AI Photo Finder — Project Plan & Tech Stack (v0.2)

Codename: LensLibrarian (name can be changed). This version (v0.2) updates the plan to use **Flutter (Dart)** from the start for a cross-platform (Android & iOS) app, while maintaining the core vision: a lightning-fast, privacy-first photo search on mobile using natural language, face/person detection, on-device tagging, and robust filtering.

1. Product Goals & Non-Goals

- **Goals:**
- **Natural Language Search:** Find photos by natural descriptions and keywords (e.g. *"yellow shirt at Galata Tower"*, *"sunset with my cat"*). The search should handle free-form queries, with an **LLM-powered parser** interpreting the query into filters and search terms.
- **People & Pets Grouping:** Automatically group photos by detected people (and pets) using on-device face embeddings + clustering. User can label clusters (e.g. "Mom") for easy reuse.
- **Custom Tags & Filters:** Allow users to add custom tags (e.g. `#office`, `#thesis`) and apply boolean filters (AND/OR) in queries. Support filter operators like date ranges (`before:2024-01-01`), locations, or "has:ocr" for text-containing images.
- **Privacy-First:** All image processing (embedding generation, face recognition, OCR) is done **on-device by default**; no photos or biometric data leave the device [1] [2]. Cloud features (if any) are opt-in and minimal. Even when using an LLM for query parsing, only the **text query** is sent to the cloud API – never the photos or embeddings.
- **Performance:** Achieve a snappy UX. After indexing, queries should return results in under **150 ms** on an average gallery (10k–30k photos) [1]. The system should handle large local photo libraries efficiently.
- **Non-Goals (MVP):**
- No social features or photo editing in the MVP.
- No cloud backup or multi-device sync at launch (all data stays local, though the architecture will allow adding cloud sync later).
- No server-side face management or heavy cloud processing. The focus is on on-device intelligence; cloud is only used for optional enhancements like LLM-based query understanding.

2. Core UX Flows

- **A. First Run:**
- On first launch, prompt the user for permission to access photos. Clearly explain that analysis happens on-device and offer opt-in for any cloud features (e.g. enabling LLM-based search or future backup options).
- Once permission is granted, begin the background **Indexing** process. The app scans the photo library and extracts metadata and embeddings for each image (this may happen in batches with a background task).

- Show an indexing progress indicator with the ability to pause/resume. Respect device constraints: e.g. only index while on charger or Wi-Fi if the user prefers, to manage battery and data usage.

- **B. Search:**

- The main UI is a search bar where users can type natural language queries and filter instructions. For example:
  - `yellow shirt galata tower`
  - `mom + birthday 2023`
  - `cat near window #office before:2024-01-01`
  - Users can also enter full sentences (e.g. "photos of **Mom and me at the beach last summer**"). The app will use an LLM API to parse such free-form queries into a structured search (identifying persons "Mom", approximate date range "summer 2021", and keywords "beach") and into a refined text description for image matching.

- Upon search, results are shown in a responsive grid of thumbnails, sorted by relevance (semantic similarity score). The UI also presents facet filters or chips (e.g. People, Places, Time, Tags) so users can refine the results further by tapping on a person's name, a location, a date, or a tag. Searching is designed to feel instant (utilizing a pre-built index for speed).

- **C. People:**

- A dedicated "People" tab or section lists clusters of faces detected in the library. The app auto-clusters faces via unsupervised learning (e.g. DBSCAN on face embeddings). These appear as "Unnamed Person 1, 2, ..." initially.

- The user can tap a face cluster to review photos and assign a label (e.g. label the cluster "Mom" or "Alice"). They can also merge clusters if the same person was split into two, or split if a cluster grouped different people by mistake. All face grouping data stays local.

- **D. Tags:**

- Users can add custom tags to photos (single photo or bulk-select multiple photos and tag them all). Tags are user-defined keywords like projects, events, or any category (`#vacation`, `#office`, etc.).
- A "Tags" management UI lists all tags and allows quick filtering: tapping a tag shows all photos with that tag, and tags can be combined with other filters in search (including via query syntax or UI chips). There will be easy ways to add/remove tags and view untagged photos.

Throughout all flows, the **UI** is built with Flutter's native widgets for a smooth, cross-platform experience. Design will be kept simple and performant (e.g. efficient scrolling of thousands of thumbnails).

3. System Architecture (Cross-Platform Design)

Overview: The system follows a mobile-first, on-device architecture. Flutter (Dart) is used for all UI and app logic, with plugins and native integrations to handle platform-specific tasks. We maintain a clear separation between on-device components (which handle heavy image processing and data management) and cloud

services (used sparingly, e.g. for LLM queries). The architecture is designed for **privacy** and **speed**, but is flexible for future cloud support (e.g. syncing metadata).

- **Indexer Pipeline (On-Device):** The indexing pipeline runs in the background (using Flutter's workmanager plugin to schedule tasks on both Android and iOS). Its steps:
- **Enumerate Assets:** Traverse the device's photo library using a cross-platform API (e.g. `photo_manager` plugin, which abstracts Android's MediaStore and iOS PhotoKit) to get a list of all photos and their basic metadata (URI/ID, resolution, timestamp, etc.).
- **Extract Metadata:** For each photo, gather EXIF metadata (timestamp, GPS coordinates, camera make/model) and compute a **dominant color** for the image. Optionally, run a lightweight scene classifier for coarse tags (not critical for MVP).
- **OCR for Screenshots/Documents:** If a photo is detected as a screenshot or contains text (e.g. by filename or aspect ratio), run on-device OCR to extract any text content. Save the recognized text (to enable text-in-image search).
- **Face Detection & Embedding:** Run on-device face detection on the photo to find any faces. For each face, crop and align the face region, then compute a **face embedding** vector (using a pre-trained model). Store the face embedding and metadata (bounding box, landmarks) in the database, linked to the photo. Multiple faces per photo are handled.
- **Global Image Embedding:** Compute a **global image embedding** for the entire photo using a CLIP-like model. This produces a high-dimensional vector representation of the image's content, which enables semantic search (matching images to text queries).
- **Database Storage:** Save all the above information into a local SQLite database (augmented with a vector search extension). The photo's metadata and embeddings are inserted if not already present. This database becomes the search index.

The pipeline runs iteratively to avoid blocking the UI: it might index in chunks (e.g. 100–500 photos at a time) and marks progress. It will resume on the next batch until all photos are indexed. We use background isolates or tasks so indexing doesn't freeze the UI. The pipeline also respects power/network constraints (for example, it can wait until the device is charging to run intensive embedding computations if desired).

- **Query Engine:** When the user enters a search query, the query engine interprets it and finds matching photos through these steps:
- **Natural Language Parsing (LLM):** The raw query string is sent to a cloud-hosted LLM (OpenAI GPT-4/GPT-3.5 or Google's Gemini, via API) to **parse the query**. The LLM returns a structured interpretation – for example, identifying keywords vs. filters. It may output a JSON with fields like `people: ["Mom"]`, `date_range: [2021-06-01, 2021-08-31]`, `tags: ["office"]`, and a refined textual summary of the visual content requested. *(If the device is offline or the user disabled cloud parsing, the app will fall back to a basic on-device parser for simpler queries.)*
- **Text Embedding (On-Device):** The core of search is still vector similarity. For the semantic part of the query, we take the refined text description (either the original query or a cleaned-up version from the LLM) and run it through the same embedding model used for images (using CLIP's text encoder on-device) [3] . This produces a query embedding vector in the same latent space as the photo embeddings.
- **Vector Search (On-Device):** Perform a **k-Nearest Neighbors (kNN)** search in the `photo_embeddings` vector index for vectors nearest to the query embedding. This is done via a SQLite **vector extension** (`sqlite-vec`), which allows a SQL query like: `SELECT photo_id FROM photo_embeddings WHERE embedding MATCH $query_vec ORDER BY distance LIMIT 500`

[4] [5]. This efficiently yields the top-N candidate images whose embeddings are most similar to the query in CLIP space. (SQLite+ `sqlite-vec` is capable of handling tens of thousands of vectors efficiently on-device [6], which covers our target gallery size.)

