

영수증 OCR 파이프라인 코드 상세 보고서

본 문서는 Python 3.11.9 환경에서 동작하는 회사용 영수증 OCR 파이프라인 코드에 대한 상세 설명입니다. 이 파이프라인은 사내 DB에서 영수증 정보 레코드를 불러와 **전처리 -> OCR 인식 -> 후처리 -> DB 저장**의 과정을 자동화합니다. 코드는 여러 모듈로 구성되어 있으며, 각 모듈의 역할과 상호 연동을 **wrapper.py** 중심으로 설명합니다. 새로운 개발자가 이 코드를 이해하고 유지보수하거나 기능 수정할 수 있도록, 각 파일별로 아래 항목들을 정리하였습니다:

- 1. 코드의 목적 및 역할
- 2. 시스템 내 위치와 연동 흐름
- 3. 사용 대상 (사용 주체와 사용 시나리오)
- 4. 실행 환경 및 의존성 (Python 버전, 필요한 라이브러리, OS 제약)
- 5. 외부 연동 요소 (사용 API, DB 테이블, 외부 모델 등)
- 6. 입력값 및 파라미터 설명
- 7. 출력값 및 결과를 설명
- 8. 코드 흐름 설명 (주요 처리 단계)
- 9. 주요 함수 및 모듈 설명
- 10. 로깅 및 예외 처리 전략
- 11. 테스트 방법 및 사용 예시
- 12. 유지보수 및 확장 고려사항

먼저 전체 시스템의 동작 흐름을 개괄한 후, 각 모듈(`pre_process.py`, `doc_process.py`, `post_process.py`, `wrapper.py`, `db_master.py`)의 상세 내용을 순서대로 설명합니다.

전체 시스템 개요

이 OCR 파이프라인은 **SAP HANA DB**로부터 특정 날짜의 영수증 처리 대상 데이터를 조회하여 시작됩니다. 그런 다음 각 데이터 레코드마다 다음 단계를 수행합니다:

1. **전처리 (`pre_process.py`)** - 영수증 파일(이미지 또는 문서)을 **다운로드 및 변환**하고, **YOLO 모델**을 이용해 영수증 영역을 **검출 및 크롭**합니다. 이 단계의 결과로 한 개의 입력에 대해 한 장 또는 여러 장의 영수증 이미지가 추출됩니다 (주로 영수증 사진이 여러 장 묶인 경우 분할).
2. **OCR 인식 (`doc_process.py`)** - 전처리 결과로 얻어진 각 영수증 이미지를 **Azure Form Recognizer**에 보내 **텍스트 및 필드 인식**을 수행합니다. Azure의 **영수증 인식 모델**을 사용하여 상호, 날짜, 총액 등의 **구조화된 데이터**를 추출합니다 ^①.
3. **후처리 (`post_process.py`)** - Azure OCR 결과(JSON 데이터)를 **요약 정보(summary)**와 **항목 리스트(items)**로 재구성합니다. 필요한 키-값 쌍을 정리하고 부족한 값은 기본값을 채워 하나의 결과 JSON 파일로 저장합니다.
4. **DB 저장 (`db_master.py`)** - 후처리된 JSON 결과를 읽어 **SAP HANA DB**의 결과 테이블에 **삽입**합니다. 영수증 요약 정보는 `RPA_CCR_LINE_SUMM` 테이블에, 품목별 상세 내역은 `RPA_CCR_LINE_ITEMS` 테이블에 각각 INSERT합니다 ^{② ③}.

이 모든 흐름을 **wrapper.py**에서 오케스트레이션하며, 멀티스레딩을 통해 여러 레코드를 병렬 처리할 수 있습니다. **wrapper.py**는 DB에서 대상 레코드를 조회하고, 각 레코드에 대해 `run_pre_pre_process -> run_azure_ocr -> post_process_and_save -> insert_postprocessed_result` 순으로 함수를 호출합니다. 각 단계에서 오류가 발생하면 해당 레코드 처리는 건너뛰거나 오류 내용을 별도로 저장하고, 파이프라인은 다음 작업을 이어서 수행하도록 설계되어 있습니다.

아래에서는 각 모듈 파일별로 상세 내용을 설명합니다.

pre_process.py (전처리 모듈)

1. 코드의 목적 및 역할:

`pre_process.py` 모듈은 영수증 이미지 전처리를 담당합니다. 주요 역할은 원본 입력(이미지 URL 또는 문서 경로)을 받아 파일 다운로드, DRM 해제, 이미지 추출/변환, 그리고 YOLO 모델을 통한 영수증 영역 검출 및 크롭입니다. 즉, 여러 형식의 입력 데이터(PDF, 이미지 파일 등)를 정규화하여 OCR이 가능한 이미지로 만들고, 이미지 내에서 영수증 부분만 잘라내는 것이 이 모듈의 목적입니다. 이 과정을 통해 후속 OCR 엔진이 처리할 이미지(들)와 관련 메타정보를 생성합니다.

2. 전체 시스템 내 위치와 연동 흐름:

본 모듈은 파이프라인의 첫 단계에 해당하며, `wrapper.py`에서 Azure OCR 실행 전에 호출됩니다. `wrapper.py`의 `process_single_record` 함수 내에서, DB에서 불러온 한 건의 레코드에 대해 `run_pre_pre_process(in_params, record)` 함수를 실행하여 전처리를 수행합니다⁴⁵. 이 함수는 전처리 결과로 영수증 이미지 파일 경로들을 포함한 딕셔너리 리스트를 반환하며, 이후 단계인 `doc_process.py`에서 이 경로들을 사용해 Azure OCR 처리를 진행합니다. 만약 전처리 단계에서 영수증 검출 실패 또는 오류가 발생하면, 해당 레코드의 해당 영수증에 대해서는 OCR을 건너뛰고 오류로 처리합니다 (`RESULT_CODE`와 메시지를 포함한 결과를 반환)⁶⁷. 따라서 `pre_process.py`는 `wrapper` → `pre_process` → `doc_process` → `post_process` → DB 저장의 흐름 중 가장 앞부분에 위치하며, 다른 모듈들과는 주로 데이터 파일 및 딕셔너리 형태로 연동합니다. (예: 다운로드 받은 파일 경로, 크롭된 이미지 경로 등을 다음 단계에 전달).

3. 사용 대상:

이 전처리 코드는 최종 사용자가 직접 실행하기보다는, 백엔드 배치 프로세스나 RPA 작업의 일부로 동작합니다. 보통 운용 시에는 `wrapper.py`가 하루치 데이터를 일괄 처리하면서 내부적으로 호출합니다. 개발/테스트 단계에서는 개발자가 `main` 블록의 샘플 코드나 단위 테스트를 통해 이 함수를 실행해볼 수 있습니다. 예를 들어, `__main__`에 정의된 테스트용 `db_record`를 사용하여 `run_pre_pre_process`를 호출하면 개발 환경에서 동작을 검증할 수 있습니다⁸⁹.

실행은 보통 자동화된 스케줄러나 CI/CD 파이프라인에서 정해진 주기에 `wrapper.py`를 실행함으로써 이루어지며, 개별적으로 `pre_process.py`를 독립 실행할 경우 `__main__`의 테스트 코드가 수행됩니다. 따라서 일반 사용자는 직접 이 모듈을 호출할 일은 없고, 개발자나 운영 스크립트가 통합 구동하는 구성입니다.

4. 실행 환경 및 의존성:

- Python 버전: 3.11.9 (요구사항)

- 필수 라이브러리:

- `requests` (HTTP 요청으로 파일 다운로드에 사용) - 버전 2.x 이상 권장.

- `Pillow` (PIL 이미지 처리) - 이미지 변환 및 병합에 사용.

- `Ultralytics YOLO` (`ultralytics` 패키지) - YOLOv8 모델 로드 및 추론에 사용¹⁰. 이 모듈은 내부적으로 PyTorch를 사용하므로 `PyTorch` 1.7+ (또는 해당 YOLO 버전에 맞는 버전) 설치가 필요합니다.

- `playwright` (및 `Playwright Chromium` 브라우저) - 사내 SSO를 통한 파일 다운로드에 사용. `Playwright` 설치 후 `playwright install`로 브라우저 세팅 필요.

- `loguru` - 로깅을 위해 임포트되어 있으나, 실제로는 Python 표준 `logging`로 처리하고 있습니다 (`loguru`는 사용되지 않음).

- `pathlib`, `urllib.parse`, `io`, `zipfile`, `traceback` 등 표준 라이브러리.

- `PyMuPDF` (`fitz` 모듈) - PDF 등 문서의 이미지 추출에 필요. (코드에 명시적 `import`는 없지만, `fitz.open()` 호출이 있어 `PyMuPDF` 설치가 전제됩니다.)

- OS 및 환경 제약:

- 기본 설정상 Windows 경로 (C:\\temp\\download_docs)를 사용하고 있으나, 경로는 in_params로 변경 가능하므로 Linux 등 다른 OS에서도 동작 가능합니다. 다만 Playwright의 기본 경로가 Windows로 되어 있어, Linux 서버에서 실행 시에는 download_dir을 적절한 경로로 지정해야 합니다.
- Playwright 사용 시 GUI 환경이 없는 서버에서는 headless=True 옵션 사용이 필요합니다. 현재 함수 download_r_link_with_sso의 기본값은 headless=False이므로, X11 디스플레이가 없는 리눅스 서버에서 실행 시 브라우저 실행 오류가 발생할 수 있습니다. 이 경우 코드를 수정하거나 해당 함수를 호출하지 않도록 환경 변수 설정이 요구됩니다 (EGSS SSO ID/PW가 없으면 자동으로 이 부분을 건너뜁니다).
- 영수증 이미지 처리에는 메모리 사용이 있을 수 있습니다. 특히 대용량 PDF를 이미지로 변환하거나 YOLO 모델을 로드할 때 메모리를 많이 소모할 수 있으므로, 10MB 이상의 파일은 처리하지 않도록 제한을 두고 있습니다 (validate_file_size 함수에서 파일 크기 10MB 이상이면 오류) 11.

5. 외부 연동 요소:

- **파일 다운로드 (HTTP/SSO):** download_file_from_url() 함수는 입력 URL로부터 파일을 받아와 download_dir에 저장합니다 12 13. 여기에는 두 가지 특별 케이스가 있습니다: - URL이 R로 시작하는 경우 (예: "R1234..." 형식) 사내 EGSS 시스템의 SSO 인증 링크로 간주하여, download_r_link_with_sso() 함수를 통해 Playwright로 SSO 로그인 후 파일 다운로드를 시도합니다 14. 이 함수는 웹 브라우저 자동화를 통해 ID/PW (환경변수 EGSS_SSO_ID, EGSS_SSO_PW로 제공)로 로그인하고 링크에 연결된 파일을 저장합니다. 성공 시 다운로드된 파일 경로를 반환하며, 실패나 타임아웃 시 None을 반환합니다 15 16. - 일반적인 HTTP/HTTPS URL의 경우, requests.get으로 파일을 다운로드합니다 17. 만약 URL에 '@' 문자가 포함되어 있으면 (URL 내 인증정보 등) '@' 이전 부분만 사용하도록 처리합니다 18. - **DRM 해제 API:** 사내 보안 상의 이유로, 첨부 문서에 DRM(Digital Rights Management)이 걸려 있을 수 있습니다. call_drm_decode_api(file_path) 함수는 내부 DRM 해제 서버 (http://10.158.120.68:8089/drm/decode)에 HTTP POST 요청을 보내 DRM 해제를 시도합니다 19. 본 API의 요청 헤더는 Content-Type: application/json이며, 바디는 {"fileLocation": <파일경로>} 형식입니다 20. 응답으로 {"status": "ok", "data": "<해제된파일경로>"} 와 같이 해제된 파일 경로를 받을 수 있습니다. 코드에서는 상태코드 200이며 res_json["status"] == "ok" 이고 data 필드가 존재하면 성공으로 판단하여 해당 경로를 반환합니다 21. DRM 해제가 실패하거나 대상 파일이 DRM이 아닐 경우, 경고 로그를 남기고 원본 경로를 그대로 반환합니다 22. (즉, DRM 해제에 실패해도 프로세스는 중단되지 않고 원본 파일로 계속 진행). - **YOLO 모델 (딥러닝 모델):** 영수증 이미지 내에서 실제 영수증 영역을 찾아내기 위해 YOLO (You Only Look Once) 객체 검출 모델을 사용합니다. ultralytics.YOLO 클래스를 통해 미리 학습된 모델 가중치 파일(.pt)을 로드하며, 모델 경로는 설정에 yolo_model_path로 주어집니다 23. 이 모델은 영수증이나 영수증 항목을 탐지하도록 훈련된 것으로 가정됩니다. crop_receipts_with_yolo() 함수 내부에서 model(png_path)로 추론을 실행하면 결과 객체에 bounding box 리스트가 포함됩니다 24 25. 코드에서는 yolo_results = model(png_path) 후 첫 번째 결과의 boxes를 받아와 탐지된 박스 좌표들을 사용합니다 25. YOLO 결과의 boxes 객체는 각 검출된 객체의 좌표와 점수, 클래스 정보를 포함하며, boxes.xyxy를 통해 이미지 내 좌상단(x1,y1)과 우하단(x2,y2) 픽셀 좌표를 얻을 수 있습니다 26. 이 좌표를 이용해 PIL.Image에서 해당 영역을 크롭합니다.
- **주의:** YOLO 모델의 출력 클래스에 대한 조건이 코드에 없는데, 이는 이 모델이 영수증만 검출하거나 또는 모든 검출 결과를 영수증으로 간주한다는 전제입니다. 만약 범용 모델(예: COCO 데이터셋 학습)이면 사람, 물체 등 영수증 이외 객체를 검출할 수 있으므로, 이상적인 상황에서는 영수증 클래스만 필터링하도록 개선이 필요할 수 있습니다. 현재 코드는 검출된 바운딩박스 개수만을 기준으로 처리합니다. - **파일 시스템 I/O:** 전처리 과정에서 여러 디렉터리가 사용됩니다. 예를 들어, download_dir는 원본 파일 및 중간 산출물 저장 위치, merged_doc_dir는 문서의 모든 페이지 이미지를 병합한 파일 저장 위치(기본값: download_dir/document_merged), cropped_dir는 잘라낸 영수증 이미지 저장 위치(download_dir/cropped)입니다 27 28. 함수 내에서 os.makedirs로 경로가 없을 경우 자동 생성합니다.
- **환경 변수:** 앞서 언급한 EGSS SSO ID/PW 외에, 특별한 환경 변수 의존성은 없습니다. (Azure OCR 키 등은 in_params로 전달함)

6. 입력값 및 파라미터 설명:

- | 함수 | 주요 | 입력: | 전처리의 | 메인 | 함수는 |
|----|----|-----|------|----|-----|
| - | | | | | |

- `run_pre_pre_process(in_params: dict, db_record: dict) -> list` 입니다 ²⁹.
- `in_params` 딕셔너리는 파이프라인 전체 설정값을 담고 있으며, 전처리에 필요한 키는 다음과 같습니다:
 - `"download_dir"` : 원본 및 중간 파일 저장 디렉토리 경로. (예: `"./output"` 또는 `"C:\\temp\\download_docs"`)
 - `"yolo_model_path"` : YOLO 모델 가중치 파일(.pt) 경로. (예: `"./yolo/best.pt"`)
 - `"merged_doc_dir"` : (옵션) 문서 파일 처리 시 병합 이미지 저장 디렉토리. 지정하지 않으면 `download_dir/document_merged` 를 사용 ³⁰.
 - 그 외 `in_params`에는 후속 단계에서 쓸 설정들도 포함되지만, 전처리 단계에서 사용되는 것은 위 세 가지가 핵심입니다. (예: `output_dir`, `preprocessed_dir`, `cropped_dir` 등의 키가 테스트 코드에 보이지만, 실제 함수에서는 `download_dir` 하나로 통합해 사용합니다. 설정 키는 혼용되고 있어, **실행 시 일관되게 맞춰주는 것**이 필요합니다.)
 - `db_record` 딕셔너리는 DB에서 조회된 단일 레코드로, 최소 다음 필드를 포함해야 합니다 ³¹:
 - `"FIID"` : 문서 (영수증 집합)를 식별하는 ID (예: `"FIN20230701-001"` 형태).
 - `"LINE_INDEX"` : 해당 문서 내 라인 인덱스 (정수). DB 상 SEQ로 제공되며, 영수증 묶음의 순번인 듯 합니다.
 - `"GUBUN"` : 구분 코드 (예: `"Y"` 등 특정 의미의 코드값). 파이프라인 논리에는 영향 없고 식별 용도로 그대로 전파 됩니다.
 - `"ATTACH_FILE"` : 원본 영수증 첨부 이미지의 URL (단일 이미지용). 없으면 None 또는 빈 문자열.
 - `"FILE_PATH"` : 원본 영수증이 포함된 파일 경로 (PDF, docx 등 또는 다중 영수증일 수 있음). 없으면 None.
 - (추가 필드가 있을 수 있으나 전처리 단계에서는 위만 사용. `"COMMON_YN"`은 출력에만 쓰이는 값으로 입력에는 없음)
- **구조 예시:**

```
in_params = {
    "download_dir": "./output",
    "yolo_model_path": "./yolo/best.pt",
    "merged_doc_dir": "./output/document_merged"
}
db_record = {
    "FIID": "FIN20230701-001",
    "LINE_INDEX": 1,
    "GUBUN": "Y",
    "ATTACH_FILE": "https://example.com/receipt1.jpg",
    "FILE_PATH": None
}
```

이 경우 `ATTACH_FILE` 만 있으므로 한 장짜리 영수증 이미지로 처리됩니다. - **보조 함수 입력:**

- `download_r_link_with_sso(url: str, sso_id: str, sso_pw: str, download_dir: str, headless: bool) -> str` : SSO 로그인에 필요한 R링크와 자격증명을 받아, 파일 다운로드 후 경로를 반환.
- `call_drm_decode_api(file_path: str) -> str` : DRM 해제 API에 파일경로를 보내고 해제된 경로 반환 (실패 시 원본 경로).
- `extract_images_from_document(file_path: str) -> list[PIL.Image]` : PDF, DOCX, PPTX, XLSX 파일에서 이미지들을 추출하여 PIL Image 객체 리스트 반환.
- `merge_images_vertically(images: list[PIL.Image], output_path: str) -> str` : 여러 이미지를 세로로 붙여 하나의 이미지로 저장, 저장 경로 반환.
- `process_document_file(file_path: str, merged_doc_dir: str) -> str` : 문서 파일 전체 처리. 주어진 문서 파일을 DRM 해제 -> 이미지 추출 -> 이미지 병합하여 `<파일명>merged.png` 생성, 경로 반환. 실패 시 None.

- `validate_file_size(path: str) -> None`: 파일 크기가 10MB 이상이면 `ValueError` 를 발생(예외).
- `download_file_from_url(url: str, save_dir: str, is_file_path: bool) -> str`: URL로부터 파일 다운로드. `is_file_path` 가 True이면 URL이 절대경로가 아닐 경우 사내 기본 서버(`http://apv.skhyunix.com`)를 앞에 붙여 사용 ³². 성공 시 저장된 경로 반환, 실패 시 None.
- `convert_to_png(input_path: str, save_dir: str) -> str`: 이미지 파일을 열어 PNG 형식으로 저장 후 경로 반환.
- `crop_receipts_with_yolo(model: YOLO, png_path: str, file_type: str, base_filename: str, original_img: PIL.Image, fiid: str, line_index: int, gubun: str, receipt_index: int or None, common_yn: int, cropped_dir: str) -> list`: YOLO로 영수증 영역을 탐지하여 잘라낸 이미지 파일들을 저장하고, 결과 정보 리스트 반환. 인자는 대부분 앞 단계에서 준비한 것들로, `file_type` 은 "ATTACH_FILE" 또는 "FILE_PATH" 에 따라 동작 분기, `receipt_index` 는 ATTACH_FILE의 경우 보통 1로 시작, FILE_PATH는 None으로 넣으면 내부에서 1부터 자동 할당, `common_yn` 은 ATTACH_FILE=0, FILE_PATH=1로 설정.
- 각 보조 함수의 입력은 위와 같으며, 필요한 값이 없으면 Assertion이나 예외 없이 None 처리를 하기도 합니다 (예: `download_file_from_url` 에서 EGSS ID/PW 없으면 경고 후 None 리턴 ³³).

7. 출력값 및 결과물 설명:

- `run_pre_pre_process` 함수는 **리스트**를 반환합니다. 리스트의 각 요소는 **딕셔너리**로, **크롭된 개별 영수증 이미**지에 대한 정보를 담고 있습니다 ³⁴. 일반적으로 ATTACH_FILE 입력은 영수증 하나만 해당하므로 결과 리스트 길이는 1개이지만, FILE_PATH 입력(예: 한 이미지에 여러 영수증)인 경우 영수증 검출 개수만큼 여러 딕셔너리가 나올 수 있습니다. 주요 키는 다음과 같습니다: - "FIID", "LINE_INDEX", "GUBUN": 입력과 동일한 식별 정보 (원본 레코드 값 그대로 전달).
- "RECEIPT_INDEX": 해당 FIID/LINE 내에서 영수증 순번. ATTACH_FILE의 경우 보통 1로 설정되고, FILE_PATH의 경우 1부터 검출된 순서대로 부여됩니다 ³⁵ ³⁶.
- "COMMON_YN": ATTACH_FILE의 경우 0, FILE_PATH의 경우 1로 설정되어 원본이 개별 첨부인지 합쳐진 파일인지 나타냅니다 ³⁷.
- "file_path": 전처리 결과 생성된 **크롭된 영수증 이미지 파일의 경로**입니다. PNG 포맷으로 저장되며, 파일명은 <원본파일기본이름>_receipt.png (첨부파일) 또는 <원본파일기본이름>_r{번호}.png (다중이미지의 각 영수증) 형태로 지정됩니다 ³⁸ ³⁹.
- (검출 실패 시) "RESULT_CODE", "RESULT_MESSAGE": 영수증 검출에 실패하거나 오류가 있으면 `file_path` 대신 결과코드와 메시지가 포함됩니다. - 예를 들어 YOLO가 아무 객체도 탐지하지 못한 경우 `RESULT_CODE`: "E001", `RESULT_MESSAGE`: "YOLO 탐지 결과 없음"으로 된 딕셔너리 한 개를 반환합니다 ⁶.
- ATTACH_FILE 모드에서 영수증이 **2개 이상** 탐지된 경우 (한 장에 두 개 이상 영수증이 있을 때) `RESULT_CODE`: "E002", 메시지에 그 개수를 명시하며 해당 레코드 처리를 스킵합니다 ⁴⁰.
- 또한 함수 내부에서 예외 발생 시 `except` 에서 에러 로그만 남기고 결과 리스트를 반환하도록 되어 있는데, 명시적으로 `RESULT_CODE` 를 추가하지는 않습니다. 따라서 치명적 오류가 나면 빈 리스트 혹은 일부만 채워진 리스트가 반환될 수 있습니다. - **저장되는 파일들**: 전처리 과정 중 임시 및 최종 산출물 파일들이 저장됩니다. 주요 파일 위치와 예시는 아래와 같습니다: - 원본 다운로드 파일: `download_dir` 경로 아래 <원본파일명> (예: `./output/receipt.pdf`, `./output/image123.jpg`).
- DRM 해제 파일: DRM이 해제되면 API에서 반환한 경로 (보통 원본과 다른 위치, 예: `C:\decoded\...` 형태)를 사용하며, 처리 완료 후 해당 파일을 삭제합니다 ⁴¹.
- 병합된 문서 이미지: `merged_doc_dir` 경로에 <파일이름>_merged.png (예: `./output/document_merged/receipt_merged.png`). 여러 페이지짜리 문서의 경우 이 한 장의 PNG 이미지에 모든 페이지가 세로로 붙어 있습니다.
- 변환된 PNG 이미지: 원본 또는 병합된 이미지를 PNG로 변환한 결과가 `download_dir` 에 저장됩니다 (예: 원본이 `file.pdf` 였으면 `file.png` 로 저장).
- 크롭된 영수증 이미지: `download_dir/cropped` 폴더에 저장되며, 파일명은 위에서 설명한대로 구성됩니다 (예: `file_receipt.png` 또는 `file_r2.png` 등).
- **반환 예시**: (ATTACH_FILE 한 건, 성공적인 검출의 경우)

```
[
{
  "FIID": "FIN20230701-001",
  "LINE_INDEX": 1,
  "GUBUN": "Y",
  "RECEIPT_INDEX": 1,
  "COMMON_YN": 0,
  "file_path": "/output/cropped/receipt1_receipt.png"
}
]
```

만약 검출 실패 시:

```
[
{
  "FIID": "FIN20230701-001",
  "LINE_INDEX": 1,
  "GUBUN": "Y",
  "RECEIPT_INDEX": null,
  "COMMON_YN": 0,
  "RESULT_CODE": "E001",
  "RESULT_MESSAGE": "YOLO 탐지 결과 없음"
}
]
```

와 같은 구조가 될 것입니다. (실제 RESULT_MESSAGE는 한국어 로그 메시지로 저장됩니다.)

