ChatGPT

Video Surveillance AI App – Learning Project Plan

Project Overview & Goals

You plan to build a mobile application (iOS and Android) that allows users to upload videos from multiple cameras and later query those videos for specific people or attributes. The core goals are: (1) Publish a basic cross-platform mobile app, (2) Integrate AI for computer vision (CV) and some natural language query processing, and (3) Learn to use AWS cloud services in the process. In short, the app will ingest videos, **analyze them with AI** to detect people and their attributes, store the results for ~30 days, and provide a query interface to find occurrences (e.g. "person in red shirt holding a bag at 2:15 in camera2.mp4"). Key requirements include OTP-based user authentication, video processing for **face recognition** and **attribute extraction** (clothing color, age, etc.), counting unique individuals across videos, and a basic UI for querying results.

Why This Project? Aside from the functional goals, this is a hands-on learning sprint. It's an opportunity to experiment with **latest AI tools** (especially computer vision models and possibly NLP for queries) and to get familiar with AWS infrastructure for a production-grade app. Given the short 2.5 day timeline, we'll emphasize using **managed services and pre-built models** (to avoid long model training) and follow best practices for quick development.

Functional Requirements Recap

- **User Authentication:** Users register with name, email, and phone; login via mobile OTP (passwordless). This ensures each user has a secure account (and also helps in scoping their data).
- **Video Upload:** Users can upload up to 4 videos (initially) from different cameras. Videos will be stored (likely in cloud storage) for processing. Eventually, in a scaled-up scenario, the system might accept live streams from up to 200 cameras, but for now we handle individual video files.
- **Video Analysis (AI):** For each uploaded video, the system should automatically extract **people and their attributes**. This includes detecting each person, possibly tracking them in the video, and identifying distinguishing attributes:
- *Face-based attributes:* identity (to recognize the same person across frames/videos), approximate age (to tell child vs adult vs senior), **emotions** (happy, sad, etc. – a proxy for sentiment), presence of **mask** (face covered or not), **hair color**, facial hair, glasses, etc. Amazon Rekognition's face analysis can provide many of these details (e.g. mood/emotion, hair color, eyeglasses, age range) [1] .
- *Clothing and appearance:* dominant **color of attire**, patterns (e.g. striped shirt vs solid), accessories (hat, backpack?), etc.
- *Objects held:* Identify if the person is holding items (phone, bag, etc.) via object detection on each person's region.
- **Data Storage:** Extracted metadata should be stored with references to the video and timestamp. For example, we might store that *"PersonID_abc: wearing blue shirt, gray pants, ~30 years old, at [video1.mp4, timestamp 02:15]"*. This data will be kept ~30 days. A database will hold these records for querying.

- **Unique Person Counting:** The system must determine the **number of unique individuals** across all videos a user uploads. If the same person appears in multiple videos (or multiple times in one), they should be identified as one unique visitor. This implies some form of **face recognition or matching** to assign a consistent ID to the same person across videos.
- **Query Interface:** A user (or admin) can later query the processed data by using natural descriptions – e.g. *"kids with red shirts"*, *"person with mask and glasses"*, *"someone holding a baby"*, etc. The system should return matching video segments or counts based on these criteria. (In initial version, this could be a simple filter interface or keyword search rather than full NLP, depending on time.)

Non-Functional & Technical Requirements

- **Accuracy of Recognition:** Use reliable AI models for face detection/recognition and attribute extraction. False negatives/positives should be minimized (e.g. correctly identify masks, fairly estimate ages). We'll use proven pre-trained models or APIs (like AWS Rekognition) to avoid training models from scratch under time constraints.
- **Scalability (Future):** The design should be mindful of future scaling to many simultaneous video streams (though not implemented now). This means using cloud storage and possibly message queues or serverless processing so it can scale horizontally.
- **Performance:** Video processing is heavy, so we'll likely do it server-side on powerful instances or in asynchronous jobs. The mobile app should not freeze while videos are analyzed – instead, the user might be notified when results are ready.
- **Storage and Retention:** Videos and extracted data should auto-expire after 30 days (cleanup mechanism) to manage storage costs.
- **Security & Privacy:** Since we deal with personal videos (possibly containing people's faces), ensure data is secured. Use proper authentication (no open endpoints), and consider privacy implications of face recognition (disclose if needed, etc.). In early dev, an admin panel might skip auth for simplicity, but production should lock down sensitive data.

Tech Stack & Tools

Mobile App: React Native (Cross-Platform)

Using **React Native** is ideal to achieve a single codebase for both Android and iOS. It saves development time and ensures feature parity across platforms [2]. React Native's ecosystem (e.g. Expo or libraries for camera, file upload) will speed up development. Key aspects for the app:
- Implement **OTP Authentication UI**: screens for entering phone, receiving OTP, entering code.
- Screens for **Video Upload**: possibly using the device camera or file picker to select videos, then uploading to the server.
- A basic screen to list uploaded videos and maybe trigger the processing (or show processing status).
- (Optional) A simple results viewer or a "Search" screen – though complex querying might be easier on a web interface initially, as noted.

Developer tools: You can use Expo CLI for quick setup (if not needing too many native modules) or React Native CLI if you plan custom native modules. Given the timeframe, Expo might be convenient (easy video picker, and testing on device quickly). Ensure to use React Native's debugging tools and Hot Reload for quick iteration.