- **Apply Filters:** Refine the candidate list by applying any structured filters from the query. For example, if the LLM or user-specified filters indicate a person ("Mom") or a tag (`#office`) or a date range, we filter out any candidates that don't match those criteria (these are checked via joins or lookups in the SQLite tables, e.g. match photo IDs that have that person or tag, or timestamps in range). Geographic filters like *near:Location* would require checking photos with GPS coordinates within that locale (possibly using a cached reverse-geocoding table for city names).
- **Scoring & Ranking:** Compute a final relevance score for each remaining candidate. The base score comes from the vector similarity (distance) – closer embeddings mean more semantically similar. Then adjust the score with small boosts/penalties: for instance, exact tag matches or person matches might boost a photo's rank, and very recent photos might get a freshness boost [7]. If multiple people/tags were requested, those with more matches rank higher. The weighting coefficients (α, β, γ, etc.) can be tuned. Finally, sort by this composite score and take the top results (e.g. top 50). In future, we might feed top results into a re-ranking model or LLM for even better relevance, but MVP will use the simple scoring.

- **Result Display:** Fetch thumbnails for the top results (thumbnails can be cached or quickly loaded via the OS since we have URIs). Present them in the UI grid. The user can then click a photo to view it (and perhaps see its metadata, tags, people, etc., or navigate from there to the system viewer).

- **People System:** This subsystem handles face data and person naming:

- After indexing, we will have a set of face embeddings for all faces detected. We perform clustering on these vectors (using an algorithm like DBSCAN or HDBSCAN) to group faces likely belonging to the same person [8]. This can be done on-device (possibly as part of the indexing job or a separate step after initial indexing).
- The initial clusters are unlabeled. The app provides a UI for the user to review clusters and assign names, creating entries in the `people` table. When a face embedding is associated with a person label, we map that face's ID to the person's ID in a mapping table.

- Over time, if new photos are added and indexed, new face embeddings will be generated. The system can incrementally update clusters (e.g. assign new faces to the nearest existing cluster or form a new cluster if it's distinct). Any clustering algorithm updates will respect already-labeled individuals (we won't merge labeled clusters without user confirmation). This keeps the "People" view up-to-date as the photo collection grows.

- **Storage (Database):** All data is stored in a single **SQLite** database file on the device (one per user/device). The schema (see Section 5) includes tables for photos, faces, people, tags, OCR text, and vector indices [9] [10]. We utilize the `sqlite-vec` extension to store the high-dimensional vectors and perform fast similarity search. The database uses write-ahead logging (WAL) for safe concurrent access, and we'll run periodic maintenance (ANALYZE, VACUUM) when the device is idle to keep the DB efficient. Because this is local, queries are extremely fast (no network latency). Optionally, we could encrypt the database for security (to protect embeddings which are a form of biometric data), though that may have performance implications; at minimum, we rely on OS-level sandboxing for protection. For future cloud sync, the design will allow exporting just the minimal metadata (people

labels, tags, etc.) so they can be backed up without exposing actual face embeddings or photos, unless the user opts in.

- **Background Processing:** In Flutter, background tasks are handled via platform-specific APIs exposed through plugins. We will use the `workmanager` plugin for scheduling the indexing jobs. On Android, this utilizes WorkManager under the hood (which allows constraints like charging or Wi-Fi) and on iOS it uses BGTaskScheduler or a similar API for background processing [11] [12]. The indexer can thus run even if the app is closed (subject to OS scheduling). We ensure to handle background execution limits on iOS (perhaps indexing in smaller chunks when the app is opened if needed, since iOS background time is limited). We will also listen for **photo library change events** (if new photos are added) – using plugin callbacks or manual re-scans – and trigger re-indexing of new items in the background.

On-Device vs Cloud responsibilities: To clarify, **all image analysis** (embedding generation, face recognition, OCR, vector search) is done **locally on the device** for privacy and speed. The **only cloud interaction** in v0.2 is using an LLM API for understanding the **text of the user's query**. This means user photos, thumbnails, and embeddings never leave the device; only the textual query (and perhaps some metadata like a list of user-defined tag names or person names for context, if we include them in the prompt) is sent to the LLM service. This design keeps sensitive data local while leveraging the power of LLMs to improve search understanding.

The system is designed to be modular: if in the future we introduce a cloud-based component (e.g. an option to back up embeddings to a personal cloud account or do heavy re-ranking on a server), it can be added without fundamentally changing the on-device workflow. For now, the cloud is strictly for the LLM query parser and is entirely optional (the app will function with basic search even if the user opts out of LLM usage).

4. Model Menu (Tech Choices for AI Components)

For each AI component, we select models that run efficiently on mobile (via ONNX or similar), balancing accuracy and performance. All on-device models will be integrated using Flutter plugins (e.g. onnxruntime or tflite). Below is the "menu" of models with our chosen defaults for v0.2:

- **A. Global Image Embedding (for text↔image search):** We will use a **CLIP**-based model to embed images and text into a joint vector space. The MVP choice is **OpenAI CLIP ViT-B/32**, which can be quantized and run via ONNX on mobile [13]. This produces a ~512–768 dimensional vector per image. An alternative is **SigLIP-base**, a CLIP-like model with possibly better mobile optimization [13]. We plan to use a quantized ONNX model (~int8 or float16) to reduce size and increase speed. For example, CLIP ViT-B/32's ONNX model quantized might be ~40 MB and can generate embeddings in tens of milliseconds on a modern phone. The embedding vectors (float32) will be stored in the DB (we target 512–768 dimensions to balance recall and storage) [14]. *(Note: If the user opts into cloud features in the future, a cloud embedding service like Google's Vertex AI Multimodal API with 1408-dim embeddings is an option [13], but not for the on-device MVP.)*

- **B. Face Detection & Alignment:** We use **Google ML Kit's Face Detection** on both Android and iOS via Flutter plugins (no custom model needed from our side). ML Kit's face detector is fast and works

offline, providing face bounding boxes and landmarks (required for alignment) [15]. On iOS, ML Kit runs as a separate pod; alternatively, we could use Apple's Vision API (`VNDetectFaceLandmarksRequest`), but for cross-platform consistency we'll stick to ML Kit's face detector. Detected face regions will be aligned (we'll rotate/warp the face using landmarks so that eyes line horizontally, etc., using an affine transform in Dart or a native helper for consistency with how the face recognition model was trained).

- **C. Face Recognition Embedding:** For each detected face, we compute a 512-dimensional embedding using an **InsightFace ArcFace** model [16]. Specifically, we can use a pre-trained model like *resnet50* trained on MS1M (e.g. *w600k_r50*) or a MobileFaceNet variant for efficiency [16]. We will convert this model to ONNX (if not already available) and run it with ONNX Runtime. This yields a 512-D vector per face [17], which we store in the `face_embeddings` index. The model will be quantized to reduce size (ArcFace models can be ~50MB float32, ~12MB quantized int8). InsightFace models are high-accuracy for face identification; combined with clustering, they allow grouping similar faces. We'll ensure to do l2-normalization on the face embeddings before storage if required by the model.

- **D. OCR (Text in Images):** Use **Google ML Kit Text Recognition (v2)** via Flutter (`google_mlkit_text_recognition` plugin). This on-device OCR can detect Latin script (and other languages as supported by ML Kit) without network [18]. We will run OCR primarily on images flagged as potential documents or screenshots (to save time). The recognized text blocks are stored (possibly just concatenated per image, or in blocks for highlighting) in the `ocr_blocks` table for full-text search. This allows queries like "receipt" or "boarding pass" to match text within images.

- **E. Optional Object/Scene Tags:** (This is optional for MVP.) We may integrate a simple on-device object detection or image classification to assign coarse tags (like "beach", "food", "sunset"). ML Kit offers an **Image Labeling** API, or we could run a tiny **MobileNet** classifier [19]. This can enrich search (so a query "beach" could match images even if "beach" wasn't explicitly in metadata). If used, these labels could be stored as additional tags or as a separate field. However, to keep the first version lean, we might skip or make this a later addition. The dominant color extracted is also a form of simple tag that could be used for color-based search (see Bonus idea in section 11).

