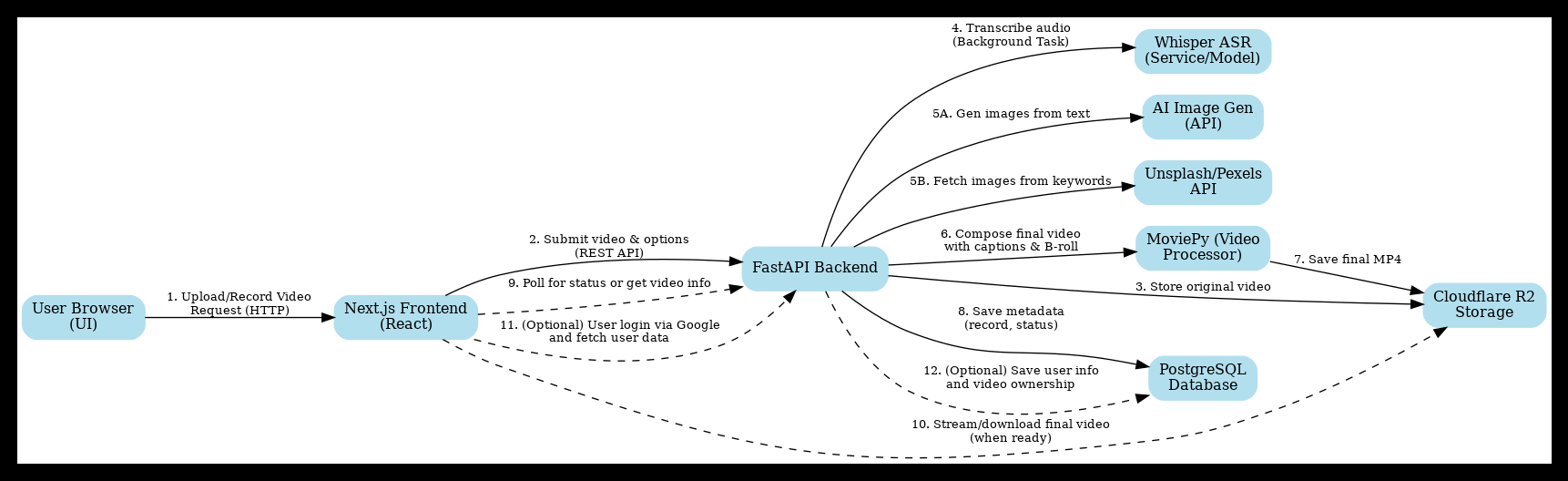
# **AI Video Generator – Technical Plan**

## **Overview**

This document outlines a comprehensive plan to build an AI Video Generator web application. The app allows users to upload or record a short talking-head video of themselves (an *“AI avatar” video*), automatically transcribe speech to captions using an AI model (e.g. OpenAI’s Whisper), and enrich the video with B-roll visuals. Two B-roll options are supported: (A) AI-generated images (via DALL·E or Stable Diffusion API), or (B) stock visuals fetched from free platforms like Unsplash/Pexels. The backend uses MoviePy to combine the original video, captions (subtitles), and B-roll clips into a final MP4 output. Users can then view or share the result on a dedicated page (e.g. /generate/{id}), which displays the video with a thumbnail, metadata, playback controls, and a download link.

Tech Stack Summary: The frontend is built with Next.js (React) for a modern, responsive UI. The backend is a FastAPI service (Python) to handle video processing and API requests. PostgreSQL stores user accounts (Google OAuth login) and video metadata. Cloudflare R2 (S3-compatible object storage) is used to store uploaded videos, generated media, and outputs. Captioning uses Whisper (via API or local model), and image generation/search uses external AI APIs and Unsplash/Pexels APIs. This architecture emphasizes clarity and scalability, suitable for a single developer to implement an MVP within ~10 hours by leveraging high-level APIs and existing libraries.

## **System Architecture**

*High-level system architecture for the AI Video Generator. Solid lines indicate primary data flow (video upload, processing, storage), and dashed lines indicate optional flows (user authentication and status polling).  
*

The application is structured as a client-server model with clear separation of concerns:

