Integración de Azure Face API con `isshesarahconnor.py`

Documentación técnica y how-to

Santi

11 de septiembre de 2025

Índice

Resumen ejecutivo

Azure Face API es un servicio gestionado para detección y reconocimiento facial. Esta documentación explica cómo usar Face API (detect, verify, identify, find_similar, liveness) y cómo integrarlo en el repositorio `isshesarahconnor` que unifica fetch, preprocess, train, export y servicio de inferencia. Incluye ejemplos de código listos para pegar y recomendaciones de seguridad y despliegue.

1. Qué hace Azure Face API

- **Detección de rostros:** bounding boxes, landmarks, atributos faciales.
- **Verificación:** comparaciones one-to-one entre dos caras.
- **Identificación:** búsqueda one-to-many contra PersonGroup / LargePersonGroup o FaceList.
- **Find_similar:** buscar caras similares (usa representaciones internas).
- **Liveness (anti-spoofing):** detección de ataques de suplantación.

Nota sobre embeddings

Face API gestiona internamente representaciones para verify/identify. Si tu objetivo es almacenamiento y uso directo de vectores (embeddings) para búsqueda no biométrica, considera usar las APIs de embeddings multimodales de Azure o extraer vectores localmente (ResNet, Facenet) y almacenarlos en una vector DB (Milvus, FAISS, Azure Cognitive Search con vector store).

2. QuickStart: recurso Azure y autenticación

1. Crear recurso **Face** o un recurso multi-service en el portal de Azure.
2. Obtener `endpoint` y `API key` desde el portal.
3. Instalar SDK (ejemplo): `pip install azure-ai-vision-face azure-core azure-identity`
4. En pruebas: usar `AzureKeyCredential`; en producción: `DefaultAzureCredential` o Managed Identity.

3. Comandos REST de la Face API de Azure

3.1. Resumen rápido

La Face API se expone típicamente en la forma:

```
https://{tu-recurso}.cognitiveservices.azure.com/face/v1.0
```

Las operaciones principales: `detect`, `verify`, `identify`, `findsimilars`, y los recursos de gestión (`personGroup`, `largePersonGroup`, `faceList`, `largeFaceList`). Autenticación: normalmente encabezado `Ocp-Apim-Subscription-Key: KEY` o Azure AD.

3.2. Operaciones principales

3.2.1. Detect

Detecta rostros en una imagen y devuelve `faceId`, bounding boxes, landmarks y atributos (opcionales). Endpoint:

```
POST /face/v1.0/detect
```

3.2.2. Verify

Comparación one-to-one. Permite comparar dos `faceId` o un `faceId` contra una `personId`. Endpoint:

`POST /face/v1.0/verify`

3.2.3. Identify

Identificación one-to-many contra un `personGroup` o `largePersonGroup`. Requiere que el grupo esté entrenado. Endpoint:

`POST /face/v1.0/identify`

3.2.4. Find Similar

Busca caras similares contra una `faceList` / `largeFaceList` o entre `faceIds`. Endpoint:

`POST /face/v1.0/findsimilars`

3.3. Colecciones y gestión (persistencia)

- **LargePersonGroup**: crear/administrar grupos a gran escala.
  `PUT /face/v1.0/largepersongroups/{largePersonGroupId}`
- **Person** dentro de un grupo: crear persona y añadir caras (persistedFaceId).
  `POST /face/v1.0/largepersongroups/{id}/persons`
- **FaceList** / **LargeFaceList**: crear listas y añadir `persistedFaceId`.
  `PUT /face/v1.0/facelists/{faceListId}` `POST /face/v1.0/facelists/{id}/persistedfaces`
- Operaciones típicas: `train` (inicio/estado), `get person`, `delete persistedFace`, etc.

3.4. Liveness / anti-spoofing

La API dispone de endpoints para crear sesiones de *liveness* y recuperar resultados (active / passive). Consultar la documentación de liveness si necesitas protección contra spoofing en autenticación.

3.5. Notas importantes

- **Autenticación:** usa header `Ocp-Apim-Subscription-Key: KEY` o Azure AD según tu despliegue.
- `faceId` **vs** `persistedFaceId`**:** `faceId` es temporal (devuelto por `detect`); `persistedFaceId` se obtiene al subir una cara a un `FaceList` o a una persona y persiste hasta su eliminación.
- **recognitionModel:** las colecciones (personGroup/faceList) y las llamadas a `detect` deben usar modelos compatibles (p. ej. `recognition_04`) para obtener resultados coherentes.
- **Límites y costes:** revisar cuota/precio por región; para producción usar paginación y batching cuando sea posible.