- **F. Query Parsing (Natural Language to Filters):** For the advanced query understanding, we rely on a **Large Language Model (LLM)** in the cloud. The primary choice is OpenAI's GPT-3.5 or GPT-4 API, given the user's API key and budget. Google's **Gemini/PaLM** could be integrated similarly when available. The LLM is prompted to parse the user's search text into a structured form (identifying dates, people names, etc.) and possibly to rephrase the query in terms of visual concepts. This isn't a model we host, but an API we call. For offline or privacy-only mode, our fallback is a simpler rule-based parser (detect `#tags`, known person names, date keywords) – but the LLM will greatly improve understanding of natural phrases or uncommon expressions. We need to craft prompts to get a concise JSON output from the LLM, which the app can then interpret. This component is an addition in v0.2 to enhance the natural language interface from day one, even though it introduces a cloud dependency for that aspect.

The table below summarizes the tech stack and tools for these components in Flutter:

Aspect	Technology (Flutter)	Purpose/Notes
Framework & Language	Flutter (Dart)	Cross-platform UI and logic for Android/iOS with single codebase.
UI Library	Flutter Widgets (Material/Cupertino)	Build native-feel UI, support dark mode, responsive layouts.
Photo Library Access	`photo_manager` plugin	Enumerate and access photos/videos on device (Android MediaStore and iOS PHAsset under the hood) [20]. Provides URIs, metadata, and thumbnail access.
Image Decoding	Flutter built-in image codecs / `image` package	Load and downscale images for embedding processing (downsample to ~384px for speed as needed).
ML Inference Runtime	`onnxruntime` plugin	Run ONNX models (CLIP, InsightFace, etc.) on-device with a fast Dart API across mobile platforms [21]. Ensures ML models execute locally using CPU/GPU/NN accelerators if available.
Global Image Model	Quantized CLIP ViT-B/32 (ONNX)	512–768D image & text embeddings for cross-modal search [13]. Runs via onnxruntime. Provides `embedImage()` and `embedText()` functions.
Face Detection	`google_mlkit_face_detection`	ML Kit face detector (on-device) for bounding boxes and landmarks. High performance detection for multiple faces.
Face Embedding Model	InsightFace ArcFace (e.g. r50 model, ONNX)	512D face feature vectors [16] [17] for face recognition. Runs via ONNX Runtime plugin.
OCR	`google_mlkit_text_recognition`	ML Kit on-device text recognition. Extracts text from images (Latin and supported scripts) for OCR functionality.
Database	`sqlite3` (dart:ffi) with `sqlite-vec` extension	Local data store for photos/metadata. The `sqlite-vec` extension provides vector similarity search inside SQLite [22] [6]. We load a pre-compiled extension for Android (.so) and iOS (dylib) via FFI.

Aspect	Technology (Flutter)	Purpose/Notes
Database ORM	(Optionally) Drift or sqlite3 direct queries	We can use direct SQL for full control (especially for vector queries). Basic queries use SQLite, vector search uses `MATCH` operator from extension.
Background Tasks	`workmanager` plugin	Schedule background indexing tasks across platforms [20]. Uses Android WorkManager and iOS BGTaskScheduler internally. Allows conditions (charging, etc.).
LLM API	OpenAI GPT-4/GPT-3.5 (HTTP API) or Google PaLM/Gemini API	Cloud service for parsing user's natural language queries into structured search terms. Only query text and necessary context sent; results (filters/parsed intent) returned and applied on device.
State Management	Provider or Riverpod (Dart packages)	Manage app state (e.g. current search results, indexing progress) in Flutter in a scalable way. Not core to tech stack, but for completeness.
Platform Channels	Flutter platform channels (MethodChannel) for any custom native code	Might be used to integrate things like loading the sqlite-vec extension or performing image cropping in native code if needed for speed.

(The above covers the main components of the tech stack in v0.2. All chosen plugins are compatible with both Android and iOS. The ONNX models (CLIP, face) will be bundled with the app assets. We will ensure any native libraries needed (e.g. onnxruntime binaries, sqlite-vec compiled library) are included for both platforms.)

5. Data Model (Database Schema)

We use a **single SQLite database** to store all metadata and embeddings for fast retrieval. Using SQLite means we have a self-contained, queryable index on-device. The schema is largely the same as v0.1 [10] [14], but here we present it in a structured form:

- **Photos Table (`photos`):** Stores one entry per photo in the library (identified by a stable ID).
- **Columns:** `id` (TEXT, primary key – e.g., the MediaStore or PHAsset local identifier), `uri` (TEXT content URI or file path needed to locate the image), `width` (INT), `height` (INT), `created_at_ts` (INT timestamp in epoch ms), `gps_lat` (REAL), `gps_lon` (REAL), `dominant_color` (TEXT, e.g. a hex code or color name), `is_screenshot` (INT as boolean), `hash_sha256` (BLOB for image hash).

- **Purpose:** Core reference for each photo and basic attributes. Allows filtering by date (`created_at_ts`), location, etc. The `dominant_color` can enable color-based queries. A SHA-256 hash can detect duplicates or identify photos uniquely across devices (for future sync).

- **Photo Embeddings Table (`photo_embeddings`):** A virtual table (using `sqlite-vec`) that stores the global image embedding vectors for each photo [14].

- **Schema:** `photo_id` (TEXT, foreign key referencing `photos.id`), `embedding` (FLOAT[] vector of length 768 by default, or 512 if using a smaller model). We create an index on `photo_id` for fast join with `photos`.

- **Purpose:** Enables vector similarity search. The `MATCH` operator on this table with a query embedding returns nearest neighbor photos by cosine distance. This is the backbone of semantic search.

- **Faces Table (`faces`):** Stores metadata for each detected face in each photo.

- **Columns:** `face_id` (TEXT primary key, can be a UID), `photo_id` (TEXT, foreign key to `photos`), `bbox_x, bbox_y, bbox_w, bbox_h` (REAL coordinates for face bounding box in the image), `landmarks` (BLOB, could store coordinates of key facial landmarks or a serialized structure).

- **Purpose:** Records location of faces in photos and links to the face's embedding. Allows drawing face rectangles if needed, and associating multiple faces to one photo.

- **Face Embeddings Table (`face_embeddings`):** A virtual table storing the embedding vector for each face [17].

- **Schema:** `face_id` (TEXT, foreign key to `faces.face_id`), `embedding` (FLOAT[512] vector representing the face features).

- **Purpose:** Supports clustering and face recognition. We can run `MATCH` queries here for face similarity (though for MVP we primarily cluster rather than interactive face search by vector). Each face embedding can be linked to a person once labeled.

- **People Table (`people`):** Stores user-defined person labels.

- **Columns:** `person_id` (TEXT primary key, could be a UUID or slug), `label` (TEXT, e.g. "Mom" or "Alice").

- **Purpose:** Keeps track of named identities. Initially empty until user labels clusters.

- **Face-Person Mapping Table (`face_person_map`):** Links face embeddings to a person.

- **Columns:** `face_id` (TEXT, primary key, also serves as foreign key to `faces.face_id`), `person_id` (TEXT, foreign key to `people.person_id`).

- **Purpose:** When a user names a cluster of faces as "Mom", we update all those faces to map to the "Mom" person_id. This allows querying photos by person. (One face can map to at most one person

in this simple design; if a photo has two people, it will have two face entries, each mapping to their respective person.)