* Next.js Frontend (Client): Presents the user interface for uploading/recording videos, choosing B-roll options, and viewing results. Communicates with the backend via REST API calls. It handles authentication (Google OAuth) and provides pages for video generation results and user dashboards. The UI will have a polished, professional style (similar to popular AI productivity apps) – focusing on simplicity and responsive design.
* FastAPI Backend (Server): Exposes RESTful API endpoints for video upload, video generation status, and fetching video results. The backend orchestrates the processing pipeline:
  + Video Upload Handling: Receives the uploaded video (or recorded blob) and stores it (initially to disk or directly to R2).
  + Asynchronous Processing: In a background task, it extracts audio and runs Whisper for transcription. Then, based on user choice, it either calls an image generation API or searches stock image APIs for relevant visuals. Finally, it invokes MoviePy to merge the original video, captions, and B-roll images/clips into the output video.
  + Storage & Delivery: Saves the final MP4 to Cloudflare R2 and updates the database record with the output location, ready for the user to retrieve.
* PostgreSQL Database: Stores User records (if logged in via Google) and Video records. Each video record links to the user (or null if generated anonymously) and contains metadata (e.g. caption text, generation options, timestamps, file URLs). This allows logged-in users to have a history of their generated videos and ensures each video (even anonymous) has a unique ID for sharing via URL.
* Cloudflare R2 Storage: Acts as a central object store for binary data – uploaded raw videos, generated images (if needed), and final video files. Using R2 (S3 API) decouples file storage from the application server, enabling scalability (multiple app servers can access the same files) and persisting large media efficiently.
* External AI/Media Services: The backend integrates with external services for AI capabilities:
  + *Whisper API/Model:* For speech-to-text captioning. This could be OpenAI’s Whisper API or an open-source model running locally. Whisper provides timestamped transcripts which are used to create synchronized subtitles.
  + *Image Generation API:* (Option A) e.g. OpenAI’s DALL·E or Stability AI, used to create images from text prompts (derived from the video transcript).
  + *Stock Photo/Video APIs:* (Option B) e.g. Unsplash or Pexels API, used to search for relevant photos or short video clips based on keywords extracted from the transcript.
  + These external services are accessed via their Python SDKs or REST calls (requiring API keys). They run asynchronously so as not to block the main web request thread.

By designing the system as above, we ensure clarity (each component has a well-defined role) and scalability. The web frontend can be deployed on a platform like Vercel (optimized for Next.js), and the FastAPI backend with MoviePy can run on a cloud VM or container (since video processing may need more memory/CPU). The use of R2 and a database means adding more backend instances or workers is straightforward without data loss or collision.

## **Database Schema (PostgreSQL)**

The PostgreSQL database uses a simple schema with two primary tables: one for users and one for videos. Below is a breakdown of the schema:

* Users – stores user accounts for Google-authenticated users. (Casual/anonymous users can use the app without an account, in which case videos will have no associated user.)
  + id (UUID or serial primary key): Unique user identifier.
  + google\_id (string): Google OAuth subject ID or email (unique).
  + name (string): User’s name (optional).
  + email (string): User’s email (for identification, unique if used).
  + created\_at (timestamp): Account creation time.
* Videos – stores metadata and links for each generated video.
  + id (UUID primary key): Unique video identifier used for sharing (e.g. used in /generate/{id} URL). A UUID ensures unguessable URLs for privacy.
  + user\_id (foreign key to Users, nullable): Owner of the video (null if not logged in).
  + original\_filename (string): Original video file name (if applicable, for reference).
  + caption\_text (text or JSON): The transcription of speech (could be full text or JSON with timestamps).
  + caption\_srt (text, nullable): SubRip subtitle content (optional, if we choose to also store/generate an .srt file).
  + broll\_option (string): “AI” or “Stock” – which B-roll generation method was used.
  + topics (text, nullable): Key topics or keywords extracted (could be used for searching or for image generation prompts).
  + video\_url (string): URL or path to the final MP4 in R2 (could also store R2 object key).
  + thumbnail\_url (string): URL/path to a thumbnail image (could be first frame or a generated preview image).
  + status (string): Processing status (pending, processing, completed, error). Allows the frontend to know if video is ready.
  + created\_at (timestamp): Time the video was requested.
  + completed\_at (timestamp, nullable): Time the video processing finished.

Note: Additional tables or fields can be added for logging, admin, or billing if needed (not required for this MVP). For scalability, using UUIDs for video IDs allows direct sharing without revealing sequence or requiring logged-in status. The caption\_text or topics field will help in generating image prompts or search queries.

## **API Endpoints (FastAPI)**

The FastAPI backend will expose a set of RESTful endpoints under a namespace (e.g. /api). Each endpoint’s request/response schema is clearly defined for integration with the Next.js frontend. All responses will be in JSON except the actual video file download (though we primarily serve videos via R2 URLs, not through the API directly). Below is a specification of key endpoints:

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /api/videos | POST | Upload a video for processing. Expects a video file (multipart form data) and parameters (e.g. chosen B-roll option). Returns a JSON with a new video\_id and initial status. This triggers asynchronous processing in the background. *(Authentication:* optional; if user is logged in, include their token to link the video to their account.)\* |
| /api/videos/{id} | GET | Get video status & info. Returns JSON containing metadata for video with id {id} – e.g. status (processing or completed), caption text (if ready), final video URL (if completed), thumbnail URL, etc. This is used by the frontend to poll for completion or to load video details. Publicly accessible if {id} exists (the ID itself is unguessable). If video not found or not ready, returns appropriate error or status. |
| /api/videos/{id}/download | GET | (Optional) Proxy to download the final video file. In most cases, the frontend can directly use the video\_url from the info endpoint (which points to R2). But this endpoint could be used to add custom headers or force download. Could be omitted if direct R2 links are used. |
| /api/users/me | GET | Get current user profile. Returns user info and perhaps saved video IDs. Requires auth (Google OAuth session or token). This helps the frontend know if user is logged in and display their data. |
| /api/users/me/videos | GET | List user’s videos. Returns a list of video metadata (id, thumbnail, status, created\_at) for the authenticated user. Used on the user’s dashboard page. |
| /api/auth/login & /api/auth/callback | GET/POST | Google OAuth handshake. (Only needed if handling auth in FastAPI; however, we plan to use Next.js for auth, see Frontend section.) If implemented on backend, these would initiate OAuth with Google and handle the callback to create a session or JWT. Given time constraints, we rely on Next.js for auth, so these endpoints may not be needed. |

