stops in the middle, we stop the beat list there unless it's known to resume with same tempo).

- **Audio Metadata:** Additionally, we capture overall audio metadata like the detected primary language of speech (Whisper provides that, e.g. "en" or "es"), which we might put in `meta` or in the transcript info. If multi-language speech (some videos have code-switching), we could mark segments by language too (Whisper segment includes language probability). This might be beyond core needs, but worth noting for completeness (especially if captions might be in multiple languages).

All audio sub-modules operate with the same timebase as video (we ensure no offset issues – if the video has any initial silence or offset, we handle it). The audio analysis is cached extensively: the transcript generation (most expensive) is cached by video hash; we may also cache audio features or onset calculations since those are quick anyway.

Provenance for audio will list the ASR model (e.g. "Whisper-large v2"), any diarization model, and classification models used (like "YAMNet (trained on AudioSet)" etc.) with version. If any external API (Azure STT, etc.) was used, it will note that along with the model version if known (Azure often has a standard model behind the scenes).

Characters & Entities: Faces, Objects & People Tracking

This module identifies **visual entities** in the video – primarily people (faces) and significant objects – and links them with the audio-derived speakers to form a consistent set of "characters" or entities present in the content.

- **Face Detection & Tracking:** We scan the video frames for human faces. The goal is to **track each distinct person's appearances** across the video. The default approach is to run a face detector on frames and then group detections by identity.

Detection: We can use a fast face detection model like **MediaPipe Face Detection** or **RetinaFace** (both have high accuracy). For performance, we don't necessarily run on every frame; instead, we can detect on key frames (e.g. one frame per shot, or one every half second) and assume continuity within a shot (if a face is present throughout, missing a few frames doesn't matter). If a shot is long or a face moves in/out, we might do detection on a small interval. MediaPipe (by Google) can run real-time and gives face boxes; RetinaFace is more accurate and also provides facial landmarks which could help in knowing orientation. Both are lightweight enough for short videos (RetinaFace using CPU might be slower, but we can use a GPU or a smaller model like MTCNN if needed).

Face Tracking & Clustering: Detected face bounding boxes in consecutive frames can be linked by proximity (Kalman filter + Hungarian algorithm for tracking by position/size). However, since we ultimately want to know unique identities across cuts, we perform **face embedding & clustering**. For each detected face (especially the clearest instance of that face in a shot), we compute a face embedding using a pretrained face recognition model (e.g. **Facenet** or **ArcFace**, which are typically under permissive licenses for research). All embeddings from the video are then grouped by similarity (using a threshold or clustering algorithm) to assign an ID to each unique face/person. For example, if the same person appears in multiple shots, the embeddings should cluster together. This way, even if shots are not continuous, we recognize it's the same person (assuming frontal face and no extreme changes – if they changed clothes drastically or wore sunglasses in one shot, the model might miscluster, but short videos usually maintain one look per person).

We also handle edge cases: if two people appear simultaneously (some videos have multiple people on screen), the tracking/clustering will naturally assign two different IDs if their faces differ. We must be careful in labeling so that Person1 in one context vs Person2 doesn't get merged. A robust embedding model helps here.

Output: The result is a set of **character identities** with associated time segments of visibility. In the JSON, we represent them under `entities.characters` (or `entities.people`). Each character gets a unique ID (like "Person1", "Person2"). For each, we list their **appearance segments** (time ranges or shot indices where that person's face is visible) and any other descriptive info. For example:

```
"characters": [
  { "id": "Person1", "appearances": [ {"start": 0.0, "end": 15.0}, {"start":
30.0, "end": 45.0} ], "face_bbox": [100,200,200,300] },
  { "id": "Person2", "appearances": [ {"start": 10.0, "end": 12.0} ],
```

```
"face_bbox": null }
]
```

Here Person1 appears from 0–15s and 30–45s (maybe they left or the video cut to something else in between). We might include a representative bounding box or image (like first appearance's bbox) for reference, but likely we won't embed actual image data in JSON – instead, we could store a thumbnail image in cache and reference a path if needed for preview. The JSON can include something like `"face_thumb": "thumbs/person1.jpg"` as optional.

If there are multiple faces on screen at once (e.g. a dialogue), their appearances can overlap in time. Our format allows overlapping intervals in different character entries, which is fine.

Confidence: Each face detection can have a confidence; we assume if we confirmed a face cluster as a person, they are real – we might not list per detection confidence in the main spec (it could be voluminous if many frames). Instead, we store an overall quality like `"face_detected": true, "detections": 50 frames, "model": "RetinaFace, conf>0.9"`. For v1, just confirming presence is enough.

- **Speaker–Face Association:** This is where audio and video meet to answer "who is speaking when". Once we have speaker-labeled transcript segments and we have visual character tracks, we attempt to link them:
- If there is only one character (one face) and one speaker, the mapping is trivial: Person1 is the speaker for all dialogue. We will mark Person1 accordingly (e.g. in the characters list, add a field `"speaks": true` or link to the transcript segments).
- If multiple people are present, we align by time: e.g. Speaker0 spoke from 10–12s, and during 10–12s, we see that Person2's face was on screen while Person1's face was off (or Person1 was on but not talking? We might catch mouth motion but that's too detailed). We infer that Speaker0 corresponds to Person2. We do this for each speaker cluster. It might require that the video cuts to whoever is speaking (common in edited interviews). If two faces are on screen and one voice is heard, we have to guess – possibly based on context or pre-knowledge (not always solvable purely with automation unless we had a model that can do active speaker detection by lip movements). For v1, we assume either one speaker at a time and the active speaker is either the only face visible or the face doing a talking motion (we won't analyze lip sync explicitly, but it's a possible enhancement with something like SyncNet).
- Another clue: if our transcript or OCR caught a person's name (like a subtitle identifying a speaker), but that's rare in short videos.

We will attach a `speaker_id` to the character if identified. E.g. Person1 might get `"speaker_id": "Speaker0"` (which corresponds to transcript labels). Or we simply unify the naming and label them all as PersonIDs. Actually, better: once linked, we can update transcript to use Person1/Person2 labels for speakers instead of generic Speaker0. This yields a cleaner final result where "Person1" is both the visual and the audio identifier. In the output, we then might not even need separate speaker mention in characters, because the transcript lines already reference the same Person IDs.