Backend and Cloud Services: AWS Stack

Leveraging AWS will cover most server-side needs without reinventing the wheel:

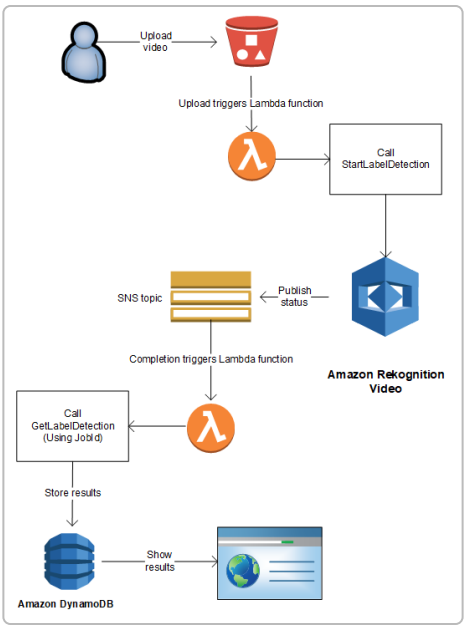
- **User Authentication: Amazon Cognito** User Pools can manage user signup/login and handle SMS OTP via Amazon SNS. Cognito supports **passwordless SMS OTP** flows [3]. For example, you can enable a **custom auth challenge** in Cognito so that when a user enters their phone, a Lambda triggers to send an OTP code via SNS, and Cognito verifies it [4]. This saves you from implementing OTP logic from scratch and is production-grade (Cognito handles user accounts securely). Using AWS Amplify on the front-end can simplify integrating this. *(Alternatively, a third-party like Twilio for SMS could be used, but since AWS is preferred for learning, Cognito+SNS is a good route.)*

- **Storage for Videos: Amazon S3** will store the raw video files. S3 is scalable and easy to integrate. The mobile app can upload videos to S3 either via a backend API or directly using pre-signed URLs. A best practice is to request a **pre-signed URL** from your backend for each upload; the app then PUTs the video file to S3, and finally your backend is notified (e.g. via S3 event) that a new video is in. We will configure an S3 bucket with proper access permissions (only allow your app/back-end to read them, etc.).

- **Video Processing & AI Analysis:** This is the core of the project. We have two main approaches:

- **Using AWS Managed AI (Amazon Rekognition):** This is the fastest way to get robust results. Amazon Rekognition is a cloud CV service that can detect **people, objects, text, scenes, faces, emotions, and more** in images and videos [5]. We can feed the videos to Rekognition and get a wealth of metadata without training models ourselves. Specifically:

    - *Rekognition Video APIs:* Rekognition has asynchronous APIs for video. We'd upload the video to S3, then call e.g. `StartLabelDetection` (to detect objects/people per frame) and `StartFaceDetection` (for faces and facial attributes) on that video. These operations run in the background on AWS's side. When done, they publish a completion notification to an SNS topic. Our app can subscribe a Lambda to that SNS to get the results [6] [7]. The results (from `GetLabelDetection` or `GetFaceDetection`) will include timestamps for each detected object or face [8]. We can then post-process and store what we need (e.g. entries for each person seen with their attributes). **Diagram:** The image below shows a typical AWS workflow for video analysis – user uploads video to S3, a Lambda triggers to call Rekognition, and another Lambda handles the results, storing them in a DB for querying [9] [7].



Architecture – Automated video processing with AWS Rekognition. In this serverless workflow, uploading a video to S3 triggers a Lambda which calls a Rekognition Video analysis (e.g. label detection). Rekognition publishes completion to an SNS topic, which triggers a second Lambda. That Lambda fetches results (e.g. via `GetLabelDetection`), stores them in a database (like DynamoDB), and a web/client app can then query or display the analyzed data [6] [7].*

- *Rekognition Image APIs:* In some cases (or for simplicity), we could also treat video frames as still images. For a short MVP, one strategy is to extract key frames (say one frame per second or whenever motion is detected) and call the image APIs like `DetectFaces` and `DetectLabels` on those frames. This avoids dealing with the async video jobs and SNS, but it's less efficient for longer videos. Still, it's an option if you want to write a quick processing script in Python without setting up the full pipeline.
- *Face Recognition:* Rekognition can be used to identify if the same person appears across videos. The typical method is to use a **Face Collection**. We can call `IndexFaces` on face images to add them to a collection (Rekognition stores face feature vectors). Later, we can use `SearchFaces` or `SearchFacesByImage` to find matches [10] [11]. For example, as we process video1, we add each detected face to "User123's collection". While processing video2, for each face we check against that collection; if Rekognition returns a match above a confidence threshold, we know it's a repeat person (and we link them via a common PersonID). This way, the **number of unique people** = number of unique faces in the collection [10] [11]. (We'll need to manage some logic to avoid adding the *same* face multiple times from many frames – perhaps track which frame IDs have been indexed or only index one frame per person track in the video.)
- *Attribute Extraction:* Rekognition's face analysis will directly give many attributes like **Emotions**, **AgeRange**, **Gender,** whether the person has **Beard/Mustache**, **EyesOpen**, etc. Notably, it can predict **mood (happy, calm, angry, surprised, etc.) and hair color** [1], which covers the "sentiment" and hair color requirements. For clothing color/pattern, Rekognition's **label detection** might tag clothing items (e.g. "Person" with "Shirt" and sometimes even color like "Red Shirt" if it's confident). If that is insufficient, an alternate approach is to apply some

custom logic: for each detected person's bounding box, analyze the pixel colors in the torso region (using OpenCV, for instance) to determine the dominant color (e.g. lots of red pixels -> wearing red). This could be a simple workaround for color detection if needed.