Endpoint Implementation Details:

* POST /api/videos (Upload): This is the core endpoint where a user’s video file is received. It will accept the file as UploadFile (FastAPI’s form data type) and a JSON body or form fields indicating the chosen B-roll mode (“AI” or “Stock”). On receiving the request, the backend will:
  + Validate the file (size/type limits, ensure it’s a short video).
  + Save the raw video file. Ideally, this is saved directly to Cloudflare R2 to avoid storing large files on disk. We can use boto3 (AWS SDK) with R2’s S3 endpoint to upload the file stream directly. For example, using s3\_client.upload\_fileobj(file.file, bucket, object\_name) to stream the upload to R2.
  + Create a database entry for the new video (status = “processing”) with a new UUID. If the user is authenticated (backend can identify via a token/cookie), link the user\_id.
  + Kick off an asynchronous background task to perform the processing (transcription + B-roll + composition). In FastAPI, this can be done with BackgroundTasks or an external task queue (like Celery or RQ). Given the 10-hour implementation window, using FastAPI’s built-in background task or threading is simplest: e.g. background\_tasks.add\_task(process\_video\_job, video\_id, options).
  + Immediately respond to the client with JSON: {"id": video\_id, "status": "processing"}. The client can then redirect the user to the /generate/{id} page to wait for the result.
* GET /api/videos/{id} (Status/Info): The processing task will update the database once complete, so this endpoint can check DB for the video’s status. If status == completed, it returns metadata including:
  + video\_url: a URL to access the final video (likely a signed URL or public link to R2).
  + thumbnail\_url: URL of the thumbnail image.
  + caption\_text or caption\_segments: the transcription (could be used for display or debugging).
  + created\_at, completed\_at, etc., and possibly user info if needed.
* If processing is still ongoing, return {"status": "processing"} (optionally include a progress percentage if we have one, though not required for MVP). The frontend will poll this endpoint periodically (e.g. every few seconds) to see if processing is done so it can update the UI accordingly.
* Authentication: Since login is optional, most endpoints (like fetching a video by id) do not require auth – any user with the link can get the video info. For user-specific endpoints (like listing one’s own videos), we will require the request to include an auth token or session cookie. For simplicity, if using NextAuth on the frontend, the client can either:
  + Use Next.js API routes to securely fetch data (with the session) and then pass to the page.
  + Or include a JWT in the request (NextAuth can issue JWTs) that the FastAPI backend validates. Due to time constraints, a simpler approach is to rely on Next.js’s server side to fetch /api/users/me/videos with an API key known to both, or skip implementing this until needed.
* We will prioritize core functionality (video generation) over fully integrated auth for the MVP. We ensure, however, that if auth is present, it’s respected in the relevant endpoints.
* Error Handling: The API should handle error cases gracefully. For example, if Whisper fails or generation API returns nothing, we update the video status to “error” and include an error message. The GET /api/videos/{id} would then return status “error” with a message for the frontend to display.

## **Video Processing Pipeline (Integration Strategy)**

Once a video is uploaded and the background task is triggered, the following steps occur on the backend to generate the final video. This pipeline is the heart of the application, leveraging Whisper, image generation/APIs, and MoviePy:

### **1. Audio Transcription (Whisper Integration)**

* Audio Extraction: Using MoviePy or FFmpeg, we extract the audio track from the uploaded video. MoviePy’s VideoFileClip can directly give the audio clip: e.g. video = VideoFileClip('video.mp4'); audio = video.audio; audio.write\_audiofile('audio.wav'). This separates audio for transcription. (If using OpenAI’s Whisper API, we could even send the video file directly, but extracting audio ensures compatibility and smaller upload.)
* Transcription via Whisper: We have two possible approaches:
  + *OpenAI Whisper API:* Easiest for a quick implementation – send the audio file to OpenAI’s API (requires an API key) and get back the transcription (and possibly timestamps). The API is asynchronous (HTTP call) but we can call it within our background task and wait for the response. This ensures high accuracy with minimal setup (just install openai Python SDK).
  + *Local Whisper model:* Use the whisper Python library or openai-whisper to run on the server. Given the time constraint and potential lack of GPU, this can be slow for longer audio or large models. For a short video (e.g. 30-60 seconds) it might be fine to use a smaller model (like base or small). The local method gives us detailed timestamps for each sentence/word. If using this, ensure to install the model and possibly run it in a separate thread so as not to block other tasks.
* Captions with Timestamps: Whisper provides timestamps for segments of text. We will convert these into caption segments (like subtitle entries). For example, Whisper might return a list of segments: [{"start": 1.0, "end": 4.5, "text": "Hello world"}, ...]. We can format this into SRT or just keep it in Python as a list of (start, end, text). These will be used to overlay text on the video with precise timing. The Whisper AutoCaption project demonstrates this end-to-end: it *“rips the audio from the input video, uses Whisper to generate timestamped subtitles, and then MoviePy overlays these subtitles into the video”*. We will follow a similar approach for captioning.
* *Note:* If translation to another language was desired, Whisper can also translate, but our scope is just same-language transcription (e.g. English captions for English speech) unless extended.

