LIFE MIRROR — Agent-based Backend: `instructions.md`

Purpose: This document is a step-by-step implementation guide for refactoring the LifeMirror backend into a modern, agent-based architecture using **LangGraph** + **LangChain**, with **Guardrails** for safety, **DSpy** for prompt optimization, and **LangSmith** for tracing & evals. It excludes any voice/audio analysis — do **NOT** implement the voice agent. It **does** include image, photo, and **video** handling and storage.

Table of contents

---

1) Goals & constraints

Primary goal: Replace the monolithic Flask backend with an agent-based, traceable, safe, modular pipeline that:

- Uses **LangGraph** for orchestration and **LangChain** for tools/pipelines.
- Adds strict **Guardrails** (input + output validation, format enforcement).
- Uses **LangSmith** to trace, log, and evaluate runs continuously.
- Uses **DSpy** to automatically optimize prompts during development.
- Stores user profiles, past conversations, images and **videos**, embeddings, and structured analysis results in an auditable, efficient manner.

Important constraints:

- The system must **not** include any voice/ audio analysis (explicitly excluded).

- Respect user privacy: require explicit consent for biometric-like analysis.
- Keep outputs parsable (strict JSON) for downstream tooling and for Guardrails to validate.

---

2) High-level architecture

```
Client (web/mobile)
   └─> API Gateway (FastAPI)
         └─> LangGraph Orchestrator (service)
                ├─> Vision Agent(s)  (FaceAgent, FashionAgent, PostureAgent)
                ├─> Bio/Text Agent     (BioAgent)
                ├─> Embedder Agent    (images/videos/text -> embeddings)
                ├─> Memory/Retriever  (Vector DB + PG cache)
                └─> Aggregator/Formatter -> Response

Infrastructure:
 - Object storage (S3/MinIO)
 - Relational DB (Postgres + pgvector OR MongoDB)
 - Vector DB (Qdrant/Weaviate/Milvus/Pinecone) *or* pgvector
 - Task queue (Celery/RQ/Dramatiq) + worker pool (GPU workers for CV)
 - Model serving: local GPU inference (YOLO/Pose), cloud LLM API (OpenAI/OpenRouter)
 - Guardrails service (guardrails.ai policies enforced in nodes)
 - LangSmith for tracing and evals
 - DSpy for prompt tuning (dev only)
```

Notes:

- LangGraph orchestrates flows and calls the Agents (each agent can be a node that internally uses LangChain tools).
- Heavy CV models (YOLO, pose estimation) should run in separate GPU-enabled worker processes or a model-serving layer.

---

3) Prerequisites & accounts

- Cloud account for object storage (AWS/GCP/Wasabi) or self-hosted MinIO
- PostgreSQL (managed or cloud) or MongoDB
- Vector DB account (Pinecone / Qdrant / Weaviate / Milvus) OR pgvector on Postgres
- LangSmith account + API key
- Guardrails.ai account / library available
- DSpy access (for prompt optimization) and API/CLI
- LLM provider credentials (OpenAI, OpenRouter, Anthropic, or whichever you use)
- GPU-enabled host(s) for heavy CV inference (NVIDIA GPUs) or cloud GPU instances

---

4) Repo layout and coding patterns (suggested)

```
/ (repo root)
├─ /src
│   ├─ /api                # FastAPI app that exposes HTTP endpoints
│   ├─ /agents             # LangGraph node implementations
│   │    ├─ orchestrator.py
│   │    ├─ vision_agent.py
│   │    ├─ face_agent.py
│   │    ├─ fashion_agent.py
│   │    ├─ posture_agent.py
│   │    ├─ bio_agent.py
│   │    └─ aggregator_agent.py
│   ├─ /tools               # LangChain tools & wrappers for models & CV
│   ├─ /models              # model-serving clients (YOLO, pose, BLIP wrappers)
│   ├─ /storage             # object store helpers, thumbnailing, video keyframe utils
│   ├─ /db                  # migrations, ORM models (SQLAlchemy or Prisma)
│   ├─ /schemas             # pydantic models & Guardrails schemas
│   └─ /tests
├─ /infrastructure         # k8s manifests, docker-compose, terraform
├─ /docs
└─ /scripts
```

Coding patterns:

- Use **FastAPI** for HTTP endpoints.
- Each Agent exposes a well-typed interface: `run(input: dict, context: dict)-> dict`.
- All LLM and external API calls should be wrapped with a client library that adds: request metadata, timeouts, retries, and LangSmith instrumentation.
- Keep I/O pure — agents should not write to DB directly but return structured outputs to the Orchestrator which will persist them.