- *Mask Detection:* As of recent updates, Rekognition has **PPE detection** which can tell if a person is wearing a face mask or not [12] [13]. We could call `DetectProtectiveEquipment` on video frames (or images of people) – it returns whether it sees a face mask on a detected person. This would directly fulfill the "face is masked or not" attribute.
- *Object-in-hand:* Rekognition label detection will identify common objects in each frame. If a person is holding a phone, bag, etc., those might appear as detected labels ("Cell Phone", "Bag"). We may need to cross-reference if that object is in proximity to the person's bounding box, but for a rough solution, simply noting that *in the frame with Person X at time Y, labels include "Cell Phone"* is enough to infer Person X was holding a phone (assuming only they are in frame or using spatial filtering if multiple people).
- *Counting & Querying:* All the above analysis results (faces, labels, etc.) will be aggregated per user. For querying, since we'll likely store structured attributes (e.g. a record like `{personId: X, video: cam1.mp4, time: 02:15, attributes: {shirtColor: red, hair: black, mask: false, emotion: happy,...}}` in a DB), we can query by those fields. A simple search API might accept parameters (color=red, hasMask=false, emotion=happy) and filter the records. We can also implement a rudimentary **NLP** layer to interpret a natural sentence into those parameters – for example, using a small **OpenAI GPT-4 or Claude** prompt to parse a query text ("man with red shirt and mask") into `{"shirtColor": "red", "mask": true, "gender":"male"}`. This is an *optional* enhancement; given the time crunch, a straightforward approach is to have a form-based filter or require the admin to use specific keywords.

- **Using Custom/Open-Source Models:** As an alternative or supplement to AWS, you could use open-source CV models deployed on an EC2 instance or AWS Lambda (if small enough) or via AWS SageMaker endpoints. Examples:

  - **YOLOv8 or YOLOv5** for real-time object detection (it can detect people and many objects with high speed). A YOLO model could identify people, and with custom training it could even classify clothing colors. In fact, there are community models for detecting clothing and colors (e.g. a YOLOv8 model fine-tuned to detect shirt color [14]). But training a model is not feasible in 2 days, so you'd rely on pre-trained capabilities.
  - **Facial Recognition libraries** like **Facenet** or **dlib** could be used to compute face embeddings and cluster them to identify unique individuals. For example, you could extract faces from frames, run them through a pretrained network to get 128-D vectors, and group similar vectors by cosine similarity. This is essentially what Rekognition does behind the scenes. Implementing this yourself is educational but will consume time – since learning AWS is a goal, you might lean on Rekognition for now.
  - **OpenCV** for some basic tasks: e.g. extracting video frames, image preprocessing, or color detection as mentioned. OpenCV could also do motion detection to skip redundant frames.
  - **MediaPipe** (Google's CV library) has ready solutions for face detection, pose, etc. For instance, MediaPipe Face Mesh can detect faces and even identify mask vs no mask indirectly, but again, using Rekognition is simpler for now.
  - **NLP Query**: If you decide to incorporate an NLP model for query understanding, you might use a lightweight approach like a dictionary of keywords to attributes (e.g. "red" -> shirtColor=red) or use a cloud AI service. OpenAI's GPT-3.5/4 API or Anthropic's Claude could

parse queries if you provide a prompt with examples. However, calling an external API for each query might be overkill; a simpler solution is to design the query UI as checkboxes/dropdowns for each attribute (color, age category, mask Y/N, etc.), which avoids misinterpretation and heavy NLP work. Given your time, this simple filtering UI is advisable, and NLP integration can be a stretch goal if time permits.

AWS vs Custom Trade-off: Since one objective is to **learn AWS services**, the recommendation is to use AWS managed services (Cognito, S3, Rekognition, DynamoDB, Lambda, etc.) as much as possible. This speeds up development (no need to train or fine-tune models, and AWS will handle scaling in the background) [15]. Rekognition in particular lets you focus on the application logic rather than the "hard parts" of deep learning and video processing [15].

- **Database:** For storing analysis results (metadata), you have two main choices: relational (AWS RDS) or NoSQL (AWS DynamoDB). DynamoDB is likely a good fit here – the data is semi-structured (varying attributes per person) and you might want fast key-value lookups (e.g. query by user and video). The AWS example solutions often use DynamoDB for storing label detection results for quick web lookup [16] [17]. For instance, you could use a primary key of `UserID` and sort key of `PersonID` or a composite like `VideoID#Timestamp`, and include attributes for each detection. Dynamo's flexible schema means you can add new attribute types easily (for new AI features later). If you prefer SQL, an Amazon Aurora (MySQL/Postgres) would work but setting it up and managing schema in 2 days might slow you down. Dynamo with the AWS Amplify DataStore (GraphQL) could also be an option, but that might be overkill. I'd suggest starting with DynamoDB for simplicity and speed.
- **Serverless or Server?:** Using AWS Lambda for processing is convenient (especially with S3 triggers and SNS triggers as shown in the diagram). You could write a Python Lambda that, upon S3 upload, kicks off Rekognition jobs, and another that on SNS notification, fetches results and writes to Dynamo. This fully serverless route is nice because you don't manage servers and it scales. However, Lambdas have limited execution time (15 minutes max) which might constrain processing very large videos. In our case (few short videos), that's fine. If you find Lambda development too rushed, an alternative is a small **Node.js/Express or Python FastAPI server** running on an EC2 or AWS Elastic Beanstalk. That server can handle upload endpoints and processing synchronously (or spawn threads). But given AWS has blueprints and tutorials for Lambda+Rekognition, it's a great learning exercise to implement it the "cloud-native" way.
- **Additional AWS Services:**
- *Amazon API Gateway* – to expose your backend (Lambda or otherwise) to the mobile app and web UI securely. For example, an API endpoint for "uploadVideo" (which returns a presigned URL or directly accepts file bytes) and an endpoint for "queryResults". If using Amplify, it can create a GraphQL API (AppSync) which might be too much to learn in 2 days, so a simple REST API Gateway + Lambda will do.