8. 코드 흐름 설명:

`run_pre_pre_process` 함수 내에서 구현된 **전처리 단계별 흐름**은 다음과 같습니다:

- **(1) YOLO 모델 로드:** 함수가 호출되면 가장 먼저 YOLO 모델을 메모리에 로드합니다. `model = YOLO(in_params["yolo_model_path"])`를 통해 모델 객체를 생성합니다 ³⁰. 이 단계에서 모델 파일(.pt)이 없거나 경로 잘못되면 예외가 발생하며, 상위 try에서 잡혀 전체 전처리 실패로 이어집니다.
- **(2) 입력 레코드 처리 루프:** 이어서 `for file_type in ["ATTACH_FILE", "FILE_PATH"]:` 루프로 한 레코드 내 두 종류의 필드에 대해 순차 처리합니다 ⁴². 각 레코드에는 보통 **둘 중 하나만** 값이 있고, 나머지는 None입니다.
- `url = db_record.get(file_type)`: 해당 URL을 가져와서, 값이 없으면 (None 또는 빈 값) 그 타입은 건너뛰고 다음으로 넘어갑니다 ⁴³. 예를 들어 ATTACH_FILE이 존재하면 FILE_PATH는 없으므로 FILE_PATH는 skip하게 됩니다.
- `common_yn`을 0 (ATTACH_FILE) 또는 1 (FILE_PATH)으로 설정하고, `receipt_index`를 ATTACH_FILE인 경우 1로, FILE_PATH인 경우 None으로 미리 지정합니다 ³⁷. (receipt_index는 ATTACH_FILE의 경우 기본 1로 시작, FILE_PATH는 함수 내에서 각 검출마다 1부터 할당).
- **(3) 파일 다운로드:**

- 위에서 얻은 `url`에 대해 `orig_path = download_file_from_url(url, download_dir, is_file_path=(file_type=="FILE_PATH"))`를 호출하여 로컬에 저장합니다 ⁴⁴.
- 이 함수 안에서 앞서 설명한 로직에 따라, EGSS 링크면 SSO 다운로드, 아니면 HTTP GET으로 다운로드 시도합니다.
- 다운로드 성공 시 반환된 `orig_path`는 로컬 파일 경로이며, 실패하거나 스킵된 경우 `None`이 반환됩니다 ⁴⁵.
- 만약 `None`이면 해당 `file_type` 처리를 건너뛰고 다음 루프로 넘어갑니다 ⁴⁶. (예: EGSS 링크인데 환경 변수 ID/PW 없거나, HTTP 링크 404 등). 이때 `[{file_type}] URL 다운로드 스킵됨` 등의 정보 로그를 남깁니다.
- **(4) 문서 파일 처리:** 다운로드된 파일의 확장자를 확인하여, 문서인지 이미지인지 분기합니다 ⁴⁷.
 - 지원하는 문서 확장자: `[".pdf", ".docx", ".pptx", ".xlsx"]`. 이들인 경우 `process_document_file(orig_path, merged_doc_dir)`를 호출합니다 ⁴⁸.
 - `process_document_file` 내부에서는 1) DRM 해제를 시도 (`decoded_path = call_drm_decode_api(file_path)`) ⁴⁹, 2) `extract_images_from_document(decoded_path)`로 문서의 이미지를 추출합니다 ⁵⁰. 이때 PDF의 경우 PyMuPDF로 각 페이지 렌더링하여 이미지로 변환하고, Office 파일은 zip 구조를 열어 `word/media` 또는 `ppt/media` 폴더에서 이미지를 추출합니다 ⁵¹ ⁵². 추출된 PIL Image 객체들을 리스트로 반환하며, 추출 과정에서 오류가 나도 계속 진행 (문서 페이지별 try/continue)하고, 전혀 없으면 빈 리스트 반환합니다. 3) 추출된 이미지 리스트가 비어있으면 경고 로그 출력 후 `None` 반환하여 처리 중단 ⁵⁰. 4) 이미지가 있으면 `merged_path`를 계산 (`<merged_doc_dir>/<파일명>_merged.png`)하고, `merge_images_vertically(images, merged_path)`를 호출하여 한 장의 이미지로 병합 저장합니다 ⁵³. 병합은 각 이미지를 위에서 아래로 붙이는 단순 방식으로 구현되어 있습니다 ⁵⁴. 저장된 `merged.png` 파일 경로를 반환합니다. 5) DRM 해제로 생성된 중간 파일(`decoded_path`)이 원본과 다르면, 처리가 끝난 후 해당 파일을 삭제하여 임시 파일을 정리합니다 ⁵⁵. 6) 모든 과정이 오류 없이 끝나면 병합 이미지 경로를 반환하고 종료하며, 도중 예외 발생 시 에러 로그를 남기고 `None` 반환합니다 ⁵⁶.
 - `run_pre_pre_process`는 `process_document_file`의 반환값을 `merged_path`로 받아옵니다. 만약 `None` (실패 또는 이미지 없음)이면 해당 `file_type` 처리를 skip하고 다음으로 넘어갑니다 ⁴⁸. 정상 반환된 경우, 이후 단계로 진행합니다.
 - 병합된 이미지(`merged_path`)가 이미 PNG 형식으로 저장되지만, 코드상 이어서 `png_path = convert_to_png(merged_path, download_dir)`를 호출하는데 ⁵⁷, 이는 결과적으로 `<download_dir>/<파일명>_merged.png.png`와 같이 이중 확장자 파일을 하나 더 만들 가능성이 있습니다. (예: `receipt_merged.png` -> 저장 -> `convert_to_png` -> `receipt_merged.png` 라는 이름에서 다시 `.png` 붙여 `receipt_merged.png.png`). 이 부분은 구현상의 불필요한 중복이며, 실제 실행 시 파일명이 어색하게 될 수 있습니다. (하지만 기능적으로는 동일한 이미지 복사본일 뿐, 큰 문제는 없습니다. 유지보수 참고사항에 언급합니다.)
 - 이미지 파일인 경우 (jpg, png 등 일반 이미지 확장자):
 - `convert_to_png(orig_path, download_dir)`를 바로 호출하여 PNG로 변환합니다 ⁵⁸. 이때 원본이 이미 PNG여도, 함수가 동일 이름에 `.png`를 붙여 저장하므로 (예: input `image.png` -> output `image.png` (뎀어쓰기? 아니면 `image.png.png`로? 코드상 `.png`만 추가하므로 `image.png` -> `image.png.png`), 이 역시 중복 저장이 될 수 있습니다).
 - 변환 결과 `png_path`를 얻습니다.
- 정리하면, 이 단계 결과로 **OCR에 투입할 PNG 이미지 경로**(`png_path`)가 준비됩니다. 문서였던 경우 여러 페이지->병합->PNG 경로, 이미지였던 경우 바로 PNG 경로입니다.

• (5) 영수증 영역 검출 (YOLO 크롭):

- 준비된 `png_path` 를 열어 `PIL.Image.open` 으로 `original_img` 객체를 얻습니다 (with 블록 사용) ⁵⁹.
- `base_filename = os.path.splitext(os.path.basename(png_path))[0]` 로 파일 이름(확장자 제외)을 가져옵니다 ⁵⁹.
- `cropped_dir = os.path.join(download_dir, "cropped")` 로 크롭 이미지 저장 폴더 경로를 지정합니다 ²⁸.
- 이제 핵심 함수인 `crop_receipts_with_yolo(...)` 를 호출합니다 ⁶⁰. 인자로써 준비된 모델, 경로, 이미지 객체, 식별자들(FIID 등), `file_type`, `receipt_index`, `common_yn` 등이 모두 전달됩니다.
 - 함수 내부에서 `yolo_results = model(png_path)` 로 YOLO 추론 수행 → 첫 번째 결과 `yolo_results[0]` 의 `boxes` 객체를 얻습니다 ²⁵.
 - **결과 없음 처리:** 만약 `boxes` 가 None이거나 길이가 0이면, "YOLO 탐지 결과 없음" 경고를 남기고 결과 리스트에 `RESULT_CODE: "E001"` 항목을 추가하여 반환합니다 ⁶. 이 때 FIID, LINE_INDEX 등 식별자도 같이 넣어줍니다.
 - 검출된 box가 있을 경우, `cropped_dir` 경로를 생성 (`os.makedirs`) 하여 준비합니다 ⁶¹.
 - **ATTACH_FILE 분기:** `file_type`이 "ATTACH_FILE"인 경우, **영수증 한 개만 있어야 정상입니다:**
 - 만약 `len(boxes) > 1` 이면, 이는 한 이미지에 여러 영수증이 검출된 상황이므로, 경고 로그를 남기고 `RESULT_CODE: "E002"` 와 메시지 ("YOLO 결과 N개 발견 (ATTACH_FILE는 1개만 가능)")를 결과 리스트에 추가한 뒤 반환합니다 ^{40 62}.
 - 그렇지 않고 정확히 1개의 box가 있으면, 해당 box의 좌표를 가져와 (`box_coords = boxes[0].xyxy[0].cpu().numpy()`) `x1,y1,x2,y2` 정수로 변환합니다 ⁶³. 그 영역을 `original_img.crop((x1, y1, x2, y2))` 로 잘라내어 `cropped_img` 를 얻습니다 ⁶⁴.
 - 저장 경로 `cropped_path` 는 `cropped_dir/<base_filename>_receipt.png` 로 설정합니다 ⁶⁵. 그리고 `cropped_img.save(cropped_path)` 로 이미지를 파일로 저장합니다 ⁶⁶.
 - `validate_file_size(cropped_path)` 를 호출하여 크기가 너무 크지 않은지 확인합니다 ⁶⁷. 만약 10MB 이상이라면 이 함수에서 예외를 던지는데, 이 예외는 `crop_receipts_with_yolo` 내에서 except로 잡히고 에러 로그를 남긴 뒤 함수가 결과 빈 리스트를 반환합니다 (예외 상황에서는 `result.append` 하지 않고 except 빠져나와 결국 빈 리스트 리턴) ⁷.
 - 정상이라면, 결과 리스트에 검출 결과 딕셔너리를 추가합니다: FIID 등 식별자, `RECEIPT_INDEX` 는 `receipt_index` 인자가 있으면 그것을, 없으면 1 (ATTACH_FILE의 경우 호출 시 이미 1로 넣었으므로 1), `COMMON_YN`, 그리고 `"file_path": cropped_path` 를 담습니다 ⁶⁸.
 - **FILE_PATH 분기:** `file_type`이 "FILE_PATH"인 경우, 한 이미지에서 여러 영수증을 검출하는 모드입니다:
 - 검출된 `boxes` 리스트를 `enumerate(boxes, 1)` 로 반복하면서 각각 처리합니다 ³⁵.
 - 각 box에 대해 좌표를 뽑아 (`box.xyxy[0].cpu().numpy()`) `x1,...y2` 계산 후 `original_img.crop((x1,y1,x2,y2))` 으로 자릅니다 ³⁵.
 - `cropped_path` 는 `<base_filename>_r{idx}.png` 로 설정하여 번호를 붙입니다 ⁶⁹. 예를 들어 `base_filename`이 "receiptMerged", `idx=1,2,...` 로 파일명을 구분합니다.
 - 크롭 이미지를 저장하고, 마찬가지로 파일 크기 검증을 거칩니다 ^{69 70}.
 - 결과 리스트에 딕셔너리를 추가하는데, 여기서는 `RECEIPT_INDEX` 를 `idx` 값으로 하고 나머지 식별자는 동일하게 포함합니다 ⁷¹. `common_yn`은 1로 동일.
 - 모든 box에 대해 이 과정을 반복하여, 여러 영수증 조각에 대한 정보가 `results` 리스트에 누적됩니다.
 - **예외 처리:** 함수 내부 try-except에서 예외 발생 시 (예: 모델 실행 오류, PIL crop 오류 등) error 로그 `[ERROR] YOLO 크롭 오류: ...` 를 남기고 traceback을 출력한 뒤, **지금까지 누적된 results 리스트** (또는 없으면 빈 리스트)를 반환합니다 ⁷. 이 경우 특정 box까지 처리되다 실패했으면 이미 추가된 결과는 있을 수 있고, 실패 난 것은 추가되지 않은 채 끝납니다.

- YOLO 처리가 끝나면 함수는 `logger.info("[종료] crop_receipts_with_yolo")` 로그 후 results 리스트를 반환합니다 72 .

- `run_pre_pre_process`는 `result = crop_receipts_with_yolo(...)`의 반환 리스트를 받아 `results.extend(result)`로 모아서 누적합니다 73 74 . ATTACH_FILE와 FILE_PATH 각각에 대해 수행되므로, ATTACH_FILE 처리 결과 + FILE_PATH 처리 결과가 합쳐져 최종 `results`가 완성됩니다. 대부분의 경우 둘 중 하나만 처리되므로 그냥 하나의 리스트와 동일합니다.

• (6) 반환 및 종료:

- 루프가 끝나면 (두 file_type 처리 완료 후) `[종료] run_pre_pre_process` 로그를 남기고 결과 리스트를 반환합니다 34 .
- 상위 `wrapper.py`에서는 이 리스트를 받아 다음 OCR 단계로 순회 처리하게 됩니다.

정리하면, 전처리 모듈은 입력으로 받은 DB 레코드의 첨부 이미지 또는 파일 경로를 처리하여, OCR 가능한 이미지(들)의 경로 목록을 산출합니다. 이 과정에서 다양한 상황(SSO로 보호된 링크, DRM 문서, 여러 페이지 문서, 이미지 속 다중 영수증 등)을 처리하도록 구성되어 있습니다.

9. 주요 함수 및 모듈 설명:

위 흐름에서 이미 각 함수의 동작을 설명하였지만, 이해를 돕기 위해 핵심 함수를 다시 요약합니다:

- `run_pre_pre_process(in_params, db_record) -> list`: 전처리 전체를 총괄하는 함수입니다. 전처리 파이프라인의 진입점으로, 설정과 DB 레코드를 받아 파일 다운로드, 이미지 변환, YOLO 크롭까지 모든 단계를 거쳐 결과 리스트를 반환합니다. 반환 리스트에는 성공한 경우 `file_path`가 있는 딕셔너리, 실패한 경우 `RESULT_CODE`가 있는 딕셔너리가 들어있습니다. 내부에서 `download_file_from_url -> (process_document_file or convert_to_png) -> crop_receipts_with_yolo` 순으로 호출합니다.
내부에서 external I/O와 ML모델 호출까지 하므로, 실행에 시간이 걸릴 수 있으며 try/except로 감싸 전체 실패를 캐치하여 빈 리스트라도 반환하도록 안전장치를 뒀습니다.
- `download_file_from_url(url, save_dir, is_file_path) -> str`: 주어진 URL에 대해 파일을 다운로드하는 함수입니다. SSO 인증 링크와 일반 링크를 모두 처리하며, 다운로드 성공 시 로컬 파일 경로를 돌려줍니다. `is_file_path=True`인 경우 URL이 절대경로가 아니라면 사내 호스트를 붙여 내부 경로를 다운로드합니다. `requests.get`으로 HTTP GET하며, 다운로드 후 파일 크기가 10MB 이상이면 `ValueError` 예외를 발생시켜 상위에서 처리하도록 합니다 75 45 .
- `download_r_link_with_sso(url, sso_id, sso_pw, download_dir, headless) -> str`: Playwright를 이용해 사내 SSO 로그인 후 주어진 R-링크의 파일을 다운로드합니다. 15초 내 다운로드가 시작되지 않으면 `TimeoutError`로 간주하며, 성공 시 `download.suggested_filename`으로 받은 파일명을 이용해 지정 폴더에 저장합니다 76 77 . 실패 시 None 반환. 이 함수는 내부적으로 브라우저 UI 요소(id가 `#mid`, `#PASSWORD_INPUT`, `#loginBtn`)를 조작하므로, 해당 요소들이 사내 로그인 페이지에서 맞아야 정상 동작합니다.
- `call_drm_decode_api(file_path) -> str`: DRM 해제 REST API 호출 함수입니다. `requests.post`로 로컬 DRM 해제 서버에 JSON payload를 보내고, 응답으로 해제된 파일 경로를 얻으면 반환합니다 19 78 . 오류 또는 실패 시 경고를 남기고 원본 경로를 그대로 반환합니다.
- `extract_images_from_document(file_path) -> list[Image]`: 문서 파일(PDF, Office)에서 모든 이미지를 추출합니다. PDF의 경우 PyMuPDF (`fitz`) 라이브러리를 사용해 각 페이지를 이미지로 렌더

링 79 하고, DOCX/PPTX/XLSX는 zip 해제를 통해 `.../media/` 폴더의 이미지 파일들을 추출합니다 52
 80. 추출된 PIL Image 객체들을 리스트로 반환하며, 지원하지 않는 확장자일 경우 경고 로그를 남기고 빈 리스트 반환합니다 81. 이 함수는 문서 내 **이미지 인쇄본만 추출**하며, 영수증이 스캔 PDF처럼 이미지 형태로 들어있을 때 유용합니다 (텍스트만 있는 PDF의 경우 이미지로 추출되지 않을 수 있음).

- `merge_images_vertically(images, output_path) -> str`: PIL 이미지 리스트를 받아 하나의 큰 이미지로 합치는 함수입니다. 모든 이미지의 최대 너비와 총 높이를 계산하여 흰색 배경의 새 이미지를 만들고, 리스트 순서대로 위에서부터 붙입니다 54. 완성된 이미지를 지정 경로에 PNG로 저장하고 경로를 반환합니다 82. 여러 페이지 영수증을 한 장으로 만들어 OCR이 한 번에 처리하도록 돕습니다.
- `process_document_file(file_path, merged_doc_dir) -> str or None`: **문서파일 처리 파이프라인** 함수입니다. 위의 DRM -> 추출 -> 병합 과정을 한 번에 수행합니다. 반환값은 병합 이미지 경로이며, 중간 DRM 파일 삭제까지 책임집니다 83 84. 실패하면 None을 반환하므로, 이를 통해 상위에서 문서 처리 성공/실패를 판단합니다.
- `crop_receipts_with_yolo(...) -> list`: **YOLO 탐지 및 이미지 크롭** 핵심 함수입니다. Ultralytics YOLO 모델 객체와 대상 이미지 경로를 입력받아, **영수증 bounding box**들을 검출한 뒤 잘라낸 이미지 파일들을 생성합니다. 이미 설명한 대로, ATTACH_FILE 모드와 FILE_PATH 모드에 따라 로직이 다릅니다. Return은 결과 딕셔너리들의 리스트입니다. YOLO 결과 해석 부분에서 `result.bboxes.xyxy`를 사용하였으며, 좌표를 정수로 변환하여 PIL로 이미지 crop 후 파일 저장, 크기 검증까지 합니다. 이 함수 내에서 오류 상황(결과 없음 E001, 다중 검출 E002, 예외 등)을 적절히 처리하여 상위에 전달합니다 6 40.
- `validate_file_size(path)` (반환없음): 지정 파일이 10MB 이상이면 `ValueError`를 던지는 함수입니다 11. 상위에서 이 예외를 잡아 처리하는 식으로 설계되었습니다 (예: `download_file_from_url`은 자체 try에서 잡고 None 리턴, `crop_receipts_with_yolo`는 try-except로 잡고 error 로그). 이 제한은 너무 큰 이미지를 Azure OCR에 보내지 않도록 하기 위함입니다.

10. 로깅 및 예외 처리 전략:

전처리 모듈에서는 로거 이름을 `"PRE_PRE_PROCESS"`로 설정하여 사용하고 있으며 85, 전반적으로 **INFO 레벨 로그**로 각 단계 시작/종료 및 주요 이벤트를 기록하고, **WARNING/ERROR 레벨**로 예외적 상황을 기록합니다. 예외는 가급적 상위로 전파하지 않고 이 모듈 안에서 처리하거나 `RESULT_CODE` 형태로 반환하여, 파이프라인 전체가 중단되지 않도록 설계되었습니다.

- **로그 메시지**: 거의 모든 함수의 시작과 끝에서 `[시작] 함수명 / [종료] 함수명` 로그를 **INFO**로 출력합니다 (예: `logger.info("[시작] run_pre_pre_process")`, `logger.info("[종료] crop_receipts_with_yolo")` 등) 86 72. 이를 통해 로그 파일에서 각 함수의 실행 흐름을 추적하기 쉽습니다. 또한 처리 완료된 주요 결과를 포함하는 로그도 INFO로 남깁니다. 예를 들어, 파일 다운로드 완료 시 `[종료] download_file_from_url`과 함께 로그, DRM 해제 성공 시 `[DRM] 해제 성공 -> <경로>` 로그, 병합 이미지 저장 완료 시 `[종료] 병합 이미지 저장 완료: <경로>` 로그 등을 남겨, 어떤 파일이 생성되었는지 알 수 있습니다 87 41.