Output integration: We plan to unify identities, so `characters[id].speaking_segments` (or similar) can list times they spoke. Alternatively, we rely on the transcript's speaker tags. For completeness, we may include in each character object a list of segments they speak (which is derived from transcript). For example:

```
{ "id": "Person1", "appearances": [...], "speaks": [ {"start": 0.5, "end": 5.0,
"text": "Hello..."} ] }
```

But this duplicates transcript content. A simpler way: characters have an ID and their mapping to speaker label is in provenance: e.g. `"Person1": "Speaker0"`. Then one knows all transcript segments labeled Speaker0 belong to Person1. We will likely choose clarity over minimalism since this is an analysis spec – duplicating a bit is fine for clarity. So we might list a short snippet or reference of their speech.

If a speaker has no face (like an off-screen narrator or a phone voice), we will have a character entry with no `appearances` but with `speaks` segments. Conversely, if a face never speaks (background person), they'll have appearances but no speech. This way, **every significant entity** in the video is accounted for, either visually, audibly, or both.

- **Object Detection & Tracking:** The video may contain important **objects** or props (e.g. a product being shown, a pet, a car in the scene). We use an object detection model (like **YOLOv8** or **Detectron2** with COCO classes) to find notable objects. We likely run this on a set of key frames (similar to face detection) – possibly the same frames we used for face OCR to not double decode. For each frame, we get bounding boxes and class labels (e.g. "cat", "car", "food"). We then consolidate across time:
- If the same object (class + position) persists across frames, we group that as one object instance with a time range. For example, if "cat" is detected at 5s and 6s in about the same location, that's presumably the same cat appearing from 5–6s. We might not try to differentiate two different cats if they never appear together; if two objects of same class appear simultaneously (like two people – but people we handle with face tracking separately – or two phones on a table), we can label them cat#1, cat#2 behind the scenes. But for output, might not need that level for objects (unless needed).
- We focus on objects that are relevant. A generic detector will find lots of things (every person will also be detected as "person" by YOLO – we skip those since faces cover persons; or the background "chair", "table" might be found but not meaningful unless it's crucial like a video about a chair). We may filter to classes of interest or confident detections that stand out in context. For instance, if the video is a cooking short, detecting "pan", "egg" is relevant. If it's a tech review, detecting "phone", "laptop" is relevant. We could have a predefined list of classes that are usually important (people, animals, vehicles, food, tools) and ignore trivial ones (like "cup" in background). For v1, we might simply output all detected with confidence above a threshold and let downstream use decide importance.

Output: Under `entities.objects`, we list distinct objects identified. Each entry includes the object type (class label), and the time segments it's visible. For example:

```
"objects": [
  { "label": "Dog", "appearances": [ {"start": 2.0, "end": 4.0}, {"start":
10.0, "end": 10.5} ] },
  { "label": "Car", "appearances": [ {"start": 15.0, "end": 18.0} ],
"confidence": 0.87 }
]
```

If multiple of the same class, we either merge if they never overlap (i.e. assume it's one object reappearing) or differentiate if they do overlap (like two dogs on screen from 2–4s, YOLO would give two boxes; we'd then output two dog entries that both appear 2–4s, or perhaps one entry with quantity 2? That might complicate things, maybe better separate entries like Dog1, Dog2 if needed). Given short videos, usually one of each thing stands out. The `confidence` can be an average or representative detection confidence for that object. If the object is known (like a specific brand or character) and we have a way to identify that (maybe via a custom model or because text on it was read), we could add that info, but that's advanced (e.g. recognizing a logo on a product – Azure can detect some brands as "brands" [31] [32], but we'll not dive into brand detection in v1 aside from listing object class).

Each object entry does not have a unique "ID" like persons, since typically we refer to them by label. If needed, we can give them IDs (Obj1, Obj2), but likely not necessary unless tracking multiple of same class distinctly.

- **Integrating Characters & Entities:** We essentially have two lists now: `characters` (people) and `objects`. "Characters" covers human entities (with visual face and/or voice). We could combine them into one `entities` list with a type field, but keeping humans separate from generic objects is semantically useful (especially since humans have different attributes like voice, while objects don't). In the JSON, grouping under `entities` as separate sub-lists is fine.

A special note: sometimes non-human entities can be "characters" (e.g. a cartoon character or pet that is the star of the video). If it's an anthropomorphic context, we might treat it similar to a person (but we won't do voice for a dog etc, unless a voice is dubbed). For clarity, animals will be under objects with their label (though an "object" named Dog might sound odd, but it's fine as a category).

Confidence & Model Info: The provenance will list what object detection model was used (e.g. "YOLOv5 (trained on COCO)" and minimum confidence). Face detection model and face recognition model will be listed as well. We also must be mindful of **privacy/PII**: if the pipeline could identify actual persons (say by comparing faces to known celeb databases), we should either avoid that or treat carefully. By default v1 does not attempt to name faces (so no PII risk beyond detection). If content-machine later integrates face recognition for known personalities (for example, if analyzing a news clip with known figures), that would be an extension requiring a face DB and careful handling. In this spec, we stick to anonymous IDs.

Narrative Arc & Engagement Analysis

In this final analytical module, we interpret the content's **story structure** and engagement tactics. Short videos often follow patterns: a **hook** to grab attention, rising action or **escalation** (e.g. increasing suspense or information), and a **payoff** or climax (the reveal or punchline). We aim to identify these segments and any other notable narrative features, using a combination of heuristic rules and AI (LLM) analysis on the compiled data from previous modules.

- **Hook Detection:** The "hook" is typically at the very beginning (first 2–3 seconds) and is designed to stop the viewer from scrolling away. It could be a startling visual, a bold textual claim, a question, or an interesting sound. To detect the hook, we look at:
- **Position & Time:** By default, assume the hook is roughly from 0s to ~3s or up to the first cut. If the first shot is super short (say a rapid cut in 1 second), that initial bit is likely the hook montage. Many

short videos put the hook entirely before the first transition, so the first cut itself might mark end of hook.

- **Content Cues:** We analyze the transcript of those first seconds for hook-like language (e.g. "Did you know…", "What if I told you…", "Watch till the end to see…" – these are classic hook phrases). We also look at on-screen text – sometimes creators literally put "Hook:" or a big title at start. Or an exciting visual (like explosion or dramatic before-after comparison) could serve as hook.
- **LLM Analysis:** We feed the early portion of transcript and video description to a prompt asking "What is the hook here? Does it effectively grab attention?". The LLM can identify if there's a clear hook and possibly its boundary [33] [34]. However, to avoid reliance solely on AI guess, we combine with timing heuristics.

Output: We mark a segment as the **Hook**, usually starting at 0 and ending at a certain timestamp. For example, we might output `{ "phase": "hook", "start": 0.0, "end": 2.7, "description": "Opens with a question about AI tools" }`. The description is a brief explanation of what the hook was (we can derive this from the content – e.g. if transcript says "Stop scrolling! You won't believe this…", description could be "Shock phrase to pique interest"). We might also score its strength (optional, e.g. "hook_strength: high" if it's particularly attention-grabbing). This goes into `narrative.arc` (details on structuring the JSON for narrative below).