- *AWS Amplify Framework* – this is a developer tool that ties together Auth, Storage, API, etc., with handy CLI and libraries for React Native. If you haven't used it, it can save time for setting up Cognito, API Gateway, and DynamoDB via GraphQL schema. The downside is it generates a lot of boilerplate and might be confusing to debug if unfamiliar. Considering your timeframe, you might cherry-pick Amplify only for Auth (it has React Native UI components for sign-in that support OTP) and Storage (for easy S3 upload). Alternatively, handle these manually through AWS SDK calls if you prefer more transparency.

- **AI Development Workflow Tools:** You mentioned tools like Anthropic's Claude, OpenAI, or Cursor for coding assistance. If you have access, these can be very helpful to speed up writing code (for example, using Github Copilot or Cursor IDE to generate boilerplate for AWS SDK calls, etc.). **Claude Code** and **Cursor** are essentially AI coding assistants that can autocomplete or even implement functions given prompts. My suggestion: *Use them to accelerate routine coding tasks*, such as writing a function to parse a video file and call Rekognition, or setting up Amplify config – but be cautious to review the code they produce. Given the tight schedule, an AI pair-programmer could indeed be handy, just ensure the output aligns with AWS best practices (these tools sometimes make minor mistakes in API usage). There are also many **sample codes** on AWS blogs and GitHub for similar tasks – you can leverage those with minimal modifications (that's often faster than prompting an AI from scratch). In summary, tools like Claude/Cursor can augment your development, but since you're new to AI workflows, don't get too sidetracked setting them up; focus on coding the core logic first, and use them if you feel they save you time.

High-Level Solution Design

Let's outline the system architecture and data flow from end-to-end, then break down the development steps:

1. Architecture Diagram & Components:
At a high level, the system consists of: a **React Native mobile app** for user interaction, a set of **AWS cloud services** for backend logic and AI analysis, and a simple **web interface** (for admin queries). Below is an overview of how these pieces interact:

- **Mobile App (RN) ↔ Backend API**: The app communicates with backend endpoints for actions like registration/login (OTP), uploading videos, and fetching query results.
- **Auth Service:** Amazon Cognito handles user sign-up (with email/phone) and login via OTP. Cognito can directly verify OTP via Lambda triggers and SNS. The RN app uses Amplify SDK or Amazon Cognito SDK to initiate auth flows.
- **Video Upload Flow:** When a user uploads a video:
- The app either calls an API endpoint (e.g. `/getUploadURL`) which returns a signed S3 URL, then the app uploads the file to S3; **or** the app calls an API that streams the video to a backend server which then puts it in S3. The presigned URL approach is usually more efficient (letting S3 handle the upload load).
- S3 triggers a Lambda (via **S3 Event Trigger**) when a new video file arrives. The Lambda's job is to start the analysis. It will call **Rekognition Video** APIs (like `StartFaceDetection`, `StartLabelDetection`, and possibly `StartFaceSearch` if using a collection for known faces) [6] [7]. We might pass metadata like the UserID and VideoID along (perhaps via the SNS topic or database) so the results can be linked to the right user.
- Rekognition processes the video in the background. When done, it sends a notification to an **SNS Topic** we configured.
- An **SNS-triggered Lambda** (second Lambda) receives that notification. This Lambda then calls the corresponding `Get*` APIs (e.g. `GetFaceDetection`, `GetLabelDetection`) to retrieve the analysis results [7]. It then **parses and transforms** those results into our data model and stores them in the **Database** (DynamoDB). For instance, it might iterate through each face detected and create/update a record for that person. Here we also use face comparison: if a face matches someone already seen (we can use a Rekognition Collection or a DynamoDB table of face

encodings), tag it as the same PersonID. If it's new, generate a new PersonID. Also store time intervals the person appeared, etc. We might also store aggregate info like total unique people in that video.

- After storing results, this Lambda could also send a push notification or update some status so the user knows analysis is done (if we had more time, integration with AWS AppSync or WebSocket to notify app). In the 2.5 day MVP, we might skip active notification and instead the app can poll an API to check if results are ready.
- **Query Flow:** For querying, you'll likely implement a minimal **admin web app** (could be a simple React web page or even using a tool like AWS Amplify Studio or AWS AppSync's GraphiQL). This web UI will let you input query filters and display results:
- The web client calls a **Query API** (could be a Lambda behind API Gateway) that reads from DynamoDB. For example, it might scan for all records for User=XYZ where attributes match the query. Because DynamoDB's query capabilities are limited to primary key, you might design a GSI (Global Secondary Index) for certain attributes (like one GSI for `ClothingColor` to enable "find all red"). But given the small scale, scanning a few hundred records and filtering in code is fine.

- The results (list of matching persons or frames) are returned and the web UI can present them. E.g., *"Found 1 person matching red shirt: Person 5 in video2.mp4 at 00:37"*. If you stored thumbnails or snapshot images of the person, you could even display those (Rekognition can return the bounding box, and you could fetch that frame from S3 to show a thumbnail – nice-to-have feature).

- **Data Model:** In DynamoDB, one design could be: a table `PeopleDetection` with primary key `UserId` and sort key `PersonId` (or a composite of VideoID#PersonIndex). Each item could store attributes like `{ videos: [ {videoId, timestamps: [...] } ], clothingColor: "red", hairColor: "brown", ageRange: "20-30", emotions: ["Happy"], mask: false, ... }`. If a person appears in multiple videos, you update the same item's `videos` list. Alternatively, you store each appearance separately: PK = User, SK = VideoID#Timestamp, and have attributes for that instance. But then linking the same person requires an additional ID field. It might be simpler to aggregate per person after processing each video. For counting unique, just count items with unique PersonId for that user.

