Moving the video cleaning script to a serverless, event-driven architecture using AWS Lambda and S3 is a common and powerful pattern.

However, there's a **major challenge** we need to address upfront: **FFmpeg**. Standard Lambda runtimes do not include the FFmpeg binary. Your Python script relies heavily on calling the `ffmpeg` executable via `subprocess`.

To make this work in Lambda, we need to package FFmpeg along with your function. The most common way is using **Lambda Layers** or **Lambda Container Images**. For this lab, we'll focus on using a **Lambda Layer**, as it's often simpler for adding binaries.

This lab guide will be detailed, assuming you have basic familiarity with AWS concepts but providing specific steps.

Serverless Video Cleaning Lab: S3 -> Lambda (with FFmpeg) -> S3

Goal: Automatically clean MP4 videos uploaded to an S3 bucket using the Python script (adapted for Lambda) and store the cleaned result in another S3 bucket.

Architecture:

1. **Source S3 Bucket:** User uploads `.mp4` files here.
2. **S3 Event Notification:** Detects `.mp4` uploads and triggers the Lambda function.
3. **AWS Lambda Function:**
    - Runs the Python code (with OpenCV, NumPy).
    - Uses an **FFmpeg Lambda Layer** to access the FFmpeg binary.
    - Downloads the source video from S3 to its temporary storage (`/tmp`).
    - Executes the cleaning logic (motion detection, segment identification, FFmpeg calls).
    - Uploads the cleaned video to the destination S3 bucket.
4. **Destination S3 Bucket:** Stores the processed, cleaned `.mp4` files.
5. **IAM Role:** Grants the Lambda function necessary permissions (S3 access, CloudWatch Logs).
6. **CloudWatch Logs:** Captures logs from the Lambda function for monitoring and debugging.

Prerequisites:

- AWS Account with permissions to create S3 buckets, Lambda functions, IAM roles, and CloudWatch Logs.
- AWS CLI installed and configured (`aws configure`).
- Python 3.8+ installed locally.
- `pip` for installing Python packages.
- A sample `.mp4` video file for testing.
- **Crucially:** Access to a static FFmpeg binary compatible with Amazon Linux 2 (the standard Lambda runtime environment). See Episode 2.

Module 1: Setting up S3 Buckets

Goal: Create two S3 buckets: one for source videos and one for cleaned output videos.

Steps:

1. **Navigate to S3:** Open the AWS Management Console and go to the S3 service.
2. **Create Source Bucket:**
   - Click "Create bucket".
   - Enter a **globally unique** bucket name (e.g., `my-video-cleaning-source-YOUR_INITIALS-DATE`).
   - Choose an AWS Region (e.g., `us-east-1`). Remember this region for all other resources.
   - Keep default settings for Block Public Access (should be ON).
   - Disable Bucket Versioning for simplicity in this lab (optional).
   - Click "Create bucket".
3. **Create Destination Bucket:**
   - Click "Create bucket".
   - Enter a **globally unique** bucket name (e.g., `my-video-cleaning-destination-YOUR_INITIALS-DATE`).
   - **Important:** Use the *same AWS Region* as the source bucket.
   - Keep default settings (Block Public Access ON, Versioning disabled).
   - Click "Create bucket".
4. **Record Bucket Names:** Note down the exact names of your source and destination buckets. You'll need them later.

Module 2: Preparing the FFmpeg Lambda Layer

Goal: Create a Lambda Layer containing the FFmpeg static binary so our function can execute it.

Challenge: Getting a compatible FFmpeg binary.

- **Option A (Recommended for Linux/macOS users): Use a pre-built static binary:**
    - Go to a trusted source for Linux static builds, like John Van Sickle's FFmpeg Builds (https://johnvansickle.com/ffmpeg/) or BtbN's builds (https://github.com/BtbN/FFmpeg-Builds/releases).
    - Download a **static** build (not shared) for `amd64` architecture. Look for builds compatible with older GLIBC versions if possible, for maximum compatibility with Amazon Linux 2.
    - Extract the downloaded archive. You only need the `ffmpeg` executable file itself.
- **Option B (More complex): Build FFmpeg on an Amazon Linux 2 environment:** Use an EC2 instance running Amazon Linux 2 or a Docker container with an Amazon Linux 2 image to compile FFmpeg statically. This ensures perfect compatibility but is more advanced.
- **Option C (Check Community Layers):** Search the AWS Serverless Application Repository or public layer repositories for existing FFmpeg layers, but **use with caution**, verifying their source and contents.