### **2. B-roll Image Generation or Retrieval**

After obtaining the transcript or key captions, the next step is to get relevant visual content for B-roll. The user can choose Option A: AI-generated images or Option B: stock images/videos. The implementation for each:

* Option A – AI-Generated Images: For each caption or for the overall video theme, we generate an image:
  + We will use an image generation API (to save time instead of hosting a model). For example, OpenAI’s DALL·E API can generate an image from a text prompt. We could use the entire transcript or chunk it by caption to create specific prompts. A simple strategy is to take each subtitle text and use it as a prompt (or a slightly expanded prompt). For instance, if one caption is “The stock market is volatile today”, we send that to the image API to get an illustrative image. Another approach is to extract keywords or the overall topic (e.g. via a simple key-phrase extraction or using GPT-4 to summarize the video) and generate a few images that represent the main points.
  + Given time constraints, an easy method: use the first one or two caption segments as the prompt for one image, and maybe the next segments for another image, etc. If the video is short (say 30s, maybe 3-4 caption segments), generating one image per segment is reasonable.
  + Integration: The FastAPI task will call the OpenAI Image API (via openai.Image.create(prompt="...")) or Stability API. The API will return a URL or base64 image. We download the image file (or keep the URL if accessible) and either store it temporarily (possibly upload to R2 as well) or keep in memory for MoviePy processing.
* Option B – Stock Media (Unsplash/Pexels): Instead of generating new images, we fetch existing visuals:
  + Use Unsplash API or Pexels API for photos (or even short videos from Pexels if we dare include video B-roll). We’ll need an API key (both have free tiers).
  + For each key caption or keyword, perform a search. For example, if the caption text is “The stock market is volatile today”, we might pick keywords “stock market” or use the whole text as query. The Unsplash API endpoint for search looks like: GET https://api.unsplash.com/search/photos?query=stock%20market&client\_id=YOUR\_ACCESS\_KEY. It returns JSON with results (urls to images in various sizes).
  + We parse the JSON and pick one image (e.g. the first result). We then download that image (to feed into MoviePy). Alternatively, Pexels can provide videos – but integrating video B-roll adds complexity (we’d have to possibly trim and overlay video clips; given 10-hour limit, we will stick to images for now).
  + Each fetched image might need to be resized or cropped in MoviePy to match the video frame size (we can handle that in composition stage).
* Matching Images to Captions/Timeline: We should decide how to align B-roll with the video:
  + One simple approach: overlay each image during the time its corresponding caption is being spoken. For example, if caption 1 goes from 0:01 to 0:04, show image1 during that interval; caption 2 from 0:04 to 0:08, show image2, etc. This ensures the B-roll visuals appear in sync with the spoken content. We can achieve this by making the image a clip in MoviePy that spans the same start/end as the caption segment.
  + Alternatively, to avoid covering the entire video with images (which would completely hide the talking avatar), we might choose to only overlay on certain segments (e.g. every other caption) or for a portion of the duration to cut back to the speaker’s face occasionally. For MVP simplicity, we can overlay for each segment – this effectively creates a video where the user’s face video plays, but whenever a caption is being spoken, a relevant image is shown instead. The audio of the user remains continuous.
  + We will implement it such that the audio from the original video is always used, and we just switch the visual track between the user’s footage and the B-roll images. This gives a news-report style effect.