- **Admin Panel:** Since you mentioned possibly launching on Product Hunt and maybe offering subscriptions, eventually you'll want a proper web dashboard for users (or admins) to search and see analytics (like "20 people visited my store today, 5 of them were kids, etc."). For now, you can keep the admin panel *unauthenticated* and simple (as requested), but note that in real deployment you'd secure it. A quick implementation could be a static HTML/JS page that calls your query API with a hardcoded user ID or selection.

Diagrammatically, the workflow is:

- *User app* – Upload video -> *S3* -> trigger -> *Lambda1* (start AI jobs) -> *Rekognition* processes -> *SNS* -> *Lambda2* (get results, store in DB) -> *DB (Dynamo)*. Then *User/admin query -> API (Lambda) -> DB ->* results back.

This aligns with the AWS reference architecture for serverless video analysis, just adding the face collection step for cross-video identity.

Step-by-Step Development Plan

Given ~2.5 days (60 hours) of focused development, here's a breakdown of how to approach it:

Day 1: Frontend Setup & Basic Backends

1. Project Setup (Mobile App): Initialize a React Native project (e.g. using `npx react-native init` or `expo init`). Set up basic navigation if needed (maybe a stack navigator with screens for Login, Upload, Results). Implement a simple UI for OTP login: - Allow input of phone number and a button "Send OTP". - On button, call Cognito or a placeholder API that triggers OTP. Since Cognito's custom flow might take time to configure, you could initially simulate it or use Amplify's pre-built Auth UI which has an OTP flow if set to phone verification. **Note:** To save time, consider using AWS Amplify's UI components for Auth. Amplify CLI can configure a Cognito user pool with OTP (as in the Medium guide) [18] [19] . You'll need to set up an SNS IAM role and verify a sender ID for SMS. If that's too much, a quick hack is to use email OTP (since Amazon SES is easier to start than SNS for SMS due to some restrictions) – but SMS is preferred as per requirements. - Implement the OTP verification screen (input code, verify it). - Test this flow manually (Amplify has `Auth.signIn` and `Auth.sendCustomChallengeAnswer` methods if using custom challenges). - If Amplify is troublesome, you could skip actual verification in the prototype (e.g. accept any code "1234" for now) just to not block progress, and note to integrate Cognito fully later.

2. AWS Backend Setup: In parallel, go to AWS Console: - Create an S3 bucket (enable "Bucket event notifications" on this bucket for `ObjectCreated` events -> target a Lambda function (which we'll create next)). Also, set CORS if the app will upload directly (allow PUT from your app domain or wildcard during dev). - Create a DynamoDB table for results (with primary key as discussed, e.g. UserId and sort key). - Set up an SNS Topic for Rekognition completion notifications. - (Optional) Create a Rekognition Face Collection for each user or a single global one. Actually, a better approach is one collection per user for privacy (so you don't mix faces between users). You can programmatically create a collection when a new user registers or use a naming convention (like Collection_<<UserID>>). For now, maybe create one test collection for your own user ID.

3. Lambda Functions: Use the AWS Lambda console or your IDE to create two functions (you can choose Python since Boto3 is convenient, or Node if you prefer JS – Python often has easier image processing libs if needed). Outline: - *Lambda1 (On S3 Upload):* Triggered by S3 event. It gets the S3 object key (video path). It should parse the user ID and video ID from the key or S3 metadata (you might structure uploads as `s3://bucket/UserID/videoName.mp4` to know who it belongs to). Then it calls `rekognitionClient.startFaceDetection(Video={S3Object...}, NotificationChannel={SNSTopic,...}, FaceAttributes="ALL")` to detect faces with attributes. Also call `startLabelDetection` on the same video to detect objects. Both calls will return a JobId. Lambda1 can store these JobIds in Dynamo or somewhere if needed, but actually we don't need to if we rely on SNS: Rekognition will publish JobId and status to SNS. However, we might want to tag the SNS message with user/video context – AWS allows embedding "MessageAttributes" or we could include the video filename which encodes user info. Actually, the SNS notification from Rekognition might include the S3 key in the payload by default (the docs show it includes JobId, status, and a "Video" structure with the S3 info). We will verify that in testing. Ensure the Lambda1 has permissions: it needs `rekognition:Start*` permissions and also an IAM Role that Rekognition can assume to publish to SNS (as per AWS docs, you must grant Rekognition access to SNS topic ARN). - *Lambda2 (On SNS Notification):* Triggered by SNS (subscribe this Lambda to the Rekognition SNS topic). This one will get an event containing the JobId and

