# Integrating ML Features into "Moments": Alt-Text & Keyword Search

Evan Parra

September 2025

**Course:** COT 6930 — AI & ML in Production (Fall 2025)
**Repo:** https://github.com/DevDizzle/moments
**Final commit:** https://github.com/DevDizzle/moments/tree/a1-submit

**Abstract**

This extends the Flask photo app *Moments* with two ML-backed features using Google Cloud Vision: (1) automatic alt-text on upload when description is blank; (2) keyword search via auto-generated tags. No custom model training. The goal was to embed a pre-trained API cleanly into an existing codebase across data model, routes, and UI, while considering harms, cost, and production constraints.

## 1 Introduction

I added two things that improve the app without adding friction:

- **Alt-text on upload:** if a user doesn't write a description, Vision labels are used to compose a short alt-text sentence.
- **Keyword search:** Vision labels become tags; search queries match tags, descriptions, or alt-text.

Design goal: keep the user flow identical; let the ML run in the background.

## 2 What I built (short)

- **Models:** add `Photo.alt_text`; add `Tag` + `photo_tags` (many-to-many).
- **Utils:** a helper to call Vision and return `(alt_text, tags)`.
- **Routes:** upload integrates Vision; search matches {tags OR description OR alt_text}; simple tag delete (owner-only).
- **Templates:** real `<img alt="...">` in gallery/detail.

## 3 How it works (technical)

### Dependencies & config (PDM only)

- This project uses **PDM** and a pinned lockfile; do not use `pip install`.
- Install once with: `pdm install`.
- A local `.env` sets:
  - `GOOGLE_APPLICATION_CREDENTIALS=path/to/vision-key.json`
  - `FLASK_APP=moments`, `FLASK_ENV=development`, `UPLOAD_FOLDER=static/uploads`
  - Optional: `VISION_LABEL_SCORE_THRESHOLD=0.70`
- The key file is present locally but *not committed*. `.env` is ignored by git.

### Data model (Python/SQLAlchemy)

```python
# moments/models.py
photo_tags = db.Table(
    "photo_tags",
    db.Column("photo_id", db.Integer, db.ForeignKey("photo.id"),
        primary_key=True),
    db.Column("tag_id", db.Integer, db.ForeignKey("tag.id"),
        primary_key=True),
)

class Photo(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    filename = db.Column(db.String(256), nullable=False)
    description = db.Column(db.String(300))
    alt_text = db.Column(db.String(300))  # NEW
    tags = db.relationship("Tag", secondary=photo_tags, backref="photos
        ")

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, index=True, nullable=
        False)
```

Listing 1: Data model changes: Photo.alt_text, Tag, association table

### Vision helper (labels ⇒ tags, alt-text fallback)

```python
# moments/utils.py
from google.cloud import vision
import os

def analyze_image(image_bytes: bytes, label_score_threshold: float |
    None = None) -> tuple[str, list[str]]:
    if label_score_threshold is None:
        label_score_threshold = float(os.getenv("
            VISION_LABEL_SCORE_THRESHOLD", "0.70"))

    client = vision.ImageAnnotatorClient()
    resp = client.label_detection(image=vision.Image(content=
        image_bytes), max_results=10)
    labels = resp.label_annotations or []

    # Collect tags above threshold (lowercased, de-duped)
    seen, tags = set(), []
    for l in labels:
        if not l.description or (l.score or 0) < label_score_threshold:
            continue
        t = l.description.strip().lower()
        if t and t not in seen:
            seen.add(t); tags.append(t)

    # Compose concise alt-text from top 1-3 labels (only if needed)
    top = [l.description.lower() for l in labels[:3] if l.description]
    if not top: return ("", tags)
    if len(top) == 1: return (f"a photo of {top[0]}", tags)
```

```
    if len(top) == 2: return (f"a photo of {top[0]} and {top[1]}", tags
        )
    return (f"a photo of {top[0]}, {top[1]}, and {top[2]}", tags)
```

Listing 2: Google Cloud Vision utility (analyze_image)

### Upload flow (POST /upload)

```
# moments/routes.py (excerpt)
from .utils import analyze_image

# ... after saving the file ...
file.seek(0)
alt, tags = analyze_image(file.read())

p = Photo(filename=save_name,
          description=desc or None,
          alt_text=(None if desc else (alt or None)))
db.session.add(p); db.session.flush()

for t in tags:
    tag = Tag.query.filter_by(name=t).one_or_none()
    if not tag:
        tag = Tag(name=t)
        db.session.add(tag); db.session.flush()
    p.tags.append(tag)

db.session.commit()
```

Listing 3: Upload integrates Vision; tags are attached; alt-text filled if description absent

### Keyword search

```
# moments/routes.py (excerpt)
from sqlalchemy import or_, func

@app.route("/search")
def search():
    q = (request.args.get("q") or "").strip().lower()
    results = []
    if q:
        results = (
            Photo.query
            .outerjoin(photo_tags).outerjoin(Tag)
            .filter(or_(func.lower(Tag.name).contains(q),
                        func.lower(Photo.description).contains(q),
                        func.lower(Photo.alt_text).contains(q)))
            .distinct()
            .order_by(Photo.id.desc())
            .all()
        )
    return render_template("search.html", q=q, photos=results)
```