- **Escalation (Build-Up) Detection:** After the hook, the video's middle portion usually **develops the content** – providing a sequence of events or information that increases intrigue or value, up until the climax. We identify escalation as the segment between the hook and payoff, but it might not be the entire middle if there are multiple beats:
- We look at pacing: often escalation maintains a quick pace to sustain interest. If the video slows down or goes off-topic, interest could drop (not good escalation). But if it continuously builds (e.g. each subsequent clip is more intense or reveals more), that's good escalation.
- Transcript analysis: using an LLM, we can ask it to divide the transcript into "setup/build-up vs climax" and see where the turning point is. Often the payoff is in the last third; everything before that is build-up. The LLM might identify phrases that signal transition ("And finally…" often heralds the payoff).
- We also check for **audience cues**: if our sound effect analysis found audience laughter or gasps at some point, that might indicate a payoff moment just occurred before it, meaning escalation ended right before that.
- Another heuristic: The **last cut** or second-to-last cut might lead into the payoff scene. For instance, if a video cuts to a different angle or something at 85% of its runtime, likely that last segment is payoff. Thus, escalation would end just before that.

Output: We define the escalation segment as from the end of hook to the start of payoff. We'll mark it with start and end time and a description like `{ "phase": "escalation", "start": 2.7, "end": 12.0, "description": "Demonstrating increasingly bold AI tool applications" }`. The description summarizes how the content escalated (e.g. "each step shows a more impressive result"). We might derive this summary by prompt: "summarize the middle of the video and how it builds interest." The LLM can articulate it, and we refine for brevity.

- **Payoff Detection:** The **payoff** is the climax or the main reveal/punchline that rewards the viewer for watching. Often at the end or near the end of the video. To find it:

- Look for a **notable change** in content around the last few seconds: maybe a surprise visual (our editing analysis might catch a sudden change or special effect there), or a tone shift in transcript (like the conclusion or answer to a question).
- If a video poses a question in the hook, the payoff is likely the answer at the end – we can have the LLM find if there's a question-answer pair and locate the answer.
- **Engagement indicators:** Many videos accompany payoffs with strong cues: e.g. a **sound effect** (a final "ta-da!" sound, or applause), or the **music might change** (beat drop, or music stops for a comedic silence). Our sound analysis might have flagged something like applause or a drum hit near the end, indicating payoff [31] (the example from Bellingcat mentions Azure identifying when names were mentioned at certain times – analogous to payoff if it's an introduction).
- Also, check for **on-screen text**: sometimes they literally put text at the payoff (like "The End" or the final punchline as caption).
- If available, **engagement data** (not in this context since it's just the video file) would show retention dropping after payoff – but we don't have that here. However, short videos often have a call-to-action after payoff ("Follow for more!"). If our OCR or transcript finds "follow @..." or "subscribe" near the end, that often comes *after* the payoff, meaning payoff just happened. So payoff likely ends right before any CTA or sign-off phrase.

Output: Mark the payoff segment, e.g. `{ "phase": "payoff", "start": 12.0, "end": 15.0, "description": "Reveals the final transformed photo with a surprise element" }`. The description highlights what the payoff was (e.g. "reveals the surprise result"). We might also include an "engagement_score" if we had a way to estimate how satisfying the payoff is, but that's speculative. Instead, we ensure we note it clearly.

- **Additional Narrative/Engagement Patterns:** Besides the basic arc phases, we can derive other insights:
- **Call-to-Action (CTA):** If the video ends with "like and subscribe" or "try this yourself", that's an engagement tactic. We detect these via text or keywords in speech. Not exactly narrative arc, but in engagement context, it's important. We could list CTA presence as a boolean or event (e.g. `narrative.cta: "Follow prompt at 14.5s"`).
- **Emotion Arc:** We could analyze sentiment or emotion over time (via voice tone analysis or text sentiment). If the video's escalation corresponds to rising positive excitement or tension, that's a pattern. An LLM could label each segment with an emotion (e.g. hook: curiosity, escalation: suspense, payoff: humor). For v1, we might not explicitly output this, but we'll incorporate into how we describe arc.
- **Story Type Classification:** Perhaps tag the video as one of known short-form content archetypes – e.g. "storytelling (beginning-middle-end)", "list/countdown format" (if the video is like "5 tips...", then it's structured differently, each item is mini-payoff, but the final item might be biggest), "Q&A", "challenge", etc. The narrative analyzer (likely an LLM) can categorize the format. We can output a `narrative.format` field like `"listicle"` or `"prank story"` if discernible. This is useful for analytics (compare performance of different formats).
- **Highlight Moments:** While payoff is the main highlight, sometimes there might be secondary spikes (especially in longer 60s videos, there could be multiple peaks). We can utilize the highlight detection logic similar to what **AI-Highlight-Clip** does: sliding window scoring of "highlight potential" with an LLM [33] [34]. That tool rated each segment on how "viral" or interesting it is [33]. We could replicate a light version: break the video into e.g. 5-second windows and ask an LLM to rank which window is most "exciting". It likely picks the payoff segment. But if it picks something else strongly too, that indicates another highlight (maybe a mid-video joke). We can optionally list a "highlights" array (top

N moments). However, given short length, probably the highlight is the payoff by definition, so we may skip separate highlights output. If needed, we just note if there's any other moment that stood out (like "mid-video surprise at 8s").

Implementation of narrative analysis relies significantly on **LLM analysis of transcript + our structural markers**. We will prompt a large language model (like GPT-4 or Gemini) with a structured summary of the video: the transcript, list of scenes, maybe notes on pacing or reactions, and ask for identification of hook/escalation/payoff and reasoning. The LLM's output can be parsed into our schema. We keep the prompts and model versions recorded. If local LLMs are an option by 2026 (maybe fine-tuned smaller models on story analysis), we could use them, but likely the OpenAI/Google APIs yield the best insight currently. The cost is minimal for a 15s transcript anyway.

We also ensure **provenance** here: we will note that narrative segmentation was done by an AI analysis with possible subjectivity. The user might disagree on what exactly constitutes the hook; our spec is an aid, not gospel.