---

5) Infrastructure & services to provision (step-by-step)

1. Provision relational DB (Postgres) with `pgvector` extension OR provision managed vector DB.
2. Provision object storage bucket for user media (images + videos). Enable server-side encryption.
3. Provision or install vector DB (if not using pgvector). For early stages use **pgvector** to reduce system complexity.
4. Provision a Redis instance for Celery/RQ broker and caches.
5. Provision GPU-enabled worker pool for CV inference (YOLO, pose). These can be Docker containers on GPU VMs or k8s nodes.
6. Create secrets store and rotateable env var system (AWS Secret Manager, Vault).
7. Configure monitoring (Prometheus + Grafana) and error tracking (Sentry).
8. Acquire LangSmith and Guardrails keys; set up DSpy access.
9. Configure CI/CD pipelines (GitHub Actions / GitLab CI) and deployment manifests.

---

6) Data model & storage design

High-level decisions:

- Use **object storage** for raw images and video (store path/URL in DB).
- Use **Relational DB (Postgres)** for structured data (users, consents, analysis metadata, sessions). Use `pgvector` if you prefer to keep embeddings in Postgres.
- Use **Vector DB** (Qdrant/Weaviate/Milvus/Pinecone) if you need scale or multi-modal vector search features.

Suggested tables (Postgres)

users

```
id UUID PRIMARY KEY,
email TEXT,
name TEXT,
created_at TIMESTAMP,
consent JSONB -- track what user consented to
```

media (images & videos)

```
id UUID PRIMARY KEY,
user_id UUID REFERENCES users(id),
media_type VARCHAR('image','video'),
storage_url TEXT,
thumbnail_url TEXT,
keyframes JSONB, -- list of thumbnails + offsets
size_bytes BIGINT,
mime TEXT,
created_at TIMESTAMP,
metadata JSONB -- exif, camera info
```

analyses

```
  id UUID PRIMARY KEY,
  media_id UUID REFERENCES media(id),
  agent VARCHAR,
  result JSONB,
  confidence FLOAT,
  created_at TIMESTAMP,
  langsmith_run_id TEXT
```

conversations (text history)

```
  id UUID PRIMARY KEY,
  user_id UUID,
  session_id TEXT,
  messages JSONB, -- [{role, content, ts}]
  created_at TIMESTAMP
```

embeddings_meta

```
  id UUID PRIMARY KEY,
  media_id UUID NULL,
  conversation_id UUID NULL,
  text_excerpt TEXT,
  vector VECTOR, -- if pgvector used
  vector_id TEXT -- pointer to external vector DB
  created_at TIMESTAMP
```

Important: keep `raw images` & `videos` ONLY in object storage and never embed binary in the DB.

---

7) Media ingestion (images + videos) — secure upload flow

Upload flow (recommended):

1. Client requests presigned upload URL from API (FastAPI endpoint). API validates user, file type and size limit, then returns presigned URL.
2. Client uploads directly to object storage (S3 or MinIO) using the presigned URL.
3. Client sends a `media.create` request to API with the `storage_url` , mime type, and metadata.
4. API stores media record in DB, returns `media_id` .
5. API enqueues background jobs for heavy processing (embedding generation, keyframe extraction, face detection, posture detection) with `media_id` .
6. Workers pick up jobs, download media from object storage, process (generate thumbnails, keyframes, embeddings), write analysis results back to `analyses` table and store embeddings in vector DB.

Video-specific processing:

- When job runs on a video: extract keyframes using shot detection (ffmpeg + pyannote or ffmpeg select filters).
- Generate a **poster** (single representative thumbnail) and 3-10 **keyframes** (scene-level or sampled at fixed intervals) depending on duration.
- For each keyframe: generate image embeddings and object/pose detections.

