FLUX WIKI EMBEDDINGS

TECHNICAL SPECIFICATION FOR THE FLUX VISUAL EMBEDDINGS LAYER — MODEL SELECTION, VECTOR STORAGE, GENERATION PIPELINE, AND QUERY INTERFACE.

flux.dantesisofo.com/wiki/embeddings/

FLUX_WIKI_v2.0

JUNE 2026

FLUX DOCUMENTATION SYSTEM Layer 5 — INTELLIGENCE | embeddings
flux.dantesisofo.com/wiki/embeddings/

EMBEDDINGS

1. WHAT EMBEDDINGS ARE

A visual embedding is a mathematical representation of a photograph's content as a vector of floating-point numbers.

```
photograph (JPEG, ~5MB)  →  embedding model  →  vector (512 floats)
```

The vector encodes what the photograph contains — composition, tonal distribution, subject matter, texture, spatial relationships, light quality — as a point in high-dimensional space.

Two photographs that look similar will have similar vectors. Their points in the embedding space will be close together.

Two photographs that are visually unrelated will have distant vectors.

Distance = dissimilarity. Proximity = similarity.

The embedding captures visual content. It does not capture filenames, dates, or folder locations.

2. WHY EMBEDDINGS TRANSFORM THE ARCHIVE

A conventional archive organizes photographs by filename and date. To find a photograph, you remember where you put it.

At 400,000 photographs, this fails. No human can hold 400,000 photographs in memory. Browsing by date is impossible at that scale. Search by filename requires you already know the filename.

Embeddings make the archive queryable by visual content.

```
BEFORE EMBEDDINGS:
  Query:  "find the photograph I made in fog with an umbrella in 2023"
  Answer: scroll through 2023 folders until you find it

AFTER EMBEDDINGS:
  Query:  "umbrellas in fog"
  Answer: 400,000 vectors searched in <100ms; top results returned
```

The transformation:

```
files        →  relationships
storage      →  memory
retrieval    →  understanding
date search  →  visual search
folder drill →  natural language query
```

3. WHAT BECOMES POSSIBLE

These are the canonical example queries. They are not hypothetical. They will work once the embeddings layer is built.

```
show all umbrellas in fog
show visual echoes of Rome
show all silhouettes from winter 2022
show images visually similar to this one
show recurring gestures across six years
show evolution of layering over time
find all photographs where a person is waiting
find all photographs with wet pavement reflections
find all photographs made within one block of 40th and Walnut
```

Each of these queries is a vector search operation:

1. Text query → encode with CLIP text encoder → query vector
2. Query vector → nearest-neighbor search across 400,000 photo vectors → top-N results
3. Results → return photo IDs → retrieve thumbnails and metadata

No manual tagging required. No folder structure required. The visual content of the photograph is the index.

4. TECHNICAL ARCHITECTURE

Model

```
Model:       CLIP (Contrastive Language-Image Pretraining)
Publisher:   OpenAI (open weights via Hugging Face)
Variant:     ViT-L/14 or ViT-B/32 (TBD based on compute budget)
Input:       JPEG, resized to 224×224 for inference
Output:      512-dimensional float vector (ViT-B/32) or 768-dimensional (ViT-L/14)
License:     MIT (open weights, can run locally)
```

CLIP is selected because it supports both image-to-image and text-to-image search from a single model. The same model that encodes photographs also encodes text queries. This enables natural-language queries against the archive without separate models for text and image encoding.

Alternative: SigLIP (Google), DINOv2 (Meta) for image-only similarity. CLIP is preferred for text query support.

Vector Dimensions

```
ViT-B/32:  512 dimensions
ViT-L/14:  768 dimensions
```

Preferred: ViT-L/14 (768d) for higher-quality representations at the cost of 50% larger storage and slower inference.

Decision deferred until hardware (Mac mini M4 Pro) is available for benchmarking.

Storage

```
Primary:    SQLite-vec (sqlite3 extension for vector search)
Alternative: pgvector (PostgreSQL extension) if corpus exceeds 5M rows
Index type: HNSW (Hierarchical Navigable Small World) — approximate nearest-neighbor
```

SQLite-vec is preferred for Phase 1: - No additional server process - Compatible with the existing SQLite metadata database - HNSW index supports sub-100ms queries on 400,000 vectors - Single file, simple backup

pgvector is the upgrade path if query performance degrades at scale.

Storage Estimate

```
400,000 vectors × 512 floats × 4 bytes = 819 MB (ViT-B/32)
400,000 vectors × 768 floats × 4 bytes = 1.2 GB (ViT-L/14)
HNSW index overhead: ~2x raw vector size
Total (ViT-L/14 + index): ~2.4 GB
```

Stored at: `/FLUX_EMBEDDINGS/` on NAS.