- **Tags Table (`tags`):** Master list of all unique tags the user has created.
- **Columns:** `tag` (TEXT primary key – the tag label itself).
- **Purpose:** Ensures tags are managed consistently (e.g., so we can list all tags or ensure no duplicates differing only by case, etc.). Tags could also include automatically inferred labels if we add those later (like scene categories).
- **Photo-Tags Table (`photo_tags`):** Associative table mapping photos to tags (many-to-many relationship).
- **Columns:** `photo_id` (TEXT), `tag` (TEXT), with a composite primary key (`photo_id, tag`) [23].
- **Purpose:** Allows fast querying of all photos for a given tag and vice versa. Both fields would be indexed (photo_id naturally via primary key and tag via implicit index or separate index if needed).
- **OCR Blocks Table (`ocr_blocks`):** Stores extracted text from photos (particularly screenshots or documents).
- **Columns:** `photo_id` (TEXT), `text` (TEXT).
- **Purpose:** Enables text search within images. For example, if the user searches a word that appears in a screenshot, we can find it by querying this table. We might store one row per photo with all text concatenated, or multiple rows per photo for each text block (for highlighting occurrences – but initially, simpler to store as one blob of text per photo).

Database considerations: We use foreign keys where appropriate (though SQLite requires enabling FK constraints explicitly). The `sqlite-vec` extension provides the `embedding MATCH query_vec` capability for the two virtual tables (`photo_embeddings`, `face_embeddings`). The size of the database for 30k photos: each photo embedding vector (768 floats) is ~3KB (float32) or ~768B (int8 quantized). So 30k images ≈ 90MB (float) or 23MB (quantized) for the embeddings, plus minimal overhead. This is acceptable on modern devices. We will use **WAL mode** for better write performance and crash safety, and consider splitting the DB into multiple files (or using **Room/Drift** ORM) if it simplifies implementation – but a single DB is straightforward. We'll also implement a backup/export routine (user-initiated) to export just the metadata (tags, people labels) to a JSON or to cloud, since images themselves are not stored in the DB.

6. Retrieval Algorithm

The retrieval process combines vector search with filtering logic, incorporating the LLM for query understanding:

1. **Query Parsing:** As described, when a user enters a query string, we first obtain a structured interpretation. For MVP, we call the LLM API with a prompt to parse the query into filters. For example, input: *"photos of Mom at the beach last summer"* might yield an output like: `{ "people": ["Mom"], "keywords": ["beach"], "date_range": ["2022-06-01", "2022-09-01"] }`.

We parse this JSON. If the LLM fails or is unavailable, we fall back to searching the raw text and simple regex rules for things like `before:YYYY` or `#tag`. (The user will be informed if the "smart search" is offline and maybe prompted to simplify query.)

2. **Text to Vector:** Take the core **keywords/description** part of the query – in this case, maybe "Mom beach summer" (the LLM could even return a suggested concise caption like *"person at beach in summer"*) – and encode it into a vector using the CLIP text encoder on-device. This yields the query embedding vector `q_vec` in the same space as the `photo_embeddings`. The text encoder is part of the ONNX model and runs quickly.

3. **Vector Similarity Search:** Query the `photo_embeddings` virtual table using the `MATCH` operator with `q_vec`. This returns a list of candidate photo IDs with their cosine distances (or inner product scores) to the query [4]. We might retrieve the top 200 or 500 candidates to ensure we don't miss some that might be filtered out next.

4. **Apply Filters:** Apply the filters from the parsed query:

5. If specific **people** are mentioned, filter candidates to those photos that contain faces belonging to those people (we check `photo_id` in `faces` joined with `face_person_map` for the given person labels).
6. If a **date range** is given (or a year, etc.), filter by `photos.created_at_ts`.
7. If **tags** or keywords with `#` are present, filter by `photo_tags`.
8. If a **location/place** was specified (and recognized by LLM), we have to interpret it. If it's a city name and we have GPS for photos, we could approximate filtering to photos where `gps_lat, gps_lon` is near that city (this may require a lookup table of city coordinates or using reverse geocoding APIs offline). For MVP, explicit `near:Place` syntax could be supported via a pre-built mapping of a few key places the user mentions frequently (or simply rely on any geotags in EXIF to match the place name string if present).
9. If **"has:ocr"** or a text snippet is included, ensure those photos have OCR text matching.

10. All these filters are intersected with the candidate set from vector search. This intersection can be done in SQL (e.g., adding JOIN/WHERE clauses to the vector query if the extension allows, or doing a second step filter in memory on the candidate IDs).

11. **Scoring:** For each remaining candidate photo, compute a final score. We start from the vector similarity (e.g. use `score = -distance` since smaller distance is better). Then add bonus points for any direct matches: e.g., if the query included a tag and the photo has that tag, add a constant to the score; if the query included a person and the photo has that person, add a boost [7]. We can also slightly boost newer photos if recency seems generally desirable (freshness boost δ). These weights (α for semantic similarity, β for tag, γ for people, etc.) will be tuned by testing. The result is a single relevance score.

12. **Ranking & Results:** Sort the filtered candidates by the final score descending. Take the top N (e.g. 50) for display. The UI will show these in a grid. We also prepare auxiliary data for the UI: e.g., we can compile facet counts (how many of the results belong to each person, each tag, etc., for UI filters) if needed. The thumbnails for these results are loaded (likely via the OS since we have the URI; Flutter's

`Image.memory` or `Image.file` can display them, possibly using a cached thumbnail if available from the MediaStore to save decoding time).

13. **Feedback Loop:** Although not in MVP, the design allows capturing which results the user clicked or found relevant to potentially fine-tune future ranking (learning user preferences). For now, that's out of scope, but logging search analytics locally could be an upgrade path (with user consent).

The retrieval algorithm prioritizes speed: All heavy-lifting (vector math, database filtering) is done in optimized native code (via SQLite extension and ONNX runtime). The amount of data processed per query is small (a few hundred candidate vectors, then some simple filter checks). This should comfortably meet the <150ms goal on a modern mid-range phone for ~30k photos, based on initial estimates and the efficiency of SQLite/vec. (Notably, SQLite with indexes can handle tens of thousands of records quickly, and `sqlite-vec` is optimized in C for vector ops [6].)

7. Platform Stack (Flutter Implementation Details)

This section highlights how the app is implemented on Flutter and how it interfaces with platform-specific capabilities, replacing the native Android/iOS components from v0.1 with Flutter equivalents:

- **Flutter UI:** The entire app UI is written in Dart using Flutter's widget framework. We leverage Material Design components for a familiar look on Android, and adapt styling on iOS (or use Cupertino widgets) as needed for native feel. The UI will include screens/tabs for Gallery, Search Results, People, and Tags. Flutter's performance is sufficient for rendering a grid of photo thumbnails (it can use the Skia engine for fast graphics). We will use `GridView` and `ListView` with builder patterns to efficiently load images on scroll.

- **Photo Access Plugin:** Using the `photo_manager` plugin, we can list and fetch media on both platforms [20]. This plugin provides methods to request permissions, enumerate albums and assets, and get thumbnails or full image files. It abstracts differences (like Android URIs vs iOS local identifiers). We will use it in the Indexer to get all assets IDs and metadata, and also in the Gallery UI to display the latest photos.

- **ONNX Runtime Integration:** The `onnxruntime` Flutter plugin allows us to load ONNX models and execute them with a Dart API [21]. We will include the quantized ONNX model files (for CLIP and face embedding) in the app assets. At runtime, we initialize the ONNX Runtime session for each model (this may involve loading a shared library for onnxruntime itself, which the plugin manages per platform). The models are run either on CPU or with acceleration (the plugin may use NNAPI on Android or CoreML/Metal on iOS if available for ONNX, or just use optimized C++). We expect embedding computation for one image (256–384px resized) to be <100ms on device. For text embedding, it's even faster (<10ms). We will reuse the model sessions to avoid re-loading for each inference. ONNX is chosen for consistency across platforms; as an alternative, TensorFlow Lite models could be used with `tflite_flutter` plugin, but since we have CLIP in ONNX, we stick with ONNX Runtime.