- Optionally: compute an aggregated video-level embedding by averaging keyframe embeddings or using a pooling network.
- Record the `keyframes` list (offset seconds + thumbnail URL) in the `media` table.

Sizing & quotas:

- Max image size: 10MB (adjustable by plan)
- Max video size: 100MB or user-plan-specific; use chunked/ multipart uploads for large files.
- Enforce server-side scan for viruses if possible.

---

8) Agents — responsibilities, inputs/outputs, models, guardrails & starter prompts

> **Guiding rule**: every agent must accept a typed `input` JSON and MUST return a typed `output` JSON that conforms to a Guardrails-compliant JSON schema.

8.0 Agent list (no voice)

- `OrchestratorAgent` (LangGraph root)
- `FaceAgent` (face detection + Face++/landmarks + heuristics)
- `FashionAgent` (YOLO clothing detection + LLM fashion critique)
- `PostureAgent` (pose detection + biomechanical heuristics)
- `BioAgent` (text/bio vibe analysis)
- `EmbedderAgent` (text/image/video -> embeddings)
- `MemoryAgent` / `RetrieverAgent` (semantic search, recall past analyses)
- `AggregatorAgent` (combine outputs, compute final scores)
- `FormatterAgent` (produce final JSON and human summary)
- `CompareAgent` (compare with celebs / past self)

8.1 `FaceAgent`

Responsibility: detect face(s), return Face++-style attributes (age, gender, emotion, beauty), landmarks, and MediaPipe landmarks; compute image-based skin/sampling metadata.

Input:

```
{ "media_id": "...", "storage_url": "...", "options": { "require_landmarks": true } }
```

Output (Guardrails JSON skeleton):

```
{
  "faces": [
    {
      "face_id": "uuid",
      "bbox": [x,y,w,h],
      "landmarks": {"mp_index": {"x":123,"y":456}, ...},
      "age": 28,
      "gender": "male",
      "emotions": {"happiness": 0.8, "sadness":0.0, ...},
      "beauty": {"male_score": 70, "female_score": 65}
    }
  ],
  "inference_metadata": {"detector": "facepp", "model_version":"v1"}
}
```

Models/Tools: Face++ API (or a local face detector), MediaPipe FaceMesh, a wrapper to normalize different providers' outputs.

Guardrails: require `faces.length > 0` for downstream attractiveness/posture flows; if `faces.length == 0`, return a structured message with `error_code: no_face_found`.

Starter prompt pattern (LLM use-case): Use LLM only for summarizing face attributes or drafting human-friendly text. Always ask the LLM to output strict JSON and add a schema in the prompt.

8.2 `FashionAgent`

Responsibility: detect clothing items (object detection with YOLO/DETR), call LLM to produce critique and improvement suggestions, rating (1–100), and itemized suggestions.

Input:

```
{ "media_id": "...", "keyframes": [...], "detected_items": ["jacket","jeans"] }
```

Output:

```
{
  "outfit_rating": 78,
  "items": ["jacket","jeans","sneakers"],
  "good": ["color coordination","fit"],
  "bad": ["wrinkled shirt"],
  "improvements": ["try a darker shoe","iron the shirt"],
  "roast": "friendly roast string (optional)"
}
```

Models/Tools:

- Local YOLOv8 detection (GPU worker) for item detection.
- CLIP/zero-shot classification for aesthetic labels (optional).
- LLM (e.g., Llama or OpenAI) for critique — controlled by Guardrails to return JSON only.

Guardrails: the LLM must return `outfit_rating` in [0,100] and `items` must be subset of detected YOLO items. If mismatch, agent should fallback to a deterministic rule: e.g., item suggestions = YOLO items.

Starter prompt skeleton (strict JSON):

```
System: You are a fashion critic that MUST output only valid JSON. Schema: {outfit_rating:int, items:list, good:list
User: Analyze the following detected items: [LIST] and the image URL: [URL]. Produce concise JSON.
```

8.3 `PostureAgent`

Responsibility: use a pose detection model to extract keypoints, compute biomechanical scores (head-neck, shoulder symmetry, spine verticality, hip tilt), and return per-component scores plus an overall posture score.