- **Output Schema for Narrative:** In `VideoSpec.v1.json`, we dedicate a section `narrative` with the arc breakdown. A proposed structure:

```
"narrative": {
  "arc": {
     "hook":    { "start": 0.0, "end": 2.7, "description":
"Grabs attention with XYZ" },
     "escalation": { "start": 2.7, "end": 12.0, "description": "Builds up
by ..." },
     "payoff":  { "start": 12.0, "end": 15.0, "description": "Final reveal
of ..." }
  },
  "cta": "Follow me for more recipes",
  "format": "storytelling",
  "emotion": { "overall_tone": "uplifting", "ending_tone": "humorous" }
}
```

  The `arc` is the key part with times and descriptions. We included `cta` (if any, as text or boolean), `format` classification, and maybe an `emotion` or other pattern notes. These additional fields are optional. We only fill what we can reliably detect. In many cases, `cta` and `format` can be filled (CTA via simple detection, format via an LLM or rule: e.g. if the video uses a numbered list "1,2,3...", format = list). `emotion` might require an LLM's sentiment analysis per segment (which we can do as it's easy to prompt). If not confident, we leave it out or keep it high-level (like overall tone: motivational vs comedic).

By analyzing narrative in this way, the spec provides a human-understandable summary of **why the video is engaging**: it shows what the hook was, how it kept interest, and what the payoff/climax was. Downstream, this can feed into content-machine's decisions (e.g. ensuring generated videos have a clearly defined hook and payoff).

VideoSpec.v1 JSON Schema

Bringing together all the above, below is the **schema definition** for the `VideoSpec.v1.json` output. This is a structured JSON (or equivalent dict) containing all extracted metadata and meta-information. All time values are in seconds (float). Fields with `?` are optional or null if not applicable. The top-level structure is divided into sections for clarity:

```
{
  "meta": {
    "version": "VideoSpec.v1",
    "source": "<input file or URL>",
    "duration": <video length in seconds>,
    "frame_rate": <FPS>,
    "resolution": { "width": <px>, "height": <px> },
    "file_size": <bytes>?,
    "language": "<primary spoken language code>"?,
    "analysis_date": "<timestamp ISO8601>",
    "notes": "<any general notes or warnings>"
  },
  "timeline": {
    "shots": [
      { "id": 1, "start": <time>, "end": <time>, "transition_in": "<cut|fade|
other>" },
      { "id": 2, "start": ..., "end": ..., "transition_in": "cut", "jump_cut":
true? },
      ...
    ],
    "pacing": {
      "shot_count": <int>,
      "avg_shot_duration": <seconds>,
      "median_shot_duration": <seconds>,
      "fastest_shot_duration": <seconds>,
      "slowest_shot_duration": <seconds>,
      "classification": "<very_fast|fast|moderate|slow>"
    }
  },
  "editing": {
    "captions": [
      { "text": "<caption text>", "start": <time>, "end": <time>, "speaker":
"<PersonID?>" },
      ...
    ],
    "text_overlays": [
      { "text": "<overlay text>", "start": <time>, "end": <time>, "position":
"<top/bottom/coords>?", "note": "<type (e.g. title, hashtag)>?" },
      ...
```

```
    ],
    "camera_motion": [
      { "shot_id": <id>, "motion": "<static|zoom_in|zoom_out|pan_left|pan_right|
tilt>", "start": <time>, "end": <time> }
      // Alternatively, if one motion per shot, we could just list under each
shot in timeline.
    ],
    "other_effects": {
      "jump_cuts": [ <list of shot transition ids that are jump cuts> ]?,
      "speed_changes": [ { "start": t, "end": t, "speed": 0.5 } ]?
    }
  },
  "audio": {
    "transcript": [
      { "start": <time>, "end": <time>, "speaker": "<Person1>", "text":
"<utterance>" },
      { "start": <time>, "end": <time>, "speaker": "<Person1>", "text":
"<...>" },
      { "start": <time>, "end": <time>, "speaker": "<Person2>", "text":
"<...>" },
      ...
    ],
    "music_segments": [
      { "start": <time>, "end": <time>, "track": "<name/id>?", "background":
<bool>?, "description": "<e.g. 'upbeat EDM'>" }
    ],
    "sound_effects": [
      { "time": <time>, "type": "<SFX label>", "description": "<optional>",
"confidence": <0-1>? }
    ],
    "beat_grid": {
      "bpm": <float>?,
      "beats": [ <time1>, <time2>, ... ]
    }
  },
  "entities": {
    "characters": [
      { "id": "Person1",
        "appearances": [ {"start": t1, "end": t2}, ... ],
        "speaker_label": "Speaker0"?,
        "speaking_segments": [ {"start": t3, "end": t4}, ... ]?
      },
      { "id": "Person2", "appearances": [ ... ], "speaker_label": "Speaker1"?,
"speaking_segments": [ ... ]? },
      ...
    ],
    "objects": [
      { "label": "<ObjectClass>", "appearances": [ {"start": t5, "end":
```

```
t6}, ... ], "confidence": <0-1>? },
      { "label": "Cat", "appearances": [ {"start": 8.0, "end": 10.0} ] },
      ...
    ]
  },
  "narrative": {
    "arc": {
      "hook":        { "start": <time>, "end": <time>, "description": "<hook
summary>" },
      "escalation":  { "start": <time>, "end": <time>, "description": "<build-
up summary>" },
      "payoff":      { "start": <time>, "end": <time>, "description": "<payoff
summary>" }
    },
    "format": "<format label>"?,
    "cta": "<call-to-action text if any>"?,
    "themes": [ "<keyword1>", "<keyword2>" ]?,
    "tone": "<emotional tone or style overall>"?
  },
  "provenance": {
    "modules": {
      "shot_detection": "PySceneDetect v0.6 ContentDetector [35]",
      "ocr": "Tesseract.js v2.1 (fallback EasyOCR v1.6) [18] [26]",
      "asr": "OpenAI Whisper-large v2 [12]",
      "diarization": "pyannote.audio 2.0 (Speaker Embedding model x)",
      "face_detection": "MediaPipe FaceDetection (0.5 threshold)",
      "face_recognition": "ArcFace model (insightface v1.0)",
      "object_detection": "YOLOv8m (COCO)",
      "narrative_analysis": "GPT-4 (OpenAI, 2026-01 API)"
    },
    "notes": [
      "Faces grouped into 2 characters; Person2 voice was off-screen (no face
detected).",
      "OCR: EasyOCR used for frames 100-120 (stylized font).",
      "Azure Video Indexer not used in this run (available as integration)."
    ]
  }
}
```

(Comments in parentheses or after `//` are for explanation here and not in actual JSON.)