3.6. Ejemplos `curl`

Detect (enviar imagen como binario)

```
curl -X POST "https://<endpoint>/face/v1.0/detect?returnFaceId=true&
   returnFaceLandmarks=false" \
  -H "Ocp-Apim-Subscription-Key: <KEY>" \
  -H "Content-Type: application/octet-stream" \
  --data-binary "@imagen.jpg"
```

Verify (comparar dos faceId)

```
curl -X POST "https://<endpoint>/face/v1.0/verify" \
  -H "Ocp-Apim-Subscription-Key: <KEY>" \
  -H "Content-Type: application/json" \
  -d '{"faceId1":"<uuid1>", "faceId2":"<uuid2>"}'
```

Identify (one-to-many contra largePersonGroup)

```
curl -X POST "https://<endpoint>/face/v1.0/identify" \
  -H "Ocp-Apim-Subscription-Key: <KEY>" \
  -H "Content-Type: application/json" \
  -d '{
    "faceIds":["<faceId>"],
    "largePersonGroupId":"mi_grupo",
    "maxNumOfCandidatesReturned":5,
    "confidenceThreshold":0.6
  }'
```

Find Similar (buscar en largeFaceList)

```
curl -X POST "https://<endpoint>/face/v1.0/findsimilars" \
  -H "Ocp-Apim-Subscription-Key: <KEY>" \
  -H "Content-Type: application/json" \
  -d '{
    "faceId":"<faceId>",
    "largeFaceListId":"mi_large_list",
    "maxNumOfCandidatesReturned":3,
    "mode":"matchPerson"
  }'
```

Crear LargePersonGroup (ejemplo)

```
curl -X PUT "https://<endpoint>/face/v1.0/largepersongroups/mi_grupo" \
  -H "Ocp-Apim-Subscription-Key: <KEY>" \
  -H "Content-Type: application/json" \
  -d '{"name":"Mi grupo", "recognitionModel":"recognition_04"}'
```

3.7. Recomendaciones prácticas

- Llama `detect` al inicio para obtener `faceId` en flujos inmediatos. Para búsquedas persistentes añade caras a `FaceList` o a `Person` (persistedFaceId).
- Mantén consistencia de `recognitionModel` entre creación de colecciones y detecciones posteriores.
- Maneja los errores y los códigos de estado (403/429/500) y respeta las cuotas; implementa backoff/retries.
- Para grandes volúmenes, usa `LargeFaceList`/`LargePersonGroup` y diseñar procesos de batching.

4. Snippets how-to (Python)

4.1. Detectar caras (obtener `faceId` y rectángulos)

```
from azure.core.credentials import AzureKeyCredential
from azure.ai.vision.face import FaceClient
from azure.ai.vision.face.models import FaceDetectionModel,
    FaceRecognitionModel

endpoint = "https://<tu-subdominio>.cognitiveservices.azure.com/"
key = "<TU_API_KEY>"
face_client = FaceClient(endpoint=endpoint, credential=AzureKeyCredential(
    key))

with open("img.jpg", "rb") as f:
    img_bytes = f.read()

results = face_client.detect(
    img_bytes,
    detection_model=FaceDetectionModel.DETECTION03,
    recognition_model=FaceRecognitionModel.RECOGNITION04,
    return_face_id=True,
    return_face_landmarks=True,
    face_id_time_to_live=120
)

for face in results:
    print(face.face_id, face.face_rectangle)
```

4.2. Verificar two faces (one-to-one)

```
res = face_client.verify_face_to_face(face_id1, face_id2)
# res.is_identical -> bool, res.confidence -> float
```

4.3. Registrar personas y entrenar (LargePersonGroup)

```
from azure.ai.vision.face import FaceAdministrationClient
from azure.core.credentials import AzureKeyCredential

admin = FaceAdministrationClient(endpoint=endpoint, credential=
    AzureKeyCredential(key))
admin.create_large_person_group(large_person_group_id="my_group", name="Mi
    grupo")

admin.begin_train(large_person_group_id="my_group").result()
```

4.4. Buscar similares / find_similar y verify

```
res_find = face_client.find_similar_from_face_id(face_id=some_face_id,
    large_face_list_id="my_list")
res_verify = face_client.verify_face_to_face(face_id1, face_id2)
```

5. Integración con pipeline_all_in_one.py

El script ya soporta múltiples backends locales para detección (face_recognition, MTCNN, cv2 haar cascades). A continuación se muestran patrones y snippets plug-and-play para integrarlo con Face API.