Listing 4: Search over tags OR description OR alt_text

**Templates: real HTML alt attribute**

```
<!-- moments/templates/_photo_card.html (excerpt) -->
<img src="{{ url_for('static', filename='uploads/' ~ photo.filename) }}
    "
     alt="{{ photo.alt_text or photo.description or '' }}"
     loading="lazy">
```

Listing 5: Template ensures real < img alt=...> for accessibility

## 4 UI/UX design (Automate / Prompt / Organize / Annotate)

**Automate** the Vision call on every upload; users keep their flow. **Annotate** photos with tags and, if needed, alt-text. **Organize** content via tag-based search. I avoided the prompts (*"we detected a dog..."*) to reduce friction.
*Forcefulness:* low (background) *Frequency:* high (every upload) *Value:* high (accessibility + discovery) *Cost:* API calls and some latency.

## 5 Harms & risks (and mitigations)

- **Inaccurate/biased/offensive labels:** owner can delete tags; confidence threshold; optional stop list.
- **Weak alt-text:** owner can edit description; that overrides auto alt-text.
- **Privacy:** owners can remove tags that reveal sensitive context.

## 6 Production challenges (scaling & cost)

**Cost/latency:** don't block uploads on external calls at scale. Move Vision to a background queue (Celery + Redis), add rate limiting/retries, cache by image hash.
**Search performance:** DB indexes on `Tag.name`, `Photo.description`, `Photo.alt_text`.
**Monitoring/budgets:** log Vision errors; backoff; set GCP budget alerts.

## 7 How to run locally (PDM)

```
# install deps
pdm install

# ensure .env contains GOOGLE_APPLICATION_CREDENTIALS and other vars
# example keys in .env (not committed):
# GOOGLE_APPLICATION_CREDENTIALS=path/to/vision-key.json
# FLASK_APP=moments
# FLASK_ENV=development
# UPLOAD_FOLDER=static/uploads
# VISION_LABEL_SCORE_THRESHOLD=0.70

# initialize & run
pdm run flask init-app
pdm run flask lorem       # optional demo data
pdm run flask run         # http://127.0.0.1:5000
```

Listing 6: Local setup with PDM (requirements are hashed; use PDM only)

# 8 Evidence (screenshots)

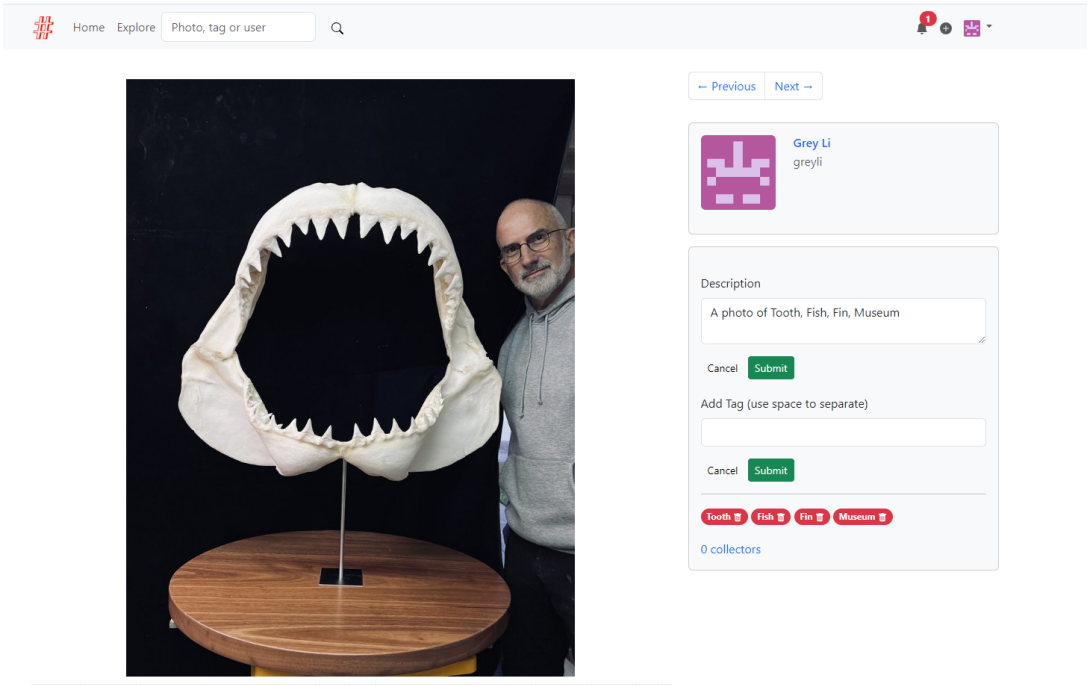


Figure 1: Upload example used to demonstrate Vision labels, tags, and auto alt-text (man standing next to shark jaws).