- **경고 (WARNING) 로그**: 문제가 있지만 치명적이지 않은 상황에 사용됩니다. 예시:

- 지원하지 않는 문서 확장자일 때 `[WARN] 지원하지 않는 문서 확장자: ...` 81.
- 문서에서 이미지 추출 실패시 `[WARN] 이미지 추출 실패: 파일명 - 오류메시지` (계속 다음 이미지 시도) 88.
- YOLO 탐지 결과 없음/다중일 때 `logger.warning(...)`으로 E001/E002 상황 기록 6 40.
- EGSS SSO ID/PW 없을 때 `[EGSS] 환경변수 ... 누락 -> 다운로드 스킵` 경고 33.

- 파일이 10MB 넘어 validate에서 걸렸을 때, `download_file_from_url`의 except에서 `[ERROR] 파일 다운로드 실패: ...`로 처리되지만, 10MB 초과 자체에 대한 명시적 경고는 없습니다 (ValueError로 바로 except). 필요 시 개선 여지.

• 예외 (ERROR) 로그 및 예외:

- 다운로드 실패 (`requests` 예외 등)시 `[ERROR] 파일 다운로드 실패: <URL> - <e>` ⁴⁵.
- DRM 해제 오류시 `[ERROR] DRM 해제 오류: <e>`와 `traceback.print_exc()` 출력 ⁸⁹.
- 문서 이미지 추출 오류시 `[ERROR] 문서 이미지 추출 오류: <e>`와 `traceback` ⁹⁰.
- YOLO 크롭 중 오류시 `[ERROR] YOLO 크롭 오류: <e>`와 `traceback` ⁹¹.
- 전처리 전체 (`run_pre_pre_process`)에서 try 밖으로 예외가 나오면 `[ERROR] 전처리 실패: <e>`와 `traceback` 후 빈 리스트 반환 ⁹².

대부분의 함수에서 예외를 잡은 후 **traceback**을 출력하여 상세 오류를 로그에 남기고 있으며, 심각한 문제는 상위로 `raise`하기보다는 오류 로그 기록 후 **동일 함수 내에서 처리 완료**하거나 안전한 값 반환을 선택했습니다. 예를 들어 `insert_pre_pre_process`는 최종적으로 어떤 예외도 상위로 전달하지 않고 빈 리스트를 리턴하므로, `wrapper`에서는 그 레코드에 대해 OCR을 실행하지 않는 정도로 흐름이 이어집니다.

- **로그 출력 위치:** `wrapper.py`의 `__main__`에서 `logging.basicConfig` 설정을 통해 **파일**(`./logs/pipeline.log`)과 **콘솔**에 동시에 로그가 출력되도록 되어 있습니다 ⁹³. 모든 모듈의 로거 (PRE_PRE_PROCESS 등)가 기본적으로 root 핸들러에 로그를 남기기 때문에, `wrapper`에서 설정한 포맷으로 통합되어 기록됩니다. 로그 포맷은 `"%(asctime)s - %(levelname)s - %(message)s"`로, **시간 - 등급 - 메시지** 형태입니다 ⁹³.
예시: (가상의 로그 시퀀스)

```
2025-07-10 09:00:00 - INFO - [시작] run_pre_pre_process
2025-07-10 09:00:00 - INFO - [EGSS] 접속 시도: R12345... # SSO 링크 접속
2025-07-10 09:00:05 - INFO - [EGSS] 다운로드 완료: C:
\temp\download_docs\file.pdf
2025-07-10 09:00:05 - INFO - [시작] process_document_file
2025-07-10 09:00:06 - INFO - [DRM] 해제 성공 → C:
\temp\download_docs\file_decoded.pdf
2025-07-10 09:00:07 - INFO - [종료] extract_images_from_document → 2개 이
미지 추출됨
2025-07-10 09:00:07 - INFO - [종료] 병합 이미지 저장 완료: C:
\temp\download_docs\document_merged\file_merged.png
2025-07-10 09:00:08 - INFO - [종료] process_document_file
2025-07-10 09:00:08 - INFO - [시작] crop_receipts_with_yolo
2025-07-10 09:00:09 - INFO - [완료] 크롭 결과 저장: ...
\output\cropped\file_merged_r1.png (etc.)
2025-07-10 09:00:09 - INFO - [종료] crop_receipts_with_yolo
2025-07-10 09:00:09 - INFO - [종료] run_pre_pre_process
```

이러한 로그를 통해 전처리 단계별 진행 상황과 산출물을 파악할 수 있습니다.

- **예외 흐름:** 전처리 단계에서 발생한 대부분의 예외는 위처럼 로그로만 기록되고 해당 함수가 **정상 종료**하도록 처리됩니다. 예를 들어, SSO 다운로드 실패 -> None 반환 -> 그 `file_type` 스킵, DRM 해제 실패 -> 원본 사용 지속, 이미지 추출 실패 -> None 반환 -> `file_type` 스킵, YOLO 탐지 실패 -> 결과코드 리턴 등으로 이어집니다. 때문에 `wrapper`는 전처리 결과 리스트를 받아 그냥 다음 단계로 진행할 뿐, 여러 상황을 코드적으로 인지하진 않습니다.

치명적인 예외가 try-except로 잡히지 않으면 최종적으로 `run_pre_pre_process`의 전체 try에서 잡혀 `[ERROR]` 전처리 실패를 남기고 빈 리스트를 반환하며, wrapper는 그 레코드를 처리하지 못하게 됩니다.

11. 테스트 방법 및 사용 예시:

- **단위 테스트:** 각 함수별로 단위 테스트를 수행하려면, 가짜 입력값을 주거나 실제 작은 파일을 준비하여 호출해볼 수 있습니다. 예를 들어: - `download_file_from_url`은 인터넷 연결이 가능한 환경에서 테스트 URL (예: 작은 이미지)을 넣어 호출해보고, 반환된 파일이 정확히 저장되는지 확인합니다. (단, 외부망 접근이 어려운 사내 환경에서는 제한될 수 있음)

- `extract_images_from_document`은 샘플 PDF/DOCX를 만들어 1-2 페이지 이미지를 포함시킨 후, 해당 경로를 인자로 호출해 반환된 이미지 리스트의 길이와 크기를 검증합니다.

- `merge_images_vertically`은 PIL Image 두 개를 인메모리 생성하여 리스트로 주고, 반환된 파일이 정확히 두 이미지를 위아래 결합했는지 확인합니다.

- `crop_receipts_with_yolo`은 YOLO 모델이 필요하므로, **사전 학습된 YOLO 모델**과 **테스트 이미지**가 준비되어야 합니다. 예를 들어, 영수증 사진 한 장과 해당 모델(.pt)을 세팅한 후, 함수에 전달하여 기대대로 결과가 나오는지 (리스트 길이, 좌표 영역이 맞게 잘렸는지) 확인합니다. 이때 검출 실패 상황도 테스트하려면 빈 이미지 또는 영수증 없는 이미지를 넣어 E001 코드 반환을 확인할 수 있습니다. - **통합 테스트:** 모듈에는 `if __name__ == "__main__":` 블록에 간단한 통합 테스트 코드가 포함되어 있습니다 ⁸ ⁹. 이 테스트 시나리오는: 1. `in_params`에 `output/cropped` 등 경로와 **YOLO 모델 경로**를 지정하고, 2. `db_record`에 예시 데이터를 채웁니다 (`FIID`: "TEST001", `LINE_INDEX`:1, `GUBUN`:"Y", `ATTACH_FILE`: 실제 이미지 URL 하나, `FILE_PATH`: None).

3. `run_pre_pre_process()`를 호출하여 결과를 받아 출력하고, crop된 파일 경로들을 출력합니다 ⁹⁴ ⁹⁵.

사용자는 이 부분을 활용해 **실제로 동작을 확인할** 수 있습니다. 단, 예제의 `ATTACH_FILE` URL로 들어간 Wikimedia 이미지 (Brandon_Sanderson_signing.jpg)은 영수증 사진이 아니므로 YOLO 모델이 검출 못할 확률이 높습니다. 실제 테스트 시에는 **영수증 이미지 URL**(또는 로컬 경로)로 바꾸고, `yolo_model_path`도 **학습된 영수증 검출 모델 경로**로 설정해야 합니다. 또한 `download_dir` 등도 실제 쓰기 권한 있는 폴더로 설정 필요합니다.

실행 방법: 터미널에서 해당 디렉토리로 이동하여 `python pre_process.py`를 실행하면, 위 `__main__`의 테스트 코드가 동작합니다.


출력으로는 이모지와 함께 테스트 시작 알림, 그리고 결과 리스트 (딕셔너리)와 각각의 크롭 파일 경로들이 콘솔에 찍힙니다.

예상 콘솔 출력:

```
run_pre_pre_process() 테스트 시작
결과 리스트:
[{'FIID': 'TEST001', 'LINE_INDEX': 1, 'GUBUN': 'Y', 'RECEIPT_INDEX': None,
 'COMMON_YN': 0, 'RESULT_CODE': 'E001', 'RESULT_MESSAGE': 'YOLO 탐지 결과 없음'}]

📁 크롭된 이미지 파일들:
오류: {'FIID': 'TEST001', 'LINE_INDEX': 1, ... 'RESULT_MESSAGE': 'YOLO 탐지 결과 없음'}
```

위와 같이, 테스트 이미지에서 영수증을 찾지 못해 오류 코드가 반환되었음을 보여줍니다. 영수증 이미지로 교체하면

" <파일경로>" 형태로 성공 경로가 출력될 것입니다.

• 자주 발생하는 오류 및 해결:

• YOLO 모델 파일 오류: 모델 경로가 잘못되었거나 .pt 파일이 없는 경우 `Ultralytics YOLO` 쪽에서 예외가 발생합니다 (ex: `FileNotFoundError: [Errno 2] No such file or directory: './best.pt'`). 이때 전처리 전체가 실패하여 `[ERROR]` 전처리 실패: ... 로그가 나오고 빈 리스트가 반환될 것입니다. 해결:

`in_params["yolo_model_path"]` 설정을 올바른 경로로 지정하고, 모델 파일 존재 여부를 확인합니다.

- Playwright 브라우저 실행 오류: 서버 환경에서 Display가 없는데 `headless=False` 로 호출되면 `browser.new_context` 부분에서 Timeout/Connection 에러가 날 수 있습니다. 이 경우, headless 모드 사용을 위해 코드를 수정하거나, SSO가 필요한 R링크 다운로드를 피해야 합니다. 환경변수 `EGSS_SSO_ID/PW`를 넣지 않으면 코드가 SSO 다운로드를 건너뛰므로, ID/PW 설정이 곤란하면 애초에 `ATTACH_FILE` 링크로 제공하거나 수동 다운로드 과정을 추가하는 것도 방법입니다.
- PIL 이미지 처리 오류: 간혹 PIL이 이미지를 열 때 `OSError: cannot identify image file` 오류를 낼 수 있습니다 (예: 파일 손상). 이 경우 `convert_to_png` 나 `Image.open` 에서 except로 잡혀 `[ERROR] ...` 로그가 나오고 해당 파일이 스킵됩니다. 입력 데이터의 파일 손상이 의심되면 소스를 확인하거나, `requests.get` 으로 받은 내용이 이미지인지 검증이 필요합니다.
- 파일 경로 관련: Windows 경로 (`C:\...`)를 Linux에서 쓰면 디렉토리 이름에 콜론(:)이 들어가 invalid path가 됩니다. 따라서 Linux에서 돌릴 경우 `download_dir` 등을 `"/temp/download_docs"` 형태로 변경해야 합니다.
- Memory/Performance: 대용량 PDF (수백 페이지 등)은 PyMuPDF로 한꺼번에 렌더링하면 메모리 사용이 높습니다. 현재 10MB 이상이면 아예 막지만, 페이지 수 제한이 없으므로 9MB짜리 500페이지 PDF도 열 수 있습니다. 이런 경우 메모리 부족이나 처리 지연이 발생할 수 있으며, 추후 페이지 수 제한이나 샘플링 전략이 필요할 수 있습니다.

12. 유지보수 및 확장 고려사항:

- **입력 형식 확장:** 현재는 PDF, Office (docx, pptx, xlsx) 및 이미지 파일들을 처리합니다. 만약 새로운 형식 (예: TIFF 여러페이지 이미지, 이미지가 포함된 기타 문서 등)을 지원하려면 `extract_images_from_document` 에 해당 형식 처리가 추가되어야 합니다. 예를 들어 TIFF는 PIL로 여러 frame을 열 수 있으므로 별도 처리, HTML 파일의 경우 브라우저 렌더링 등 방법을 고려할 수 있습니다.
- **SSO 인증 방식 변경:** EGSS 링크 다운로드에 Playwright를 사용하는 것은 비교적 무거운 방식입니다. 추후 **API 방식 SSO**가 제공되거나, 다운로드 링크를 직접 다룰 수 있게 되면 `download_r_link_with_sso` 부분을 대체하거나 옵션으로 전환하는 것이 좋습니다. 또한 현재 headless 기본값을 True로 바꾸는 것이 서버 운영에는 더 안전하므로, 유지보수 시 이 부분을 수정하는 것이 권장됩니다.
- **DRM 서버 변경:** `call_drm_decode_api` 의 IP나 경로가 바뀔 수 있으므로, 이러한 상수를 설정화할 필요가 있습니다. 예를 들어 `in_params` 나 config 파일로 DRM API URL을 받을 수 있도록 개선하면 유연성이 높아집니다.
- **YOLO 모델 교체 및 다중 클래스 처리:** 영수증 이외에도 다른 유형 문서를 처리해야 하거나, YOLO 모델을 변경해야 할 경우 (`ultralitics` 업그레이드, 혹은 TensorFlow 등 다른 프레임워크) 인터페이스가 달라질 수 있습니다. 현재 Ultralytics YOLO v8 기준 코드는 잘 동작하지만, 향후 버전에서 결과 객체 구조가 바뀔 가능성이 있습니다⁹⁶. 예컨대 `result.bboxes`가 None일 경우 처리 등을 더욱 엄밀히 해야 할 수도 있습니다. 또한 현재는 YOLO detection 결과 중 클래스 구분을 하지 않고 모두 영수증으로 간주하므로, 만약 모델이 여러 클래스를 검출하도록 학습되었다면 특정 클래스 필터링 로직을 추가해야 합니다 (예: `if result.names[int(cls)] == 'receipt':` 등의 처리).
- **파일 크기 및 해상도 한계:** Azure Form Recognizer의 입력 제한은 50 MB, 10000x10000px 등⁹⁷ 이므로, 현재 10MB 제한은 너무 보수적일 수 있습니다. 향후 정책에 따라 이 제한을 조정하거나, 해상도가 큰 이미지의 경우 리사이즈(축소)하여 처리 시간을 줄이는 최적화도 고려할 수 있습니다.
- **폴더 구조/경로 설정 통일:** 코드 내에 `output_dir`, `preprocessed_dir`, `cropped_dir` 등을 사용하는 흔적이 있으나, 실제 함수에서는 `download_dir` 하나로 일괄 처리합니다. 설정 키의 일관성이 떨어지면 유지보수자가 혼란을 겪을 수 있으므로, 차후 `in_params` 구조를 정리하는 것이 좋습니다. 예를 들어 `download_dir`를 `output_dir`로 이름 통일하거나, 미사용 키를 제거하여 코드와 설정을 맞추는 개선이 필요합니다.
- **병렬 처리 주의:** 현재 전처리 단계에서는 멀티스레드 이슈가 적으나, `ultralitics.YOLO` 모델 객체를 각 스레드에서 개별 로드하고 있습니다 (각 thread에서 `YOLO(model_path)` 호출). 만약 모델 파일 크기가 크다면 메모리를 많이 차지할 수 있으므로, **하나의 모델 객체를 공유하거나** 프로세스 풀로 변경하는 것도 고려됩니다. 다만 Ultralytics 모델은 thread-unsafe할 수 있으므로 공유 시에는 락이 필요할 수 있습니다. - **전처리 결과 활용 확장:** 현재는 크롭된 이미지 파일만 결과로 사용하지만, OCR 성능 향상을 위해 **이미지 전처리**(예: 밝기/명암 조정, 회전 보정 등)를 추가할 수 있습니다. 이러한 처리는 YOLO 전후로 삽입 가능하며, 성능 모니터링을 통해 필요 시 구현합니다. - **로그 세분화:** loguru를 도입하려다 만 흔적이 있으므로, 향후 **logging 일원화** 또는 **loguru 전환**을 결정하여 정리할 필요가 있습니다. 또한 현재 DEBUG 레벨 사용은 없는데, 개발/디버깅을 위해 YOLO 검출 좌표값이나 크롭 이미지 크기 등 상세 정보를 DEBUG로 남기면 문제 분석에 유용할 것입니다.

doc_process.py (OCR 인식 모듈)

1. 코드의 목적 및 역할:

`doc_process.py`는 Azure Form Recognizer OCR 서비스를 호출하여 이미지의 텍스트와 필드 정보를 추출하는 모듈입니다. 특히 Azure 영수증 인식 모델(prebuilt-receipt)을 사용하여 영수증 이미지에서 상호명, 전화번호, 날짜, 총액, 품목 리스트 등 구조화된 OCR 결과를 얻는 역할을 합니다 ¹. 이 모듈의 주요 함수인 `run_azure_ocr`는 Azure OCR API 호출을 캡슐화하여, 입력 이미지 파일에 대한 OCR 결과(JSON)를 반환하고 저장하며, 오류 발생 시에도 결과 코드와 메시지를 담아 반환합니다. 간단히 말해, 전처리 이미지 -> Azure OCR -> JSON 출력**의 기능을 수행합니다.

2. 시스템 내 위치와 연동 흐름:

이 모듈은 파이프라인의 OCR 단계에 해당하며, 전처리 후 후처리 전에 실행됩니다. `wrapper.py`에서 전처리가 끝난 후 각 크롭된 영수증 이미지에 대해 `run_azure_ocr(in_params, cropped)`가 호출됩니다 ⁹⁸. 여기서 `cropped`는 전처리 모듈이 반환한 개별 영수증 딕셔너리로, 내부에 "file_path" 키로 이미지 경로를 가지고 있습니다. Azure OCR 호출이 성공하면 결과를 곧바로 Python dict로 돌려주고, `wrapper.py`는 이를 임시 변수로 받아 후처리 단계에 넘깁니다. 또한 `doc_process.py` 자체에서 OCR 결과 JSON을 파일로도 저장하며, 저장 경로는 `in_params["ocr_json_dir"]`로 지정된 폴더입니다. 이 JSON 파일 경로는 후처리 모듈에서 입력으로 사용됩니다 (즉, `doc_process -> JSON 파일 -> post_process` 연계).

만약 Azure OCR 호출이 실패하거나 예외가 발생하면, 이 함수는 **에러 정보를 담은 딕셔너리**를 반환하고, 별도로 **오류 JSON 파일**도 남깁니다. `wrapper.py`에서는 반환 딕셔너리에 "RESULT_CODE" == "AZURE_ERR" 인지 검사하여, 그런 경우 후처리 단계를 건너뛰고 바로 오류 내용을 DB에 저장하는 흐름을 취합니다 ⁹⁹ ¹⁰⁰. 따라서 `doc_process.py`는 wrapper 및 post_process와 협업하여 **성공 시 정상 데이터 전달, 실패 시 오류 데이터 전달**이라는 맥락으로 동작합니다.

3. 사용 대상:

Azure OCR 모듈 역시 **백엔드 서비스**의 일부로, 최종 사용자나 관리자에 의해 직접 실행되지는 않습니다. `wrapper.py`에서 내부적으로 호출되며, 개발자가 기능을 테스트하거나 Azure 설정을 확인할 때 **main** 블록의 샘플 코드를 실행해볼 수 있습니다. `__main__`에는 Azure 서비스에 실제 요청을 보내는 예제가 포함되어 있으므로, Azure **endpoint와 key를 올바르게 설정**하면 직접 이 모듈을 실행하여 OCR 결과를 시험해볼 수 있습니다 ¹⁰¹ ¹⁰². 이 과정은 Azure 요금 과금이 수반될 수 있으므로 테스트 시 유의해야 합니다.

정리하면, 사용 주체는 wrapper (파이프라인 프로세스)이고, 사람은 보통 이를 직접 다루지 않습니다. 다만 Azure Form Recognizer 서비스 계정의 **key** 만료나 **Endpoint 변경** 등의 상황에서는 운영 담당자가 `in_params` 설정을 업데이트해야 할 것입니다.

4. 실행 환경 및 의존성:

- **Python 버전:** 3.11.9 (전체 시스템과 동일)

- **필수 라이브러리:**

- `azure-ai-formrecognizer` - Azure Form Recognizer SDK (Document Intelligence) 클라이언트 라이브러리. 코드는 `DocumentAnalysisClient` 및 `AzureKeyCredential`을 import하고 있으므로, **버전 3.x** 이상이 필요합니다 ¹⁰³. (v3.1+에서 `DocumentAnalysisClient` 도입 ¹⁰⁴). **권장 버전은 v3.2 또는 v4.0** (2023년 기준 최신)입니다.

- `azure-core` - Azure SDK 공통 라이브러리 (AzureKeyCredential 클래스 제공).

- 표준 `os`, `json`, `logging`, `traceback` 등이 사용됩니다.

- **Azure 서비스 요구사항:** 실제 실행에는 **Azure Form Recognizer 리소스**(현재 Azure AI Document Intelligence로 명명) 접근이 필요합니다. 네트워크 통신으로 Azure 클라우드 서비스에 연결하므로, 해당 환경이 인터넷에 나갈 수 있어야 하며 방화벽 허용 등이 필요합니다. (엔드포인트는 HTTPS 443 포트)

- **OS 제약:** OS와 무관하게 Python 라이브러리만 있으면 동작하지만, **인증** 방식으로 API Key를 사용하므로 환경 변수에 키를 두거나, 안전한 저장이 필요합니다. (현재 코드는 키를 `in_params`로 받아 사용)

- **성능:** Azure 서비스 호출에는 네트워크 지연과 서비스 처리 시간이 있고, 영수증 이미지 1장당 보통 1~2초 내외로 응답하나, 비동기 Poller를 사용하므로 코드가 기다리는 동안 다른 스레드 작업을 처리할 수 있습니다. Threadpool로 감싸 병렬 처리하므로, network I/O 대기 시간 동안 다른 OCR도 병렬로 처리되어 전체 속도를 높입니다.