Steps (Assuming you have the `ffmpeg` binary using Option A):

1. Create Layer Directory Structure: On your local machine, create the following directory structure:

```
ffmpeg_layer/
└── bin/
    └── ffmpeg  # Place the downloaded static ffmpeg binary here
```

- The `bin/` directory is crucial. Lambda automatically adds `/opt/bin` from layers to the execution environment's PATH.

2. Set Execute Permissions (Linux/macOS): Navigate into the `bin` directory in your terminal and make the binary executable:Bash

```
chmod +x ffmpeg
```

3. Zip the Layer Contents: Navigate back to the *outside* of the `ffmpeg_layer` directory (so `ffmpeg_layer` is inside your current directory). **Zip the contents of the `ffmpeg_layer`** directory:Bash

```
cd ffmpeg_layer
zip -r ../ffmpeg_layer.zip .
cd ..
```

(Ensure the `bin` directory is at the root level inside `ffmpeg_layer.zip`)

4. Publish the Lambda Layer: Use the AWS CLI (replace `ffmpeg_layer.zip`, region, and add a description):

```
aws lambda publish-layer-version \
   --layer-name ffmpeg-static \
   --description "Static FFmpeg binary for Lambda" \
   --zip-file fileb://ffmpeg_layer.zip \
   --compatible-runtimes python3.8 python3.9 python3.10 python3.11
python3.12 \
   --region YOUR_AWS_REGION
   # Example: --region us-east-1
```

5. Record Layer ARN: Note the `LayerVersionArn` output from the command. It will look something like `arn:aws:lambda:us-east-1:123456789012:layer:ffmpeg-static:1`.

Module 3: Preparing the Lambda Function Code

Goal: Adapt the Python script for Lambda, handle S3 events, and package it with dependencies.

Steps:

1. **Create Project Directory:** Create a new directory for your Lambda function code locally (e.g., `lambda_video_cleaner`).
2. **Save Lambda Handler Code:** Inside `lambda_video_cleaner`, create a file named `lambda_function.py` with the following code:

```
import boto3
import cv2
import numpy as np
import os
import subprocess
import traceback
import urllib.parse

# Initialize S3 client (best practice outside handler for potential reuse)
s3_client = boto3.client('s3')

# --- Configuration (Consider using Environment Variables in Lambda) ---
```

```
DESTINATION_BUCKET = os.environ.get('DESTINATION_BUCKET', 'DEFAULT_DEST_BUCKET_NAME') # Set via Lambda Env Vars
FFMPEG_PATH = "/opt/bin/ffmpeg" # Path provided by the Lambda Layer

# --- Parameters for the cleaning logic ---
# These could also be Environment Variables or part of event data if needed
ACTIVITY_THRESHOLD = 5.0
MIN_SEGMENT_DURATION_SEC = 1.0
SMOOTHING_WINDOW_SEC = 0.5
BUFFER_SEC = 1.0 # Example buffer

# --- Helper Functions (Adapted from your script) ---

def calculate_motion_score(frame1_gray, frame2_gray):
    """Calculates a motion score based on the Mean Absolute Difference."""
    diff = cv2.absdiff(frame1_gray, frame2_gray)
    score = np.mean(diff)
    return score

def clean_video_logic(input_path, output_path, activity_threshold, min_segment_duration_sec, smoothing_window_sec, buffer_sec=0.0):
    """
    Core logic to analyze video and identify segments.
    This function now expects paths within the Lambda environment (e.g., /tmp).
    It returns the list of segments to keep or raises an error.
    FFmpeg execution is handled separately after this returns.
    """
    print(f"Starting video analysis for: {input_path}")
    print(f"Parameters: Threshold={activity_threshold}, Min Duration={min_segment_duration_sec}s, Smoothing={smoothing_window_sec}s, Buffer={buffer_sec}s")

    # --- Phase 1: Video Analysis ---
    print("Phase 1: Analyzing video for motion...")
    cap = cv2.VideoCapture(input_path)
    if not cap.isOpened():
        raise IOError(f"Could not open video file for analysis:
{input_path}")

    fps = cap.get(cv2.CAP_PROP_FPS)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

    if fps <= 0 or frame_count <= 0:
        cap.release()
        raise ValueError(f"Invalid video properties (FPS: {fps}, Frames:
{frame_count}).")

    print(f"Video Info: {width}x{height}, {fps:.2f} FPS, {frame_count}
frames (Duration: {frame_count / fps:.2f}s)")

    motion_scores = []
    prev_frame_gray = None
    frame_num = 0

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        current_frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        # Use GaussianBlur from OpenCV for noise reduction
        current_frame_gray = cv2.GaussianBlur(current_frame_gray, (21, 21),
0)

        if prev_frame_gray is not None:
            score = calculate_motion_score(prev_frame_gray,
current_frame_gray)
            motion_scores.append(score)
        else:
            motion_scores.append(0)

        prev_frame_gray = current_frame_gray
        frame_num += 1
        # Lambda logs can get cluttered, reduce progress frequency or
remove
        if frame_num % int(fps * 30) == 0: # Log progress less often
            progress = (frame_num / frame_count) * 100
            print(f"  Analyzed frame {frame_num}/{frame_count}
({progress:.1f}%)")

    cap.release()
    print(f"\nAnalysis complete. Calculated {len(motion_scores)} motion
scores.")

    if not motion_scores:
        raise ValueError("No motion scores calculated.")

    # --- Phase 2: Segment Identification ---
    print("Phase 2: Identifying active segments...")
    smoothed_scores = None
    smoothing_window_frames = max(1, int(smoothing_window_sec * fps))

    if len(motion_scores) >= smoothing_window_frames and
smoothing_window_frames > 1:
        motion_scores_np = np.array(motion_scores, dtype=float)
        kernel = np.ones(smoothing_window_frames) / smoothing_window_frames
        smoothed_scores_conv = np.convolve(motion_scores_np, kernel,
mode='valid')

        if len(smoothed_scores_conv) > 0:
            pad_start_len = smoothing_window_frames // 2
            pad_end_len = len(motion_scores) - len(smoothed_scores_conv) -
pad_start_len
            padding_start = np.full(pad_start_len, smoothed_scores_conv[0])
            padding_end = np.full(pad_end_len, smoothed_scores_conv[-1])
            smoothed_scores = np.concatenate((padding_start,
smoothed_scores_conv, padding_end))
        else:
            smoothed_scores = np.array(motion_scores, dtype=float)
    else:
        smoothed_scores = np.array(motion_scores, dtype=float)

    if smoothed_scores is None:
        raise ValueError("Smoothed scores were not calculated.")

    if len(smoothed_scores) != len(motion_scores):
        raise ValueError(f"Logic error: Smoothed scores length mismatch.")

    active_frames = smoothed_scores >= activity_threshold
    raw_segments_to_keep = []
    in_active_segment = False
    start_frame = 0
    min_segment_frames = int(min_segment_duration_sec * fps)

    for i, is_active in enumerate(active_frames):
        if is_active and not in_active_segment:
            start_frame = i; in_active_segment = True
        elif not is_active and in_active_segment:
            end_frame = i
            if (end_frame - start_frame) >= min_segment_frames:
                raw_segments_to_keep.append((start_frame, end_frame))
            in_active_segment = False
    if in_active_segment:
        end_frame = len(active_frames)
        if (end_frame - start_frame) >= min_segment_frames:
             raw_segments_to_keep.append((start_frame, end_frame))

    if not raw_segments_to_keep:
        print("No active segments found meeting criteria.")
        return [], fps # Return empty list if no segments
    # Apply Buffer and Merge
    segments_to_keep = raw_segments_to_keep # Default if no buffer
    if buffer_sec > 0:
        buffer_frames = int(buffer_sec * fps); final_segments = []
        raw_segments_to_keep.sort(key=lambda x: x[0])
        current_start, current_end = raw_segments_to_keep[0]
        current_start = max(0, current_start - buffer_frames)
        current_end = min(len(active_frames), current_end + buffer_frames)
        for i in range(1, len(raw_segments_to_keep)):
            next_start, next_end = raw_segments_to_keep[i]
            adj_next_start = max(0, next_start - buffer_frames)
            adj_next_end = min(len(active_frames), next_end +
buffer_frames)
            if adj_next_start <= current_end: current_end =
max(current_end, adj_next_end)
            else: final_segments.append((current_start, current_end));
current_start = adj_next_start; current_end = adj_next_end
        final_segments.append((current_start, current_end))
        segments_to_keep = final_segments

    print(f"Identified {len(segments_to_keep)} final segments to keep.")
    return segments_to_keep, fps # Return segments and fps

def run_ffmpeg_command(command_list):
    """Runs an FFmpeg command using subprocess and handles errors."""
    try:
        print(f"Running FFmpeg command: {' '.join(command_list)}")
        # Increased timeout for potentially long operations
        result = subprocess.run(command_list, capture_output=True,
text=True, check=True, timeout=600)
        print("FFmpeg command stdout:")
        print(result.stdout)
        print("FFmpeg command stderr:")
        print(result.stderr)
        return True
    except subprocess.CalledProcessError as e:
        print(f"Error executing FFmpeg command: {e}")
        print(f"Command: {' '.join(command_list)}")
        print(f"Return code: {e.returncode}")
        print(f"Output (stdout): {e.stdout}")
        print(f"Output (stderr): {e.stderr}")
        return False
    except subprocess.TimeoutExpired as e:
        print(f"FFmpeg command timed out: {' '.join(command_list)}")
        print(f"Timeout: {e.timeout}s")
        print(f"Output (stdout): {e.stdout}")
        print(f"Output (stderr): {e.stderr}")
        return False
    except Exception as e:
        print(f"Unexpected error running FFmpeg: {e}")
        traceback.print_exc()
        return False


# --- Lambda Handler ---

def lambda_handler(event, context):
    """
    Main Lambda function handler triggered by S3 event.
    """
    print("Received S3 event.")

    # --- Basic Error Handling & Setup ---
    if not DESTINATION_BUCKET or DESTINATION_BUCKET ==
'DEFAULT_DEST_BUCKET_NAME':
         print("Error: DESTINATION_BUCKET environment variable not set.")
         return {'status': 'error', 'message': 'Destination bucket not
configured'}

    # Define temporary file paths in Lambda's writable directory
    tmp_dir = "/tmp"
```

```
   download_path = os.path.join(tmp_dir, 'input_video.mp4')
   output_path = os.path.join(tmp_dir, 'cleaned_video.mp4')
   file_list_path = os.path.join(tmp_dir, "mylist.txt")
   temp_segment_prefix = os.path.join(tmp_dir, "temp_segment_")

   try:
       # --- 1. Get Source Bucket and Key from Event ---
       record = event['Records'][0]
       source_bucket = record['s3']['bucket']['name']
       source_key =
urllib.parse.unquote_plus(record['s3']['object']['key'])
       print(f"Processing s3://{source_bucket}/{source_key}")

       # Simple check for video file extension
       if not source_key.lower().endswith(('.mp4', '.mov', '.avi')): # Add