---

5. GENERATION PIPELINE

Per-image workflow for embedding generation:

```
1. Load JPEG from NAS (/FLUX_ARCHIVE/ORIGINALS/...)
2. Preprocess:
   a. Decode JPEG to RGB array
   b. Resize to 224×224 (bicubic)
   c. Normalize to CLIP input range (mean=[0.48,0.46,0.41], std=[0.27,0.26,0.28])
3. Model inference:
   a. Forward pass through CLIP image encoder
   b. Extract [CLS] token embedding
   c. L2-normalize the vector (unit sphere projection)
4. Store vector:
   a. Write to embeddings table in SQLite-vec
   b. Link to photo_id in photos table
5. Update metadata:
   a. Set embedding_id in photos table
   b. Log completion timestamp
```

Batch processing for full corpus:

```
python3 generate_embeddings.py \
  --input /FLUX_ARCHIVE/ORIGINALS/ \
  --db    /FLUX_METADATA/flux.db \
  --vec   /FLUX_EMBEDDINGS/flux_vectors.db \
  --model clip-vit-l-14 \
  --batch 64
```

Estimated throughput on Mac mini M4 Pro: - ~60–120 images/minute (CPU inference, no GPU) - 400,000 images / 90 img/min = ~74 hours for full corpus - One-time operation. Subsequent ingest is per-image real-time.

---

6. QUERY INTERFACE

Text-to-image query

```
import clip
import torch

model, preprocess = clip.load("ViT-L/14")

def query_archive(text_query, top_k=12):
    # Encode text query
    text   = clip.tokenize([text_query])
    with torch.no_grad():
        q_vec = model.encode_text(text).numpy()
    q_vec = q_vec / np.linalg.norm(q_vec)  # L2 normalize

    # Search vector database
    results = vec_db.search(q_vec, k=top_k)

    # Return photo IDs and similarity scores
    return [(r.photo_id, r.distance) for r in results]
```

Image-to-image query

```
def find_similar(photo_id, top_k=12):
    # Load existing embedding
    q_vec = vec_db.get_vector(photo_id)

    # Nearest-neighbor search
    results = vec_db.search(q_vec, k=top_k + 1)  # +1 to exclude self

    # Filter out the query image itself
    return [(r.photo_id, r.distance) for r in results if r.photo_id != photo_id]
```

Query performance target:

```
400,000 vectors, HNSW index, ViT-L/14 (768d)
Target: <100ms per query on Mac mini M4 Pro
```

---

7. CORPUS SCALE

```
Full corpus:      ~400,000 photographs
Keeper archive:   ~15,000 photographs
Active issues:    423+ (FLUX_001–FLUX_423+)
New ingest rate:  variable (daily practice)
```

Scale implications: - Full corpus embedding generation: one-time pass, ~74 hours - Incremental embedding: per-image, <1 second, runs at ingest time - Vector database size: ~2.4 GB (manageable on NAS, fits in RAM for search) - Index rebuild frequency: never (HNSW is insert-efficient; incremental insertion works)

---

8. IMPLEMENTATION PHASES

Embeddings require these prerequisites:

```
Phase 1 (COMPLETE):    NAS hardware ordered, arriving soon
Phase 2 (PENDING):     Full corpus migrated to NAS (/FLUX_ARCHIVE/ORIGINALS/)
Phase 3 (PENDING):     Keeper archive matched and flagged in metadata database
Phase 4 (PENDING):     Metadata database initialized, all SHA-256 hashes generated
Phase 5 (THIS PHASE):  Embedding generation for full corpus
Phase 6 (AFTER):       Keeper model training (uses embeddings + keeper labels)
```

The embeddings layer cannot be built until Phases 2–4 are complete. The metadata database must exist. The corpus must be on the NAS. SHA-256 hashes must be computed (for deduplication before embedding).

9. RELATIONSHIP TO KEEPER MODEL

The embeddings layer and the keeper model are separate but dependent:

```
Embeddings:    visual representation of every photograph
Keeper labels: binary (kept/not kept) for ~15,000 photographs
Keeper model:  trained on (embedding, keeper_label) pairs
```

The keeper model takes embeddings as input features and keeper labels as training targets. Without embeddings, the keeper model has no features. Without keeper labels, the keeper model has no training signal.

The sequence is: 1. Build embeddings for all 400,000 photographs 2. Match keeper archive (15,000 images) to full corpus → assign keeper labels 3. Train keeper model on (embedding, label) pairs 4. Score all corpus photographs with keeper model → keeper_score (0.0–1.0) 5. Use keeper scores for auto-ranking during ingest and issue draft suggestions

See: KEEPER MODEL for the full specification.

SEE ALSO

Document	Layer	Relationship
INTELLIGENCE	Layer 5 — Intelligence	Layer overview; embeddings is a subdocument of this layer
KEEPER MODEL	Layer 5 — Intelligence	Uses embeddings as input features for the taste model
METADATA ENRICHMENT	Layer 5 — Intelligence	SQLite schema that links photo_id to embedding_id
TRAINING DATA	Layer 5 — Intelligence	The labeled dataset that embeddings + keeper labels create
BOOTSTRAP	Layer 4 — Infrastructure	Phased implementation plan; embeddings is Phase 5

FLUX_WIKI_v2.0 — flux.dantesisofo.com/wiki/embeddings/