- **ML Kit via Flutter:** Google's ML Kit provides SDKs for Android and iOS, and the Flutter plugins (like `google_mlkit_face_detection` and `google_mlkit_text_recognition`) wrap these. Under

the hood, they use Firebase/Google ML Kit libraries but do **not require a Firebase project** for the on-device APIs. We include these plugins, and when we call them (e.g. detect faces in an image), they will perform platform-channel calls to native code which runs the detection. The face detector returns face coordinates which we then feed into our ONNX face embedding model. The text recognizer returns a String for detected text which we store. These are both asynchronous calls. We need to ensure image data is provided in the format they expect (likely an `InputImage` object, which can be created from a file path or bytes).

- **SQLite and Vector Search:** Instead of Android's Room, we use direct SQLite in Flutter. The `sqlite3` package (dart:ffi) by Simon Binder allows using SQLite with FFI for full performance. We will ship the `sqlite-vec` extension (a compiled C library) and load it into SQLite connection at runtime (SQLite `LOAD EXTENSION` command) – note: on iOS, dynamic loading might be restricted, so we may need to statically compile sqlite-vec into the app binary or use a workaround. Alternatively, since the extension is small, we could compile a custom SQLite amalgamation with the extension included and use that. These are build details to handle, but from the Dart side, usage will be via SQL commands. Once set up, we can create the virtual tables and perform vector queries as illustrated in section 6. Using raw SQL via `sqlite3` gives flexibility especially for the `MATCH` operator. We'll wrap common queries in Dart functions (or use an ORM like Drift with a custom function for `MATCH`). The app logic ensures that any DB write (indexing) happens in the background isolate to not block the UI, and reads (search queries) are fast anyway. We also use transactions appropriately during bulk insert (for performance).

- **Concurrency:** Flutter is single-threaded for the main UI isolate. For heavy tasks like indexing, we have two approaches:

- Use the `workmanager` plugin to perform tasks in background (which actually spins up a separate isolate for the Dart code to run the task). We can schedule the indexing to run periodically or as one-off. The plugin handles ensuring on Android that WorkManager calls into our Dart code even if app is terminated (with some setup).

- Alternatively, if we want more manual control, we could spawn a Dart **Isolate** for indexing when the app is running (and ensure it continues if the app is backgrounded, which might need the plugin anyway to survive termination). In either case, we'll coordinate progress updates via some shared state or callback (e.g. the isolate can send progress messages to the UI isolate).

- **LLM API integration:** The app will call the OpenAI API using Dart HTTP requests (through `dart:io` or `http` package). This is a network call so we do it outside the main isolate (maybe as part of the search operation in a background isolate or using `compute`). The response (parsed JSON) is then merged into the search logic. We will implement a simple cache for LLM outputs for identical queries to save cost if the user repeats a search. Also, we might use a shorter LLM prompt if certain structured syntax is detected to reduce reliance on the API for trivial cases.

- **Cross-Platform Differences:** By using Flutter, 90% of code is shared. Some platform-specific handling remains: e.g., requesting photo permission differs (the plugin covers it), loading the sqlite-vec extension on iOS might require an extra config, and packaging the ONNX models is straightforward (just include in pubspec assets). We'll test on both platforms. One known iOS issue is that background tasks require configuring allowed background modes (in Xcode project capabilities)

and scheduling via BGTaskScheduler requires registering task identifiers. The workmanager plugin documentation will guide these steps. We will ensure to comply with iOS memory limits (e.g., indexing chunk size should be small enough to not use too much RAM at once, especially on older devices).

In summary, the platform stack for v0.2 is unified under Flutter, using plugins to handle what in v0.1 was done with separate Android/iOS APIs [24] [20]. This approach should speed up development and ensure feature parity on both platforms from day one.

8. Privacy & Safety

Privacy is a cornerstone of this app. **By default, no image data or sensitive metadata ever leaves the user's device.** All processing – from face recognition to image embeddings – happens locally. Below we outline key privacy measures and how we address user data safety:

- **On-Device by Default:** As stated, the app does not upload photos, face embeddings, or location history to any server [25]. The search index (including face vectors and OCR text) resides only in a local database. Users can use the core functionality entirely offline. This caters to users who want a completely private photo library search.

- **LLM Query Parsing (Opt-in):** The only feature that involves cloud communication is the LLM-based query understanding. We will make this **opt-in and transparent**. On first use of search, or in settings, we explain that enabling "Natural Language Smart Search" will send your text queries to an AI service (OpenAI or Google) for interpretation. Emphasize that **only the text** is sent – the service does not see the photos or any personal images. Users concerned about even their query text can disable this and rely on on-device search (with more manual filter syntax). For those who enable it, we will allow them to review the returned parsing (maybe in a debug mode or log) to build trust.

- **No Unintended Data Sharing:** We ensure the app only requests necessary permissions (Photos access, perhaps Location if we do on-device reverse geocoding, but not needed if we only use existing GPS in EXIF). We do not request internet permission until/unless the user opts into the LLM feature (though on modern Android, internet is normal permission not shown; we will still document that network is only used for that feature).

- **Face Data Consent:** Faces are considered sensitive biometric data. We will include a specific consent screen explaining the face recognition feature. The user can choose to enable or disable the "People" clustering. If disabled, the app will skip face embedding and clustering entirely (no face vectors stored). They can still use other search features. If enabled, they acknowledge that face data is processed locally. We also provide an in-app option to **clear all face data** (which would delete face embeddings and people labels) any time, for peace of mind.

- **Cloud Sync (Future) Caution:** While the MVP has no cloud backup, the architecture is ready to allow an *optional* cloud sync of non-sensitive data (like tags or people labels) to, say, the user's own cloud storage (Google Drive, iCloud Drive, etc.). If and when we implement that, it will be strictly opt-in and end-to-end encrypted if possible. At this time, v0.2 doesn't include it, but we mention this to users as a possible future feature with reassurance of safety.

- **Data Lifecycle:** Provide controls to the user: a "Reset App / Wipe Data" option to delete the entire index (in case they want to re-index from scratch or are concerned about stale data). Also an export feature for them to download their tag/person data as a file (so they trust they own it).

- **Security Measures:** The local database can be sensitive (e.g. face embeddings could potentially be misused if someone malicious got the phone and reverse-engineered it). We consider enabling SQLCipher (encrypted SQLite) to encrypt the DB at rest. This might be optional, as it could impact performance. At minimum, we rely on the device's security (screen lock, storage encryption on modern iOS/Android) to protect files. We also avoid storing any unnecessary PII. We do store face embeddings and OCR text out of necessity for features, but those are not accessible to other apps due to sandboxing.

- **Abuse Cases:** Because the app indexes photos, we also consider safety: if the user has very sensitive images (medical documents, etc.), those are all local – the risk is only if someone with device access opens the app. We might implement an app lock (PIN or biometric unlock for the app) if users want to protect the photo index from casual access (especially since the app can search things that Photos app might not easily surface).

- **Legal/Compliance:** We will include a privacy policy clearly stating how data is handled. If we consider underage users or jurisdictions like EU, we ensure compliance (e.g., not uploading any biometric without consent covers most points).

In summary, **v0.2 keeps user data on-device and under user control.** Even the powerful LLM integration is designed in a privacy-conscious way. We treat face embeddings as biometric identifiers and thus allow the user to opt out entirely [26] . By building trust through transparency and offering control toggles, we align with the app's privacy-first promise.

9. Milestones Timeline

We plan the project in milestones, each focusing on a functional slice of the app. Below is an outline of the milestones with an estimated sequence (each milestone could correspond to roughly a few days of focused work or one sprint cycle). The app being cross-platform from the start means we'll test on both Android and iOS as we go (no separate porting milestone, unlike v0.1).

Milestone	Description
1. Flutter Gallery Base	*Hello Gallery*: Set up a new Flutter project. Implement photo permission request and use `photo_manager` to load and display a grid of recent photo thumbnails. Verify scrolling performance with a large number of images. Basic UI navigation in place (tabs for Search, People, Tags – though initially empty).
2. Local Database Setup	Integrate the `sqlite3` database. Define the schema (tables for photos, faces, etc. as in section 5) and create them on app startup. Prepare to load the `sqlite-vec` extension (compile and include libs for Android/iOS, test loading it successfully). Write a simple DAO layer or raw SQL functions for inserting and querying. Test inserting a dummy photo record and query it back as a sanity check.