5. 외부 연동 요소:

이 모듈은 **Azure Form Recognizer REST API**와 통신합니다. 내부적으로 Python SDK (`DocumentAnalysisClient`)를 사용하여 API 호출을 수행합니다. 주요 연동 요소:

- **Azure Form Recognizer Endpoint:** `in_params["azure_endpoint"]` 로 전달되는 URL로, 보통 형태는 `https://<리소스이름>.cognitiveservices.azure.com/` 입니다 ¹⁰⁵. 지역별 Endpoint가 있으며, 올바른 Endpoint여야 API 호출이 성공합니다.

- **Azure API Key:** `in_params["azure_key"]` 로 제공되며, AzureKeyCredential로 래핑되어 인증에 사용됩니다 ¹⁰⁶. 잘못된 키이거나 만료된 키면 API 호출 시 401 Unauthorized 오류가 발생할 것입니다.

- **Form Recognizer Model ID:** 코드에서는 `client.begin_analyze_document("prebuilt-receipt", document=f)` 로 **Azure의 영수증 모델**을 지정하고 있습니다 ¹⁰⁷. "prebuilt-receipt" 라는 **모델 ID**는 Azure에서 영수증용으로 미리 학습된 모델이며, **영수증의 상호/전화/날짜/합계/품목 등 필드 추출** 기능을 제공합니다 ¹. 이 API는 asynchronous operation으로 poller를 반환하며, 코드에서는 `poller.result()` 로 결과를 최종 수신합니다 ¹⁰⁷. Azure SDK가 polling을 내부 처리하므로 별도 수동 상태 체크는 없습니다.

- **OCR 결과 JSON 구조:** Azure SDK의 `result = poller.result()` 가 반환하는 것은 `AnalyzeResult` 객체이며, 이를 `result.to_dict()` 로 **파이썬 dict**로 변환해 사용합니다 ¹⁰⁸. 이 dict는 Azure Form Recognizer 서비스의 응답 JSON을 거의 동일하게 표현한 것입니다. 구조 상 최상위 키는 "analyzeResult" 이고 그 안에 문서별 인식 내용이 들어 있습니다 (documents, fields 등). 영수증의 경우, `result_dict["analyzeResult"]["documents"][0]["fields"]` 아래에 **필드명이 key, 추출값이 value**로 들어갑니다. 예: `"MerchantName": {"valueString": "상호명", ...}, "Total": {"valueCurrency": {"amount": 1234.5, "currencySymbol": "$"}, ...}, "Items": {"valueArray": [...]}` 등이 포함됩니다 ^{109 110}.

- **저장 디렉토리:** `in_params["ocr_json_dir"]` 경로가 OCR 결과 JSON을 저장하는 폴더로 쓰입니다 ¹¹¹. 함수 내에서 `os.makedirs(json_dir, exist_ok=True)` 로 경로를 생성하고, 입력 이미지 파일명과 동일한 이름에 `.ocr.json` 확장자를 붙여 JSON을 기록합니다 ^{106 112}. 예: 입력 이미지 `sample_receipt.png` -> 결과 파일 `sample_receipt.ocr.json`.

- **오류시 파일 처리:** Azure 호출 실패 시, `in_params.get("error_json_dir", "./error_json")` 경로를 사용하여 오류 내용을 담은 JSON 파일을 만듭니다 ¹¹³. 파일명은 `fail_{FIID}_{LINE_INDEX}_{RECEIPT_INDEX}_{COMMON_YN}.json` 형식으로 구성됩니다 ¹¹³. 예: FIID=ABC123, LINE_INDEX=1, RECEIPT_INDEX=2, COMMON_YN=0 인 경우 `fail_ABC123_1_2_0.json`. 이 파일은 `post_process`나 `wrapper`에서 직접 사용하지는 않지만, **오류 기록 보존 및 추후 분석용**으로 남깁니다.

6. 입력값 및 파라미터 설명:

- `run_azure_ocr(in_params: dict, record: dict) -> dict`: 이 모듈의 핵심 함수입니다 ¹¹⁴.

- `in_params` 는 파이썬 딕셔너리이며, 이 함수에서 필요한 주요 키는: - "azure_endpoint": Azure Form Recognizer 서비스 엔드포인트 URL. 필수. (예: "https://<resource>.cognitiveservices.azure.com/")

- "azure_key": API 인증 키 문자열. 필수.

- "ocr_json_dir": OCR 결과 JSON 저장 디렉토리. 필수. (없으면 assert 실패)

- "error_json_dir": (선택) 오류 발생 시 JSON 저장 디렉토리. 지정 없으면 기본 `"./error_json"` 사용.

- `record` 는 전처리 단계에서 만들어진 딕셔너리로, 최소: - "file_path": OCR 대상 이미지 파일 경로 (크롭된 영수증 이미지) ¹¹⁵ 가 있어야 합니다. - 그 외 FIID, LINE_INDEX, RECEIPT_INDEX, COMMON_YN, GUBUN 등의 필드는 오류 발생 시 반환값에 포함시키기 위해 요구됩니다 ¹¹⁶. 코드에서 assert로 체크하지는 않지만, 후단 처리 위해 되도록 제공해야 합니다. 실제 `wrapper.py` 에서 호출 시 `cropped` 딕셔너리에는 이들 필드가 모두 포함되어 있습니다 ⁹⁸. - **구조 예시:**

```

in_params = {
    "azure_endpoint": "https://myocrresource.cognitiveservices.azure.com/",
    "azure_key": "abcdef1234567890...",
    "ocr_json_dir": "./ocr_json",
    "error_json_dir": "./error_json"
}
record = {
    "FIID": "FIN20230701-001",
    "LINE_INDEX": 1,
    "RECEIPT_INDEX": 1,
    "COMMON_YN": 0,
    "GUBUN": "Y",
    "file_path": "./output/cropped/receipt1_receipt.png"
}

```

이런 형태로 전달되며, record에 포함된 식별자들은 이후 결과 처리에 이용됩니다. -

AzureKeyCredential(key): Azure SDK 사용 시 자격 증명 객체로, 생성자에 API Key 문자열을 줍니다 ¹¹⁷. 이 부분은 함수 내부에서 직접 사용되므로, 호출자가 신경 쓸 부분은 아닙니다.

- **API 호출 파라미터:** Azure SDK의 begin_analyze_document(model_id, document=file) 호출 시: model_id="prebuilt-receipt", document에 파일 바이너리를 넘깁니다 ¹⁰⁷. Azure 서비스 쪽 파라미터로 locale 등을 지정할 수도 있으나 (특정 영수증 언어 힌트), 코드에서는 기본 설정을 사용합니다. 기본은 **자동 언어 감지**이며, 한국어 영수증도 지원 범위에 들어있습니다.

7. 출력값 및 결과물 설명:

- **정상 출력:** Azure OCR 성공 시 함수는 **Azure 응답의 딕셔너리(result_dict)**를 그대로 반환합니다 ¹¹⁸. 이 dict 구조에는 Azure 분석된 영수증 데이터의 모든 필드와 인식된 값이 포함됩니다. 예를 들어, result_dict["analyzeResult"]["documents"][0]["fields"] 안에: - "MerchantName": {"valueString": "편의점이름", "confidence": 0.99, ...}, - "TransactionDate": {"valueDate": "2023-07-01", ...}, - "Total": {"valueCurrency": {"amount": 5000.0, "symbol": "KRW"}, ...},

- "Items": {"valueArray": [{ "valueObject": { "Description": {"valueString": "물품 1", ...}, "Price": {"valueCurrency": {"amount": 3000.0}}, ... } }, ...]} 등의 데이터가 들어있습니다 ¹¹⁹ ¹²⁰. 이 전체 구조는 후처리 단계에서 가공되므로, 이 함수에서는 변형하지 않고 그대로 넘깁니다.

또한 파일 시스템에 **JSON 파일 저장**을 합니다. 저장 경로는 <ocr_json_dir>/<이미지파일명>.ocr.json이며, ensure_ascii=False 및 indent=2로 저장하여 사람이 읽기에도 편한 UTF-8 JSON 파일입니다 ¹¹². 예: input receipt1_receipt.png -> output ./ocr_json/receipt1_receipt.ocr.json.

반환 dict와 저장 파일 내용은 동일하며, **함수 리턴값으로도 결과를 돌려주기 때문에** wrapper에서는 반환 dict를 바로 변수로 받아 로직에 활용할 수 있습니다. (wrapper.py에서는 이 dict를 ocr_result로 받아 검사만 하고, 실제 후처리에는 파일 저장된 JSON 경로를 이용합니다 ¹²¹ ¹²².) - **실패 출력:** 여러 가지 이유로 Azure OCR이 실패할 수 있습니다: - 예외/오류: 네트워크 장애, 인증 실패(잘못된 키), Azure 서비스 오류 등이 발생하면 except Exception as e: 블록으로 들어갑니다 ¹²³. 이 경우 함수는 **오류 내용 JSON**을 파일로 저장하고, **결과 딕셔너리**에 RESULT_CODE와 RESULT_MESSAGE를 담아 반환합니다. - RESULT_CODE는 고정 "AZURE_ERR"로 설정합니다 ¹²⁴. - RESULT_MESSAGE는 "OCR 실패: <e>" 형태로 예외 메시지를 포함합니다 ¹²⁴. (예: "OCR 실패: Invalid credentials" 또는 "OCR 실패: HTTPSConnectionPool...timed out" 등).

- 이와 함께 입력으로 받은 식별자 값 FIID, LINE_INDEX, RECEIPT_INDEX, COMMON_YN, GUBUN도 모두 넣어줍니다 ¹²⁴. - 그리고 이 dict를 그대로 반환합니다 ¹²⁵. - 추가로, error_json_dir 디렉토리에 위와 동일한 내용을 담은 JSON 파일을 하나 생성합니다 ¹¹³. 파일명은 fail_{FIID}_{LINE_INDEX}_{RECEIPT_INDEX}

`_COMMON_YN}.json`이며,

내용은

`{"RESULT_CODE": "AZURE_ERR", "RESULT_MESSAGE": "...", "FIID": ..., ...}` 등으로 기록됩니다 ¹²⁶. 이 파일은 나중에 별도로 활용되지는 않지만, 로컬에 실패 케이스에 대한 JSON이 남아 문제 파악에 쓰일 수 있습니다.

- 반환 예시 (오류 시):

```
{
  "FIID": "FIN20230701-001",
  "LINE_INDEX": 1,
  "RECEIPT_INDEX": 1,
  "COMMON_YN": 0,
  "GUBUN": "Y",
  "RESULT_CODE": "AZURE_ERR",
  "RESULT_MESSAGE": "OCR 실패: Invalid image format or empty file."
}
```

실제 RESULT_MESSAGE 내용은 예외에 따라 다릅니다.

- 로그 기록: 함수 내에서 `[완료]` OCR 성공 및 JSON 저장: `<json_path>` 로그와 `[종료]` `run_azure_ocr` 로그를 성공 시 남깁니다 ¹²⁷. 실패 시에는 `[ERROR]` OCR 실패: `<e>` 에러 로그와 traceback을 찍고, `[종료]` `run_azure_ocr` (오류로 종료) 로그를 INFO로 남긴 뒤 반환합니다 ^{123 125}.

- **후속 사용:** 정상 반환된 `result_dict`는 wrapper에서 바로 쓰지 않고, `post_process` 단계에서는 **저장된 JSON 파일**을 다시 엽니다 ¹²⁸. 다만 wrapper에서는 OCR 실패를 감지하기 위해 반환 dict의 RESULT_CODE를 확인하여, 실패인 경우 별도 처리하는 로직이 있습니다 (error summary 생성 후 DB 저장) ^{99 129}. 성공한 경우 `post_process` 실행 전 OCR JSON 경로를 조합하는데, 조합 규칙이 여기 저장한 것과 일치해야 합니다 (코드에서 같은 규칙 사용함):

```
os.path.splitext(os.path.basename(cropped['file_path']))[0] +
".ocr.json" 122).
```

8. 코드 흐름 설명:

`run_azure_ocr` 함수의 내부 흐름은 비교적 단순하며, 다음과 같습니다:

- **(1) 필수 파라미터 검증:** 함수 시작 시 `assert` 문을 사용하여 `in_params`에 필요한 키와 `record`에 필요한 키가 있는지 확인합니다 ¹¹¹.
- `"azure_endpoint"`, `"azure_key"`, `"ocr_json_dir"`이 `in_params`에 없으면 `AssertionError`를 발생시킵니다 ¹³⁰.
- `"file_path"`가 `record`에 없으면 마찬가지로 `AssertionError` 발생 ¹³¹.
- `AssertionError` 발생 시 곧바로 `except`로 넘어가 `[ERROR]` OCR 실패: `AssertionError(...)` 로그를 내고 오류 JSON 처리로 이어질 것입니다. (즉, 필수값 누락도 실패로 간주함)

- **(2) Azure Client 초기화:** 필수값이 있으면,

- `endpoint = in_params["azure_endpoint"]`, `key = in_params["azure_key"]`로 변수에 담고,
- `json_dir = in_params["ocr_json_dir"]`로 저장 디렉토리를 가져옵니다 ¹⁰⁶.
- `os.makedirs(json_dir, exist_ok=True)`로 결과 저장 폴더 생성합니다.
- `file_path = record["file_path"]`로 OCR 대상 이미지 경로를 지정합니다.

- `client = DocumentAnalysisClient(endpoint=..., credential=AzureKeyCredential(key))` 로 Azure 클라이언트 객체를 생성합니다 ¹³². 이때 실제로 `AzureKeyCredential`의 유효성은 객체 생성 시 체크 안 하고, 요청 보내봐야 압니다.

• (3) Azure OCR 호출:

- `with open(file_path, "rb") as f:` 로 이미지 파일을 바이너리 읽기 모드로 엽니다 ¹³³.
- 그 파일 객체 `f` 를 넘겨 `poller = client.begin_analyze_document("prebuilt-receipt", document=f)` 를 호출합니다 ¹³⁴. 이로써 Azure 서비스에 OCR 요청이 전송됩니다 (HTTP POST).
- `poller.result()` 를 호출하여 최종 결과를 받아옵니다 ¹³⁵. SDK가 내부적으로 Azure 응답이 준비될 때까지 기다립니다 (Form Recognizer v3.0+은 통상 `async`이지만, SDK가 `polling encapsulate`).
- `result = poller.result()` 가 반환하는 `AnalyzeResult` 객체를 `result.to_dict()` 로 파이썬 딕셔너리화하여 `result_dict` 에 저장합니다 ¹³⁶.
- 이 시점에서 `result_dict` 는 Azure JSON 응답과 동일한 구조를 가진 dict이며, 이후 처리가 용이합니다.

• (4) 결과 저장:

- `base_filename = os.path.splitext(os.path.basename(file_path))[0]` 로 입력 파일명의 확장자를 제거한 이름을 구합니다 (예: `"/path/to/X.png" -> "X"`) ¹³⁷.
- `json_filename = f"{base_filename}.ocr.json"` 로 출력 JSON 파일명을 결정하고, `json_path = os.path.join(json_dir, json_filename)` 로 전체 경로를 만듭니다 ¹³⁷.
- `with open(json_path, "w", encoding="utf-8") as jf: json.dump(result_dict, jf, ensure_ascii=False, indent=2)` 로 JSON 파일을 기록합니다 ¹³⁸. `ensure_ascii=False`로 설정하여 한글 등도 유니코드 이스케이프 없이 저장되며, `indent=2`로 계층 구조를 보기 쉽게 들여쓰기 합니다.
- 저장이 완료되면 info 로그로 `[완료] OCR 성공 및 JSON 저장: <경로>` 를 남깁니다 ¹²⁷.

• (5) 정상 종료:

- `[종료] run_azure_ocr` info 로그를 남기고, 함수는 `return result_dict` 로 인식 결과 딕셔너리를 반환합니다 ¹¹⁸.
- `wrapper`에서는 이 반환값을 받아 `ocr_result` 변수에 저장하고, 이어지는 처리에 사용합니다.

• (6) 오류 처리 흐름:

- Azure 호출이나 파일 I/O 등에서 **Exception**이 발생하면, 곧바로 `except` 블록으로 넘어갑니다 ¹²³. Azure SDK 호출 중 HTTP 에러가 발생하면 `HttpResponseError` 예외 등이 발생할 수 있고, `KeyCredential` 잘못으로 401이면 동일하게 `Exception`으로 잡힙니다.
- `except` 블록에서는 먼저 `logger.error(f"[ERROR] OCR 실패: {e}")` 로 에러 메시지를 로그에 남기고, `traceback.print_exc()` 로 상세 스택을 출력합니다 ¹²³.
- 그 다음, 오류 상황에서도 JSON 파일을 남겨야 하기 때문에:
 - `error_json_dir = in_params.get("error_json_dir", "./error_json")` 로 오류 폴더를 결정하고, `os.makedirs(error_json_dir, exist_ok=True)` 로 디렉토리 준비합니다 ¹¹³.
 - `fail_filename = f"fail_{record.get('FIID')}-{record.get('LINE_INDEX')}-{record.get('RECEIPT_INDEX')}-{record.get('COMMON_YN')}.json"` 로 파일명

을 구성합니다 113. `.get`을 사용하므로 `record`에 키가 없으면 'None' 문자열이 들어갈 수 있지만, 일반적으로 다 있습니다.

- `fail_path = os.path.join(error_json_dir, fail_filename)` 경로를 만들고, `with open(fail_path, "w", encoding="utf-8") as f: json.dump({...}, f, ensure_ascii=False, indent=2)`로 오류 내용을 JSON으로 기록합니다 126. JSON 내용은 `{"RESULT_CODE": "AZURE_ERR", "RESULT_MESSAGE": f"OCR 실패: {str(e)}", "FIID": ..., "LINE_INDEX": ..., ... "GUBUN": ...}` 형식입니다 124.
 - [종료] `run_azure_ocr` (오류로 종료) 로그를 INFO로 남깁니다 139.
 - 마지막으로 `return { ... }`로 딕셔너리를 반환합니다 125. 이 딕셔너리는 JSON에 기록한 내용과 거의 동일하며, `FIID`, `LINE_INDEX`, `RECEIPT_INDEX`, `COMMON_YN`, `GUBUN` 등 원본 식별자들과 `"RESULT_CODE": "AZURE_ERR"`, `"RESULT_MESSAGE": f"OCR 실패: {e}"`가 담겨 있습니다.
- 이 반환값을 받은 `wrapper`는 해당 영수증 처리에 대해 후처리를 수행하지 않고, 곧바로 DB에 실패 결과를 저장하는 분기로 갑니다 (`wrapper`에서 `error summary` 작성 및 `insert` 호출) 99 129. 그러므로 `doc_process` 단계에서 오류가 발생해도, 최종 DB에는 실패 정보가 반영됩니다.

9. 주요 함수 및 모듈 설명:

`doc_process.py`에는 하나의 주요 함수 `run_azure_ocr`만 정의되어 있습니다. (module level logger 설정과 `main` 테스트 코드 외에 다른 함수는 없음)

- `run_azure_ocr(in_params, record) -> dict`: Azure Form Recognizer 호출 및 결과 처리 함수입니다. **인자**: `in_params`는 Azure 호출 설정(엔드포인트, 키, 경로 등), `record`는 대상 이미지 정보. **반환**: OCR 인식 결과를 담은 dict (성공 시 Azure OCR 모든 필드 데이터, 실패 시 오류 코드/메시지). **로직 요약**:
 - 필요한 설정 (`azure_endpoint`, `azure_key`, `ocr_json_dir`)과 입력 파일 경로 존재 확인.
 - Azure DocumentAnalysisClient 생성 및 영수증 모델 호출 (`begin_analyze_document("prebuilt-receipt", ...)`).
 - 결과를 dict로 변환하여 로컬 JSON 파일 저장.
 - 성공 시 dict 반환; 예외 시 오류 내용을 JSON 파일과 dict로 생성하여 반환.이 함수는 네트워크 I/O와 Azure 응답 대기(polling)를 포함하므로 실행 시간이 입력 이미지 크기 및 서비스 응답 속도에 따라 좌우됩니다.
- Azure 1건 호출당 수 초 내외, `ThreadPoolExecutor`로 여러 개 병렬 처리 시 Azure 서비스측 TPS 제한에 유의해야 합니다. (Form Recognizer 무료 계정은 동시 처리 제한이 있을 수 있음)

10. 로깅 및 예외 처리 전략:

- **로그 레벨 및 메시지**: - INFO 레벨: 시작(`"[시작] run_azure_ocr"`) 140, 성공 완료(`"[완료] OCR 성공 및 JSON 저장: ..."`), 종료(`"[종료] run_azure_ocr"`) 127 에 로그를 남깁니다. - ERROR 레벨: OCR 실패 시 (`logger.error("[ERROR] OCR 실패: ...")`) 예외 메시지를 기록합니다 123. - 또한 `traceback.print_exc()`로 콘솔(또는 로그파일)에 파이썬 에러 스택을 출력합니다. 이 출력은 logging 시스템과 별개로 `stderr`에 나올 수 있지만, logging 핸들러에서도 해당 정보가 파일에 기록되도록 할 수도 있습니다. 현재 기본 설정에서는 `traceback`이 **터미널에 출력**되고, `pipeline.log` 파일에는 `[ERROR] OCR 실패: ...`만 남습니다. - 에러 상황에서도 마지막에 `[종료] run_azure_ocr (오류로 종료)` INFO 로그를 넣어두어, 로그 흐름에서 정상 종료와 구분했습니다 139. - **예외 처리**: - 이 함수 내부에서는 모든 오류를 broad exception으로 잡습니다 (`except Exception as e:`). Azure SDK의 다양한 예외뿐 아니라 파일 열기 실패, `AssertionError` 등 모든 예외가 여기서 처리됩니다. - 예외 발생 시 **함수 내부에서 처리 완료 후**, 오류 결과를 반환하므로, 상위(`wrapper`)는 정상 결과와 동일하게 dict를 받습니다. 다만 `RESULT_CODE`로 실패 여부만 구분합니다. - 그러므로 이 함수에서 예외가 발생해도 `wrapper`에는 예외가 전파되지 않으며, 파이프라인 실행은 중단되지 않습니다. - 특정 예외별 세분화는 없지만, 필요하다면 Exception 타입에 따라 별도 메시지를 줄 수 있을 것입니다 (예: `HttpResponseError`를 잡아 key 오류 시 "인증 실패" 등을 해석). 현재 구현은 모든 예외에 동일한 `RESULT_CODE "AZURE_ERR"`로 처리합니다. - **Azure 서비스 관련 예외**: - 잘못된 endpoint나 네트워크 단절 시 `requests.exceptions.ConnectionError` 등이 발생

할 수 있고, AzureKeyCredential 오류는 `ClientAuthenticationError` 등이 날 수 있습니다. 이들도 모두 Exception으로 잡힙니다. RESULT_MESSAGE에는 해당 예외의 메시지가 포함되므로, 예컨대 키 잘못이면 "OCR 실패: (401) ... Invalid credentials" 같은 내용이 담길 것입니다. - Azure 서비스 처리 중 영수증이 너무 크거나 복잡해 타임아웃 발생 시 (SDK 기본 타임아웃은 수십 초일 겁니다), 그것도 Exception으로 잡혀 "timed out" 메시지를 남길 것입니다. - **로그 출력 위치:** wrapper에서 설정한 logging에 따라, 여기서 쓰는 `logging.getLogger("AZURE_OCR")`가 root logger에 속해 pipeline.log 및 콘솔에 모두 출력됩니다. AZURE_OCR 라는 이름은 지정했지만, 별도로 핸들러를 추가하지 않아 root에 propagate될 것입니다. wrapper에서 setLevel(INFO)로 했으므로 ERROR/INFO 모두 기록됩니다. 예시: pipeline.log 중 OCR 부분:

```
2025-07-10 09:00:10 - INFO - [시작] run_azure_ocr
2025-07-10 09:00:12 - INFO - [완료] OCR 성공 및 JSON 저장: ./ocr_json/
receipt1_receipt.ocr.json
2025-07-10 09:00:12 - INFO - [종료] run_azure_ocr
```

또는 오류시:

```
2025-07-10 09:00:10 - INFO - [시작] run_azure_ocr
2025-07-10 09:00:11 - ERROR - [ERROR] OCR 실패:
HTTPSConnectionPool(host='...azure.com', port=443): Max retries exceeded...
Traceback (most recent call last):
... (stack trace print) ...
2025-07-10 09:00:11 - INFO - [종료] run_azure_ocr (오류로 종료)
```

이런 로그를 통해 Azure OCR 호출의 성공/실패와 상세 원인을 추적할 수 있습니다.