### **3. Video Composition (MoviePy Integration)**

With captions (text + timestamps) and B-roll images (with their intended display times) ready, we use MoviePy to compose the final video:

* Loading Base Video: Load the user’s original video using MoviePy’s VideoFileClip. This contains both video and audio. We’ll use the audio from this clip throughout. We might actually create a base “background” clip that’s just a black screen with the audio, or simply use the video clip and overlay on top (either way works).
* Creating Caption TextClips: For each caption segment, we create a text overlay. MoviePy’s TextClip can render text to an image frame, which we can then place on video. However, doing this for every frame manually is tedious – instead, MoviePy offers SubtitlesClip which can take subtitles (e.g. from an SRT file or list of (start, end, text)) and generate a video clip of text overlays. The Paperspace tutorial shows usage of a generator = lambda txt: TextClip(txt, font=..., fontsize=..., color='white', stroke\_color='black', ...) and then SubtitlesClip(subs, generator) to produce a clip that can be composited. We will do something similar:
  + Define style (font, size relative to video width, white text with black border for readability – a common subtitle style).
  + Use our list of subtitles to create a subtitles clip. This clip essentially is transparent everywhere except when/where text should appear.
  + Alternatively, for simplicity (since the video is short), we could manually iterate through caption segments, create a TextClip for each, and set its start and duration (end-start). This yields a list of text clips.
* Creating Image Clips: For each B-roll image, create an ImageClip using MoviePy. Set the clip’s duration to the desired length (e.g. the caption segment length) and use .set\_start(t) to specify when it should appear in the timeline. Also, ensure it’s the same resolution as the final video (we can resize as needed). To cover the full frame (hiding the speaker video), we set position to center and size to fill the frame.
* Composite and Concatenate:
  + We can overlay images on the original video by using CompositeVideoClip. For example, if base\_clip is the original video and img\_clip is an overlay image starting at 3s for 4s, doing CompositeVideoClip([base\_clip, img\_clip]) will result in the image covering the base at that interval. We will actually have multiple image clips and multiple text clips. CompositeVideoClip can take a list of all layers.
  + Approach 1: Composite the original video with all text clips and image clips in one go. All overlays are placed on the base video. This is straightforward: final\_clip = CompositeVideoClip([base\_video\_clip, \*subtitle\_clips, \*image\_clips]). The text clips will overlay as subtitles (likely positioned at bottom center), and image clips will overlay covering the entire frame (position=(“center”, “center”)) at their respective times.
  + Approach 2: Alternatively, create separate video segments and concatenate:
    - For each caption segment, create a clip of either the original video (for times when we want to show the speaker) or the image (for times we want B-roll) with the text overlay. Then concatenate these segments. This approach is more complicated programmatically, so given the time we stick to Approach 1.
  + We need to ensure all clips have the same resolution. MoviePy’s concatenate can auto-resize if needed, but for Composite it expects same size for base and any overlay (or will align top-left if smaller). We’ll use VideoFileClip(..., target\_resolution=(H, W)) to force the base video to a certain size (or get its size and use that for images).
* Rendering Final Video: Use final\_clip.write\_videofile("out.mp4", codec="libx264", fps=24, audio\_codec="aac"). This will encode the video. MoviePy uses FFMPEG under the hood. Since the videos are short, this should be reasonably fast (a few seconds). We must ensure the server has ffmpeg available (MoviePy may require ImageMagick for TextClip; if not available, we might use a workaround with PIL as in some tutorials, but given the timeline, we assume we can get TextClip working by installing ImageMagick or using a simpler font).
* Performance and Async: The background task will handle this synchronously (within itself). We need to be careful with memory – loading video and images is memory intensive. But with short video and maybe 3-5 images, it should be fine. If memory issues or speed issues arise, one can consider optimizing (e.g. lowering resolution or using a more efficient video pipeline).

By the end of this step, we have the final video file ready (with audio, captions burned-in, and cutaway images). This is essentially implementing a programmatic video editing workflow. The pipeline we follow is akin to the one in the Whisper AutoCaption project which *“uses Whisper and MoviePy to take in a video, extract its audio, convert speech into text captions, and then add those captions at the correct timeslots back to the original video”* – we extend it by also adding images.

### **4. Saving and Returning Result**

* Once out.mp4 is produced, the backend task will upload this file to Cloudflare R2 (similar to the input video). Using boto3, we’ll do s3\_client.upload\_file("out.mp4", bucket, final\_object\_key). The object key could be something like videos/{video\_id}/final.mp4.
* We also generate a thumbnail – possibly using MoviePy (e.g. take the first frame or a frame where an image is visible). MoviePy’s VideoFileClip allows saving a frame: clip.save\_frame("thumb.png", t=1.0). We can then upload this to R2 as well.
* Update the database: set status to “completed”, fill in video\_url (which could be the R2 public URL or a path we know how to serve), thumbnail\_url, and completed\_at.
* Optionally, free up local disk (delete temp files) to keep the environment clean.

The backend processing is now done, and the video is ready to be served to the user.