Milestone	Description
3. Indexing Pipeline Implementation	Integrate ONNX Runtime and ML Kit to build the indexing worker. Bundle the quantized CLIP model and InsightFace model in the app. Use `workmanager` to schedule a background task (or run in an isolate for dev testing). In the task: enumerate a batch of photos via `photo_manager`, for each photo do: load downscaled image, compute image embedding (CLIP) [22], detect faces (ML Kit), crop and compute face embeddings [27] [3], run OCR if screenshot. Insert data into the SQLite DB. Mark photos as indexed (to avoid duplicate work on next run). Show a progress UI (maybe a notification or in-app progress bar) reflecting indexing status. By end of this milestone, the app should be able to index, say, the first 100 photos and store results. Measure performance and adjust batch sizes.
4. Basic Search Functionality	Implement a simple search UI screen with a text input and a grid of results. When user enters a query and hits search: embed the query text with CLIP (on-device), perform a SQLite `MATCH` query for nearest images [4], retrieve the top results, and display thumbnails. Start without LLM integration first (i.e. user must use simple keywords or exact tag syntax). Ensure the search is fast (<1s worst case). This milestone essentially proves end-to-end retrieval: type a keyword that you know is in a photo (either via OCR text or a concept like “sunset”), see that photo result appear. We can also include a couple of filter examples (e.g. if user types `#tag`, manually filter by tags by querying the DB).
5. LLM Query Parsing Integration	Connect the OpenAI (or Gemini) API for query parsing. Develop the prompt and logic to send the query and receive a response. Integrate this into the search flow: when search is submitted, call LLM (in parallel, perhaps, to not stall basic search too much). Once the LLM responds (with filters or refined text), apply that to improve the results. This milestone includes handling failure cases (if API fails or times out, we still return basic search results). Also, add a toggle in settings to enable/disable this feature. Write unit tests for the parsing function (feeding sample queries and ensuring it outputs the expected JSON structure).
6. Tagging & Filtering UI	Implement the Tags feature: ability to add/remove tags on photos. For UI, perhaps allow long-press or a “Edit Tags” button on a photo thumbnail or detail view. Also create a Tags screen listing all tags and allowing the user to tap a tag to see photos. In search UI, add a simple filter bar: e.g. icons or chips for filter options (People, Date, Tags) that pop up a selector. Link these to the search function (so selecting a person + date + entering a keyword all combine in the query engine). This milestone focuses on the user experience of filtering and ensuring the query engine respects these filters.
7. OCR Integration	Expand the indexing to include OCR for screenshots/documents. We likely already did this in milestone 3 for data; here we ensure that the search UI can search by text. For example, implement a filter or keyword “has:text” or just rely on the LLM to include OCR text in semantic search. Test by indexing some images with text and searching for a word in that text. Optimize storage of OCR text (maybe full-text index via FTS5 could be used, but given our size, a LIKE query on text is okay). If using FTS5, that would be another extension to compile. Possibly skip FTS for now and do simple text matching.

Milestone	Description
8. People (Faces) UI	Build out the People screen. Show clusters of faces (initially unnamed). For clustering, use an algorithm on the `face_embeddings` table. Since Dart isn't great for heavy math, we might do a simple clustering in Python offline and include the result for testing. But ideally, we integrate a Dart clustering library or write a basic DBSCAN in Dart (there are few thousand faces at most, manageable). Alternatively, we could tag faces by similarity by doing a self-join in SQL (not trivial). For MVP, possibly treat each unique face embedding as its own cluster and later merge via UI – that's not ideal. We aim to get at least grouping by exact matches or so. After clusters are determined, show representative face thumbs for each cluster, allow naming them (which inserts into `people` and `face_person_map`). Then, integrate person filter: e.g. in search or in gallery, filter photos by selected person. Test end-to-end: search "Mom" should bring photos of the person labeled Mom. Ensure unlabelled faces can be searched via face similarity if needed (maybe future). This milestone is one of the more complex, likely will take significant effort to refine clustering accuracy and UI.
9. Performance & UX Polish	At this point, most features are in place. Now refine performance and user experience: Combine scores for better ranking as discussed (semantic + metadata). Implement result re-ranking if needed to ensure e.g. a photo that exactly matches a tag appears higher. Add quick filter chips to the search results page (e.g. if a search returns mixed people, show top 3 people's names as chips to filter further). Improve UI transitions, empty states (what if no result? show a friendly message). Ensure that the app handles large galleries: test with ~10k dummy entries if possible by simulating. Profile memory usage (loading too many images at once?) and fix any issues (perhaps use lower-res thumbs for grid, load higher res on demand). Also, ensure that the indexing process can be paused or it yields to UI (maybe index 100 images, then yield back to message loop to keep app responsive, etc.). Measure query latency on a representative set; tune if above target. Possibly implement caching of last search results or caching query embeddings for frequent searches.
10. Battery/Data Controls & Settings	Implement settings toggles such as: "Index on Wi-Fi only" or "Index only while charging" (Android WorkManager can enforce these; on iOS we have less control but can suggest user keep phone charging). Also, a toggle for "Use Cloud AI for Search" (LLM usage) – tie it to the feature from milestone 5. Provide a manual "Re-index" button in settings in case user wants to refresh everything (maybe if they suspect data inconsistency). Provide an "Export Tags/People" option (which generates a JSON file of just tags and people labels mapping to some photo IDs or hashes) for backup. Also a "Wipe all data" to clear the index and start over (and/or remove face data specifically). These controls empower the user and also help in testing.

Milestone	Description
11. Testing & QA	Before release or beta, do thorough testing on both Android and iOS devices. Fix platform-specific bugs (e.g. iOS photo permission edge cases with "Select Photos" permission – ensure we handle limited library access gracefully). Test on different OS versions. Write unit tests for critical components (e.g. database functions, query parser logic if LLM returns certain format, etc.). Possibly enlist a small group for alpha testing to get feedback on accuracy and speed. Measure indexing time on large library (if too slow, consider multi-thread within isolate or partial indexing approach). At this stage, also update any docs (privacy policy, in-app help explaining how to use search syntax, etc.).
12. (Bonus) App Store Prep	(If time permits in timeline) Prepare for publishing: app icon design, splash screen, App Store/Play Store listings, analytics (privacy-compliant, e.g. basic crash reporting). This includes ensuring we comply with Apple guidelines (face data usage description in Info.plist, etc.). The app will initially be distributed likely as a TestFlight/internal test before full release.

The above timeline is a guideline. Some milestones can be parallelized (e.g. while indexing logic is being built, the UI design can progress). Since Flutter allows hot-reload, development and testing can be iterative and fast. We anticipate an initial working MVP by milestone 4 or 5, and a more feature-complete beta by milestone 9. The LLM integration (milestone 5) might be done earlier in tandem if it's a priority to showcase. Overall, these 12 milestones could be achieved in roughly 8–10 weeks of work with focused effort, given familiarity with the tech.