- **오류 대응:** Azure OCR 실패의 근본 원인이 키/엔드포인트 문제인 경우 즉각적인 조치가 필요합니다 (키 재발급, 네트워크 확인 등). 이미지 자체 문제로 실패했다면 (예: 빈 이미지, 해상도 문제) pipeline의 다른 영수증은 영향 없이 넘어가며, 해당 케이스만 DB에 실패로 기록됩니다.

11. 테스트 방법 및 사용 예시:

- **개발용 테스트 (main):** `doc_process.py` 끝부분에 **main** 블록이 있어, 개발자가 Azure OCR을 실제 호출해 볼 수 있도록 예제 코드를 제공합니다 ¹⁰¹ ¹⁰². 1. `in_params`를 설정: `azure_endpoint` (자신의 Azure Form Recognizer 엔드포인트 URL)과 `azure_key` (유효한 키) 값을 넣고, `ocr_json_dir`와 `error_json_dir` 경로를 지정합니다. 2. `record`를 설정: FIID 등 식별자는 임의 값으로 두고, `"file_path"`에 **OCR을 실행할 이미지 경로를** 넣습니다. 예제에서는 `./test_cropped/sample_receipt.png`로 가정하고 있으며, 실제 테스트 시 존재하는 영수증 이미지 파일 경로로 바꿔야 합니다 ¹⁰². 3. 스크립트를 실행하면, `run_azure_ocr(in_params, record)`의 결과를 받아 `pprint(result)`로 출력합니다 ¹⁴¹. 출력 결과는 Python dict 형태로 터미널에 표시되며, `analyzeResult` 내부 내용을 모두 볼 수 있습니다. - 성공 예시 출력: (요약하여)

```
{
  'apiVersion': '2022-08-31',
  'modelId': 'prebuilt-receipt',
  'analyzeResult': {
    'documents': [ {
      'docType': 'receipt.receiptType',
      'fields': {
        'MerchantName': {'type': 'string', 'valueString': 'STORE NAME', ...},
```

```

        'TransactionDate': {'type': 'date', 'valueDate': '2023-07-01', ...},
        'Total': {'type': 'currency', 'valueCurrency': {'amount': 25.0, 'currencySymbol':
'$'}, ...},
        'Items': {'type': 'array', 'valueArray': [ { 'valueObject': { 'Description': {...}, 'Price':
{...}, ... } }, ... ]},
        ... 기타 필드 ...
    },
    ... 기타 문서 정보 ...
} ],
'pages': [ ... 페이지별 OCR 텍스트 위치 ... ],
'tables': [ ... ]
}
}

```

이러한 상세 데이터가 출력됩니다 (실제 영수증에 따라 다름). - 실패 예시 출력: (엔드포인트 잘못 등)

```

{
  'FIID': 'TEST001',
  'LINE_INDEX': 1,
  'RECEIPT_INDEX': 1,
  'COMMON_YN': 0,
  'GUBUN': 'Y',
  'RESULT_CODE': 'AZURE_ERR',
  'RESULT_MESSAGE': 'OCR 실패: Authentication failed, invalid key'
}

```

등이 출력될 것입니다. - 이 테스트를 수행하려면 앞서 말했듯 Azure 자격 정보가 필요하고, 인터넷 연결이 되어있어야 합니다. 또한 Form Recognizer 인스턴스의 **정책 상 동시 요청 제한** 등을 넘지 않도록 유념합니다 (테스트는 1건이므로 문제없음). - **통합 테스트**: 전체 파이프라인 맥락에서 `run_azure_ocr` 는 wrapper에 의해 호출되므로,

wrapper.py의 **main** 테스트를 이용하면 전처리->OCR->후처리까지 한번에 테스트해볼 수 있습니다. 그러나 Azure 호출은 비용이 발생하므로, 작은 데이터로 제한하는 것이 좋습니다. - **자주 발생하는 오류 및 해결**:

- AssertionError: 'azure_endpoint'가 없습니다.: in_params 설정에 키가 빠진 경우, 함수 진입 시 AssertionError로 except 처리됩니다. 이 경우 RESULT_MESSAGE에 "'azure_endpoint'가 in_params에 없습니다."라는 메시지가 그대로 포함되며, wrapper에서 DB에 저장될 것입니다 ¹¹⁵. 해결: wrapper 세팅 또는 config에서 azure_endpoint, azure_key, ocr_json_dir 등을 빠뜨리지 않게 합니다.

- Authentication / Authorization 오류: Azure Key가 잘못됐거나 만료되면, `ClientAuthenticationError` 또는 `HttpResponseError` 401이 발생할 수 있습니다. RESULT_MESSAGE에는 "PermissionDenied" 등 Azure의 에러 내용이 담깁니다. 이때 키를 올바른 것으로 교체하고 재시도해야 합니다. Endpoint URL이 틀린 경우도 비슷한 오류가 납니다.

- 네트워크 타임아웃: 간혹 Azure 서비스 응답이 지연되면 Timeout (기본 30초~2분 등) 오류가 날 수 있습니다. 이 경우 RESULT_MESSAGE에 "Max retries exceeded" 또는 "Read timed out" 등이 포함될 수 있습니다. 해결: 네트워크 상태를 확인하거나, Azure 지역을 가까운 곳으로 사용, 요청 이미지를 줄여 처리시간 단축 등이 필요합니다.

- 이미지 파일 문제: 입력 이미지가 0바이트이거나, 잘못된 형식이면 Azure 서비스가 "BadArgument" 오류를 반환할 수 있습니다. 예컨대 빈 파일이면 `HttpResponseError: 400 File is empty.` 같은 메시지가 생깁니다. 이 역시 except 처리되어 RESULT_MESSAGE에 들어가므로, 해당 레코드만 실패 처리됩니다. 전처리에서 이미 파일 크기 0 등을 거르는 양으므로, Azure 단계에서 발견될 수 있습니다. - 글자 판독 오류: Azure OCR은 기계 학습 모델로 동작하므로, 인식 결과가 부정확할 수 있습니다. 이는 코드 오류가 아니지만, 후처리 및 사용자 검증 단계에서 종종 나오는 이슈입니다. (예: 상호명 오인식, 합계 잘못 읽음 등) 이는 Azure 모델 업데이트나 이미지 품질 향상(전처리 개선)을 통해 해결하는 방향으로 가야 합니다.

12. 유지보수 및 확장 고려사항:

- **Azure API 버전 대응:** 현재 사용 중인 `DocumentAnalysisClient` 와 `prebuilt-receipt` 모델은 Azure Form Recognizer v3.0+에 해당합니다 ¹⁰⁴. 추후 Azure 서비스 버전이 업그레이드되거나 API 변경이 있을 수 있습니다. 특히 2025년 이후 **Document Intelligence v4.x**로의 전환이 예상되는데, `prebuilt-receipt` 모델 ID와 응답 필드가 조금 변경될 가능성이 있습니다. (예: `MerchantCategory` → `ReceiptType`로 필드명 변경 등 ¹¹⁹). 향후 Azure SDK 업그레이드 시 이 코드를 점검하여, 필드명이나 호출 메서드(`begin_analyze_document` vs 새 메서드) 등에 변경이 있는지 확인해야 합니다. - **필드 활용 및 추가:** Azure 응답에 현재 코드에서는 관심을 두지 않는 필드(예: `MerchantAddress`, `TaxDetails` 등)이 있을 수 있습니다. 필요에 따라 후처리 단계에서 추가 활용하려면 이 `doc_process` 단계에서 할 일은 없지만, Azure 서비스 설정(예: 명시적 필드Extraction 포함 여부)이나 지역 설정 등을 조정할 수 있습니다. 기본은 모든 필드 추출입니다. - **모델 교체 또는 다중 사용:** 만약 **커스텀 모델**을 사용해야 하는 요구가 생긴다면, `run_azure_ocr`에 전달되는 `model_id`를 바꾸는 등 수정이 필요합니다. 현재는 상수 `"prebuilt-receipt"`지만, `in_params`에 `model_id`를 넣고 `begin_analyze_document(in_params.get("model_id", "prebuilt-receipt"), ...)`로 변경하면 커스텀 모델 ID를 사용할 수 있게 확장할 수 있습니다. - **지연에 따른 async 처리:** 현재 `poller.result()`로 동기 대기하지만, 파이프라인이 커지고 Azure 호출량이 늘면 **async 처리** 고려가 필요합니다. Azure SDK는 `async` 메서드를 제공하므로, 향후 `asyncio` 기반 리팩토링 시 `await client.begin_analyze_document(...)` 등으로 변경할 수 있습니다. 그러나 이미 `ThreadPoolExecutor`로 병렬 처리 중이므로, 성능 문제가 크진 않을 수 있습니다. - **OCR 실패 재시도 전략:** 현재는 실패 시 즉시 포기하지만, 일시적 오류(네트워크, Azure 일시 장애 등)의 경우 **한번 재시도** 로직을 넣는 것도 고려 가능합니다. 이를 구현하려면 `Exception` catch후 특정 에러메시지에 한해 일정 횟수 loop 재시도하고, 그래도 안되면 최종 실패 처리하도록 바꿀 수 있습니다. - **Azure 비용 및 호출 모니터링:** 운영상 Form Recognizer 호출에 비용이 발생하므로, 과도한 호출이 되지 않도록 제어가 필요할 수 있습니다. `wrapper`에서 `target_date`를 명시하여 하루치씩만 처리하는 것처럼, 혹시라도 잘못된 인덱스로 중복 호출되지 않게 관리하는 것이 중요합니다. - **대안 OCR 엔진 통합:** 경우에 따라 Azure 대신 오픈소스 OCR(Tesseract 등)이나 다른 클라우드 OCR을 쓰게 될 수도 있습니다. 그런 상황을 대비해 `doc_process.py`를 인터페이스화(예: `run_ocr` 라는 추상 함수 형태로)하고 구현체를 바꿀 수 있게 하면 유지보수성이 높아집니다. 현재 구조에서는 Azure에 종속적이므로, 다른 OCR을 쓰려면 이 파일을 대폭 수정해야 합니다. - **에러 코드 정교화:** 현 버전은 모든 오류에 "AZURE_ERR" 하나로 처리하지만, 향후 인증 문제, 네트워크 문제, 용량 문제 등 원인별 코드를 세분화하면 더 정확한 로깅과 대응이 가능합니다. 예: "AZURE_AUTH_ERR", "AZURE_TIMEOUT", "AZURE_FAIL" 등. 후처리나 `wrapper`에서도 이를 활용해 재시도 또는 알람 처리를 분기할 수 있을 것입니다.

post_process.py (후처리 모듈)

1. 코드의 목적 및 역할:

`post_process.py` 모듈은 **Azure OCR 결과**를 해석하여, 최종적으로 **업무에 활용할 수 있는 데이터 구조**로 변환하는 역할을 합니다. Azure Form Recognizer가 반환한 JSON (영수증의 모든 필드, 텍스트 좌표 등 포함)을 입력으로 받아, 우리가 필요한 **요약 정보 (summary)**와 **항목 리스트 (items)**를 추출 및 정제합니다 ¹⁴². 이 모듈의 핵심인 `post_process_and_save` 함수는 Azure OCR 결과 JSON 파일을 열어, **영수증 전체 요약**(예: 거래처, 날짜, 총액 등)과 **개별 품목 정보**(상품명, 가격, 수량 등)를 딕셔너리로 정리한 후, 이 둘을 `{"summary": ..., "items": [...]}` 구조로 하나의 JSON 파일에 저장합니다 ¹⁴³ ¹⁴⁴. 또한 오류 시에도 오류 내용을 포함한 JSON을 생성하고 예외를 발생시켜 상위에서 감지할 수 있게 합니다. 요약하면, **OCR 결과 JSON -> 요약/품목 JSON** 변환기입니다.

2. 전체 시스템 내 위치와 연동 흐름:

이 모듈은 파이프라인의 **후처리 단계**이며, Azure OCR (`doc_process.py`) 다음, DB 저장 (`db_master.py`) 이전에 수행됩니다. `wrapper.py`에서는 Azure OCR 성공 시, 각 결과 이미지에 대해 `post_process_and_save(..., {...})`를 호출합니다 ¹²² ¹⁴⁵. 인자로는 설정(`in_params`)과 record 정보를 넘기는데, record에는 OCR 결과 JSON 경로(`json_path`)와 영수증 식별자들(FIID 등)이 포함되어 있습니다. 이 함수가 실행되면: - Azure OCR JSON 파일을 읽어와 필요한 필드들을 추출하고 `summary`와 `items`를 생성합니다. - 결과를 `post_json_dir` 폴더에 `<FIID>_<LINE_INDEX>_<RECEIPT_INDEX>_post.json` 이름으로

로 저장합니다 146. - 결과 JSON 경로를 반환하면, wrapper.py 는 이를 받아 DB 저장 함수 (insert_postprocessed_result)에 넘깁니다 147 148. 만약 후처리 도중 오류가 발생하면, post_process_and_save 함수는 RuntimeError 예외를 발생시키게 되어 있습니다 149. 이 예외는 wrapper에서 해당 스레드의 처리로 전파되고, wrapper의 try-except에 걸려 [FATAL] 처리 중 오류로 로깅될 것입니다 150. (현재 wrapper는 post_process 예외를 개별로 잡지 않고 process_single_record 전체 try로 잡고 있음). 그러나 코드상 post_process 함수 내부에서도 실패 시 오류 JSON을 만들고 경로를 리턴하므로, wrapper의 흐름상 예외 대신 오류 JSON 경로를 리턴하도록 사실상 처리되어, DB에는 오류 summary도 저장되도록 조치했습니다 151 152. 정리하면, post_process.py는 wrapper.py에 의해 호출되고, 최종 산출 JSON을 만들어 db_master.py로 넘기는 연결 고리 역할을 합니다.

3. 사용 대상:

이 모듈 역시 내부 서비스용이며, 최종 사용자가 직접 실행하지 않습니다. 대신, 개발자나 운영자가 결과를 확인하거나 테스트할 때 main 블록을 사용 가능합니다 153 154. __main__에서는 이미 존재하는 OCR JSON 파일 (테스트용)과 설정을 넣어 함수를 실행해 볼 수 있도록 예제가 있습니다. 예제를 통해 개발자는 Azure OCR의 결과(JSON)가 정확히 어떤 summary/items로 변환되는지 확인할 수 있습니다. 일반적인 사용 시나리오: wrapper로 전체 파이프라인이 돌 때 자동으로 후처리가 수행되며, 결과 JSON 파일과 DB 입력으로 이어집니다. 개발 시에는 OCR JSON 구조 변화나 추가 필드 반영 등을 테스트하기 위해 독립적으로 이 모듈을 실행해볼 수 있습니다.

4. 실행 환경 및 의존성:

- **Python 버전:** 3.11.9 (다른 모듈들과 동일)
- **필수 라이브러리:**
 - 표준 os, json, logging, traceback, datetime 등이 사용됩니다.
 - re (정규식)도 import되어 있으나, 현재 코드에서는 사용되지 않았습니다 (필요시 잠재적으로 문자열 필터링에 쓰일 수 있음).
 - 외부 서비스 호출이나 특별한 패키지는 없고, Azure JSON 포맷에 대한 사전 지식이 코드에 반영되어 있습니다. (필드명, 구조 등)
 - **OS 제약:** OS 영향은 거의 없으며, 단순 파일 읽기/쓰기 연산만 있습니다. 경로 구분자 정도만 신경 쓰면 되고, Windows/Linux 모두 문제없습니다.
 - **입력 파일 요구사항:** 이 모듈이 다루는 입력은 Azure OCR이 생성한 .ocr.json 파일입니다. 반드시 Azure Form Recognizer v3.0 이상 영수증 모델의 출력 형식을 따라야 합니다. 만약 다른 OCR 출력이나 형식이 들어오면, fields 추출이 실패하거나 None을 반환할 수 있습니다. (예: doc.get("fields", {})) 가 비어 있으면 summary 대부분 None 처리됨)
 - **성능:** 후처리 작업 자체는 가볍습니다 (JSON 파싱 및 파이썬 딕셔너리 생성). 1건 처리당 수 마이크로~수십 마이크로 수준이므로, 성능 병목이 되지 않습니다.

5. 외부 연동 요소:

- **파일 I/O:** Azure OCR 결과 JSON 파일을 읽어오고, 후처리 결과 JSON을 새로 저장하는, 파일 입출력이 주요 연동입니다. json_path로 지정된 파일을 열어 json.load로 데이터를 파싱하며 128, 출력은 in_params["postprocess_output_dir"] 경로에 쓰여집니다 155 146. - **경로 관리:** postprocess_output_dir 및 error_json_dir가 연동 요소라 할 수 있습니다. 전자는 정상 결과 저장 위치, 후자는 오류 시 저장 위치. wrapper에서 보통 post_json_dir로 설정 값을 넣어주며, in_params에서 받아 사용합니다 156. error_json_dir은 전처리/OCR에서도 쓰였지만, 여기서도 동일 키를 참조합니다. - **외부 모델/서비스:** 이 모듈은 Azure나 ML모델 호출이 없고 로컬 처리만 합니다. - **DB 연동:** 직접 DB 접근은 없으나, 결과를 DB에 넣기 쉬운 구조로 만드는 게 목적입니다. db_master.insert_postprocessed_result는 이 모듈의 출력 JSON을 파싱해 DB Insert 하므로, 두 모듈이 결과 JSON 포맷으로 연동된다고 볼 수 있습니다. (테이블 필드와 JSON 키 이름이 매핑됨)

예를 들어, summary의 "MERCHANT_NAME"은 RPA_CCR_LINE_SUMM 테이블의 MERCHANT_NAME 컬럼으로, items의 "ITEM_NAME" 등은 RPA_CCR_LINE_ITEMS 테이블의 ITEM_NAME 컬럼으로 대응됩니다 3

157.

- **Azure 출력 연동:** Azure OCR JSON의 필드명을 코드 내에 하드코딩하여 사용하고 있습니다. (예:

"MerchantName", "Total", "Items" 등) ¹⁵⁸ ¹⁵⁹ . 이는 Azure Form Recognizer의 prebuilt-receipt 모델 스키마와 연동된 부분입니다. Azure 모델 업데이트로 필드명이 바뀌면 영향받습니다. 현재 코드에서는 'TotalTax' 등의 필드명도 사용합니다.

6. 입력값 및 파라미터 설명:

- `post_process_and_save(in_params: dict, record: dict) -> str`: 후처리 함수로, Azure OCR 결과를 받아 정리하고 JSON을 저장합니다 ¹⁶⁰ . - `in_params`: 설정 딕셔너리로, 필요한 키: - `"postprocess_output_dir"`: 후처리 결과 JSON을 저장할 디렉토리 경로. 필수.
- `"error_json_dir"`: (선택) 오류 시 JSON 저장 경로. 없으면 기본 `"./error_json"` 사용. (전처리/OCR과 동일 키)
- 이 외에 `in_params`에는 Azure나 DB 관련 키도 있을 수 있지만 이 함수에서는 안 씁니다. (wrapper에서 `{**in_params, "postprocess_output_dir": post_json_dir}` 형태로 넘겨주므로 필요한 키만 추가되고 나머지도 같이 올 수 있으나 영향 없음) ¹⁶¹ . - `record`: 처리 대상 정보 딕셔너리. 필요한 내용: - `"json_path"`: Azure OCR 결과 JSON 파일 경로 ¹⁶² . 이 파일을 읽어 parsing합니다. 필수. (없으면 `FileNotFoundError` 발생)
- `"FIID", "LINE_INDEX", "RECEIPT_INDEX", "COMMON_YN"`: 영수증 식별 정보. 필수. (assert 체크) ¹⁶²
- `"GUBUN"`: 구분 코드 (Y/N 등). summary에 포함만 시킵니다.
- `"ATTACH_FILE"`: 원본 첨부파일 경로 또는 URL. DB에 저장을 위해 summary에 넣습니다. (wrapper에서 record에 ATTACH_FILE를 합쳐서 넘겨줌) ¹⁶³ .
- 주의: `post_process`에서는 `ATTACH_FILE` 를 record에서 optional로 받아 쓰는데, wrapper에서 넣어주므로 존재합니다.
- 구조 예시:

```
in_params = {
    "postprocess_output_dir": "./post_json",
    "error_json_dir": "./error_json"
}
record = {
    "FIID": "FIN20230701-001",
    "LINE_INDEX": 1,
    "RECEIPT_INDEX": 1,
    "COMMON_YN": 0,
    "GUBUN": "Y",
    "ATTACH_FILE": "https://fileserver/receipt1.jpg",
    "json_path": "./ocr_json/receipt1_receipt.ocr.json"
}
```

wrapper에서 이와 유사하게 구성하여 호출합니다. - 함수 동작 중 주요 변수: - `data`: JSON 파일 로드 결과 (dict).

`doc = data.get("analyzeResult", {}).get("documents", [{}])[0]` 를 통해 첫 번째 document 객체를 가져옵니다 ¹⁶⁴ . Azure OCR은 한 이미지에 한 문서로 인식되므로 index 0 사용.