## **File Storage Flow (Cloudflare R2 Integration)**

Using Cloudflare R2 for file storage ensures large media files are handled efficiently. Here’s the plan for integrating R2:

Setup: We’ll create an R2 bucket (e.g. named ai-videos) and obtain API credentials (Access key ID and Secret). R2 is S3-compatible, so we can use Amazon’s boto3 library. We must configure the boto3 client with the R2 endpoint (unique to our Cloudflare account). For example:  
import boto3

s3 = boto3.client('s3',

endpoint\_url="https://<ACCOUNT\_ID>.r2.cloudflarestorage.com",

aws\_access\_key\_id=ACCESS\_KEY\_ID,

aws\_secret\_access\_key=SECRET\_KEY)

* This s3 client will be used for all uploads/downloads.
* Uploading Files: As noted earlier, when a video is uploaded via the API, we directly stream it to R2 to avoid storing locally. We use s3.upload\_fileobj() with the file-like object from FastAPI’s UploadFile. We will choose a key naming convention that groups files by video ID:
  1. e.g. uploads/{video\_id}/source.mp4 for original video,
  2. outputs/{video\_id}/final.mp4 for the final video,
  3. outputs/{video\_id}/thumb.jpg for the thumbnail,
  4. possibly outputs/{video\_id}/caption.srt if we save subtitles. This structure makes it easy to list or manage related files.
* Accessing Files: We have two ways to serve the final video to users:
  1. Public Bucket/Unsigned URLs: We could set the R2 bucket or objects to be public (or use Cloudflare’s CDN link) so that we can simply embed the video via a URL. For example, https://<ACCOUNT\_ID>.r2.cloudflarestorage.com/ai-videos/outputs/{id}/final.mp4. If this URL is accessible, the frontend can directly use it in a video player or as a download link. The app can still keep the id unguessable for security through obscurity.
  2. Signed URLs: Alternatively, generate a presigned URL that is valid for a short time (e.g. 1 hour) using s3.generate\_presigned\_url('get\_object', Params={...}). This way, even if the bucket is private, the user with the link can download/stream it. The backend can generate this URL on request (when the user goes to the video page).
  3. Proxy through API: As mentioned, we could have an endpoint that reads from R2 and streams to client, but that is less efficient and unnecessary unless we want to add access control logic.
* For simplicity, we might make the output bucket public (or at least the objects world-readable). Important: Do not expose any PII or user info in the URLs – using the UUID is fine.
* Storage of Images: The generated/fetched images used for B-roll could be saved to R2 as well (e.g. outputs/{id}/image1.png etc.) or we can avoid that by directly using them in memory and not retaining after video is made. To save time and cost, we likely won’t store them long-term (since they are visible in the final video anyway). If needed, we could keep them for audit or regeneration purposes.
* Cleanup: In a longer-term scenario, we might want a job to clean up old files or at least allow users to delete videos which would remove their entries and R2 objects. For now, we won’t implement deletion (except perhaps if a user re-generates or we want to clear incomplete ones).

Using R2 provides virtually unlimited storage and bandwidth via Cloudflare’s network, which is ideal for serving video content globally. This ensures the final video playback is smooth and not taxing our backend server.

## **Frontend Design (Next.js)**

The frontend built in Next.js will provide a smooth user experience for both anonymous and logged-in users. The design will focus on ease of use: minimal steps to go from uploading a video to watching the AI-enhanced result. We will use React hooks and possibly a context for user auth state. The UI components will be structured for reusability and clarity. Below is a breakdown of pages and key components:

### **Pages / Routes (Next.js)**

| **Page** | **Route** | **Description & Features** |
| --- | --- | --- |
| Landing / Home | / | The entry point of the app. This page introduces the app’s purpose (maybe a tagline like “Generate videos with AI captions and B-roll in seconds”). It will have a prominent upload/record widget for users to start creating a video. If the user is not logged in, it offers a “Login with Google” button in the header or in the flow (optional). If the user is logged in, it might show a “My Videos” link to their dashboard. |
| Upload | /upload *(optional)* | If separating the flow, this page (or section on the landing page) contains the video input form. Users can either select a video file or record using their webcam. After selecting/recording, the user chooses the B-roll option (AI images or Stock images). This could be a toggle or two buttons. Then a submit action triggers the upload to backend (via /api/videos). In a simplified UX, we may combine this with the landing to reduce page transitions (i.e. do it directly on /). |
| Processing | /processing *(optional)* | A transitional page or modal that appears after uploading. It might simply say “Your video is being processed...”, and maybe show a spinner and progress (if available). In practice, we might skip a dedicated route for this and instead immediately redirect the user to the generate page for their video ID, which can show a loading state until ready. That approach is simpler: after upload, go to /generate/{id} directly and handle both loading and result display in one page. |
| Result Viewer | /generate/[id] | A dynamic route to display a generated video by its ID. This page fetches video details from the backend (GET /api/videos/{id}) either via SSR (getServerSideProps) or on the client side (using SWR or react-query for polling). It shows the video player with the result. Features on this page: video player (HTML5 video element) with controls (play/pause, seek, volume), and the captions burned-in on the video. We also show the captions text below or as a transcript (optional). A Download button/link is provided to save the video file. Basic metadata like “Created by [username or 'Anonymous'] on [date]” can be shown. If the video is still processing, the page will show a loading indicator and periodically check the API until status is “completed”, then it will replace the placeholder with the actual video. The page is shareable – anyone with the URL can view the video (no login required to view). |
| Login | /login | A page to initiate Google OAuth login if we’re not using a popup. This page will simply have a “Sign in with Google” button (or automatically redirect to Google). We’ll use NextAuth, so actually in Next.js, the [...nextauth].js API route and NextAuth library will handle the OAuth callback. Thus, we might not need a dedicated page – instead NextAuth provides its own default login page or we can customize it. For the plan, we note that clicking login triggers Google OAuth flow. |
| Dashboard | /dashboard | This page is for logged-in users to see their history of generated videos. It will list recent videos with their thumbnail, title (if any, or just “Video created on date”), status (or a link to view). The data comes from /api/users/me/videos (which the Next.js server can fetch using the user’s session token). Each item links to /generate/{id}. This page requires auth – if not logged in, it redirects to /. Logged-in status can be managed by NextAuth (which provides a session object to pages). |