details of the completed job. It should determine which type of job (label or face) and call the appropriate `Get*` API to retrieve results. For face detection results: you get an array of faces with timestamps and attributes. For label detection: an array of labels (objects) with timestamps. We will combine info – e.g., we might primarily rely on face data for identifying persons and use label data for additional context (like person's clothing or carried objects). The Lambda should then **post-process**: - Perhaps reconstruct "tracks" of people. Rekognition's `GetPersonTracking` (if we had used StartPersonTracking) gives a "PersonID" for each track within one video (it's not an identity across videos, just a track number per video). If face detection doesn't directly group by individuals per video, an alternative is to use `GetPersonTracking` first to know how many distinct persons were in that video (each PersonID in that result is one individual track). We can map face detections to those PersonIDs by timestamp overlap. This might be complex for MVP – a simpler shortcut: assume each distinct face bounding box across frames belongs to someone, and use face recognition to group them. For MVP, you could even skip tracking and just take the first appearance of each face as a "person instance". - Use the Rekognition **Face Collection** for cross-video: for each face detected with sufficient quality, call `IndexFaces(CollectionId=userCollection)` on the frame image (Rekognition can directly use the video frame via `Image={S3Object}` using the video file and a timestamp/ByteRange if we extract a frame… actually `IndexFaces` only takes images, not video, so one trick: the `GetFaceDetection` result might include a "Face" object with bounding box and an "Image frame" (but not the actual pixels). We might need to grab the frame from S3. Alternatively, since we have the timestamp, we can use the video file to extract that frame. But AWS doesn't provide a direct frame extract API; we might use the Elastic Transcoder or convert the whole video to frames using something like AWS MediaConvert. This may be too heavy. Instead, consider using **StartFaceSearch**: If you had indexed a collection beforehand with some known faces (not our case) you could find them in the video. But here we are doing the opposite (finding unknown faces and grouping them). - Given the time, you might simplify by treating each video separately for now: count unique faces in that video via Rekognition's own grouping (PersonTracking), and not implement cross-video matching in the first iteration. Then, if time allows, implement a simple face embedding match: e.g., use the bounding box from Rekognition and retrieve that region of the frame (there's a way to get the frame image: either downloading the video and using OpenCV in Lambda – which might be too slow, or using the thumbnail technique with Elastic Transcoder as shown in the 2017 blog [20] [10]). The blog used Elastic Transcoder to get 1 frame per second. That is an AWS service but requires setting up a pipeline. An alternative: AWS has **MediaConvert**, a newer service for video transcoding; or simply, FFMPEG in Lambda (Lambda might not have it by default, but you could use a container image for Lambda with ffmpeg installed). - For MVP, perhaps assume each video's people are unique to that video (if the user only uploads non-overlapping camera footage, this may be okay). You can note in the plan that "for cross-video unique count, we would integrate face collection matching as a next step." - Store results in DynamoDB: For each person (or even each detection frame), store relevant attributes. You could create one item per **unique person per video** containing aggregated attributes (e.g. if a person wore multiple outfits – unlikely in one video – or had multiple emotions over time, you might store the most frequent). At least store person's attributes and a list of timestamps seen. Also store an item for the video overall with total distinct people count. - Ensure Lambda2 has permissions to call `Get*` Rekognition APIs and to read the S3 video (if needed for frames), plus permission to write to DynamoDB (and maybe to call IndexFaces if you go that route).

Both Lambdas can be developed in a local environment using AWS SAM/Serverless Framework for speed if you're comfortable, or directly edit in console for quick and dirty (though not ideal for complex code). Given time, writing in an IDE and using AWS CLI to deploy might be fastest.

4. Test the Pipeline: Simulate a video upload to S3 (you can manually put a small MP4 in the bucket via console) and see if Lambda1 triggers and starts Rekognition. You might use a short video with a couple of people to see if it works. Use CloudWatch logs to debug any issues (permissions, etc.). Once Rekognition finishes, ensure Lambda2 runs and you see entries in DynamoDB. At this point, log out what you store to verify structure.

Checkpoint: By end of Day 1, aim to have: user auth in app (even if not fully wired to Cognito yet), ability to upload a video (perhaps from a local file picker) to S3, and the back-end processing that populates the database with at least some detection info. This is ambitious, but focusing on the core (maybe skip fine details like face matching until basic flow works).

Day 2: Query Interface, Enhancements, and Polishing

5. Implement Query (Admin Panel): Create a very basic web page or tool to retrieve results: - E.g., a Node/Express simple app that serves an HTML with a form (for attributes like color, mask yes/no, age range dropdown). When submitted, the form triggers an endpoint that queries DynamoDB. Since it's without auth, you could even use the DynamoDB console to view data, but for a more presentable result, do a quick page. - Alternatively, use React (since you know RN, React web is similar). A create-react-app with a single page that calls API Gateway endpoints to query. You can bypass API Gateway and call DynamoDB from the web if you use AWS SDK in the browser, but that's not secure without auth. So better to have a query Lambda. - So, create a **Query Lambda** (or reuse an existing one) that scans/query the Dynamo table based on parameters. Hook it up to API Gateway or Amplify API. This API can be open for now (or protected by a simple API key if needed). - Test a few queries: e.g., find all records where `clothingColor = 'red'` or `emotion CONTAINS 'Happy'`. With Dynamo, if not using an index, you'll have to scan and filter in Lambda code. That's fine for small data.

6. Attribute Fine-Tuning: With basic pipeline working, refine the AI aspects if possible: - Check if **clothing color** is captured. Rekognition label detection might output something like "Person" label with parent "Clothing" and maybe color info in the label name if the model recognizes it. If not, you could incorporate a quick OpenCV in Lambda2: take the video frame from a certain timestamp (if you can retrieve it). Maybe easier: use the thumbnail approach – perhaps run Elastic Transcoder job via Lambda1 to generate 1 FPS thumbnails (it's actually straightforward to use Elastic Transcoder or AWS MediaConvert, but those might require creating a pipeline in console). The 2017 solution did exactly that: Elastic Transcoder outputs images per second, then Lambda2 indexed those images [20] [21]. Given the time, you might skip detailed clothing analysis. Instead, focus on low-hanging fruit from Rekognition: - Emotion (done by Rekognition). - Face mask (if you call PPE detection or possibly analyze if "mouth/nose visible" – but PPE API is easier). - Age (Rekognition gives an AgeRange; you can bucket that into "kid/young/adult/elder" categories). - Gender (Rekognition classifies gender, though note it can be inaccurate and sensitive; for a fun project it might be okay to use gender if needed, but consider avoiding it if not necessary). - For height/size: This one is tricky. Unless you have a reference (e.g. camera calibrated), you can't get actual height from a single camera easily. You could categorize someone as "child" vs "adult" based on height in frame and distance, but that's approximate. Perhaps just use age as proxy for kid/old. T-shirt size or weight is really hard via camera; you may omit that or just note that it's not feasible accurately without specialized models.
- If sentiment was meant in another way (like overall sentiment of a scene), that's unclear – likely they meant emotion from faces which we covered. - Ensure unique person counting per video is correct. If using PersonTracking, it will give each tracked person an ID. You can use that to count unique in that video. For across videos, if not doing face collection matching now, you at least have per-video counts. You can sum