Input:

```
{ "media_id": "...", "keyframe": {"url":"...","offset":2.3} }
```

Output:

```
{
  "keypoints": {"nose":[x,y],"left_shoulder":[x,y],...},
  "scores": {"head_neck": 80, "shoulders": 65, "spine": 72, "legs": 88},
  "final_score": 76,
  "analysis_text": "..."
}
```

Models/Tools:

- YOLOv8 pose model or OpenPose, Mediapipe Pose.
- Worker must normalize pixel distances by torso size (shoulder width) to avoid scale-dependence.

Guardrails: enforce numeric ranges and require `final_score` between 0 and 100. If keypoints missing, return `error_code: insufficient_keypoints` and a helpful user-facing message.

Engineering notes:

- Break posture scoring into small deterministic functions (head-neck, shoulders, spine, hips, legs). Unit test each.
- Avoid claiming medical diagnoses; produce only heuristic suggestions and include a disclaimer.

Starter prompt: LLM only used to convert numerical insights into human-friendly advice; the heavy lifting must be deterministic code.

8.4 `BioAgent` (text/bio analysis)

Responsibility: analyze user-written bio or conversation text to derive "vibe" categories, suggested improvements, and a short friendly summary.

Input:

```
{ "text": "user bio or profile text", "context": {"past_analyses": [...]} }
```

Output:

```
{ "vibe_summary": "Carefree & playful", "strengths": [...], "weaknesses": [...], "improvements": [...], "confide
```

Models/Tools:

- LLM (with Guardrails enforcing JSON format), embeddings for retrieval.

Guardrails: enforce non-judgmental language and disallow making sensitive attribute inferences.

Starter prompt skeleton:

```
System: You are an empathetic advisor. Always answer with valid JSON. Avoid identity-based judgments. Provide streng
User: Analyze the following text: "[TEXT]". Use context: [context].
```

8.5 `EmbedderAgent`

Responsibility: convert images, keyframes, videos, and text to embeddings and store them in the vector DB with metadata.

Input:

```
{ "media_id": "...", "type": "image|keyframe|video|text", "url": "..." }
```

Output:

```
{ "vector_id": "...", "db": "qdrant", "dim": 1536 }
```

Video-specific:

- For videos, compute frame-level embeddings then pool (mean/weighted) to produce a single `video_embedding_id` plus keep per-keyframe embeddings for fine-grained search.

Models/Tools: CLIP-based image embeddings, or provider embeddings (OpenAI or self-hosted CLIP).

Guardrails: reject unsupported file types; if embedding fails, return `error_code` and retry with lower-res thumbnail.

8.6 `MemoryAgent` / `RetrieverAgent`

Responsibility: given a query (image, text) return relevant past analyses or profile snippets using vector similarity + filters (user_id, date range).

Input: search vector + filters
Output: list of `embedding_meta` with link to media/analysis

Design:

- Implement hybrid retrieval: semantic (vector) + structured filters (Postgres). This reduces false positives.

8.7 `AggregatorAgent` & `FormatterAgent`

Aggregator:

- Responsibility: combine outputs from FaceAgent, FashionAgent, PostureAgent, BioAgent and compute final composite scores (e.g., attractiveness ensemble) and a short explanation.
- Must preserve all intermediate outputs and attach `langsmith_run_ids` for traceability.

Formatter:

- Responsibility: produce final API response and a plain-language summary for UI. Must enforce Guardrails JSON schema and attach `confidence` fields.

Guardrails:

- Final output schema must be validated by Guardrails before returning to the client. If Guardrails flags a violation (e.g., disallowed content), the Orchestrator must sanitize output and include a `warning` field.

---

9) LangGraph orchestration design & node definitions

High-level flow — For a single `analyze/selfie` request:

1. **Pre-validate** (Guardrails) the request (size, mime type, consent).
2. **Create media entry** (persist URL + metadata) and return `media_id` to Orchestrator.
3. **Parallel**: call `EmbedderAgent` (thumbnail + keyframe embeddings), `FaceAgent`, and `FashionAgent` on selected keyframes.
4. **Wait** for heavy tasks (if queued) — Orchestrator receives callbacks or polls job status.
5. **Call** `PostureAgent` (may be parallel to FaceAgent) on portrait keyframe(s).
6. **Call** `BioAgent` if user provided a bio text.
7. **AggregatorAgent** merges all results and computes composite scores.
8. **FormatterAgent** validates final JSON via Guardrails and returns result.
9. **LangSmith**: record run traces and attach `run_id` in analysis DB record.