- `fields = doc.get("fields", {})`: 문서 내 필드 딕셔너리. Azure 영수증 모델이 추출한 key-value들입니다 ¹⁶⁵ . 만약 doc이 dict가 아니면 빈 dict로 처리합니다.

- `now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")`: 현재 시각 문자열, create/update date로 사용 ¹⁶⁶ .

- 이 외 `fiid`, `line_index` 등 record에서 가져와 로컬 변수로 씁니다. - `summary` 와 `item_list` 가 이 함수의 핵심 결과이며, 마지막에 합쳐 `result_json` 으로 저장합니다 ¹⁴⁴ .

7. 출력값 및 결과물 설명:

- **정상 출력:** 함수가 성공하면 **생성된 후처리 JSON 파일의 경로**(문자열)을 반환합니다 146. 예: `"/post_json/FIN20230701-001_1_1_post.json"`. 이 파일은 DB 입력의 근거가 되며, summary와 items 두 부분으로 구성됩니다. - **summary:** 영수증 한 건의 요약 정보. 코드에서 만들어지는 summary 딕셔너리에는 다음 항목들이 있습니다 167 168: - `"FIID"`, `"LINE_INDEX"`, `"RECEIPT_INDEX"`, `"COMMON_YN"`, `"GUBUN"`, `"ATTACH_FILE"`: 입력 그대로 넘겨주는 식별 및 참조 정보. - `"COUNTRY"`: 국가 (Azure 필드 `"CountryRegion"`의 `valueCountryRegion` 값). 영수증 발행 국가코드가 추출되면 담깁니다 169. (Azure v4 모델에서 지원). - `"RECEIPT_TYPE"`: 영수증 분류 (Azure `"MerchantCategory"` 혹은 `ReceiptType`의 `valueString`). 예: "식음료", "주유" 등 분류가 인식되면 여기에 저장됩니다 170. 참고: Azure v3.0에는 `MerchantCategory`, v3.1+에서는 `ReceiptType` 필드로 나오며, 코드에서는 `MerchantCategory`로 가져와 `valueString` 저장. - `"MERCHANT_NAME"`: 가맹점/상호 명 (Azure `"MerchantName"`의 `valueString`) 171. - `"MERCHANT_PHONE_NO"`: 가맹점 전화번호 (Azure `"MerchantPhoneNumber"`의 `valueString`) 172. - `"DELIVERY_ADDR"`: 배송지 주소 - 영수증 모델에는 없는 개념으로, 기본값 None으로 설정됩니다 173. (이 항목은 특정 시나리오 대비인 듯: 예를 들어 배달 영수증?) - `"TRANSACTION_DATE"`: 거래 날짜 (Azure `"TransactionDate"`의 `valueDate`) 174. - `"TRANSACTION_TIME"`: 거래 시간 (Azure `"TransactionTime"`의 `valueTime`) 175. - `"TOTAL_AMOUNT"`: 총 금액 (Azure `"Total"`의 `valueCurrency.amount`를 문자열로) 176. - `"SUMTOTAL_AMOUNT"`: 합계 금액 (Azure `"Subtotal"`의 `amount`) 177. - `"TAX_AMOUNT"`: 세액 (Azure `"TotalTax"`의 `amount`) 178. - `"BIZ_NO"`: 사업자번호 - 영수증 모델에는 없는 필드, 기본 None 179. - `"RESULT_CODE"`: 200 (성공)으로 세팅 180. - `"RESULT_MESSAGE"`: "SUCCESS"로 세팅 180. - `"CREATE_DATE"`, `"UPDATE_DATE"`: 처리 시각 (now_str)로 동일하게 세팅 181. 이 summary 딕셔너리는 DB의 RPA_CCR_LINE_SUMM 테이블에 그대로 매핑될 수 있도록 구성되었습니다 (컬럼명 대문자로 거의 동일) 182 183. - **items:** 영수증 항목 리스트. 코드에서 `item_list`를 구성하는 로직은: - Azure `"Items"` 필드를 가져와 `items_field = fields.get("Items", {})` 184. - 만약 `items_field`가 dict이고 내부에 `"valueArray"` 키가 있으면, 그 배열을 순회합니다 185. Azure는 영수증에서 항목 라인을 배열로 반환하며, 각 원소가 `valueObject`로 구성되어 있음 186. - `for idx, item in enumerate(items_field["valueArray"], start=1):` 각 item (object)을 처리. `item.get("valueObject", {})`로 내부 객체를 가져옵니다 187. - 각 항목에 대해 딕셔너리를 만들어 `item_list`에 append: - `"FIID"`, `"LINE_INDEX"`, `"RECEIPT_INDEX"`: summary와 동일 (해당 영수증 식별자). - `"ITEM_INDEX"`: 위 루프의 idx 값 (1부터 순차 부여) 188. - `"ITEM_NAME"`: 항목명 (Azure `"Description"`의 `valueString`) 189. (Azure 2022-06 버전 이후 `Description`으로 명칭 제공 190) - `"ITEM_QTY"`: 수량 (Azure `"Quantity"`의 `valueNumber`, 문자열로) 191. 없으면 None. - `"ITEM_UNIT_PRICE"`: 단가 (Azure `"Price"`의 `valueCurrency.amount`, 문자열로) 192. 없으면 None. - `"ITEM_TOTAL_PRICE"`: 총액 (Azure `"TotalPrice"`의 `amount`) 193. 없으면 None. - `"CONTENTS"`: 해당 item 객체 전체를 JSON 문자열로 저장 194. `json.dumps(obj, ensure_ascii=False)`로, 사람이 읽을 수 있게 원본 구조를 string으로 넣습니다. (예: `{"Description": {"valueString": "Americano", ...}, "Price": {...}, "Quantity": {...}, "TotalPrice": {...}}`를 문자열로) - `"COMMON_YN"`: 이 영수증이 첨부파일인지 여부 (summary의 `COMMON_YN`값 그대로). - `"CREATE_DATE"`, `"UPDATE_DATE"`: summary와 동일한 현재시각. - 위 items 리스트의 각 요소는 DB의 RPA_CCR_LINE_ITEMS 테이블에 대응됩니다 157. - 만약 Items 배열이 없거나 비어있으면, `item_list`는 빈 리스트로 유지됩니다. (Azure에서 영수증 항목을 못 찾으면 Items가 빈배열이거나 아예 없음) - **결과 JSON 파일 구조:** `result_json = {"summary": summary, "items": item_list}`로 구성하고 파일에 저장합니다 144. - 파일명: `<FIID>_<LINE_INDEX>_<RECEIPT_INDEX>_post.json` 146. 예: `FIN20230701-001_1_1_post.json`. - `ensure_ascii=False, indent=2` 옵션으로 UTF-8 JSON 저장. - 이 파일은 wrapper -> db_master로 전달되고, DB Insert에 사용됩니다 3 157. - **반환값:** 함수는 최종 생성된 JSON 파일 경로(`output_path`)를 문자열로 리턴합니다 195.

예를 들어, 위 입력 예시의 영수증이 편의점 영수증이라고 하면, 생성되는 JSON (요약) 예시는:

```
```json
{
 "summary": {
 "FIID": "FIN20230701-001", "LINE_INDEX": 1, "RECEIPT_INDEX": 1,
 "COMMON_YN": 0, "GUBUN": "Y", "ATTACH_FILE": "https://fileserver/
receipt1.jpg",
 "COUNTRY": "KOR", "RECEIPT_TYPE": "식음료", "MERCHANT_NAME": "GS25 종로점",
 "MERCHANT_PHONE_NO": "02-123-4567", "DELIVERY_ADDR": null,
 "TRANSACTION_DATE": "2023-07-01", "TRANSACTION_TIME": "13:45:22",
 "TOTAL_AMOUNT": "5600.0", "SUMTOTAL_AMOUNT": "5091.0", "TAX_AMOUNT":
"509.0",
 "BIZ_NO": null,
 "RESULT_CODE": 200, "RESULT_MESSAGE": "SUCCESS",
 "CREATE_DATE": "2025-08-13 00:00:00", "UPDATE_DATE": "2025-08-13
00:00:00"
 },
 "items": [
 {
 "FIID": "FIN20230701-001", "LINE_INDEX": 1, "RECEIPT_INDEX": 1,
 "ITEM_INDEX": 1,
 "ITEM_NAME": "아메리카노", "ITEM_QTY": "1", "ITEM_UNIT_PRICE": "3000.0",
 "ITEM_TOTAL_PRICE": "3000.0",
 "CONTENTS": "{\n\"Description\": {\n\"valueString\": \"아메리카노\"...},
\n\"Price\": {\n\"valueCurrency\": {\n\"amount\": 3000.0}}, \n\"Quantity\":
{\n\"valueNumber\": 1.0}, \n\"TotalPrice\": {\n\"valueCurrency\": {\n\"amount\":
3000.0}}}",
 "COMMON_YN": 0,
 "CREATE_DATE": "2025-08-13 00:00:00", "UPDATE_DATE": "2025-08-13
00:00:00"
 },
 {
 "FIID": "FIN20230701-001", "LINE_INDEX": 1, "RECEIPT_INDEX": 1,
 "ITEM_INDEX": 2,
 "ITEM_NAME": "샌드위치", "ITEM_QTY": null, "ITEM_UNIT_PRICE": null,
 "ITEM_TOTAL_PRICE": "2600.0",
 "CONTENTS": "{\n\"Description\": {\n\"valueString\": \"샌드위치\"...},
\n\"TotalPrice\": {\n\"valueCurrency\": {\n\"amount\": 2600.0}}}",
 "COMMON_YN": 0,
 "CREATE_DATE": "2025-08-13 00:00:00", "UPDATE_DATE": "2025-08-13
00:00:00"
 }
]
}
```
```

(※ ITEM_QTY, UNIT_PRICE 등이 null인 것은 영수증 인쇄에 수량 표시가 없거나 OCR이 인식 못한 경우로 가정).

• **오류 출력:** 함수 내부에서 예외가 발생하면, catch하여 **오류 JSON 파일**을 생성한 뒤 **RuntimeError**를 발생시킵니다 196 151. 하지만 코드가 조금 특이한데, except에서 오류 JSON을 기록하고 `return error_path`를 하고 있어, 사실상 오류 발생 시에도 경로 문자열을 반환할 수도 있습니다 197. 실제 코드를 보면:

- except Exception as e:
- error_path를 정하고, error_summary 딕셔너리를 만듭니다 151 198.
- error_summary는 summary의 모든 키를 갖추되 값은 None 또는 기본, 그리고 RESULT_CODE "POST_ERR", RESULT_MESSAGE를 오류 메시지로 채웁니다 199.
- 이 summary와 빈 items 리스트를 `fail_...json` 파일로 기록합니다 197.
- 그리고 `return error_path`를 하고 있습니다 152.
- 또한 함수 docstring에는 "오류 발생 시 RuntimeError 발생시킵니다"라고 적혀 있지만, 실제 코드에는 RuntimeError를 raise하지 않고 **error_path**를 리턴하게 되어 있습니다. 이 불일치는 아마 코드 수정 과정에서 바뀐 것으로 보입니다.
- wrapper에서 이 함수 호출을 try로 감싸지 않고 있기 때문에, error_path를 문자열로 받으면 wrapper는 이를 성공으로 간주하여 DB insert를 시도할 것입니다. 다행히 wrapper에서는 insert_postprocessed_result를 호출할 때 error_json_dir에 저장된 파일이라도 그냥 넣어줍니다 100. 즉, wrapper도 Azure 실패 때와 마찬가지로 fail JSON을 DB에 넣고자 설계되어 있습니다.
- 그러므로 현재 구현상 **후처리 오류 시:**
- `post_process_and_save`는 오류 JSON (summary 부분 RESULT_CODE "POST_ERR") 파일을 생성하고 그 경로를 반환합니다 197.
- wrapper는 그것을 받아 DB insert 함수에 넘겨 실행합니다 100. DB insert 함수도 summary.RESULT_CODE "POST_ERR" 그대로 DB에 넣을 것입니다.
- wrapper 상위 try에서 예외가 안나므로, `[FATAL]` 로그는 발생하지 않습니다 (예외를 잡을 게 없으므로).
- 결과적으로, 후처리 오류도 DB에 표시되고 파이프라인은 계속 진행됩니다.
- **오류 JSON 내용:** error_summary에 FIID, LINE_INDEX 등 식별자와 `"RESULT_CODE": "POST_ERR"`, `"RESULT_MESSAGE": str(e)`가 들어갑니다. 나머지 필드 (COUNTRY 등)는 None으로 세팅되었습니다 200. 이 구조는 Azure 실패시 error JSON과 거의 유사하나, RESULT_CODE 값으로 구분됩니다. items 리스트는 빈 배열로 저장됩니다 152.
- error 파일명은 `fail_{FIID}_{LINE_INDEX}.json`으로 Receipt_index 등 일부 포함 안 한 간략 형태인데, 잘못된 부분일 수 있습니다 201. (Azure error에서는 receipt_index까지 넣었는데 여기서는 {receipt_index}가 빠짐). 이 역시 작은 불일치지만, 어차피 DB에 들어가므로 파일은 참고용일 뿐입니다.
- **반환/예외 정리:** 의도는 RuntimeError를 일으켜 상위에서 잡게 하는 것이었지만, 실제로 wrapper는 이 예외를 특정 처리하지 않고 전체 try만 있기에, 차라리 error JSON 경로 리턴이 파이프라인 흐름엔 맞습니다.
- 결과 DB 저장 후 wrapper로서는 한 레코드 처리 끝입니다.

• 로그 기록:

- 성공 시: `[완료]` 후처리 결과 저장: `<output_path>` 와 `[종료]` `post_process_and_save`를 INFO로 로그하고 종료합니다 202. output_path는 생성된 JSON 파일 경로.
- 실패 시: `[ERROR]` 후처리 실패: `<e>`를 로그(ERROR) 및 `traceback.print_exc()` 출력합니다 151. 그리고 error JSON 저장 후 바로 return하므로, `[종료]` 로그는 없습니다 (finally나 else

블록 없음). wrapper에서 예외로 처리하지 않아, 이 함수 내 [종료]가 안 찍혀도 전체 파이프라인은 계속됩니다.

8. 코드 흐름 설명:

post_process_and_save 함수의 내부 로직을 순서대로 보면:

• (1) 입력값 검증:

- assert "postprocess_output_dir" in in_params, ... 후처리 저장 경로 필수 확인 203 .
- for key in ["json_path", "FIID", "LINE_INDEX", "RECEIPT_INDEX", "COMMON_YN"]: 루프로 record에 각 필드가 있는지 assert 확인 162 . 하나라도 없으면 AssertionError 발생 -> except로 가서 처리.

• 이 검증을 통과하면, 필요한 모든 값이 있다고 간주.

• (2) 경로 및 파일 준비:

- json_path = record["json_path"], output_dir = in_params["postprocess_output_dir"] 가져오고, os.makedirs(output_dir, exist_ok=True) 로 출력 디렉토리 생성 204 .
- if not os.path.exists(json_path): raise FileNotFoundError("OCR JSON 파일이 존재하지 않음: ...") 파일 부재 시 예외 발생 205 .

• 이 FileNotFoundError는 except에서 잡혀 처리되는데, except의 e 메시지로 RESULT_MESSAGE에 들어갈 것입니다.

• (3) OCR JSON 로드:

- with open(json_path, "r", encoding="utf-8") as f: data = json.load(f) 164 .
- JSON 파싱 결과를 data(dict)로 얻고,
- doc = data.get("analyzeResult", {}).get("documents", [{}])[0] 165 : analyzeResult 밑 documents 리스트 첫 요소를 doc에 할당. 만약 analyzeResult나 documents 키가 없으면 기본 빈 dict->[{}]->0 으로 doc = {}가 됩니다.
- fields = doc.get("fields", {}) if isinstance(doc, dict) else {} 206 : doc이 dict일 경우 fields를 가져오고, 아니면 빈 dict. Normal case: doc is dict with fields, else Azure output not as expected (rare).

• 이 시점에 fields에는 영수증 인식된 필드들의 {key: {type, value..., confidence,...}} 구조 데이터가 있습니다.

• (4) 공통 정보 설정:

- now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S") 현재시간 문자열 생성 166 .
- record로부터 fiid, line_index, receipt_index, common_yn 등을 가져와 로컬 변수에 저장 166 . attach_file, gubun도 가져옵니다.

• 이는 summary와 items 작성에 쓰입니다.

• (5) summary 생성:

- summary = { ... } 딕셔너리를 구성합니다 167 207 .
- 내용은 위 7번에서 상세 기술한 키: FIID, LINE_INDEX, ..., ATTACH_FILE, COUNTRY, RECEIPT_TYPE, ... UPDATE_DATE까지.

- `fields.get("X", {})` 구조로 값을 가져올 때, 예를 들어 `fields.get("Total", {}).get("valueCurrency", {}).get("amount")`: 만약 값이 있으면 number, 없으면 None 반환될 겁니다. 이를 `str()`로 감싸 문자열화 208 . None에 `str()`하면 "None" 문자열이 될 수 있는데, 금액의 경우 None이면 `str(None) = "None"`이 되어 DB에 "None" 문자열로 저장될 위험이 있습니다. (이건 유지보수 포인트: None이면 빈 문자열이나 NULL로 넣는 게 좋겠으나, 현재 "None"으로 될 가능성 있음).
- TAX_AMOUNT, SUMTOTAL_AMOUNT 등도 마찬가지.
- BIZ_NO, DELIVERY_ADDR 등 필드는 OCR에서 제공 안 하므로 None으로.
- RESULT_CODE=200, RESULT_MESSAGE="SUCCESS", CREATE_DATE, UPDATE_DATE = now_str.

• (6) item_list 생성:

- `item_list = []` 초기화.
- `items_field = fields.get("Items", {})` 184 : Azure OCR 영수증 모델의 Items 필드.
- `if isinstance(items_field, dict) and "valueArray" in items_field:` 검사 185 .
 - True이면, `for idx, item in enumerate(items_field["valueArray"], start=1):` 로 반복 185 .
 - 각 `item` 은 Azure JSON에서 Items 배열의 요소 (ex: {"valueObject": {...}}).
 - `obj = item.get("valueObject", {})` if `item` else `{}` 188 : item이 truthy(존재)하면 valueObject, 없으면 {}.
 - `item_list.append({...})` 로 딕셔너리 생성 209 210 :
 - FIID, LINE_INDEX, RECEIPT_INDEX: 위에서 가져온 값 사용.
 - ITEM_INDEX: idx (1-based).
 - ITEM_NAME: `obj.get("Description", {}).get("valueString")` (Description 필드 문자열) 189 .
 - ITEM_QTY: `str(obj.get("Quantity", {}).get("valueNumber"))` if `obj.get("Quantity")` else None 191 . 수량이 있으면 숫자를 str로, 없으면 None.
 - ITEM_UNIT_PRICE: `str(obj.get("Price", {}).get("valueCurrency", {}).get("amount"))` if `obj.get("Price")` else None 192 . Price 필드 있을 때 amount 추출.
 - ITEM_TOTAL_PRICE: `str(obj.get("TotalPrice", {}).get("valueCurrency", {}).get("amount"))` if `obj.get("TotalPrice")` else None 193 .
 - CONTENTS: `json.dumps(obj, ensure_ascii=False)` 194 . 원래 obj 딕셔너리를 JSON 문자열로 직렬화. `ensure_ascii=False`로 한글 등 보존.
 - COMMON_YN: `common_yn` (첨부 여부 플래그).
 - CREATE_DATE, UPDATE_DATE: `now_str` (같은 생성시각).
 - 모든 items를 append하면 loop 끝.
- If Items field was missing or not dict, the list remains empty.
- (혹 Items is present but not array type (shouldn't happen with Azure v3+), code wouldn't append any.)

• (7) 결과 저장 및 반환:

- `result_json = {"summary": summary, "items": item_list}` 생성 144 .
- `output_filename = f"{fiid}_{line_index}_{receipt_index}_post.json"`,
`output_path = os.path.join(output_dir, output_filename)` 146 .
- `with open(output_path, "w", encoding="utf-8") as out_f:`
`json.dump(result_json, out_f, ensure_ascii=False, indent=2)` 146 . JSON 파일 기록.
- Info 로그 [완료] 후처리 결과 저장: `output_path` 출력 211 .
- Info 로그 [종료] `post_process_and_save` 출력 211 .

- `return output_path` ¹⁹⁵ .wrapper는 이 경로를 받음.

• (8) 오류 처리 흐름:

- try 전체를 감싼 `except Exception as e:`

- 에러 로그 `[ERROR] 후처리 실패: e` 출력, traceback 출력 ¹⁵¹ .
- `error_path = os.path.join(error_json_dir, f"fail_{FIID}_{LINE_INDEX}.json")` (여기 RECEIPT_INDEX 안붙음) ²⁰¹ .
- `os.makedirs(os.path.dirname(error_path), exist_ok=True)` 로 폴더 생성 ²⁰¹ .
- `now_str = datetime.now()` ...
- `error_summary` 딕셔너리 생성: FIID, LINE_INDEX, RECEIPT_INDEX, COMMON_YN, GUBUN, ATTACH_FILE from record, 그리고 Country~BIZ_NO 모두 None, RESULT_CODE "POST_ERR", RESULT_MESSAGE str(e), CREATE_DATE, UPDATE_DATE now_str ²¹² ²⁰⁰ .
- `with open(error_path, "w", encoding="utf-8") as f: json.dump({"summary": error_summary, "items": []}, f, ensure_ascii=False, indent=2)` ¹⁹⁷ .
- `return error_path` ¹⁵² . (RuntimeError를 raise하지 않고 경로 반환)

- 이 설계로 wrapper는 계속 진행, `error_path`를 `post_json_path`로 받음. wrapper가 DB insert 호출하면 `error_summary`가 DB에 들어가게 됨.

- wrapper의 `insert_postprocessed_result`는 `summary.RESULT_CODE "POST_ERR"`를 DB 저장 후, items 0개이면 warning 로그(`[완료] DB 저장 - FIID=..., ⚠ 품목 없음`) 남깁니다 ²¹³ .