10. Pseudocode (Flutter/Dart Implementation Snippets)

Below are illustrative pseudocode snippets to demonstrate how parts of the system might be implemented in Flutter/Dart. These are not full code, but outline the structure in a technical manner:

Indexing Worker (Dart, using workmanager): This could be the function that runs in background to index photos in batches. It fetches photos via the plugin, processes them, and stores results in the database.

```
// This function would be registered with Workmanager to run periodically or on
demand
Future<void> indexPhotosTask() async {
  final db = await openDatabase('photo_index.db');
  await loadSqliteVecExtension(db);  // Load the vector extension
  // Determine next batch of photos to index:
  List<AssetEntity> newPhotos = await PhotoManager.getAssetListRange(start: 0,
end: 500);
  for (var asset in newPhotos) {
    final id = asset.id;
    // Skip if already indexed:
    if (await isPhotoIndexed(db, id)) continue;
    // Insert photo metadata:
    await db.insert('photos', {
```

```
      'id': id,
      'uri': asset.relativePath, // or asset.getMediaUrl() depending on plugin,
      'width': asset.width,
      'height': asset.height,
      'created_at_ts': asset.modifiedDateSecond * 1000,
      'gps_lat': asset.latitude,
      'gps_lon': asset.longitude,
      'is_screenshot': asset.title != null &&
asset.title!.toLowerCase().contains('screenshot') ? 1 : 0,
      'dominant_color': null  // will compute later if needed
    });
    // Load image thumbnail for processing (downscale for efficiency):
    final thumbData = await asset.thumbDataWithSize(384, 384);
    final img = decodeImage(thumbData); // use image package to get pixels if
needed
    // Global image embedding (CLIP):
    Float32List clipVec = await onnxModel.run(img);  // Runs the image through
the CLIP model, outputs float[768]
    await db.execute('INSERT INTO photo_embeddings (photo_id, embedding) VALUES
(?, ?)', [id, clipVec]);
    // Face detection:
    final faces = await FaceDetector.detectFaces(img);
    for (var face in faces) {
      // Crop and align face:
      final faceCrop = cropImage(img, face.boundingBox, face.landmarks);
      final faceVec = await onnxFaceModel.run(faceCrop);  // float[512]
      // Create face entry:
      String faceId = Uuid().v4();
      await db.insert('faces', {
        'face_id': faceId,
        'photo_id': id,
        'bbox_x': face.boundingBox.left,
        'bbox_y': face.boundingBox.top,
        'bbox_w': face.boundingBox.width,
        'bbox_h': face.boundingBox.height,
        'landmarks': serializeLandmarks(face.landmarks)
      });
      await db.execute('INSERT INTO face_embeddings (face_id, embedding) VALUES
(?, ?)', [faceId, faceVec]);
    }
    // OCR (if screenshot or likely contains text):
    if ((asset.title??'').toLowerCase().contains('screenshot') ||
isLikelyDocument(img)) {
      final text = await TextRecognizer.recognizeText(img);
      if (text.isNotEmpty) {
        await db.insert('ocr_blocks', { 'photo_id': id, 'text': text });
      }
    }
```

```
    // (Optional) compute dominant color:
    // await updatePhotoDominantColor(db, id, computeDominantColor(img));
  }
  // If more photos left, schedule another task or loop continues...
  return;
}
```

Explanation: This pseudocode loops through photos, inserts metadata into `photos`, computes embeddings via ONNX models (CLIP and face model), and inserts into respective tables [27] [3]. It uses placeholders for how onnxruntime might be called (`onnxModel.run(img)` returning a Float32List of embedding). The `FaceDetector.detectFaces` and `TextRecognizer.recognizeText` represent ML Kit plugin calls. We handle each face found and do a separate embedding. The code is synchronous for clarity, but in practice we'd `await` each async call. Also, batching and error handling omitted for brevity. This runs in background (not blocking UI). We'd likely call `Workmanager.registerOneOffTask` to start this, or just run it on a separate isolate from the main app, depending on app state.

Search Query Execution (simplified pseudocode):

```
Future<List<PhotoResult>> searchPhotos(String query) async {
  final db = await openDatabase('photo_index.db');
  String? parsedKeywords = query;
  List<String> filterTags = [];
  String? filterPerson;
  DateTime? filterDateBefore, filterDateAfter;

  if (useLLM) {
    final parseResult = await callLLMParseAPI(query);
    // parseResult might be a JSON with fields
    parsedKeywords = parseResult['search_text'] ?? query;
    filterTags = parseResult['tags'] ?? [];
    filterPerson = parseResult['person'];  // assume one person for simplicity
    filterDateBefore = parseResult['before'] != null ?
DateTime.parse(parseResult['before']) : null;
    filterDateAfter  = parseResult['after'] != null ?
DateTime.parse(parseResult['after']) : null;
    // (Handle location similarly if present)
  } else {
    // Basic parsing: e.g., split by spaces, check for # or before:/after:
    for (var token in query.split(' ')) {
      if (token.startsWith('#')) filterTags.add(token.substring(1));
      if (token.startsWith('person:')) filterPerson = token.substring(7);
      if (token.startsWith('before:')) filterDateBefore =
DateTime.parse(token.substring(7));
      // ... and so on
    }
    parsedKeywords = query;  // use the raw query as keywords
```

```
  }

  // Get query embedding (CLIP text):
  final textVec = await onnxModel.embedText(parsedKeywords!);
  // Perform vector search using sqlite-vec extension (cosine distance):
  final rows = await db.rawQuery('''
      SELECT p.id, p.uri, v.distance
      FROM photo_embeddings v
      JOIN photos p ON p.id = v.photo_id
      WHERE v.embedding MATCH ?
      ORDER BY v.distance ASC
      LIMIT 200;
    ''', [textVec]);

  // Filter results in memory for simplicity (could also add to SQL query via
JOIN):
  List<PhotoResult> results = [];
  for (var row in rows) {
    String photoId = row['id'];
    double dist = row['distance'];
    // Apply filters:
    if (filterPerson != null) {
      // Check if photoId has a face mapped to filterPerson
      final personRows = await db.query('face_person_map fpm JOIN faces f ON
fpm.face_id=f.face_id',
                                        where: 'fpm.person_id = ? AND f.photo_id
= ?',
                                        whereArgs: [filterPerson, photoId]);
      if (personRows.isEmpty) continue; // skip this photo
    }
    if (filterTags.isNotEmpty) {
      // ensure all filterTags are in photo_tags for this photo
      for (var tag in filterTags) {
        final tagRows = await db.query('photo_tags', where: 'photo_id=? AND
tag=?', whereArgs: [photoId, tag]);
        if (tagRows.isEmpty) {
          continue; // this photo doesn't have required tag
        }
      }
    }
    if (filterDateBefore != null || filterDateAfter != null) {
      int ts = row['created_at_ts'];
      if (filterDateBefore != null && ts >=
filterDateBefore.millisecondsSinceEpoch) continue;
      if (filterDateAfter != null && ts <=
filterDateAfter.millisecondsSinceEpoch) continue;
    }
    // If passed all filters, calculate a combined relevance score:
```

```
    double score = -dist;  // base score (larger = more relevant)
    if (filterTags.isNotEmpty) score += 0.1 * filterTags.length;
    if (filterPerson != null) score += 0.2;
    // (freshness boost or other adjustments can be added)
    results.add(PhotoResult(photoId, row['uri'], score));
  }
  // Sort by score descending:
  results.sort((a, b) => b.score.compareTo(a.score));
  return results.take(100).toList();  // top 100 results
}
```

This pseudocode outlines how a search query might be handled: calling the LLM if enabled to parse filters, embedding the text via CLIP on-device, then performing the SQL vector search [5]. The filtering is done after obtaining candidates (for clarity). In a real app, we'd incorporate the filters into the SQL (e.g., join with `photo_tags` and `face_person_map` in the query to filter server-side), which would be more efficient. But the concept stands. We calculate a simple combined score and then sort. The result is a list of `PhotoResult` with photo IDs/URIs ready to display. We would then use `photo_manager` to get actual thumbnail images for those URIs to show in the UI.

These snippets demonstrate the main logic in a simplified form. The actual implementation will be more robust (with error handling, optimizations, etc.), but the goal is to confirm the feasibility and guide development.

11. Open Questions & Future Upgrades

Even with v0.2's expanded scope, there are features and improvements left for future versions. Below are some open questions and potential enhancements that we can consider down the road:

- **Event & Location Grouping:** How to make the app understand events or trips? We could use time + GPS clustering to group photos into "events" (e.g. a weekend trip) automatically [28]. This could power a "Moments" or timeline view, or be used to boost search (if a query mentions "last Christmas", we know the date range). Implementing this would involve clustering by timestamp proximity and possibly location proximity.