Node-level rules:

- Every node should accept a `context` object containing: `user_id`, `media_id`, `langsmith_run_id`, `consent_flags`, and `retrieval_snippets`.
- Nodes must be idempotent. If a node fails due to external API error, it should retry with exponential backoff (configurable).
- Long-running nodes should offload work to background workers and return a `job_id` so the orchestrator can resume later.

Error handling:

- If a critical node fails (e.g., FaceAgent crashed), Orchestrator decides whether to:
  - Retry (transient error)
  - Skip the node and continue (if optional)
  - Abort with error (if required)
- Log error details and LangSmith trace.

---

10) LangChain Tools & wrappers (how to implement)

- Wrap each external model call inside a LangChain Tool that implements `run(input)` and returns structured output.
- Each tool should:
  - Accept typed inputs (Pydantic models)
  - Add LangSmith metadata (tool name, version)
  - Have a retry wrapper with exponential backoff
  - Sanitize and validate outputs using Guardrails before returning to agent logic

Example Tool interface (Python / pseudo):

```
class FaceTool(BaseTool):
    name = "face_tool"
    def run(self, media_url: str, options: dict):
        # request face++ or mediapipe
        # validate output
        # return structured dict
```

---

11) Guardrails: schemas and integration patterns

Why: Guardrails enforces both input and output contracts for LLMs and agents. This is crucial for predictable JSON outputs.

How to integrate:

- Add Guardrails checks at:
  - API input layer (file type, size, consent)
  - After each LLM call (validate JSON schema) — if invalid, retry the LLM once with a stricter prompt
  - Before final response (Aggregator/Formatter) — if invalid, block response and return sanitized error.

Example Guardrails JSON schema (simplified):

```
- name: fashion_output
  fields:
    - name: outfit_rating
      type: integer
      required: true
      constraints:
        min: 0
        max: 100
    - name: items
      type: list
      items:
        type: string
    - name: improvements
      type: list
      items:
        type: string
```

Fallback strategy: If LLM repeatedly fails to produce valid JSON, use deterministic rules built from detector outputs (e.g., item list = YOLO detections).

---

12) Prompt design & context engineering principles (for the later prompt-generation step)

Core principles:

1. **System role** always sets boundaries (tone, safety, required JSON schema).
2. **Format constraint**: demand exact JSON and provide schema in the prompt.
3. **Few-shot examples**: include 1–3 minimal examples of *correct input -> correct JSON output*.
4. **Temperature**: keep inference `temperature=0` or very low when strict output required.
5. **Chunk context**: supply only the minimal context required (detected items, small embeddings snippet), not the whole user history.
6. **Memory summarization**: supply a short one-paragraph summary of past interactions rather than raw conversation.
7. **Safety**: include explicit instructions to avoid identity-based judgments; include `do_not_explain_personal_attributes` rule.

Prompt Template (fashion LLM example):

```
SYSTEM: You are a professional fashion critic. You MUST return exactly valid JSON that conforms to schema: {outfit_r
USER: Detected items: ["jacket","shirt"]. Photo url: [URL]. Consider fit, color, formality, and current fashion trer
OUTPUT: <only JSON>
```

Context windows:

- If you need more context because multiple analyses are being combined, pass the *aggregated numeric results* not the raw outputs.

---

13) DSPy workflow for prompt optimization (developer flow)

Goal: automatically iterate prompt variants to maximize objective metrics (JSON compliance, user-perceived helpfulness, alignment with YOLO detections, minimal hallucination).

Steps:

1. Create a dataset of input cases: small representative set of images + expected outputs (gold labels where possible).
2. Define evaluation metrics: schema_validity (binary), semantic_accuracy (e.g. items overlap with YOLO detection), ROUGE/embedding-distance to gold text, and toxicity.
3. Use DSpy to generate prompt variants and run them against the dataset.
4. Collect metrics and rank prompts.
5. Manually inspect top candidates, then run a second-stage refinement with more examples.
6. Push the chosen prompt/version into production and label the LangSmith runs with the prompt version.