Ojo con dejar APIs en código. Usar v.env siempre.

5.1. Función auxiliar de detección remota

(para usar en lugar de detect_faces_pil)

```
from azure.core.credentials import AzureKeyCredential
from azure.ai.vision.face import FaceClient
from azure.ai.vision.face.models import FaceDetectionModel,
    FaceRecognitionModel
from io import BytesIO
from PIL import Image

def azure_detect_faces_bytes(img_bytes: bytes, endpoint: str, key: str,
    face_ttl:int=120):
    face_client = FaceClient(endpoint=endpoint, credential=
        AzureKeyCredential(key))
    result = face_client.detect(
        img_bytes,
        detection_model=FaceDetectionModel.DETECTION03,
        recognition_model=FaceRecognitionModel.RECOGNITION04,
        return_face_id=True,
        return_face_landmarks=False,
        face_id_time_to_live=face_ttl
    )
    boxes = []
    face_ids = []
    for face in result:
        r = face.face_rectangle
        l = r.left; t = r.top; w = r.width; h = r.height
        boxes.append((l, t, l + w, t + h))
        face_ids.append(face.face_id)
    img = Image.open(BytesIO(img_bytes))
    return boxes, face_ids, img.size
```

5.2. Ejemplo de integración en crop_faces_from_image

```
# inside crop_faces_from_image(...)
if use_azure:
    with open(src_path, "rb") as f:
        img_bytes = f.read()
    boxes, face_ids, _ = azure_detect_faces_bytes(img_bytes, endpoint, key
        )
    # Luego recortar usando boxes como en el flujo local
```

5.3. Registrar personas (ejemplo)

```
from azure.ai.vision.face import FaceAdministrationClient
admin = FaceAdministrationClient(endpoint=endpoint, credential=
    AzureKeyCredential(key))
admin.create_large_person_group(large_person_group_id="my_group", name="Mi
    grupo")
# crear person y usar admin.add_face_to_large_person_group(...)
admin.begin_train(large_person_group_id="my_group").result()
```

6. Descripción del pipeline y responsabilidades de cada módulo

- **Downloader:** serpapi_images_for_query, icrawler_download, download_url_to con blacklist handling.

- **Face detection & crops:** `detect_faces_pil`, `crop_faces_from_image`, `crop_faces_in_folder`.
- **Prepare dataset:** `internal_prepare_dataset` — MD5 dedupe, optional pHash dedupe, split train/val/test.
- **Training:** `run_training` — builds model (ResNet), freeze/fine-tune, AMP support, checkpointing, early stopping.
- **Export:** TorchScript / ONNX export helpers.
- **Serve:** `create_flask_app` — endpoints `/predict` y `/predict_json`.

7. Comandos de uso (ejemplos)

- Descargar imágenes: `python pipeline_all_in_one.py -fetch -query .ªda Lovelace-out raw/adalovelac`
- Preparar dataset desde crops (ejemplo en REPL):
  `python -c "from pipeline_all_in_one import internal_prepare_dataset; internal_prepare_dataset`
- Entrenar (ejemplo): crear dict `cfg` y llamar `run_training(cfg)` desde un script Python.
- Levantar servidor Flask: `python -c "from pipeline_all_in_one import create_flask_app; app=create_flask_app('models/best_model.pth', labels=['a','b']); app.run(host='0.0.0.0', port=8080)"`

8. Recomendaciones: seguridad, privacidad y costes

- **Credenciales:** no incluir keys en repositorios; usar variables de entorno o Managed Identity.
- **Privacidad:** cumplir GDPR y normativas locales; solicitar consentimiento para procesamiento biométrico.
- **Costes:** Face API se factura por transacción; calcular coste si procesas grandes volúmenes.
- **Despliegue:** para producción, containerizar, usar autenticación y HTTPS, monitorizar latencia / errores.

9. Referencias y lectura adicional

- Azure Face documentation: https://learn.microsoft.com/azure/cognitive-services/face/
- Azure ML documentation: https://learn.microsoft.com/azure/machine-learning/
- Custom Vision: https://learn.microsoft.com/azure/cognitive-services/custom-vision-service/

Apéndice: snippets completos (resumen)
Los snippets principales (detección Azure, administración LargePersonGroup, find_similar/verify, integración) se incluyen en las secciones previas y también se pueden exportar como archivos `.py` si lo deseas.