them across videos as an approximation (though if same person appears in 2 videos, they'd be double-counted in that sum, which is a known gap if not implemented). - Logically, if time allows, implement a **basic cross-video face match**: e.g. take the first face of each Person from each video, and compare it with all persons from other videos using `SearchFacesByImage` (if you indexed them into a collection). This could reduce duplicates.

7. Frontend Improvements: Now integrate the front-end with the backend: - After uploading a video, the app should ideally show a status like "Processing..." and later "Done. X people detected." You could implement a polling mechanism: after upload, call an API like `/checkStatus?videoId=123` that looks up if analysis done (maybe if all jobs for that video have completed and data is in DB). Or simpler, when Lambda2 writes to DB, also write an item like {VideoID, status:"Done", peopleCount:N}. The app can fetch that periodically. If time is short, you can skip real-time updates and just have the user use the web query to see results. - If possible, display some results on mobile – e.g. a list of "Detected people" for each video (with attributes summary). This could just be text for now (like "Person1: blue shirt, male, 25-35, no mask; Person2: red shirt, female, 5-12, wearing mask; ..."). - Ensure the app UI is minimally polished: not too many bugs in navigation, handle permissions (if using camera to record video, get permission; but if just picking file, that's easier).

8. Testing & Debugging: Use a couple of sample videos (maybe record yourself with different clothes or use any open-source video of a crowd) to test end-to-end: - Register a user, login via OTP (or bypass if not fully working). - Upload a video (verify it reaches S3 and triggers processing). - Wait for processing (check CloudWatch logs or Dynamo entries). - Query via the web or directly scan Dynamo to see if data makes sense (e.g. correct count of people, attributes detected). - Try a query like "mask = false" and see if it returns only unmasked faces, etc. - Record any issues (maybe Rekognition misidentifies something or Lambda times out for a longer video). Adjust parameters as needed (maybe limit video length for now or sample one frame per second to speed up processing). - Check error handling: e.g. what if two videos uploaded at nearly same time (should be fine, separate Lambdas). What if no face is detected? (Then face job returns nothing – handle that gracefully).

9. Performance & Best Practices Check: Although this is a prototype, consider a few production best practices: - **Error Handling:** Add try/catch in Lambdas. If Rekognition fails (e.g. video format not supported), log that and maybe mark analysis as "Failed" in DB so the app can show an error. - **Security:** For now the admin query is open, but at least ensure your user-upload API is secured (maybe use the Cognito JWT to authorize it via API Gateway). Amplify can attach Cognito auth to API Gateway easily. If short on time, at least use API Gateway keys or restrict by some token. - **Resource Cleanup:** If storing lots of images or intermediate data, plan to cleanup. For example, if you extracted thumbnails to S3, perhaps set a lifecycle rule to delete them after 1 day. Also maybe set the videos to auto-delete after 30 days via S3 lifecycle rules (fulfilling the 30-day retention). - **Config management:** Use AWS Parameter Store or Amplify config files for storing things like SNS topic ARN, collection IDs, etc., rather than hardcoding in code. - **Scalability considerations:** Note in documentation which parts need change for scaling: e.g., using SQS to buffer many videos, or switching to Kinesis Video Streams for real-time camera feeds, etc. Also consider concurrency limits of Lambdas and Rekognition (Rekognition can process multiple videos at once but has service limits – default 20 concurrent videos, I believe). For now, not a big issue.

Day 3 (Half Day): Buffer for Unfinished Tasks & Documentation

If everything went smoothly (rarely the case!), use remaining time to: - Address any incomplete features (maybe you postponed face matching, or the OTP flow needs finishing touches). - Improve the UI/UX a bit: e.g., show thumbnails of detected people (you could generate a snapshot of each person by capturing the video frame at their first timestamp – using ffmpeg offline or as a Lambda layer if adventurous). - Write a **README or documentation** for your portfolio: explain the architecture, how to deploy, how to use the app, etc. This is important since you mentioned using this for learning and portfolio. A diagram and a clear description will impress others who see your project. - If deploying for others to try (Product Hunt), consider setting up proper front-end hosting (Netlify or Amplify Hosting for the web app) and ensure no hard-coded secrets are exposed. Maybe limit it to a demo user unless you build out multi-user fully. - Prepare some sample queries and test them live to be confident in the demo.