Explanation:

- `meta`: Basic info about the video and the analysis. `analysis_date` when this spec was generated, for record. `language` if we detected the spoken language (e.g. "en" or "es"). `notes` can include any general warnings (e.g. "Speech overlapping with music, transcript might be less accurate in parts").
- `timeline.shots`: Array of shots with times and transition. `id` is sequential. `transition_in` describes how this shot begins (from previous shot). If a shot started with a crossfade, previous shot likely

ended with fade-out; we could mark this one as fade-in. But to keep it simple, we mark at cut points either the outgoing or incoming type. For clarity, we chose incoming. We mark `jump_cut: true` if we identified it (that tag on shot 2 means shot2 starts with a jump cut from shot1).
- `timeline.pacing`: Stats computed from shots. The classification can be subjective; we define thresholds (e.g. >30 cuts/min = very_fast, etc.).
- `editing.captions`: On-screen subtitles matching speech. In each, `speaker` indicates who said it (so if Person2's dialogue was captioned, we link it). If the pipeline didn't link speakers, this could be left out or use Speaker0 etc.
- `editing.text_overlays`: Other text overlays not matching speech. `position` is optional (like "top_left", "center", or an [x,y] if needed). `note` could classify the overlay type, e.g. "username handle", "title", "hashtag sticker", or if it's a platform watermark, note that.
- `editing.camera_motion`: List of camera moves with their time range (if we choose per-shot labeling, each shot that has a motion will have an entry). Alternatively, we could incorporate `motion` into shot properties directly to reduce nesting. But we separated for modularity.
- `editing.other_effects.jump_cuts`: Could list shot IDs or times where jump cuts occur. `speed_changes`: segments where playback speed is !=1 (if detected). These are optional and mostly placeholders unless we implement these detections.
- `audio.transcript`: Array of spoken segments. We label speakers as Person1, Person2 (to unify with entities), or some generic if not linked. Each has the text. This is essentially the subtitle script.
- `audio.music_segments`: Time ranges of music. If known track, `track` could be filled (e.g. a content ID or name if identified via external API – not default). `background` indicates if it's behind speech (we can tell if it overlaps with any transcript segment; if yes for entire duration, we might say background:true). `description` optional quick note on type/mood of music.
- `audio.sound_effects`: List of notable SFX. `type` is a short label (from a predetermined taxonomy or classifier output). E.g. "Laughter", "Boing sound". If our classifier isn't sure but knows it's a "cartoon sound", we might put that. Each has a single time (or we could use a short interval if the sound lasts a bit, but usually SFX are like instantaneous events or <1s).
- `audio.beat_grid`: The rhythmic structure. If no discernible beat, it could be empty or bpm:null. If present, the array of beats could be long (but short video likely has <200 beats). That's fine. Or we could just store BPM and maybe phase of beat at start (but listing beats explicitly allows precise alignment checks with cuts).

- `entities.characters`: List of people in the video. Each has an `id` (we use Person1, Person2 for anonymity). `appearances` is a list of intervals they are on-screen. (We get these by grouping the face detections per person; if a person exits and re-enters, that yields multiple intervals; if mostly continuous, one interval per shot or spanning shots). We might compress contiguous appearances that span shot boundaries if the person is present across a cut (rare; cuts usually imply scene change, except jump cuts where same person reappears after a cut – we could either merge those two as one logical appearance or keep separate since a cut happened. To simplify, we may just not merge across cut even if same person, just list shot by shot. Or have an appearance entry per shot that person is in. This is detail; either way is fine as long as it's clear).
  `speaker_label`: This ties the character to a speaker from the transcript if we didn't directly name them the same. For instance, if our transcript still says Speaker0, we put "Speaker0" here. But if we replaced transcript labels with Person1, this is redundant. We will likely replace transcript labels with PersonX to unify, so that field may not be needed.
  `speaking_segments`: If we want to quickly see what each person said, we can list their dialogue times (and possibly text) here. But since transcript already has it, this might be omitted to avoid

duplication. It's more for conceptual clarity. Possibly we include it because it makes the character profile self-contained: one could see Person1 appears at X times and speaks at Y times. For now we keep it, with the understanding it's derived from transcript.

- `entities.objects`: List of non-human objects. Each with a label and intervals. We don't assign them unique IDs beyond the label because usually we reference them by name if needed. If two objects have same label but are different instances, we might append something, e.g. "Car1", "Car2", or include an `id` field. But that complexity may not be necessary unless the video explicitly has two similar objects that both matter (rare in short vids focusing on one thing). We assume one of each important thing. The `confidence` is optional global confidence for that object's presence (e.g. if detection was borderline, lower confidence might indicate it's a guess).

- `narrative.arc`: Contains three named phases: hook, escalation, payoff. Each has times and a short description. The times should cover the main portion of video from 0 to end (the escalation end and payoff end might coincide with video end or CTA start, etc.). This gives a structural segmentation.
  Additionally, `narrative.format`: one of a set of narrative structures (some possibilities: "story", "tutorial", "list", "loop video", "ad/promo", etc.). We fill it if identifiable.
  `narrative.cta`: If we detected a call-to-action or any engagement prompt in the video (commonly at the very end or occasionally at beginning like "Wait till the end"). If multiple, we can combine or list them. Typically one main CTA (like "subscribe" at end).
  `narrative.themes`: Key topics or themes of the video. We can derive by analyzing the transcript or using Azure's keyword extraction (Azure Video Indexer can output keywords discussed [36] [37]). Alternatively, an LLM or even simple TF-IDF on transcript could give main topics. E.g. for a tech tutorial video, themes might be ["AI tools", "productivity"]. For a cooking video: ["recipe", "pasta"]. This is more content semantics than narrative, but it falls under understanding what the video is about – which is useful for indexing and analytics. If using Azure or another service, we could get these (Azure would list "Machine learning", "Artificial intelligence" etc as in that Bellingcat example [38]). We include if available.
  `narrative.tone`: The overall tone or emotion style (e.g. "humorous", "dramatic", "informative"). An LLM can infer this easily from language and audio cues. For engagement, knowing tone is helpful (e.g. funny vs serious content). It's optional.

- `provenance`: Finally, this section logs the tools and versions used for each module, as discussed in the design. Each key in `modules` corresponds to one of our pipeline modules. We list name and version (and maybe model names). For example, shot_detection lists PySceneDetect with ContentDetector [35]. OCR lists Tesseract.js and fallback EasyOCR [18] [26]. ASR Whisper [12], etc. We cite a reference for each to document why we chose it or what it is. These citations in the provenance might not all be needed in the JSON itself (they are more for this spec document), but could be stored in an internal log. However, content-machine might keep an attribution internally for open-source credit. If needed, we can keep the references in comments.
  The `notes` array in provenance can include any pipeline irregularities or human interventions. E.g. if the OCR struggled with some text and we manually annotated it (if that ever happened), or if certain modules were skipped. In our example, we note Azure Video Indexer wasn't used (just to clarify data source). We also note that Person2 was voice-only, no face found (so one is off-screen narrator). These notes ensure someone reading the spec understands odd cases (like why a Person2 has speaking segments but no appearance).