9. 주요 함수 및 모듈 설명:

`post_process_and_save` 가 이 모듈의 유일한 함수입니다. 이미 위에서 상세 설명했으므로 간략히 정리:

- `post_process_and_save(in_params, record) -> str`: Azure OCR JSON을 입력받아 영수증 요약 정보와 품목 리스트를 추출하고 결과 JSON 파일로 저장하는 함수입니다.
- **인자:** `in_params` 에서 후처리 결과 디렉토리(`postprocess_output_dir`)를 가져오고, `record` 로 OCR JSON 파일 경로와 해당 영수증의 식별자, 원본파일 정보 등을 받습니다.
- **반환:** 생성된 결과 JSON 파일의 경로 문자열.
- **내부 동작 요약:**
 1. OCR 결과 JSON 파일 로드 (경로 유효성 체크 포함).
 2. Azure 결과 구조에서 필드들(fields) 딕셔너리 추출.
 3. **summary 딕셔너리 작성:** FIID 등 식별자 + Azure 인식 주요 필드(COUNTRY, RECEIPT_TYPE, MERCHANT_NAME, ... TOTAL_AMOUNT 등) 배치. 없으면 None, 성공 코드 200.
 4. **items 리스트 작성:** Azure Items 배열을 순회하여 각 품목의 ITEM_NAME, QTY, PRICE 등을 뽑아 딕셔너리화. 전체 원본 객체도 CONTENTS에 JSON 문자열로 저장.
 5. `summary`와 `items`를 합쳐 output JSON 생성, 지정 폴더에 파일 저장 (파일명은 식별자 기반).
 6. 성공 시 output 경로 반환; 오류 시 `error_summary` JSON 생성하여 별도 fail JSON 저장 후 경로 반환. (RESULT_CODE "POST_ERR")
- 모듈 의존: 이 함수는 Azure JSON 구조에 의존적이며, `db_master`와 협조하여 JSON 키를 DB 칼럼과 맞춰 둔 부분이 핵심입니다.
- 예외 처리: 함수 내에서 Exception 발생 시, error JSON 생성 후 (return)종료하므로, wrapper는 예외를 따로 처리하지 않고 DB에 실패 내용 저장으로 이어감.

10. 로깅 및 예외 처리 전략:

- **로그:** - INFO 레벨: - 함수 시작 시 `[시작] post_process_and_save` 로그 (코드에 있음) ²¹⁴ . - 완료 시 `[완료] 후처리 결과 저장: <path>`, `[종료] post_process_and_save` 로그 ²⁰² . - 이 로그들은 wrapper 로그 파일에 기록되어, 어떤 파일이 생성되었는지, 후처리가 끝났는지 등을 보여줍니다. - ERROR 레벨: - 오류 발생 시

[ERROR] 후처리 실패: e 로그와 traceback 출력 151 . - (Azure OCR 단계와 마찬가지로, traceback은 콘솔/파일 에 남습니다.) - error_summary 저장 후 함수가 종료되므로, [종료] post_process_and_save 는 찍히지 않습니다. - WARNING 레벨: - 이 모듈 자체에서 warning 로그는 사용하지 않았습니다. (items 없을 때 warning? 그 처리는 DB insert쪽에서 함) - **Logging 이름:** logging.getLogger("POST_PROCESS") 로 logger를 정의했지만, logging.basicConfig 에서 root handler만 썼으므로, POST_PROCESS logger도 root로 propagate되어 pipeline.log에 적합합니다. wrapper에 따로 logger = getLogger("WRAPPER")만 setLevel하고, 하위 loggers 안 바꿨으니 기본INFO. - **예외 처리:** - try-except로 함수 전체를 감싸 모든 예외를 처리합니다 215 151 . - except 블록에서 return error_path 하므로, 실제로는 함수는 예외를 상위로 throw하지 않습니다. (RuntimeError를 발생시키지 않음) - 그러나 개발자의 의도는 RuntimeError를 상위에 던지려 한 흔적이 comment에 있으므로, 설계 상 "후처리 오류 시 wrapper가 알아채야 한다"는 생각이었을 겁니다. - wrapper에서는 실제로 이 예외를 받지 않고, error_path를 처리하지만 그마저 DB 저장하므로 결과적으로는 문제없습니다. - DB에 RESULT_CODE "POST_ERR"이 저장되므로 사후 분석 시 후처리 단계 오류임을 구분 가능합니다. - except에서 발생 가능한 예외(FileNotFound, AssertionError 등) 모두 같은 흐름으로 처리합니다. - **오류 시 흐름:** - 후처리 실패하면, error_summary JSON이 생성되어 DB에 입력됨으로써 파이프라인은 해당 영수증을 "실패"로 기록하고 넘어갑니다. - wrapper가 process_single_record를 try-except로 감싸긴 하지만, post_process 내부에서 예외를 다루므로 wrapper try에서는 caught 안되고 그냥 정상흐름으로 간주됩니다. - Wrapper try-except는 log만 찍고 계속이므로, 실질적으로 둘 다 케이스 파이프라인 중단은 없고, concurrency thread join에는 영향 없음. - **로그 출력 예:**

```
2025-07-10 09:00:14 - INFO - [시작] post_process_and_save
2025-07-10 09:00:14 - INFO - [완료] 후처리 결과 저장: ./post_json/
FIN20230701-001_1_1_post.json
2025-07-10 09:00:14 - INFO - [종료] post_process_and_save
```

오류 시:

```
2025-07-10 09:00:14 - INFO - [시작] post_process_and_save
2025-07-10 09:00:14 - ERROR - [ERROR] 후처리 실패: Expecting value: line 1
column 1 (char 0)
Traceback (most recent call last):
... # JSON 파싱 실패 예시
```

(이후 [종료] 로그 없음, wrapper에는 [완료] DB 저장 - ... (⚠ 품목 없음) 경고로그가 찍힐 것)

11. 테스트 방법 및 사용 예시:

- **개별 테스트 (main):** 모듈 하단에 main 블록으로 테스트 코드가 있습니다 153 154 . 1. in_params 설정: "postprocess_output_dir": "./test_postprocess_json", "error_json_dir": "./test_error_json" 등 경로를 지정합니다. (이 경로들은 미리 생성되거나 모듈이 생성함) 2. record 설정: FIID, LINE_INDEX 등 식별자와 "ATTACH_FILE" (예: dummy URL), 그리고 "json_path": "./test_ocr_json/sample_receipt.ocr.json" 을 넣습니다 154 . 이 경로에 실제 OCR JSON 파일이 있어야 테스트가 됩니다. (즉, 사전에 Azure OCR을 돌려 sample_receipt.ocr.json을 준비해야 함) 3. try: output_path = post_process_and_save(in_params, record) 호출, - 출력파일 경로를 출력, - 파일 열여 summary와 items 부분을 pprint로 보여줍니다 216 217 . - except: 실패 시 오류 메시지를 출력. 4. 실행: python post_process.py 로 실행하면, 위 테스트가 돌아갑니다. - **성공 케이스:** - 생성된 파일 경로: ./test_postprocess_json/TEST001_1_1_post.json

- **요약 데이터:** 와 summary dict 출력 (pprint), - **아이템 데이터:** 와 items 리스트 출력. - (이모지 표기로 구분) - 개발자는 이를 보고 summary/items 구조가 의도대로인지 확인합니다. - **실패 케이스:** (예: json_path 잘못 지정) - **테스트 실패:** [에러 내용]이 출력될 것입니다. - 그리고 error_json_dir에 fail_TEST001_1.json 파일이 생성되었는지 확인해볼 수 있습니다. - 이 테스트 예제는 OCR JSON이 미리 필요하므로, doc_process.py 테스트

를 먼저 돌려 sample_receipt.ocr.json 파일을 얻거나, Azure 응답 예시를 handcraft해서 넣어둘 필요가 있습니다. - 하지만 대부분 개발 환경에서 `post_process_and_save` 는 wrapper 전체를 돌렸을 때 함께 테스트되므로, 이 개별 실행은 특정 이슈 디버깅 때 주로 사용할 것입니다. - **통합 테스트:** wrapper.py main 실행 시, doc_process + post_process가 연이어 실행되므로, wrapper __main__로 End-to-End 테스트 가능. 다만 wrapper __main__은 DB 접속이 포함되어 있어 환경 갖춰야 해서, DB 대신 sample data를 `query_data_by_date` 에서 반환하게 하거나, wrapper를 수정한 테스트 harness를 쓸 수 있습니다. - **자주 발생하는 오류 및 해결:**

- OCR JSON 파싱 오류: json_path 파일이 손상되어 있거나 비어있으면 `json.load` 에서 예외가 발생합니다. 이 경우 후처리 실패: Expecting value: line 1 column 1 (char 0) 등의 메시지가 뜰 수 있고, 오류 JSON이 만들어집니다. 발생 원인은 주로 Azure OCR 단계에서 JSON이 잘못 생성되었을 때인데, 앞 단계에서 이미 Azure error 처리했을 것이므로 거의 안 생겨야 합니다. 만약 수동으로 JSON 수정하다 오타냈다면 발생. 해결: OCR JSON 파일을 유효한 JSON으로 확보.
- 필드 키 오류: 현재 Azure 영수증 필드 키들이 하드코딩돼 있는데, 만약 Azure 측에서 키 이름을 바꾸거나 (예: MerchantCategory->ReceiptType) 하면, 코드가 None을 받게 됩니다. 예를 들어 v3.1에서는 ReceiptType 키로 나오는데, 코드에서는 `fields.get("MerchantCategory")` 로 찾으므로 RECEIPT_TYPE이 None이 될 수 있습니다. 이럴 때 결과는 None으로 들어가 DB에 "None" 문자열이 저장되거나 NULL로 처리될 것입니다. 해결: Azure 최신 스키마에 맞춰 키를 수정해야 합니다. (ReceiptType으로 변경 등).
- 자료형 변환 유의: 코드에서 수치들을 str로 변환해 저장하는데, DB 컬럼이 수치형이면 Insert 시 형 변환이 일어납니다. `insert_postprocessed_result` 에서는 모든 바인딩을 문자열로 해도 HANA가 타입 캐스트 할 수 있을 겁니다. 하지만 "None" 문자열이 들어가는 건 의도치 않을 수 있으니, None일 때 아예 key를 넣지 않거나 빈 문자열로 넣게 개선해야 합니다. 현재는 "None"이 저장될 가능성이 있습니다.
- 항목이 너무 많음: Items 배열이 굉장히 길 경우(수백개 품목), stringifying CONTENTS가 JSON 길이를 증가시키고 DB insert 성능에 영향 줄 수 있습니다. HANA에도 max string length 제한이 있을 수 있으므로, CONTENTS 컬럼 타입을 충분히 길게 잡아야 합니다. 현재 code에서는 다 담아버리니, DB 컬럼이 NVARCHAR(5000) 정도는 돼야 안전. 만약 overflow 발생 시 DB insert 시 에러나 잘리는 문제가 생길 수 있습니다.
- 후처리 논리 오류: 만약 Azure OCR 결과에서 특정 값이 의도와 다르게 나와 후처리 로직이 잘못 대응하고 있다면 (예: Tip 필드를 현재 무시 중), 그 부분을 추가 구현해야 합니다. 예: Tip도 받아야 하면 `fields["Tip"]` 처리 추가 등.
- RuntimeError 미사용: 코드 주석대로 RuntimeError를 발생시키지 않아 wrapper에서 특별취급 못 하는 부분은 현재는 문제없지만, 만약 wrapper가 post_process 예외를 잡아 뭔가 하길 원했다면 이 동작이 안 됩니다. (현재 wrapper엔 개별 catch 없으니 무관).
- COMMON_YN 의미: items에도 COMMON_YN을 붙여놨는데, DB에서 사실 summary에만 두어도 될 정보로 보입니다. 이건 중복이지만 문제는 없으며, 혹 DB 테이블에 둘 다 있어서 넣은 것 같습니다.
- ATTACH_FILE null: 첨부파일 없는 경우 (COMMON_YN=1, FILE_PATH로 처리한 경우) summary.ATTACH_FILE에 None이 들어갑니다. DB 컬럼이 허용하나? insert 시 None은 null로 갈테니 OK.

12. 유지보수 및 확장 고려사항:

- **Azure 모델 변경 반영:** Azure Form Recognizer 영수증 모델이 새로운 필드를 지원하거나 필드명이 변경될 경우 이 모듈을 업데이트해야 합니다. - 예: Azure v4.0에서는 Tax 필드명이 TotalTax로 이미 바뀌어 코드도 TotalTax를 쓰고 있습니다 ¹⁸⁶. ReceiptType 필드(영수증 종류)가 추가됐는데 코드에 MerchantCategory로 읽고 있으므로, 향후 ReceiptType으로 바꿔줄 필요가 있습니다.
- Azure가 영수증 라인 아이템의 서브필드 명을 변경하거나 추가할 수도 있습니다 (현재 Description/Price/Quantity/TotalPrice 사용). 추가 필드 (예: 제품 코드 등)를 추출 가능해지면 items에 넣는 로직을 확장하면 됩니다.
- **국가별 처리:** CountryRegion 필드를 받아놓고 있지만 특별히 사용하는 로직은 없습니다. 만약 해외 영수증 등 국가별 다른 통화 처리나 날짜 형식 차이가 있다면, Country에 따라 금액 통화기호, 소수점 구분 등을 처리할 수도 있습니다. 현재는 단순 amount만 문자열로 받고 currencySymbol은 무시합니다 (Total.amount만 저장). 여러 통화 가능성이 없으니 괜찮지만, 국제화 시 고려.
- **데이터 형식 개선:** 위에 언급했듯 None 값을 문자열 "None"으로 저장하는 이슈를 개선하려면, `str()` 로 바로 쓰지 말고, 조건문으로 None이면 None을 유지하거나 "" 넣도록 바꾸는 게 좋습니다. DB Insert시 None은 SQL NULL로 들어갈 수 있습니다 (SQLAlchemy text() 쓰니 :TOTAL_AMOUNT param None -> DB NULL). "None" 문자열이 들어가면 나중에 해석 어려워질 수 있습니다.
- **필드 확장:** - 영수증에 따라 필요한 추가 정보 (예: 적립 포인트, 부가 정보)이 있다면 summary에 새 키를 추가해야 합니다. 현재 BIZ_NO(사업자번호)는 추출 불가해 None만 넣는데, 만약 OCR로 사업자등록번호를 추출하려면 custom 모델이 필요할 수도 있습니다.
- Items에 현재 Discount(할인) 같은 건 없는데, 향후 Azure가 할인정보 추출 지원하면 items에 컬럼 추가하거나 별도 구조 고려해야 합니다.
- Delivery address나 BIZ_NO 처럼 예상되는 필드를 자리만 만들어둔 건, 추후 RPA나 다른 방식으로 구해 넣으려는 의도로 보입니다. 유지보수 시 실제 값 채울 수 있는 경로가 있다면 채우도록 구현할 수 있습니다.
- **오류 처리 흐름 통일:** 현재 pre_process/

doc_process와 post_process의 오류 처리 방식이 조금 다릅니다. post_process에서는 error JSON 만들어 경로 리턴, Azure OCR은 dict 반환, pre_process는 dict 리스트 반환. wrapper에서 모두 받아 처리 가능하나, 모듈들 간 일관성이 다소 부족합니다. 향후 개선 시, 실패 시 모두 예외를 던지고 wrapper 한 곳에서 모아서 JSON 파일 작성+DB 처리하는 방법도 있습니다. 또는 pre_process도 JSON 파일 남기게 할 수도 있고요. - **러닝 없는 후처리 추가 작업**: 후처리 자체는 규칙 기반이지만, 필요하면 정규표현식이나 추론 로직을 넣을 수 있습니다. 예: 상호명에 특수문자 제거, 전화번호 포맷 정규화, 주소 파싱 등. 현재 코드에 re import가 있으나 미사용, 아마 이런 정규화 시도 하다 남긴듯 합니다. 향후 품질 개선으로 이런 부분 (예: 전화번호 형식 통일 등)을 추가 구현할 수 있습니다. - **대용량 처리**: 후처리 모듈은 가벼워 문제없으나, 혹시라도 하나의 JSON에 items 수천개라면 파일 크기 커집니다. 이 경우 DB insert도 오래 걸릴 수 있으니, items가 너무 많으면 (ex 1000개 이상) CSV로 저장 후 DB 로드하는 등 방법 생각할 수도 있지만, 현실 영수증엔 항목이 많아야 수집개라 큰 이슈 아님. - **출력 중복 데이터**: FIID, LINE_INDEX 등이 summary와 item에 중복 저장됩니다. DB 설계상 summary와 items 테이블 연계 위해 중복 저장 허용한 것이지만, JSON 관점에서는冗長. 그러나 문제는 없으며 DB insert 간단히 하기 위함이니 유지. - **DB 스키마 변경 대응**: 만약 RPA_CCR_LINE_SUMM이나 ITEMS 테이블 구조가 바뀌면 (컬럼 추가/삭제/이름변경), 여기에 생성하는 JSON 키도 맞춰야 합니다. 또는 insert_postprocessed_result를 수정해 JSON 키-DB 컬럼 매핑을 조정해야 합니다. 예: "BIZ_NO" 컬럼 없으면 summary에서 제거, 혹은 새 컬럼 "STORE_TYPE" 추가되면 summary["STORE_TYPE"] 생성 등.

wrapper.py (파이프라인 구동 및 통합 모듈)

1. 코드의 목적 및 역할:

wrapper.py는 전체 영수증 OCR 파이프라인의 조율자(orchestrator) 역할을 합니다. 이 모듈은 데이터베이스에서 처리 대상 레코드를 조회하고 (query_data_by_date 사용), 각 레코드마다 전처리 -> OCR -> 후처리 -> DB 저장의 단계들을 순차적으로 실행합니다 218 4. 또한, 멀티스레딩을 통해 여러 레코드를 병렬로 처리하여 성능을 향상시키며 219, 전반적인 로깅 설정도 담당하고 있습니다 220. 요약하면, wrapper.py는 메인 프로그램으로서 각 모듈을 호출하는 흐름을 정의하고, 오류 발생 시 그에 따른 조치를 취하며 (예: 오류 시 해당 영수증 처리 생략, DB에 오류 기록), 최종적으로 모든 처리가 끝났음을 로고로 남기는 역할입니다. 회사 운영 환경에서는 이 wrapper를 실행하는 것만으로 전체 파이프라인이 동작하게 됩니다.

2. 전체 시스템 내 위치와 연동 흐름:

wrapper는 파이프라인의 최상위 레벨 스크립트입니다. 시스템 실행 진입점이 이 모듈에 있다고 볼 수 있습니다 (예: CLI나 배치 작업이 이 wrapper.py를 호출). 내부에서 다른 모든 모듈을 import하여 사용하며, 모듈 간 데이터 흐름의 다리 역할을 합니다: - DB 연동 (db_master.query_data_by_date, db_master.insert_postprocessed_result)을 이용해 입력 데이터 조회 및 출력 저장을 진행합니다 221 100. - 전처리 (pre_process.run_pre_pre_process) → OCR (doc_process.run_azure_ocr) → 후처리 (post_process.post_process_and_save) 함수를 순서대로 호출하여, 하나의 레코드에 대한 전체 처리를 완결합니다 4 99 122. - 이 일련의 처리를 각 레코드별로 concurrent하게 수행하는데, Python의 ThreadPoolExecutor를 사용하여 스레드 풀을 관리하고, 지정된 max_workers (동시 스레드 수) 내에서 병렬 처리합니다 219. - wrapper는 날짜별로 실행되도록 설계되었습니다. in_params["target_date"]를 기준으로 그 날의 영수증 로그를 조회하여 처리하며, target_date가 주어지지 않으면 기본으로 어제 날짜를 사용합니다 222. 이를 통해 일 배치(batch) 단위로 데이터 처리가 이뤄집니다. - wrapper 내부에서는 각 처리 단계의 결과 상태를 확인하여 다음 단계를 수행할지 말지 결정합니다. 예를 들어, 전처리 결과 중 "RESULT_CODE"가 있으면 (오류), OCR을 스킵하고 다음 이미지로 넘어갑니다 223; OCR 결과의 RESULT_CODE "AZURE_ERR"이면 후처리를 건너뛰고 DB에 오류 summary만 저장합니다 99 129. - 최종적으로 모든 레코드 처리가 끝나면 "전체 파이프라인 완료" 로고를 남기고 종료합니다 224.

3. 사용 대상:

이 모듈은 실제 운영 환경에서 주기적으로 실행되는 스크립트입니다. 통상, ETL 배치나 Cron 등의 스케줄러가 python wrapper.py (혹은 해당 모듈 import 후 run_wrapper 호출)로 이 파이프라인을 구동할 것입니다. 사용 대상은 개발/운영팀이고, 실행 시 필요한 설정 (DB 접속 정보, Azure Key 등)은 외부 구성파일 (예: Module_config_dex.toml)이나 환경변수, 인자 등으로 제공됩니다 225.

- 개발자는 이 wrapper를 통해 전체 프로세스를 테스트할 수 있습니다. **main** 블록에는 TOML 설정을 읽어 DB 연결을 맺고, in_params 설정 후 `run_wrapper(in_params)`를 호출하는 예제 코드가 있으므로, config 파일만 알맞게 채우면 로컬에서 실행 가능하게 해두었습니다 ^{225 226}. - 최종 사용자(업무 부서)는 직접 wrapper를 돌리지는 않지만, 이 wrapper를 통해 나온 DB 입력 결과를 활용하게 됩니다. 예를 들어, DB에 쌓인 RPA_CCR_LINE_SUMM, ITEMS 데이터를 RPA나 다른 시스템이 읽어가는 구조일 것입니다. - 운영 측면에서는 wrapper 실행 로그 (`logs/pipeline.log`)와 DB 입력 결과를 모니터링하여, 문제 발생 시 대응합니다. wrapper는 문제가 생겨도 프로세스 전체가 죽지 않고 개별 레벨에서 처리하여, **완료 로그**가 나오면 대부분 레코드 처리가 완료된 상태입니다.