other types if needed
           print(f"Skipping non-video file: {source_key}")
           return {'status': 'skipped', 'message': 'Not a target video
file type'}

       # Construct destination key (e.g., same name, or in a subfolder)
       destination_key = f"cleaned/{os.path.basename(source_key)}"

       # --- 2. Download Video from Source S3 ---
       print(f"Downloading s3://{source_bucket}/{source_key} to
{download_path}")
       s3_client.download_file(source_bucket, source_key, download_path)
       print("Download complete.")

       # --- 3. Analyze Video and Identify Segments ---
       segments_to_keep, fps = clean_video_logic(
           download_path,
           output_path, # Not used directly by logic, but good to pass if
needed later
           ACTIVITY_THRESHOLD,
           MIN_SEGMENT_DURATION_SEC,
           SMOOTHING_WINDOW_SEC,
           BUFFER_SEC
       )

       if not segments_to_keep:
           print("No active segments found to keep. Skipping FFmpeg processing.")
           # Optionally delete source or move to a different prefix
           return {'status': 'processed', 'message': 'No active segments found'}

       # --- 4. Execute FFmpeg Extraction and Concatenation ---
       print("Phase 3: Extracting and concatenating segments using FFmpeg...")
       temp_files = []
       success = True

       with open(file_list_path, "w") as f:
           for i, (start_f, end_f) in enumerate(segments_to_keep):
               start_time = start_f / fps
               duration = (end_f - start_f) / fps
               temp_output_path = f"{temp_segment_prefix}{i}.mp4"
               temp_files.append(temp_output_path)

               ffmpeg_extract_cmd = [ FFMPEG_PATH, '-ss', str(start_time), '-i', download_path, '-t', str(duration), '-c', 'copy', '-avoid_negative_ts', 'make_zero', '-y', temp_output_path ]
               if not run_ffmpeg_command(ffmpeg_extract_cmd):
                    print(f"    Warning: FFmpeg extraction (copy) failed for segment {i + 1}. Retrying with re-encoding.")
                    ffmpeg_extract_cmd_reencode = [ FFMPEG_PATH, '-ss', str(start_time), '-i', download_path, '-t', str(duration), '-y', temp_output_path ]
                    if not run_ffmpeg_command(ffmpeg_extract_cmd_reencode):
```

```python
                        print(f"    Error: FFmpeg extraction failed even with re-encoding for segment {i + 1}. Aborting.")
                        success = False
                        break # Stop processing further segments

                # Write successfully created filename to list for concatenation
                f.write(f"file '{os.path.basename(temp_output_path)}'\n")  # Use relative path for concat demuxer

        if not success or not temp_files or not os.path.exists(file_list_path) or os.path.getsize(file_list_path) == 0:
            raise RuntimeError("Segment extraction failed or produced no valid segments.")

        # Concatenate segments
        ffmpeg_concat_cmd = [ FFMPEG_PATH, '-f', 'concat', '-safe', '0', '-i', file_list_path, '-c', 'copy', '-y', output_path ]
        if not run_ffmpeg_command(ffmpeg_concat_cmd):
            print("  Warning: FFmpeg concatenation (copy) failed. Retrying with re-encoding.")
            ffmpeg_concat_cmd_reencode = [ FFMPEG_PATH, '-f', 'concat', '-safe', '0', '-i', file_list_path, '-y', output_path ]
            if not run_ffmpeg_command(ffmpeg_concat_cmd_reencode):
                raise RuntimeError("FFmpeg concatenation failed even with re-encoding.")

        print(f"Successfully created cleaned video locally: {output_path}")

        # --- 5. Upload Cleaned Video to Destination S3 ---
        print(f"Uploading {output_path} to s3://{DESTINATION_BUCKET}/{destination_key}")
        s3_client.upload_file(output_path, DESTINATION_BUCKET, destination_key)
        print("Upload complete.")
        return {'status': 'success', 'source_key': source_key, 'destination_key': destination_key}

    except Exception as e:
        print(f"!!! Critical Error processing {source_key} !!!")
        print(e)
        traceback.print_exc()
        # Optional: Move source file to an error prefix/bucket here
        return {'status': 'error', 'message': str(e)}

    finally:
        # --- 6. Cleanup Temporary Files ---
        print("Cleaning up /tmp directory...")
        for f in [download_path, output_path, file_list_path] + temp_files:
            if os.path.exists(f):
                try:
                    os.remove(f)
                    print(f"  Removed {f}")
                except OSError as e:
                    print(f"  Warning: Could not remove temp file {f}: {e}")
        print("Cleanup complete.")
```