This JSON can be quite verbose, but it is **comprehensive**. It's effectively a machine-readable storyboard of the video. Content-machine or other systems can query it to answer questions like "How many people appear?", "What was said and by who?", "Does the video follow the hook-payoff format?", "How fast are the cuts relative to beat?", etc., without needing to reprocess the video.

Integration with content-machine Architecture

The pipeline is designed to integrate seamlessly with the existing **content-machine** project structure and workflows:

- **Module Encapsulation:** Each analysis function (scene detection, OCR, ASR, etc.) can be implemented as a **service or library** within content-machine. For example, we might add a `video_analysis` module with submodules for each task. Where possible, we use content-machine's preferred languages: content-machine has a Node.js/TypeScript core, but many analysis tools are Python. We can wrap Python tools in CLI scripts or a microservice. For instance, content-machine could spawn a Python process to run PySceneDetect and return JSON cuts, or use Node bindings if available. Some tools like Tesseract have direct JS libraries (as we saw with tesseract.js), so we use those in Node. Others (Whisper ASR) might be easier to run via Python (perhaps using the `whisper` CLI or a Py API). We ensure each component's integration is **fault-isolated** – e.g. call via subprocess so if it crashes it doesn't take down the Node runtime, and we capture errors.

- **Orchestration & Workflows:** As noted, we can leverage **Temporal** or **BullMQ** to orchestrate the pipeline. A simple integration is to have a high-level function `analyzeVideo(filePath)` that internally orchestrates the steps (either sequentially in code or dispatching jobs). Given content-machine already vendors Temporal and BullMQ [39] [10], we could implement a **Temporal workflow** called `VideoAnalysisWorkflow`. Each stage (scene detection, audio transcription, etc.) would be a Temporal Activity (or a BullMQ job if we go that route). This yields robust execution: e.g. the ASR stage might be long, but Temporal can handle it and even retry if it fails, and maintain state if server restarts [40] [9]. If content-machine prefers a simpler approach, a straightforward sequential script with try/catch and invoking each module is fine for initial implementation, and we can upgrade to Temporal when scaling.

- **Data Storage & Caching:** We decide where to store intermediate and final results. Since content-machine likely manages project data, we can store the final `VideoSpec.json` in a database or file store associated with the video. For caching intermediate files:

- Possibly use a directory structure: e.g. `.cache/video_analysis/{video_id}/transcript.srt`, `faces.json`, etc.
- Alternatively, use a key-value store (maybe Redis, since BullMQ uses Redis, we might piggyback) for small items, though transcripts can be big so file might be better.
- Because content-machine might handle many videos, we ensure cache invalidation: maybe include a content hash of the video file in the file names so that if content changes (or different video with same name), caches don't mix. Also tie it to pipeline version.

The final `VideoSpec.v1.json` can be stored similarly (perhaps in a results folder or posted to a content-machine API). If content-machine has an internal database for content metadata, we might parse key fields into DB columns (e.g. store the transcript text for search, store the number of cuts, etc.). But as an

integration step, simply storing the JSON and maybe indexing some parts (like transcripts to a search index) is sufficient.

- **Downstream Usage – Templating & Analytics:** The value of this analysis shines when used by other parts of content-machine:

- **Content Templating:** content-machine can use the spec to inform video generation. For example, suppose content-machine has template scripts for creating videos; using this spec, it could allow a user to "Apply style from video X to content Y". The spec provides timeline structure and editing cues from video X. We could create a generator that reads `VideoSpec` and auto-constructs a timeline in our video composition system (Remotion or MoviePy) that matches it. E.g., if a user wants to mimic the pacing of a trending TikTok, we take that TikTok's spec: we create the same number of shots of the same lengths, insert the user's content (images/videos) accordingly, add similar captions at the same times (but with new text), use similar music/beat (or at least cut on the beats from the spec). Essentially, the spec can serve as a **blueprint** for new videos. content-machine's rendering stack (like Remotion components described in the deep dive [41] [42]) could accept parameters derived from the spec – e.g., a Remotion timeline could be programmatically generated given a list of shot durations and overlay timings. We'd also propagate style: if the spec noted "zoom_in" in shot 3, we could apply a Ken Burns zoom to the new shot 3 content. If the spec shows captions were used, we can auto-enable caption generation for the new video. This allows a form of **style transfer** from example videos.

- **Analytics & A/B Testing:** By storing specs of many videos (especially along with performance data like view count, watch time), content-machine's analytics module can find patterns. For instance, using the JSON data, one could query "Give me all videos with >10M views and find common traits in narrative or pacing." If many high-performers have a hook under 3 seconds or use certain SFX, those insights can guide content creators. Our spec is designed to be query-friendly: key fields like shot_count, avg_shot_duration, presence of captions, number of people, etc. are directly accessible. We can load these into a DataFrame or database for analysis. For example, Qdrant (the semantic vector DB content-machine uses for memory) could store some vector representation of the video style – though likely not needed since we have explicit features. Even so, one could embed the narrative description or transcript into a vector for similarity search (find videos with similar story). The spec could also feed a **recommendation system**: e.g. if a user wants to create a video on topic X, the system can retrieve successful videos on similar topics and analyze their spec for guidance (like "These recipe videos all had an average of 15 cuts/min and used top-down camera shots with captions; do likewise").

content-machine's docs suggest storing successful video patterns for reuse [43], which aligns with saving these specs. For example, they mention using Qdrant to store analyzed trends and successful patterns [43] [44]. We can indeed embed a representation of the spec (like key metrics or a learned vector) into Qdrant. This way, the system can do similarity search: "find me a reference video spec similar to this new idea I have" and then potentially use that spec to shape new content.

- **Licensing & Model Management:** content-machine will maintain the models used (some likely vendored already as we saw FunClip, PySceneDetect in vendor). We integrate our pipeline with those: e.g. use the FunClip ASR if needed (they included it vendored [28]). We ensure any new models (like face recognition) are properly added (maybe as a submodule or direct install). We also

incorporate configuration in content-machine's config files or .env for API keys (like Azure API key if user opts to use it, or OpenAI key for narrative LLM if not using local). The spec generation respects user settings: e.g. a flag `--use-external-indexer` could route the pipeline to use Azure for everything (and skip local analysis). In such case, the pipeline might simply call Azure Video Indexer API which returns a JSON of insights (scenes, faces, transcript, topics, etc.) [45] [27], and then convert that to our `VideoSpec` format. This is an alternate code path: essentially a quick integration using Azure's output as a surrogate for multiple modules. It's optional but we design for it. If user has Azure configured, we could merge Azure results with our own (maybe to validate or fill gaps like known celeb name identification if Azure provides that).