Automation: schedule periodic re-runs of DSpy on new data (monthly) and automatically deploy improved prompts behind a feature flag for A/B testing.

---

14) LangSmith tracing & evaluation plan

Instrumentation:

- Every agent run should call LangSmith start/finish with `metadata` containing `user_id`, `media_id`, `agent_name`, `prompt_version`, and vector DB ids.
- Save LLM `input`, `output` and `model` details in LangSmith. For heavy CV models, log the model version & worker id.

Eval suites:

- **Schema compliance**: test that the LLMs consistently produce valid JSON.
- **Detection alignment**: compare LLM-found items to YOLO detections.
- **Bias/toxicity checks**: automated toxicity scans on output.
- **Regression tests**: make sure new prompt changes do not reduce performance on gold set.

Alerting:

- Configure alerts if schema compliance drops below threshold or if avg latency for agent > SLO.

---

15) Testing, QA and CI/CD

Unit tests:

- Each deterministic heuristic function (posture sub-scorer, landmark converter) must have unit tests.

Integration tests:

- Mock external API calls (Face++, YOLO, LLM) and run orchestrator flows.

E2E tests:

- Run a small suite of images through the full pipeline in staging and validate final JSON schema and correctness.

CI/CD:

- On push to `main` run unit tests and integration tests with mocks.
- Deploy to staging manually. After sanity checks, promote to production.

---

16) Deployment & scaling guidelines

- Dockerize agents and worker processes.
- Use k8s for production with GPU node pools for CV workers.
- Scale LangGraph orchestrator horizontally without shared state.
- Use autoscaling for worker pods depending on queue length.
- Batch small inference jobs to reduce per-call overhead for external LLM APIs.

---

17) Security, privacy & compliance checklist

Do immediately:

- Move all API keys out of code and into secret store.
- Use HTTPS for all traffic and presigned URLs for uploads.
- Encrypt media at rest.
- Log minimal PII and redact sensitive fields in logs.

Consent & opt-in:

- Add per-feature consent toggles. Do not run attractiveness/personality analysis unless user explicitly opts in.
- Add a privacy center page that explains how data is used and retention schedules.

Retention:

- Default retention for raw media: 90 days (or per user plan). Allow user to delete media and all derived artifacts (embeddings, analyses) on request.

Disallowed:

- Do not store face embeddings without strong consent and securing them; treat biometric-derived vectors as sensitive.

---

18) Monitoring & observability

- LangSmith for request traces and run metadata.
- Prometheus + Grafana: latency, error rate, queue length, GPU utilization, embedding ingestion rate.
- Sentry for exceptions.
- Audit logs for data deletion requests.

19) Step-by-step implementation roadmap (detailed)

> Each step below is an actionable task that the engineer (or ChatGPT acting as an engineer) must complete in order. Each task should be committed as small PRs and tested.

Initial setup

Step 0 — Project bootstrap

- Create repo skeleton (see repo layout above).
- Add basic FastAPI app with a health-check endpoint.
- Add `.env.example` and secret loader.
- Add GitHub Actions CI pipeline skeleton (run lint + unit tests).

Step 1 — Provision infra

- Create Postgres DB (with pgvector if chosen) and object storage bucket.
- Provision Redis for queue and a development Qdrant instance (optional).
- Store secrets in Secret Manager.

Step 2 — Core utilities & model wrappers

- Implement wrapper modules for:
  - LLM client (OpenAI/OpenRouter) with timeout/retry + LangSmith instrumentation.
  - Face detection wrapper (Face++ API + fallback to Mediapipe) as `FaceTool`.
  - YOLO object & pose wrapper as `DetectTool` (deployed to GPU worker).
  - Embedding tool that can call OpenAI/CLIP and write to vector DB.
- Unit test these wrappers with mocked responses.

Step 3 — Storage & ingestion pipeline

- Implement `presigned_url` endpoints and `media.create` endpoint.
- Implement background worker code that consumes jobs and writes thumbnails + keyframes.
- Implement embedding storage for image thumbnails.