*(Note: For a 10-hour implementation, the Dashboard is a nice-to-have; the core flows can work without it. But planning for it ensures the app is user-friendly for registered users.)*

### **Components and UI Elements**

* Video Recorder/Uploader Component: On the landing/upload page, we provide a component for video input:
  + If using file upload: a simple <input type="file" accept="video/\*"> that triggers a file picker. We can capture the file and display a preview or the filename.
  + If using webcam record: utilize the MediaRecorder API via a React component (there are libraries like react-media-recorder that simplify this). The recorder will allow the user to start/stop recording short video from their webcam/microphone. Once stopped, we get a Blob of the video which we can then treat as a file for uploading.
  + UI-wise, perhaps show two tabs or buttons: “Upload Video” and “Record Video”.
  + This component also includes the dropdown or toggle for B-roll option: e.g. a segmented control or two radio buttons for “Auto-generate visuals with AI” vs “Use stock imagery”. We might default to one but allow switching.
  + A Submit button or form submission triggers the upload. We’ll handle this with an API call via fetch or Axios to our FastAPI endpoint. We should provide feedback during upload (a progress bar if possible). Next.js can handle form posts too, but using fetch gives more control.
  + If using Next.js API routes as a middleman (not strictly necessary), we could send the file to a Next.js API route which then forwards to FastAPI. However, to save time, we’ll have the frontend directly call the FastAPI endpoint (CORS must be enabled on FastAPI to allow this).
* Video Player Component: On the result page, a component responsible for displaying the video:
  + This will likely just wrap an HTML <video> element with appropriate attributes. We set the src to the video URL we get (which could be the R2 URL or a proxied API).
  + Add controls attribute for user control. We can also include crossOrigin="anonymous" if needed (for any cross-origin video serving, though R2 can be configured with correct CORS).
  + Since captions are burned in, we do not need to load a separate track file. If we had an SRT and wanted optional captions, we could use <track> tag, but here it’s not needed.
  + This component might also handle showing a placeholder or loading indicator if the video isn’t ready yet (in case we use it also for the loading state).
  + Additionally, a Download button can be either a simple anchor <a href={videoURL} download> or a button that triggers fetch. Because of cross-origin, we might need to ensure the videoURL has proper CORS or uses our domain (if we proxy).
  + A Thumbnail image can be shown while loading or as a poster on the video element (<video poster={thumbURL}>).
* Login Button/Component: If using NextAuth, we can utilize their signIn('google') method. For example, a button that calls signIn('google') will redirect to Google. NextAuth handles callbacks under the hood with its own API routes (/api/auth/\*). The front end just needs to provide a UI element. If not using NextAuth, we could have the button link to our FastAPI /api/auth/login endpoint or use Google’s client-side OAuth flow. To minimize custom work, NextAuth is recommended. Integrating NextAuth takes maybe ~1 hour: configure a provider with client ID/secret, and wrap the app in a . NextAuth supports Google out-of-the-box – we only need to supply credentials and it will manage the OAuth token exchange, returning a session containing user info. We’ll use NextAuth’s provided hooks like useSession() to check login state and protect the dashboard route.
* Dashboard List Component: On the dashboard page, map through the list of videos and display each as a card or list item. Could simply show the thumbnail image, maybe the date or caption snippet as title, and a “View” button (which is a Link to /generate/{id}). If a video is still processing (unlikely by the time user sees dashboard, but possible if they started one and left), show a label “Processing” or a spinner on that item.
* Styling and Layout: We aim for a modern, clean design. Using a CSS framework or component library can speed this up (Tailwind CSS, Chakra UI, or MUI). However, given the 10-hour window, a lightweight approach like Tailwind might be fastest for custom design without spending time on theming. We’ll also ensure the layout is responsive (flexbox or grid as needed). The main pages are fairly simple: a centered form on landing, a video player on result, etc. A top navigation bar with the app name and Login/Logout button on the right (if logged in, maybe show user’s name or avatar from Google and a logout).
* Error messages and edge cases: The frontend should handle cases like:
  + File too large or wrong format (could check client-side and show a message).
  + If the backend returns an error (e.g. Whisper fails), the /generate/{id} page could display “An error occurred while generating the video. Please try again.” possibly with a retry option (maybe allow them to re-submit).
  + If a user tries to access a video ID that doesn’t exist, the page should show “Video not found” (for SSR, check status code from API, for CSR, handle 404 in fetch).
  + Multi-language: not a priority for MVP (focus English UI).