- **Continuous Integration (CI) & Testing:** We will create test cases within content-machine to validate this pipeline. For CI, we can include a couple of **sample short videos** (maybe very short ones or synthetic ones that we have rights to) in the repo or as fixtures. For example, a 5-second clip with two cuts and one caption – and we know the expected JSON output. We write unit tests to run the analysis on it and assert that key outputs match expected (e.g. it finds exactly 2 shots, transcripts certain text, etc.). We can also test each module in isolation with mocked data: e.g. feed a known audio file to the ASR component and expect the transcript. However, since external models can be nondeterministic (an LLM might not always label narrative exactly the same way), we focus tests on deterministic parts (scene detection, OCR on a known frame, etc.). content-machine's testing patterns (for video generation they use snapshot tests [46]) can inspire us: we might do a "snapshot" of the JSON and verify it against a stored one for a given input video. If changes in the pipeline alter the JSON (intentionally due to improvements), the snapshot would change – we can then update the expected or flag differences. We also test performance (ensuring a short video analysis finishes under certain time).

Additionally, we can implement **benchmark tests** for the pipeline: measure how long each module takes on example videos (ensuring we meet any performance targets, e.g. < X seconds for a 60s video on given hardware). We can test memory usage too, to avoid leaks (especially if running multiple in parallel). If content-machine uses CI/CD, we integrate these tests so that any changes to analysis modules that break outputs are caught.

- **Optional Evaluation Benchmark:** If needed, we can evaluate the accuracy of our pipeline against known ground truth. For example, use a public dataset or manually annotated set of a few videos where we know the correct shots, transcripts, etc. Compare our output. This isn't a typical unit test but a separate evaluation script. It could output precision/recall of shot detection, WER (word error rate) of ASR, etc. We might incorporate this in documentation to justify model choices (e.g. "PySceneDetect achieved 95% recall on test clips for cut detection"). As an extreme, the *ShortVid-Bench* research [47] [48] could serve as inspiration for tasks (they focus on captioning and QA tasks to evaluate multimodal understanding). While we are not training a model here, if content-machine wanted to measure how well the pipeline's LLM narrative analysis works, they could create a small set of videos with human-labeled "hook, payoff" and see if the pipeline agrees. This can guide future improvements (maybe fine-tune or prompt engineering for the LLM).

In summary, integration involves hooking the pipeline into content-machine's command flow and ensuring that once analysis is done, the `VideoSpec` is accessible for the next steps. We will likely provide:

- A **CLI command** in content-machine, e.g. `cm analyze-video <input> -o <output.json> [--fast] [--use-azure]`. This would run the pipeline on a given file or URL. The `--fast` flag might run only pass 1 for quick output. `--use-azure` might offload to Azure VI. This CLI is useful for developers and also for batch processing (e.g. analyze a list of trending videos each day by automating this command). The CLI will print summary info and save the JSON.
- A **Node.js API** function (for internal use): e.g. `await analyzeVideoToSpec(videoInput, options)` that returns the JSON object. This can be called within other workflows, e.g. as part of an upload pipeline (maybe when a user uploads a new video to the system, we immediately analyze it to tag it).
- **CI tests** that run `analyzeVideo` on known small videos and verify output.

CLI/SDK Integration Patterns

For easy use by developers and automation, we implement both CLI tools and SDK hooks:

- **Command-Line Interface:** We add a command to content-machine's CLI (assuming it has one). For example:

```
content-machine analyze-video input.mp4 --json out.json --no-cache --pass 2
```

This would load `input.mp4`, run the full analysis, and save results to `out.json`. Options like `--no-cache` to force re-analysis ignoring caches, or `--pass 1` to only do first-pass quick analysis. We might also allow `--report` to print a human-readable summary in the console (pulling highlights from the JSON). The CLI will log progress for each module (so the user sees e.g. "Detecting scenes... found 5 cuts", "Transcribing audio... done", "Analyzing narrative... done"). This feedback is helpful since some steps (ASR) can take a bit of time. The CLI could also support analyzing a URL directly (if content-machine has functionality to download a video from a URL or if given a YouTube/TikTok link, it could use youtube-dl or an API to fetch it, then run analysis).

- **SDK / API:** Within content-machine's code (for example in a web service context), one can call a function or use a class. We might have:

```
import { VideoAnalyzer } from 'content-machine/video_analysis';
const analyzer = new VideoAnalyzer({ useCache: true, useAzure: false });
const spec = await analyzer.analyze(videoFilePath);
console.log(spec.timeline.shots.length + " shots detected.");
```

This would return a JS object representing the JSON. The developer can then manipulate it or store it. We ensure the object conforms to a defined TypeScript interface (to catch any typing issues). For instance, define interfaces for Shot, Caption, Character, etc., for autocompletion and type safety in TS. This makes it easier for other parts of the system to use the spec.

- **Continuous Integration (CI):** We include automated tests for at least basic functionality. Possibly using a lightweight video (few seconds of known content). In CI, we may not want to run heavy models (like Whisper large) due to time constraints, so we might use a small model or a stub for testing. For instance, in test mode we could configure the pipeline to use Whisper-tiny (fast but maybe less accurate – fine for verifying pipeline runs). Or we could mock the ASR stage entirely by providing a predefined transcript for the test video. The tests will check that the JSON has expected keys and certain values. We also test the CLI command with a sample file in a CI environment to ensure it produces an output file and no errors. If the project has a build pipeline, we might also generate documentation from the schema (maybe using the JSON schema as formal spec – we could write a JSON Schema .json file to formally define types and include that in docs for external integrators).

- **Performance Considerations:** For typical short videos (15-60s), the pipeline should run in a reasonable time on modern hardware (say, a few seconds to maybe 1-2 minutes if using large models). If content-machine processes videos in bulk, we might incorporate parallelism – e.g. running separate video analyses concurrently (since each video is independent). That can be managed by running multiple instances of the workflow or by internally parallelizing modules (some parts like face detection and object detection on frames could run in parallel threads, but Python GIL etc. might limit that; better to parallelize at video level). We'll ensure the CLI or API can be called concurrently without interfering (cache keys handle that by using unique video IDs).