Additional Considerations & Future Enhancements

Finally, let's list a few things that were out-of-scope for the 2.5 day build but are worth noting for a production version: - **Live Stream Processing:** In future, to handle 200 camera streams, you'd likely use **AWS Kinesis Video Streams** with Rekognition Video (which can do streaming analysis in near real-time) [22] [23]. That requires a different approach (Kinesis streams, and possibly an IoT Greengrass component if edge processing). - **Improved Query (NLP):** Implement a natural language query interface using an NLP model or service. For example, integrate with OpenAI's API where the user's query is sent along with context (the list of possible attributes) and the model returns a structured response or directly a DynamoDB query expression. Azure's Video Indexer actually provides a natural-language search on videos [24] [25] – you might aim to replicate some of that via your own means in future. - **Mobile App Polish:** Add ability for user to actually **view clips** of the moment a person was detected. You could store the timestamps and use a video player component in React Native to jump to those timestamps in the video. Or generate short GIFs of the detection. This would make the query results more user-friendly (not just text). - **Analytics:** Beyond queries, the app could provide summary analytics: e.g., "You had 10 visitors today (7 adults, 3 children; 2 people appeared multiple times)". This involves simple aggregation of the stored data. - **Multi-tenancy and Auth:** Lock down the system so each user's data is siloed (Cognito user IDs tie to only their S3 folder, their DynamoDB items, etc.). Use Cognito Identity Pools to grant the app temporary S3 upload rights to only their folder, etc. - **Cost Monitoring:** Rekognition is not free – video analysis is roughly $0.10 per minute of video [26]. In a production app with many uploads, you'd want to keep an eye on costs or limit video length per user tier. Perhaps incorporate a subscription model where higher tier allows more minutes of processing. - **Alternate AI Tools:** Explore more advanced or specialized models for any gaps. For instance, if clothing pattern detection is needed (striped vs plain), a custom model could be trained using a service like AWS SageMaker or AutoML. Or if you want to estimate a person's height, you might use stereo cameras or a reference object. These go beyond quick prototyping, however.

Conclusion

This plan provides a **step-by-step approach** to building a functional prototype of the video analysis app within a couple of days. By leveraging **React Native** for a rapid cross-platform UI [2] and **AWS's managed AI services** for heavy lifting (face recognition, object detection, etc.), you minimize the need to develop and train AI models from scratch while still achieving cutting-edge functionality. Amazon Rekognition will automatically identify people, objects, text, and more in your videos [5], including detailed face attributes like emotions and hair color [1], which addresses the key requirements. Using a serverless AWS backend

with S3, Lambda, SNS, and DynamoDB will allow you to scale and modify the pipeline easily as you learn and extend the project.

By the end of this sprint, you should have a working application where a user can upload a video and, after processing, search for occurrences of people with specific attributes. This checks the boxes for your learning objectives: you'll have touched mobile development, cloud services, and AI integration all in one project. Good luck, and enjoy the marathon coding session!

Sources:

- Amazon Rekognition provides automated image/video analysis, identifying people, objects, text, activities, etc., from provided media [5]. It can detect faces and analyze attributes like mood and hair color [1], and even perform facial recognition by storing and comparing facial features in collections [10]. Using these capabilities via AWS APIs lets us focus on application logic rather than ML algorithms [15].
- AWS architecture best practices for video analysis involve an asynchronous pipeline: upload to S3, trigger Lambda to start Rekognition (which sends results via SNS), then another Lambda to get results and store in a database for querying [6] [7]. This serverless flow is illustrated in AWS docs (user uploads video -> Lambda -> Rekognition -> SNS -> Lambda -> DynamoDB -> results to client) [16] [17]. We adopt a similar design in our solution.
- React Native enables building high-quality apps for iOS and Android from one codebase, greatly accelerating development and ensuring consistent functionality across platforms [2]. This suits our need to deliver mobile apps on both stores quickly.
- Amazon Cognito with Amplify supports OTP-based authentication, sending one-time passcodes via SMS to the user's phone for a passwordless login experience [3] [4]. We leverage this for user registration and login for better security and user convenience.

---

[1] Facial Recognition using AI with AWS: By Retired Member
https://www.finextra.com/blogposting/23494/facial-recognition-using-ai-with-aws

[2] React Native: Building Mobile Apps with a Single Codebase | by Chirag Dave | Medium
https://medium.com/@chirag.dave/react-native-building-mobile-apps-with-a-single-codebase-6c61c4b24d49

[3] [4] [18] [19] Building a Passwordless OTP-Based Login System with AWS Cognito, Amplify, and SNS in React Native | by Manthan Kasle | Medium
https://medium.com/@manthankaslemk/building-a-passwordless-otp-based-login-system-with-aws-cognito-amplify-and-sns-in-react-native-bbd7210c4bcb

[5] [26] Deep Analysis of Image and Video Using Amazon Rekognition - DEV Community
https://dev.to/aws-builders/deep-analysis-of-image-and-video-using-amazon-rekognition-4nfm

[6] [7] [9] [16] [17] Creating an Amazon Rekognition Lambda function - Amazon Rekognition
https://docs.aws.amazon.com/rekognition/latest/dg/stored-video-lambda.html

[8] [15] Building a scalable and adaptable video processing pipeline with Amazon Rekognition Video | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/building-a-scalable-and-adaptable-video-processing-pipeline-with-amazon-rekognition-video/

10 20 21 Find Distinct People in a Video with Amazon Rekognition | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/find-distinct-people-in-a-video-with-amazon-rekognition/

11 22 23 Working with stored video analysis operations - Amazon Rekognition
https://docs.aws.amazon.com/rekognition/latest/dg/video.html

12 Detecting personal protective equipment - Amazon Rekognition
https://docs.aws.amazon.com/rekognition/latest/dg/ppe-detection.html

13 DetectProtectiveEquipment - Amazon Rekognition
https://docs.aws.amazon.com/rekognition/latest/APIReference/API_DetectProtectiveEquipment.html

14 AbdulRehman-git/real-time-tshirt-color-detection-yolov8 - GitHub
https://github.com/AbdulRehman-git/real-time-tshirt-color-detection-yolov8

24 25 Video Analysis by Using Azure Machine Learning - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/ai-ml/architecture/analyze-video-computer-vision-machine-learning