- **On-Device Captioning:** Currently, we rely on the user's query and CLIP embeddings. We could improve recall by generating text **captions for each photo** (e.g., "A man on a beach at sunset") using a vision-language model like BLIP-2 or a smaller captioning model [28]. A mobile-friendly caption model (perhaps a distilled version or something like SigLIP's text decoder) could run on-device to add textual metadata to photos, which the search can then utilize. This would be a heavy feature (in terms of model size and compute), so it's an optional future upgrade – but it aligns with the goal of better semantic search.

- **Improved LLM Integration:** In v0.2 we use LLM for query parsing. In future, we might also use an LLM for **re-ranking results** or answering questions about photos. For example, a server-side LLM could consider the top 10 images and a detailed query to choose the best results (when cloud is enabled) [29]. We could also explore using smaller on-device language models in the future to reduce

reliance on external APIs (as on-device models become more capable, the parsing could happen locally).

- **Cross-Device Sync:** Many users have multiple devices; a future enhancement is to sync tags and people labels across devices (without syncing the actual photos). This could be done via iCloud/Google Drive by storing a small metadata file [29]. The challenge is merging indexes and handling different photo identifiers on different devices. For now, it's a later consideration once the local functionality is solid.

- **Personalized Face Recognition:** To improve face clustering accuracy, we could incorporate a few-shot learning approach per user – for example, fine-tuning or using an adapter for the face model on the user's specific family/friends to better distinguish them [29]. This is complex and would be a later-stage research project, possibly unnecessary if the base model works well. But it's a thought for making the People feature more accurate over time (learning from corrections user makes).

- **Scalability Limits:** We assume 10k–30k photos; beyond that (say 100k), performance might dip. An open question is how far we can push SQLite and on-device models. We might eventually need to optimize by downsampling embeddings (PCA to 256 dims) or use approximate NN search. We should monitor memory usage too (storing many embeddings in memory if we ever do that). This is something to profile in larger real-world scenarios.

- **UI Enhancements:** Add features like a map view (show photos on a map if GPS available), or a timeline view (scroll through photos by date), or even a "memories" feature that surfaces random past photos. These are not core to search but enrich the app.

- **Bonus Idea – Color Filter:** A small addition that could be implemented with minimal effort is a **color-based filter**. Since we already extract the dominant color of each photo, we can allow the user to filter/search by color. For example, a user could search "red flowers" and the system can boost/filter photos whose dominant_color is red. Or we could present a simple UI to filter by color (show a palette of common colors in the library). This leverages existing data with a straightforward UI and query tweak.

Each of these ideas will be weighed against complexity and user value. The plan for v0.2 focuses on the essentials; subsequent versions (v0.3, v0.4, etc.) can pick up these enhancements once we validate the core functionality.

12. Ready-to-Start Checklist (Flutter Edition)

Finally, as we gear up for development, here's a checklist of setup tasks and key steps to ensure we have everything in place to start building the AI Photo Finder app in Flutter:

- [ ] **Project Initialization:** Create a new Flutter project. Set up package dependencies in `pubspec.yaml` for all required plugins: `photo_manager`, `onnxruntime`, `google_mlkit_face_detection`, `google_mlkit_text_recognition`, `sqlite3` (and possibly `path_provider` for DB file path), `workmanager`, etc. Ensure the project is configured

for Android (enable storage permission in AndroidManifest) and iOS (add photo library usage description in Info.plist, and any background modes needed for tasks).

- [ ] **UI Skeleton:** Implement basic screens using Flutter: a Home screen with a bottom navigation bar or tabs for "Photos", "Search", "People", "Tags". Set up routing or state management for these. Ensure the app can launch and show an empty photo grid, to be populated later.
- [ ] **Permissions:** Implement runtime permission request for photo library access using `photo_manager`'s permission request API. Test that on Android (especially Android 13+ with media permission) and iOS (the permission dialog appears and we handle if user gives limited access).
- [ ] **Database Schema Setup:** Using `sqlite3` FFI, create the SQLite database file. Execute the SQL commands to create tables as defined in section 5. Verify that the extension `sqlite-vec` can be loaded: this may involve adding the compiled `.so/.dylib` to the project. (On Android, put it in `android/src/main/jniLibs`; on iOS, possibly integrate via Pod or Xcode project). Test opening the DB and `SELECT 1` to ensure it works.
- [ ] **ONNX Runtime Integration:** Add the ONNX model files to the project assets (e.g., `assets/models/clip_image.onnx`, `clip_text.onnx` if separate, and `face_recognition.onnx`). Load these using the `onnxruntime` plugin and run a test inference (maybe use a known image for which we expect some output). Check that the plugin works on both platforms (the first run might be tricky if missing libs, so verify early).
- [ ] **Indexing Workflow:** Implement a preliminary version of the indexing in Dart. Possibly start with a simpler approach (index small number of photos on button press) before wiring into workmanager. Once logic is tested in foreground, integrate with `workmanager` to run in background. Take care of iOS setup for background task (application constraints, e.g. call `Workmanager.initialize` in `didFinishLaunchingWithOptions`). Ensure the indexing can be started and that it writes to the DB. Add a simple progress indicator in UI (even just a text "Indexed X of Y photos").
- [ ] **Search Functionality:** Implement the text embedding call with onnxruntime. Then write a function to perform a query on `photo_embeddings` with `MATCH`. Ensure that we can retrieve results from the database. Start with no filters to confirm the vector search works (maybe manually insert a test embedding and query the same to see if it returns). Then connect this to a basic search UI (a TextField and a search button).
- [ ] **Tag Management:** Create UI to add a tag to a photo (perhaps in a photo detail view). This involves writing to `tags` and `photo_tags` table. Also, implement a UI list of tags (maybe just a simple list for now) to verify that we can store and retrieve tags. Adjust the search parsing to recognize `#tag` (or use a button to filter by a tag).
- [ ] **Face Detection & People:** Integrate the ML Kit face detection plugin. Write a test to run face detection on a sample image (maybe embed a test image in assets with a face) to ensure it returns results. Then integrate this into indexing as planned. Create a simple People screen that queries the `people` table and displays entries. At first, we might manually insert a dummy person to test UI. Then implement the flow: list distinct face clusters (for now, just list all faces as a placeholder), and allow naming one (insert into `people`, map in `face_person_map`). This is a complex feature, so starting early with basics helps.
- [ ] **LLM API Setup:** Store the API keys securely (perhaps just configure via an environment variable or have user input). Write a function to call the OpenAI API (using `dart:io` HttpClient or `http` package). Test parsing a fixed prompt to ensure we get a response. Handle JSON decoding. This can be done in a separate dev environment (like writing a unit test that calls the API with a sample prompt). Once working, integrate into the app's search flow behind a toggle.
- [ ] **Testing on Devices:** Throughout development, regularly test on a physical Android device and an iPhone. Pay attention to performance (especially of ONNX model runs and SQLite queries). Verify

that on iOS, all permissions and background tasks work (since iOS can be tricky with Photos permission – e.g., ensure the app can access all photos or handle limited selection). Also, test on an older or lower-end device to gauge speed of indexing and search, making adjustments (like smaller image resize if needed).

- [ ] **Performance Tuning:** Before releasing, index a large number of photos (if available, say 10k) and measure indexing time and query latency. Optimize as needed: e.g., use smaller model variant if too slow, or reduce vector dimensionality if DB query is slow. Ensure memory usage stays reasonable (maybe process in chunks to avoid OOM). Also test the LLM parse latency – might add a loading spinner while it's parsing if it takes more than a second.
- [ ] **Finalize UI/UX:** Clean up the UI, add icons (person icon for People tab, tag icon, etc.), and ensure consistency. Write any onboarding messages (first run dialogs explaining privacy, etc.). Double-check that all user-facing text is clear and technical jargon-free (for general users). Possibly implement a dark mode toggle if not automatically handled.

With all items checked, we should be ready to build and distribute a test version of **AI Photo Finder v0.2**. The combination of Flutter for development speed and our careful choice of on-device ML tech will, hopefully, result in a private, fast, and smart photo search experience across both Android and iOS.