- **Optional Benchmarking & Monitoring:** We suggest adding logging of performance metrics. E.g., the pipeline could output how long each module took. This could be stored in provenance or at least logs. Over time, this helps identify bottlenecks or regressions. We could also allow a "dry run" that only prints what it *would* do (for debugging pipeline logic without heavy compute, not very applicable here though since everything is analysis). Monitoring memory usage might also be needed if analyzing longer videos or multiple at once to ensure we don't OOM (especially with big models; e.g. Whisper large can use ~2GB GPU VRAM, which is fine if we have it, but on CPU it's slow). We could allow config to choose a smaller ASR model for resource-limited environments.

- **Security & Licensing Check:** If content-machine is used commercially, using certain models or services might require proper attribution or keys. We incorporate in provenance (and maybe in UI if needed) acknowledgments for open-source we used (like "Transcripts by OpenAI Whisper MIT License", etc.). Since the user specifically cares about licensing, we maintain a **VENDORING.md** (maybe already present) or similar document listing all integrated tools and their licenses (some of which content-machine did as we see in deep dives). We ensure our integration aligns with those guidelines (which is indeed why we choose MIT/Apache tools for default).

Finally, we plan for **future extension**: the spec is versioned v1. If future needs arise (like identifying more granular editing features, or including new sections like sentiment timeline, or AR filter detection), we can introduce v2 with new fields. The modular design means adding a module or swapping one out (like if a new better scene detector emerges, or an end-to-end AI that does everything at once – could be possible in the future) is straightforward, as long as it outputs in our schema. The architecture can accommodate a monolithic analysis model as well, we'd just integrate it as another path (e.g. "Gemini multi-modal model analysis" which might directly output captions, tags, etc. – we could parse it into our format and possibly use it to cross-check our more modular approach).

Testing & Benchmarking

To ensure the pipeline's quality, we set up a test suite focusing on correctness and efficiency:

- **Unit Tests for Each Module:** For instance, test the scene detection on a known video with 3 cuts (could generate a synthetic video of black/white alternating scenes) and verify `timeline.shots` has those 3 cuts. Test OCR on a sample frame image with known text (perhaps a screenshot of a TikTok caption) to verify the caption text is correctly extracted by our OCR pipeline (we can include that image in test assets) [49] [50]. Test the ASR on a short audio clip (maybe "hello world" synthesized audio) to ensure transcript text matches. If we cannot include model weights in CI (due to size), we can mock these or use minimal models (like Whisper tiny). We can also test the narrative LLM logic by mocking the LLM response to a known input (simulate GPT-4 returning a known segmentation and ensure we parse it to JSON properly).

- **Integration Test:** Use a real short video (few seconds, or one from content-machine's examples if any) and run the full `analyzeVideo` pipeline. Then check the high-level output: e.g. ensure `meta.duration` matches actual video length, ensure the number of caption entries equals the number of subtitles in the video (if we manually know it), etc. This catches any mis-wiring between modules.

- **Benchmarking:** Create a script to run the pipeline on a set of test videos (maybe varying lengths 15s, 30s, 60s) and time each stage. This is not a pass/fail test in CI, but for our knowledge. We ensure it meets any performance targets (for example, ideally analyzing a 60s video with default models in under, say, 30 seconds on a decent machine). If any stage is too slow, we consider optimizations like using GPU if available (Whisper on GPU is far faster, we'll detect if CUDA available and use it, else warn that it's on CPU and slow). We can incorporate these checks: the pipeline could log if it's falling back to CPU so that user knows why it might be slow.

- **Quality Evaluation (Optional):** As a separate process, we might sample a few outputs and have humans review them (especially the narrative labels and any ambiguous parts like whether an overlay was correctly categorized). We then refine the rules or LLM prompts accordingly. This is more of an ongoing improvement step.

In conclusion, the **reverse-engineering pipeline** provides a robust, extensible foundation for understanding short-form videos. By producing the `VideoSpec.v1` output and integrating it with content-machine's tools, we enable data-driven content creation and analysis. Creators and analysts can inspect these specs to learn what makes videos tick (with citations to the pipeline's sources and confidence) and leverage the insights for smarter content production. The modular approach and comprehensive schema ensure that as the social video landscape evolves (new editing trends, new platform features), we can adapt the pipeline (update modules or add new ones) without breaking the overall architecture. This spec will serve as a blueprint for implementation and a reference for future iterations.

Sources: The design choices above reference numerous tools and research. For example, PySceneDetect's role as a scene detector [1], FunClip's ASR+diarization approach [29], AI-Highlight's use of Whisper and LLM for scoring [12], Azure's comprehensive video AI capabilities (OCR, face tracking, etc.) [51] [27], and content-machine's own findings on OCR tech for captions (preferring Tesseract with fallback) [18] [26]. These

informed our module selections and defaults. The integration leverages content-machine's existing infrastructure for orchestration [39] and memory (storing successful patterns for reuse) [43] . All these ensure the pipeline is built on proven components and aligns with content-machine's ecosystem for maximum synergy.

---

[1] [12] [28] [29] [30] 45-video-processing-clipping-capture-DEEP-20260102.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/deep-dives/45-video-processing-clipping-capture-DEEP-20260102.md

[2] [13] [14] [18] [19] [20] [21] [22] [23] [24] [25] [26] [49] [50] RQ-31-OCR-TECHNOLOGIES-VIDEO-CAPTION-20260107.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/investigations/RQ-31-OCR-TECHNOLOGIES-VIDEO-CAPTION-20260107.md

[3] Text Recognition for Video in Microsoft Video Indexer - Microsoft Azure
https://azure.microsoft.com/en-us/blog/text-recognition-for-video-in-microsoft-video-indexer/

[4] [27] [31] [32] [36] [37] [38] [45] [51] Azure AI Video Indexer | Bellingcat's Online Investigation Toolkit
https://bellingcat.gitbook.io/toolkit/more/all-tools/azure-ai-video-indexer

[5] [6] [7] [8] [9] [10] [11] [39] [40] 58-video-processing-orchestration-clipping-DEEP-20260102.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/deep-dives/58-video-processing-orchestration-clipping-DEEP-20260102.md

[15] [16] [17] [33] [34] [35] [43] [44] infrastructure-and-integrations-DEEP-20260102.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/deep-dives/infrastructure-and-integrations-DEEP-20260102.md

[41] [42] 84-video-composition-rendering-infrastructure-DEEP-20260102.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/deep-dives/84-video-composition-rendering-infrastructure-DEEP-20260102.md

[46] VIDEO-TESTING-PATTERNS-20260104.md
https://github.com/45ck/content-machine/blob/4536fe95bb1fff969a1e1d32213f520627d6610c/docs/research/deep-dives/VIDEO-TESTING-PATTERNS-20260104.md

[47] [48] ShortVid-Bench: Short Video Benchmark
https://www.emergentmind.com/topics/shortvid-bench