3. **Create `requirements.txt`:** In the same `lambda_video_cleaner` directory, create `requirements.txt`:

```
4. numpy
5. opencv-python-headless
6. # boto3 is included in the Lambda runtime environment
```

    - We use `opencv-python-headless` because Lambda doesn't have a graphical interface.

4. Install Dependencies Locally: Navigate into the `lambda_video_cleaner` directory in your terminal and install the packages *into this directory*:

```
pip install -r requirements.txt -t .
```

- The `-t .` flag tells pip to install packages here, not globally.

5. Create Deployment Package: Create a ZIP file containing `lambda_function.py`, `requirements.txt`, and all the installed packages/folders from the previous step. Make sure `lambda_function.py` is at the root of the ZIP file.

```
zip -r ../lambda_deployment_package.zip .
```

(Run this command from inside the `lambda_video_cleaner` directory)

Module 4: Creating the Lambda Execution Role

Goal: Create an IAM role that grants the Lambda function permission to interact with S3 and CloudWatch Logs.

Steps:

1. **Navigate to IAM:** Go to the IAM service in the AWS Console.
2. **Create Role:**
   - Go to "Roles" and click "Create role".
   - **Trusted entity type:** Select "AWS service".
   - **Use case:** Select "Lambda". Click "Next".
3. **Add Permissions:**
   - Search for and select the `AWSLambdaBasicExecutionRole` policy (allows writing logs to CloudWatch).
   - Click "Create policy".
     - Select the "JSON" tab.

Paste the following policy JSON, **replacing** `YOUR_SOURCE_BUCKET_NAME` and `YOUR_DESTINATION_BUCKET_NAME` with the actual names you created in Episode 1.

JSON

```
   {"Version": "2012-10-17",
   "Statement": [
       {
           "Effect": "Allow",
           "Action": [
               "s3:GetObject"
```

```
            ],
            "Resource":
"arn:aws:s3:::YOUR_SOURCE_BUCKET_NAME/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource":
"arn:aws:s3:::YOUR_DESTINATION_BUCKET_NAME/*"
        }
    ]
}
```

        -
        - Click "Next: Tags", then "Next: Review".
        - Give the policy a name (e.g., `LambdaS3VideoCleaningPolicy`).
        - Click "Create policy".
    - Go back to the "Create role" browser tab/window. Refresh the policy list if needed, then search for and select the policy you just created (`LambdaS3VideoCleaningPolicy`).
    - Click "Next".
4. **Name Role:**
    - Enter a role name (e.g., `LambdaVideoCleaningRole`).
    - Review the trusted entities and policies.
    - Click "Create role".
5. **Record Role ARN:** Find the role you just created and note its ARN.

Module 5: Creating the Lambda Function

Goal: Create the Lambda function, configure it, and attach the code, layer, and role.

Steps:

1. **Navigate to Lambda:** Go to the Lambda service in the AWS Console.
2. **Create Function:**
    - Click "Create function".
    - Select "Author from scratch".
    - **Function name:** Enter a name (e.g., `VideoCleaningFunction`).

    - **Runtime:** Select "Python 3.12" (or 3.11, 3.10, 3.9, 3.8 - must match layer compatibility).
    - **Architecture:** Keep `x86_64`.
    - **Permissions:** Expand "Change default execution role". Select "Use an existing role" and choose the `LambdaVideoCleaningRole` you created in Episode 4.
    - Click "Create function".
3. **Upload Code:**
    - In the function's overview page, go to the "Code" tab.
    - Click "Upload from" and select ".zip file".
    - Upload the `lambda_deployment_package.zip` file you created in Episode 3.
    - Click "Save". (This might take a moment).
4. **Add FFmpeg Layer:**
    - Scroll down to the "Layers" section. Click "Add a layer".
    - Select "Specify an ARN".
    - Paste the `LayerVersionArn` for your FFmpeg layer (recorded in Episode 2).
    - Click "Verify" (optional but good practice), then click "Add".
5. **Configure Settings:**
    - Go to the "Configuration" tab, then "General configuration". Click "Edit".
    - **Memory:** Increase significantly. Start with **1024 MB** or **2048 MB**. Video processing needs memory.
    - **Timeout:** Increase significantly. Video processing takes time. Set it to **10 minutes** (or the maximum 15 minutes: `15` min `0` sec).
    - Click "Save".
6. **Add Environment Variables:**
    - Go to the "Configuration" tab, then "Environment variables". Click "Edit".
    - Click "Add environment variable".
    - **Key:** `DESTINATION_BUCKET`
    - **Value:** Enter the exact name of your destination S3 bucket.
    - Click "Save".

Module 6: Configuring the S3 Trigger

Goal: Set up the Source S3 bucket to automatically trigger the Lambda function when MP4 files are uploaded.