In summary, the Next.js frontend will provide a seamless flow: Landing -> Upload -> [Processing] -> Result, with optional Login and Dashboard enhancements for returning users. The use of Next.js also means we can prerender pages for better performance. For instance, we can use SSR on /generate/[id] to fetch video info (especially if it’s already done) so that the page loads with the video ready. This might be advanced for 10 hours, so we can start with client-side data fetching and add SSR if time permits.

## **Integration Considerations & External Services**

Integrating the external services (Whisper, image generation, stock API) is crucial but we will use them at a high level to save development time:

Whisper API vs Local: Given the time constraint, using the OpenAI Whisper API is recommended (no need to handle model downloading and it’s faster on OpenAI’s servers). The Python OpenAI library can send a file like:  
import openai

audio\_file = open("audio.wav", "rb")

transcript = openai.Audio.transcribe("whisper-1", audio\_file)

text = transcript["text"]

This returns the text transcription. If we need word-level timings, the API might not return that, so alternatively we use the open source model:  
import whisper

model = whisper.load\_model("small")

result = model.transcribe("audio.wav")

text = result["text"]

segments = result["segments"] # list of segments with start, end, text

* The segments give us timing for subtitles.

Image Generation API: OpenAI’s image API usage example (in Python):  
response = openai.Image.create(prompt="A scenic view of a bustling stock market trading floor", n=1, size="1024x1024")

image\_url = response['data'][0]['url']

* We can get the URL and download it (using requests or urllib). StabilityAI’s API or others have similar usage. We should be mindful of API rate limits and add a key check in config.

Unsplash API: For example, using requests:  
import requests

headers = {"Authorization": "Client-ID YOUR\_UNSPLASH\_ACCESS\_KEY"}

res = requests.get("https://api.unsplash.com/photos/random?query=KEYWORD", headers=headers)

data = res.json()

image\_url = data['urls']['regular'] # get a reasonably sized image

* Or use search endpoint for more control. Pexels similarly provides an Authorization header and endpoints.
* MoviePy and ImageMagick: One risk: MoviePy’s TextClip requires ImageMagick or a font. We should install ImageMagick on the server (or in the requirements, maybe use a smaller trick like using PIL as shown in an example to generate text overlays). To be safe, we will attempt to use TextClip and document that if ImageMagick isn’t available, an alternative is to use PIL to draw text onto an image and then use that image as an overlay (like the dev.to example did for overlay text).
* Background Task Execution: As the processing might take some time (transcription and video encoding), it should not block the FastAPI main thread handling web requests. We will use background\_tasks or possibly run a separate Celery worker if time permits. But given single-developer and time, simplest is to use FastAPI’s ability to spawn a background task on an event loop (note: if Whisper local is CPU heavy, it could still block other requests if using same process). Another approach is to use ThreadPoolExecutor to offload heavy CPU tasks. Since the expected usage is low initially, this is acceptable. For more scalability, one would deploy a worker service for these jobs, but that’s beyond our immediate scope.
* Security and Keys: All API keys (OpenAI, Unsplash, Google OAuth secrets) will be stored in environment variables and not exposed to the client. Only the NextAuth (frontend) will have the Google client ID (which is fine to expose) and the backend has the client secret. The OpenAI and Unsplash keys remain in backend config.

## **Timeline and Scalability Notes**

Given a ~10 hour development window for an MVP:

* We prioritize implementing the core pipeline (upload -> captions -> one image -> combine -> output) first, ensuring end-to-end functionality for a single option (e.g. start with stock images which might be simpler than dealing with AI API quotas).
* Then integrate the alternative option (AI generation) and add polish (multiple images, better styling).
* Authentication can be integrated using NextAuth in under an hour, but if time is extremely tight, we can allow anonymous usage only and add auth later. However, planning for it (as above) ensures the architecture supports it seamlessly.
* The UI can start simple (no fancy design) then we can incrementally add styles. Because Next.js hot-reloads, we can iteratively refine the components quickly.
* The system as designed is scalable: in the future, heavy processing can be offloaded to a separate service or queue workers (the API would then just enqueue a job and quickly respond). The use of R2 and Postgres makes horizontal scaling easier (shared storage and DB).
* Each video generation might take e.g. 20-30 seconds (assuming Whisper small model and 30s video). This is acceptable for an MVP. If needing faster, one can use Whisper API (faster) or faster-whisper library with GPU.

By following this plan, we ensure a clear separation of concerns (frontend vs backend, processing vs storage) and maintainability. A single developer can build each piece without confusion, and the integration points (API JSON, file locations, etc.) are well-defined. The end result will be a web application where a user can quickly turn a raw talking video into an enhanced clip with subtitles and illustrative cutaway imagery, demonstrating the power of AI in multimedia content creation.