Step 4 — Implement EmbedderAgent

- Build EmbedderAgent and test with sample images and keyframes.
- Validate vector DB insertions and retrieval.

Step 5 — Implement FaceAgent

- Implement face detection, Mediapipe landmarks, and standard Face Agent output.
- Unit test with sample images.

Step 6 — Implement FashionAgent

- Implement YOLO-based item detection, CLIP zero-shot optional, and LLM critique wrapper.
- Add Guardrails schema for LLM output and ensure strict JSON output.

Step 7 — Implement PostureAgent

- Implement pose detection, smaller deterministic sub-functions for each score.
- Unit test each sub-function with mocked keypoints.
- Add a disclaimer in output.

Step 8 — Implement BioAgent

- Use LLM with Guardrails to create vibe summary and suggested improvements.
- Use Embeddings for retrieval to incorporate past context.

Step 9 — Implement AggregatorAgent & FormatterAgent

- Combine all agents' outputs into a final JSON and human summary.
- Validate final JSON with Guardrails.

Step 10 — LangGraph orchestration

- Translate the above flow into LangGraph nodes and edges.
- Ensure context passing and error-handling policies are in place.

Step 11 — Add Guardrails to each agent

- Add input + output validation for every LLM call.
- Implement deterministic fallbacks.

Step 12 — Prompt optimization (DSpy)

- Create gold datasets for the LLM tasks.
- Run DSpy experiments and pick best prompt variants.
- Tag prompt versions in LangSmith traces.

Step 13 — LangSmith instrumentation & evals

- Hook LangSmith into each agent.
- Define automated evals (schema compliance, detection alignment, toxicity) and run on staging.

Step 14 — Integration testing & security review

- Run integration tests (E2E) in staging with a subset of real images (consented test data).
- Security review for keys, PII, and data retention.

Step 15 — Deploy to staging & production

- Smoke-test with limited users.
- Monitor LangSmith metrics for regressions.

Step 16 — Ongoing

- Automate DSpy runs monthly and re-evaluate prompt performance.
- Schedule retraining or re-evaluation of heuristics based on new user data.

---

20) Appendix — starter examples

20.1 — Sample guardrails YAML for fashion LLM output

```
name: fashion_agent_output
fields:
  - name: outfit_rating
    type: integer
    required: true
    constraints:
      min: 0
      max: 100
  - name: items
    type: list
    items:
      type: string
  - name: improvements
    type: list
    items:
      type: string
```

20.2 — Example Pydantic model for final API output (Python)

```
class FinalAnalysis(BaseModel):
    media_id: UUID
    summary: str
    face: Optional[Dict]
    fashion: Optional[Dict]
    posture: Optional[Dict]
    bio: Optional[Dict]
    confidences: Dict[str, float]
    langsmith_run_id: Optional[str]
```

20.3 — Starter prompt for FashionAgent LLM (strict JSON)

```
SYSTEM: You are a professional, kind fashion critic. You MUST return *only* valid JSON that follows this schema: {"c
USER: Detected items: ["jacket","shirt","sneakers"]. Image URL: [URL]. Evaluate fit, color, formality. If unsure abc
OUTPUT: <valid JSON only>
```

20.4 — Sample DB migration snippet (SQL)

```
CREATE TABLE media (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id uuid,
  media_type varchar(10),
  storage_url text,
  thumbnail_url text,
  keyframes jsonb,
  created_at timestamptz DEFAULT now()
);
```

Final notes & recommended decisions (short)

- Start **small**: get embeddings + face detection + fashion detection working with Guardrails before building the full posture heuristics.
- Use **pgvector** initially for simplicity; move to Qdrant/Weaviate if multi-modal features or scale demand it.
- Keep the **user consent toggle** at the top of the request flow: no biometric analysis without explicit opt-in.
- Use **LangSmith** from day one to capture traces and enable fast debugging and comparison.

If you want, I will now:

1. Convert this file into a repository `README.md` + a set of `issue` checklists per roadmap step.
2. Generate the **first PR** contents for Step 0 and Step 1 (FastAPI skeleton + presigned upload flow).
3. Or immediately generate **detailed LLM prompts** for each agent (ready for DSpy optimization).

Which of the above do you want next?