Steps:

1. **Navigate to Source S3 Bucket:** Go back to the S3 service and open your *source* bucket.
2. **Create Event Notification:**
    - Go to the "Properties" tab.
    - Scroll down to "Event notifications". Click "Create event notification".

  - **Event name:** Enter a name (e.g., `LambdaVideoTrigger`).
  - **Prefix (Optional):** If you only want uploads to a specific folder to trigger the Lambda, enter the folder name here (e.g., `uploads/`). Leave blank to trigger on uploads anywhere in the bucket.
  - **Suffix:** Enter `.mp4` (or `.mov` etc., depending on supported input types). This ensures only video uploads trigger the function.
  - **Event types:** Select `s3:ObjectCreated:Put`. You might also select `s3:ObjectCreated:CompleteMultipartUpload` for larger uploads.
  - **Destination:** Select "Lambda function".
  - **Lambda function:** Choose the `VideoCleaningFunction` you created.
  - Click "Save changes". (You might be asked to confirm Lambda permissions - this should be okay as the role grants S3 access).

Module 7: Testing the Setup

Goal: Upload a video and verify the entire process works.

Steps:

1. **Upload Video:** Go to your *source* S3 bucket in the AWS Console. Upload your sample `.mp4` video file (respecting any prefix you set in the trigger).
2. **Monitor Lambda Execution:**
   - Navigate to the Lambda service and find your `VideoCleaningFunction`.
   - Go to the "Monitor" tab. Click "View CloudWatch logs".
   - Look at the latest log stream. You should see log output from your function starting shortly after the upload completes. Look for messages about download, analysis, segment identification, FFmpeg execution, and upload.
3. **Check Destination Bucket:** After a short while (depending on video length, Lambda config), navigate to your *destination* S3 bucket. You should see the cleaned video file appear (e.g., inside a `cleaned/` prefix if you used the example `destination_key`).
4. **Download and Verify:** Download the cleaned video from the destination bucket and play it to ensure the inactive segments were removed as expected.

Module 8: Review and Cleanup

Goal: Understand potential issues and how to remove the created AWS resources.

Discussion / Troubleshooting:

- **Timeouts:** If the Lambda function times out (check CloudWatch Logs), you may need to increase the timeout setting further (max 15 mins) or optimize the script. For very long videos (>15 mins processing time), Lambda might not be the best fit; consider AWS Batch or Fargate.
- **Memory Errors:** If you see memory errors (`OutOfMemoryError`), increase the Lambda function's memory allocation.
- **FFmpeg Errors:** Check the CloudWatch logs for detailed errors from FFmpeg's stderr output (captured by the `run_ffmpeg_command` function). This might indicate issues with the video file, incompatible codecs, or problems with the FFmpeg binary/layer.
- **Permissions Errors:** Access Denied errors usually point to missing permissions in the Lambda Execution Role (IAM Policy). Double-check the policy against required S3/CloudWatch actions.
- **Large Files:** Lambda's `/tmp` directory is limited (default 512MB, can be configured up to 10GB but costs more). Processing very large videos that exceed this space (for the downloaded input + temporary segments + final output) will fail. For huge files, Lambda with EFS integration or other services like AWS Batch are necessary.
- **Costs:** Lambda (compute time, requests), S3 (storage, requests, data transfer), CloudWatch (logs), Data Transfer costs apply. Delete resources when done to avoid charges.

Cleanup Steps:

1. **Empty S3 Buckets:** Delete all objects from both the source and destination buckets.
2. **Delete S3 Buckets:** Delete the source and destination buckets.
3. **Delete Lambda Function:** Delete the `VideoCleaningFunction`.
4. **Delete Lambda Layer Version:** Go to Layers, select `ffmpeg-static`, and delete the version you created.
5. **Delete CloudWatch Log Group:** Go to CloudWatch -> Log groups, find the log group for your function (`/aws/lambda/VideoCleaningFunction`), and delete it.
6. **Delete IAM Role:** Go to IAM -> Roles, find `LambdaVideoCleaningRole`, detach the policies (`AWSLambdaBasicExecutionRole`, `LambdaS3VideoCleaningPolicy`), and then delete the role.
7. **Delete IAM Policy:** Go to IAM -> Policies, find and delete `LambdaS3VideoCleaningPolicy`.