4. 실행 환경 및 의존성:

- **Python 버전:** 3.11.9 (전체 코드 Python 3.11 기준 작성) - **필수 라이브러리:** - `concurrent.futures` (특히 `ThreadPoolExecutor`) - 멀티스레드 병렬 처리를 위해 사용 ^{227 219}. - `sqlalchemy` - DB 연결 및 쿼리 실행에 사용 (HANA DB 연동). `create_engine`와 DBAPI (`hdbcli`)를 활용합니다 ^{227 228}. - `tomllib` - 설정 파일 (TOML) 읽기에 사용 (Python 3.11 내장) ^{229 225}. - 로컬 모듈 import: `db_master`, `pre_pre_process` (또는 `pre_process`; import 이름이 `pre_pre_process`지만 파일명은 `pre_process.py`로 추정), `doc_process`, `post_process` - 이 네 모듈을 불러 사용합니다 ²³⁰. 따라서 해당 파일들이 같은 패키지/경로에 있어야 합니다. - `logging` - 로깅 설정 및 logger 사용. - `datetime` - wrapper 내부에서 현재 날짜 계산에 쓰입니다 (`query_date` default). - `os`, `json` - JSON 쓰는데 json은 wrapper main config parse 외에 안 쓰이네요. (Potentially for something, but not heavily used). - `time` - import 안 되어 있지만, concurrent futures `as_completed` 사용으로 별도 sleep 등은 안 썼음. - **OS 및 환경 제약:** - DB 연동을 위해 **SAP HANA ODBC 드라이버 (hdbcli)**가 설치되어 있고, Python `sqlalchemy-hana` Dialect가 구성되어 있어야 합니다. `create_engine("hdbcli://user:pass@host:port")` 사용이 가능하려면 `hdbcli` 패키지가 설치되어 있어야 하고, SQLAlchemy가 해당 dialect를 인식해야 합니다 ²³¹. (보통 `pip install hdbcli sqlalchemy sqlalchemy-hana` 등으로 설정) ²³². - `Module_config_dex.toml` 파일이 필요한데, 이것이 존재하지 않거나 경로가 다른 **main** 테스트는 못 씁니다. 운영에선 아마 config 경로 정확히 맞춰둘 것. - OS는 상관없으나, 만약 Windows에서 ODBC 접근한다면 `hdbcli` 설치 신경써야 하고, Linux면 `drivers path`. - `ThreadPoolExecutor`는 GIL의 영향으로 CPU-bound 병렬에 한계 있으나, 이 파이프라인은 대부분 I/O bound (파일 I/O, 네트워크 I/O)와 외부 연산(Azure, DB)이라 CPU병목은 적을 것이라, 멀티스레드로 효용이 있습니다. - 스레드 개수 default 4 (`max_workers`), 환경 따라 조정 가능 (`in_params`). - Logging은 파일로 `./logs/pipeline.log`를 쓰므로, 실행 경로에서 logs 폴더 쓰기 권한이 필요. - DB credential 및 Azure key가 코드상 노출되지 않게 .toml이나 env로 관리, 이 wrapper config parsing 부분에서 DB user/password가 log에 찍힐 위험은 없음 (logging before using). - **External Systems:** - SAP HANA DB - network access to DB host/port is needed. The `create_engine` uses a connection string like `hdbcli://user:pass@host:port`, so correct host (likely IP or hostname) and port (3NN15 for tenant etc) must be reachable. - Azure - wrapper itself doesn't call Azure, `doc_process` does. But wrapper passes keys and ensures they are set in `in_params`, so implicitly Azure connectivity needed in `doc_process` stage. - Possibly, since config includes section for `database` or `hana`, wrapper can adapt to Oracle or other DB if needed (comment says Oracle support removed). Right now specific to HANA. - The environment likely sets the Python path such that these module imports find the correct files.

5. 외부 연동 요소:

- **DB (Input):** wrapper uses `db_master.query_data_by_date(in_params)` to fetch records from the `LDCOM_CARDFILE_LOG` table (SAP HANA). The query is parameterized by a date (`LOAD_DATE = target_date`) ²³³. `in_params` must contain `sqlalchemy_conn` (active DB connection) and optionally `target_date` (YYYY-MM-DD string). The query returns list of dict records: each with FIID, GUBUN, LINE_INDEX, ATTACH_FILE, FILE_PATH ²³⁴. These records are what pipeline processes. - If none returned, wrapper logs "처리할 데이터가 없습니다." and returns without further processing ²²¹. - **DB (Output):** wrapper calls `db_master.insert_postprocessed_result(post_json_path, in_params)` after each `post_process` output ²³⁵. This function reads the JSON file and inserts into two tables: `RPA_CCR_LINE_SUMM` and `RPA_CCR_LINE_ITEMS` ^{3 157}. - Wrapper does not directly handle SQL

(except creating engine), delegating to db_master for actual queries. - The synergy: wrapper ensures to call insert for every processed receipt (including fail cases where post_json_path is a fail JSON). - The DB connection is passed around in in_params for both query and insert. - **Threading/Multiprocessing:** uses `ThreadPoolExecutor`. Actually all heavy tasks (OCR, YOLO) release GIL likely (I/O, waiting on external), so CPU concurrency isn't big issue, thread usage is fine. They spawn up to max_workers threads. After submitting tasks, wrapper waits for all to finish (as_completed loop) ²²⁴. - **Logging (File/Console):** wrapper sets up logging to output to a file and console at the beginning of **main** ²²⁰. This means any log messages in other modules propagate to these handlers. - **TOML Config:** wrapper **main** expects a `Module_config_dex.toml` file with database credentials (and possibly other config). They attempt to read `[database]` or `[hana]` section for user/password/host/port ²³⁶. This is an external configuration file that must be present for **main** usage. - **Operating schedule:** Possibly not code-managed but environment should call wrapper daily for yesterday's data (if no target_date given). - **Memory:** in_params passes DB connection. They do engine.connect() and pass a Connection object. Using one connection across multiple threads concurrently for queries/inserts. This is possible if HANA driver supports it; if not, slight risk. (Better would be to give each thread its own connection or use engine thread-safety). - **Cross-module data passing:** uses Python dict (in_params) and dict records. in_params is enriched by wrapper (like adding post_json_dir to pass to post_process).

6. 입력값 및 파라미터 설명:

- `run_wrapper(in_params: dict)`: 이 함수는 wrapper의 메인 동작으로, 파이프라인 전체를 실행합니다 ²³⁷. - `in_params` 딕셔너리에는 파이프라인 전반 설정이 포함되어야 합니다: - `sqlalchemy_conn`: SQLAlchemy 데이터베이스 Connection 객체 (또는 Engine). 필수. DB 조회/입력에 사용. - `target_date`: (선택) 문자열 "YYYY-MM-DD". 지정 시 해당 일자 데이터 처리, 없으면 어제 날짜 기본. - **경로 설정들:** - `output_dir` / `download_dir`: 전처리 원본 및 중간 파일 저장 디렉토리 (pre_process에서 사용). - `preprocessed_dir` / `cropped_dir`: (과거에 사용, 현재 pre_process는 download_dir 하나로 처리하나 여전히 제공가능). - `merged_doc_dir`: (선택) 전처리 병합이미지 저장 폴더. - `yolo_model_path`: YOLO 모델 파일 경로. - `ocr_json_dir`: OCR 결과 저장 폴더. - `post_json_dir`: 후처리 결과 저장 폴더. - `error_json_dir`: 오류 JSON 저장 폴더 (전처리/OCR/후처리 공용). - **API 설정들:** - `azure_endpoint`: Azure OCR 엔드포인트 URL. - `azure_key`: Azure OCR 키. - `max_workers`: (선택) 동시 스레드 수 (기본 4). - 위 항목들은 **main** 예제에서 모두 세팅하는 것을 볼 수 있습니다 ²²⁶. 또한 wrapper `__main__`가 config 파일 읽어 db_conn 생성까지 담당합니다. - **구조 예시:**

```
in_params = {
    "sqlalchemy_conn": engine.connect(),
    "target_date": "2025-07-09",
    "azure_endpoint": "https://<resource>.cognitiveservices.azure.com/",
    "azure_key": "your-azure-key",
    "output_dir": "./output",
    "preprocessed_dir": "./preprocessed",
    "cropped_dir": "./cropped",
    "ocr_json_dir": "./ocr_json",
    "post_json_dir": "./post_json",
    "error_json_dir": "./error_json",
    "yolo_model_path": "./yolo/best.pt",
    "max_workers": 4
}
run_wrapper(in_params)
```

일반적으로 production에서는 위 in_params를 모아서 wrapper를 호출할 것입니다. __main__에서는 TOML로부터 DB 정보 얻고, endpoint/key 하드코딩 (placeholder) 상태입니다. - **함수 내부 흐름 (다음에 설명)**, 이 함수 반환값은 None (완료 후 return nothing). - process_single_record(record: dict, in_params: dict): 개별 레코드(한 장/한 묶음 영수증)에 대한 처리 함수입니다 218. run_wrapper 내부에서 멀티스레드로 호출되며, 전처리->OCR->후처리->DB 저장 전체를 이 함수에서 처리합니다. - **인자:** record는 DB 조회결과 딕셔너리 (FIID, GUBUN, LINE_INDEX, ATTACH_FILE, FILE_PATH) 234. in_params는 위와 동일한 설정 딕셔너리. - **반환:** None (모든 처리는 부수효과로 파일/DB 저장). 오류 시도 None 반환이나 예외 발생과 무관하게 그냥 끝. - **로직:** 1. 로그 [시작] process_single_record - FIID=X, LINE_INDEX=Y (INFO) 238. 2. cropped_list = run_pre_pre_process(in_params, record) 실행 4. 전처리 결과 딕셔너리 리스트 획득. 3. loop for cropped in cropped_list: - If "RESULT_CODE" in cropped: 이는 전처리에서 오류/스킵된 항목. 로그 [SKIP] YOLO 오류 발생: {cropped} 출력하고 continue 239. 즉, 이 검증 실패나 오류 항목은 OCR/후처리/DB 저장 모두 건너뛴. - Else (정상 cropped): - ocr_result = run_azure_ocr(in_params, cropped) 호출 240. - If ocr_result.get("RESULT_CODE") == "AZURE_ERR": Azure OCR 실패의미 99. - 로그 [ERROR] Azure OCR 실패 → 오류 summary 저장 시도 WARNING 출력 241. - now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S") 현재시각. - error_summary = { ... } 딕셔너리 구성: FIID, LINE_INDEX, RECEIPT_INDEX, COMMON_YN, GUBUN from cropped/record, ATTACH_FILE from original record, 그리고 COUNTRY~BIZ_NO = None, RESULT_CODE = ocr_result.RESULT_CODE (AZURE_ERR), RESULT_MESSAGE = ocr_result.RESULT_MESSAGE (OCR 실패: ...), CREATE/UPDATE_DATE = now_str 242. 243. - error_result_path = os.path.join(in_params["post_json_dir"], f"fail_{error_summary['FIID']}_{error_summary['LINE_INDEX']}_{error_summary['RECEIPT_INDEX']}_post.json") 만들어 fail JSON 경로 지정 244. - 열어서 json.dump({"summary": error_summary, "items": []}, f, ensure_ascii=False, indent=2) 쓰기 245. - insert_postprocessed_result(error_result_path, in_params) 호출 하여 이 실패 JSON을 DB에 반영 246. - continue → 다음 cropped로 넘어감 (후처리 생략). - Else (OCR 성공): - json_path = os.path.join(in_params["ocr_json_dir"], f"{os.path.splitext(os.path.basename(cropped['file_path']))[0]}.ocr.json") 247. OCR JSON 경로를 전처리 산출 이미지 파일명 기반으로 결정 (doc_process 저장한 .ocr.json). - post_json_path = post_process_and_save(**in_params, "postprocess_output_dir": in_params["post_json_dir"], **cropped, "json_path": json_path, "ATTACH_FILE": record.get("ATTACH_FILE")) 호출 145. - in_params에 postprocess_output_dir 키를 추가/덮어써서 post_process 모듈이 사용할 디렉토리를 지정 (post_json_dir), - record 인자로 cropped 딕셔너리에 json_path와 ATTACH_FILE을 합친 새 dict 전달. - 이 함수가 post_json_path (결과 JSON 경로) 반환. - insert_postprocessed_result(post_json_path, in_params) DB 저장 호출 235. 4. try-except wraps the loop: except Exception as e: log [FATAL] 처리 중 오류 발생 - FIID=..., e (error level, exc_info=True) 248. This catches any unforeseen error in process_single_record (e.g., programming error or unhandled in called functions). 5. Finally log [종료] process_single_record - FIID=..., LINE_INDEX=... (INFO) 249. - process_single_record is not called by user directly, but by wrapper in threads. - **ThreadPool usage:** - max_workers = in_params.get("max_workers", 4) threads allocated 250. - with ThreadPoolExecutor(max_workers=max_workers) as executor: open thread pool 250. - futures = [executor.submit(process_single_record, rec, in_params) for rec in data_records] - all tasks submitted asynchronously 251. - for future in as_completed(futures): future.result() - wait and rethrow exceptions if any occurred inside thread (to surface errors) 224. - Actually, inside process_single_record, exceptions are caught internally for known issues, so future.result() likely either returns None or rethrows only if process_single_record itself had a bug not caught by its try (like memory error or logic bug). - After loop, log " 전체 파이프라인 완료" and "[종료] run_wrapper" (these logs are missing from snippet but logically they are in code at L125-L127) 224. - **main usage:** - It loads config from TOML, sets up DB engine and connection 225. - Logging config (file + console) set up 220. - logger = logging.getLogger("WRAPPER"), set to INFO 252 (So root and "WRAPPER" logger might be same? Actually they added 'WRAPPER' logger possibly so that in

db_master they use logging.getLogger("WRAPPER") to get same logger). - in_params is built with all keys including connection, target_date, azure keys, directories, etc ²²⁶ . - run_wrapper(in_params) called ²⁵³ . - close DB connection at end ²⁵⁴ .

7. 출력값 및 결과를 설명:

- **로그 출력:** wrapper.py 자체는 결과물을 파일이나 DB로 직접 쓰진 않지만, **로그와 최종 상태 코드**가 주요 출력입니다. - 로그: wrapper orchestrates the pipeline so it logs: - 총 조회된 레코드 수 (`logger.info(f"총 {len(data_records)}건 처리 시작")`) ²⁵⁰ . - 각 record 시작/종료 (via `process_single_record` logs). - pipeline complete message " 전체 파이프라인 완료" (with a check mark icon, present in code snippet) ²²⁴ . - Return: run_wrapper returns None. If no records, it returns early after logging no data. If exception happens outside threads, it might bubble or they catch at main? Actually no try around run_wrapper call in **main**. - **DB 저장 결과:** The tangible outputs of wrapper run are: - New rows in RPA_CCR_LINE_SUMM and RPA_CCR_LINE_ITEMS tables for each processed receipt. - If failures, those rows have RESULT_CODE != 200 and often no items inserted (or items empty). - Summaries with code 200 and items inserted count logged, or warnings for no items. - **Error handling output:** - If a thread had an unhandled exception (fatal, outside our safe except), `future.result()` would rethrow in main thread and potentially stop further threads waiting. But they catch within to avoid that for known issues. - The program as a whole prints nothing to stdout except logging. Possibly **main** would have printed something on exceptions if it had a try, but it doesn't. So any exceptions not caught in threads might bubble up and cause Python error stack. - If run in scheduling context, the exit code might indicate success (0) or failure (non-zero) if exception not caught. But in our design, most exceptions are caught and logged, not propagated, so likely wrapper exits normally with code 0 even if some receipts failed. - Potential improvement might be to set a return code or raise at end if any record failed, but not implemented. - **Example scenario outcome:** - Suppose 10 records, one has no attach_file and file_path, so skip quickly. - 9 process, out of those 9, 1 fails at OCR and 1 fails at postprocess. Then: - DB: 7 receipts with RESULT_CODE 200 in summary, and items inserted; 2 receipts with RESULT_CODE (one AZURE_ERR, one POST_ERR) and likely no items. - pipeline.log: info of 10 total, and details with error logs for those 2, but still "전체 파이프라인 완료" at end. - The process likely returns exit code 0 as exceptions were handled gracefully. - This ensures no manual intervention needed unless critical.

8. 코드 흐름 설명:

run_wrapper (main orchestrator): - Logs `[시작] run_wrapper` ²⁵⁵ . - Calls `data_records = query_data_by_date(in_params)` . DB fetch happens ²⁵⁵ . - If data_records empty: - Log "📧 처리할 데이터가 없습니다." (mailbox emoji, meaning no mail/data) ²⁵⁶ . - Return None ²⁵⁷ . - (Thus pipeline does nothing further if no tasks for that date) - If records exist: - Log `총 N건 처리 시작` ²⁵⁰ . - Determine max_workers. - Launch ThreadPoolExecutor with up to max_workers threads. - Submit all records to thread pool, calling `process_single_record(record, in_params)` for each. - Loop `as_completed` to retrieve futures: - `future.result()` ensures any exception in thread is raised in main. But inside `process_single_record`, they catch exceptions mostly. The only exception that might propagate is if something outside their try in that thread happened, e.g. our catch is around entire function except start and end logging, which covers nearly everything, but if logger usage error or something outside try we can't foresee, it might raise. - If a thread future had an exception, not caught inside, it will raise here. They don't catch around `future.result()` either, meaning it would bubble out of run_wrapper causing likely crash. But expecting minimal. - After loop (all futures done): - Log " 전체 파이프라인 완료" (checkmark sign to indicate success) and `[종료] run_wrapper` ²²⁴ . - Return None.

process_single_record (for each DB record): - Log `[시작] ...` including FIID and LINE_INDEX ²³⁸ . - try: - Call `run_pre_pre_process`: - If that call itself raises uncaught exception, it would go to except in this function. But `run_pre_pre_process` catches exceptions inside and returns [] on error, so likely no throw. - `cropped_list` result from `pre_process`. - Loop through each element of `cropped_list`: - If contains

RESULT_CODE: - This indicates either YOLO none (E001) or YOLO multiple (E002) or an error code that pre_process might add in future (or if pre_process code had an unhandled exception but still returned empty results, not likely). - Log skip: warns that YOLO error occurred with the content of that dict ²³⁹. - continue to next cropped (skip OCR and subsequent). - Else: - Call run_azure_ocr: - If it returns RESULT_CODE "AZURE_ERR": - This means Azure call failed. - Log warning: "Azure OCR 실패 → 오류 summary 저장 시도" ²⁴¹. - Create now_str. - Compose error_summary with all fields: - They notably include ATTACH_FILE from original record here ²⁵⁸, which is good since post_process expects it if it were to use, but here we directly do DB. - All relevant fields included, with code and message from OCR. - error_result_path: create path in `post_json_dir` (the final output folder) with prefix "fail_" and suffix "_post.json" to indicate failure postprocess JSON ²⁴⁴. - Dump {"summary": error_summary, "items": []} to that path ²⁴⁵. - Insert into DB via insert_postprocessed_result(error_result_path). - That will insert the summary (with RESULT_CODE AZURE_ERR) and no items (since empty list, loop inserts none, logs warning about no items). - continue to next cropped (skip actual post_process). - Else (OCR success): - Compute OCR JSON path of the file that doc_process saved (based on cropped file name) ²⁴⁷. - Call post_process_and_save with: - in_params copied but replaced/added "postprocess_output_dir" to use post_json_dir (the final output dir). - record = cropped + {"json_path": ..., "ATTACH_FILE": original attach_file} - This returns a post_json_path or error_path if failed (since post_process returns path always either success or fail). - Then call insert_postprocessed_result(post_json_path). - If post_process had failed internally, it returned fail JSON path with summary RESULT_CODE "POST_ERR". Then insert_postprocessed_result will insert that fail summary (items empty). So DB gets the failure logged, and pipeline continues normally. - If success, it inserts summary (RESULT_CODE 200) and items. - except Exception e: - If any unexpected error in above (like pre_process or azure function raising something not handled, or DB insert raising an exception not caught inside insert_postprocessed_result): - Log error [FATAL] with FIID and message (exc_info True to include stack) ²⁴⁸. - Then function ends (not rethrowing, just logging). - finally (implicit by function end or after try-except): - Log `[종료]` `process_single_record - FIID=...` ²⁴⁹ always executes (placed outside try but at function end, they didn't use finally but code flows to after except). - Return None.

So, per record: If any sub-step fails: - Pre-process: returns error result code dict or skip => we skip that image. - If all images in record skipped, basically nothing inserted for that record (maybe no summary at all). But initial record might have 2 images if attach and file path; if attach fails skip it but file path might exist and proceed. So partial for that record possible but normally either or. - OCR fails: we create a fail_post JSON and insert to DB as "AZURE_ERR". - Post fails: that function returns fail JSON path and we insert as "POST_ERR". - Insert fails: insert_postprocessed_result catches exceptions inside and logs error but does not raise (in db_master, they catch exceptions in insert and just error log, not raise) ²⁵⁹. So from wrapper perspective, insert failure won't throw, thus process_single_record won't hit its except. It will continue and log complete as if success. But data not in DB. - This is somewhat concerning: if DB insert fails (maybe DB down or constraint violation), our pipeline won't know except logs. They commit after each record though, so one failing doesn't rollback others, but they did each summary and items separate, so if items insertion fails, summary is already inserted (committed at end after all items, oh they commit after items inserted but inside same try). - Actually in insert_postprocessed_result, they commit after inserting items loop ²⁶⁰. So if items insert fails mid, exception triggers except in insert_postprocessed_result: - They log `[ERROR] DB 저장 실패: e` and print traceback ²⁵⁹, but they do not raise further, and they do not roll back either? Actually if exception thrown, likely the DB connection is in error or implicitly rolled back? They didn't explicitly rollback, maybe HANA autocommit is false until commit or it's autocommit each execute by default if autocommit mode on connection used. - They then log `[종료] insert_postprocessed_result` as if finished (this isn't in snippet but presumably symmetrical to start). - So wrapper sees no exception, logs process done, but DB might have partial or none for that record. This is a possible data inconsistency if summary inserted but items not. - Might want to handle that by either a transaction that encloses both, or if fail, remove summary row or mark it incomplete. Currently they'd have summary inserted (committed because commit after items; but

commit might not have executed if exception before commit line, so summary inserted but transaction uncommitted? Actually if autocommit off, nothing committed if exception before commit. If autocommit on by default then summary insertion was auto committed and items partial none commit maybe). - Without deeper look at SAP HANA driver default, I'd guess autocommit off by default since they call commit at end. So if items insertion fails, program didn't commit, and summary insert might be rolled back if no commit executed because connection will auto rollback on disconnect or after error if not handled? Actually with autocommit off, the connection is in an open transaction state after summary inserted. If an exception occurred, they didn't call commit, but they also didn't call rollback. If connection remains open to process more records, next commit might commit old partial results? or HANA might auto rollback upon next command after exception. - This is quite uncertain; ideally, on exception, one should call conn.rollback() to avoid partial commit. They didn't do that. - So maintenance wise, if DB errors should consider adding rollback or manage transactions per record differently. - If any top-level fatal (like code bug causing a NameError or memory error), that might propagate to thread future and then to run_wrapper as exception, which would not be caught (they didn't catch around run_wrapper in **main**). That would cause program to crash, pipeline.log not showing "완료". - But likely scenario is minimal.

9. 주요 함수 및 모듈 설명:

- `run_wrapper(in_params)`: 전체 파이프라인 실행 함수. **역할**: DB에서 대상 레코드 가져와 스레드풀로 병렬 처리, 각 레코드를 `process_single_record`로 처리, 모든 작업 완료 후 종료. **인자**: 설정 딕셔너리 (DB conn, target_date, Azure/YOLO config, directories, etc). **반환**: 없음.

내부 모듈 연계: 호출 순서 - `query_data_by_date` -> for each record: `process_single_record` -> inside that: `pre_process`, `doc_process`, `post_process`, DB insert.

특징: 다중 스레드 사용으로 효율 높임; 처리할 데이터 없으면 빠르게 종료.

- `process_single_record(record, in_params)`: 하나의 입력 레코드(한 사용자 제출 혹은 한 전표에 대한 영수증들)를 처리. **역할**: 전처리→OCR→후처리→DB 저장 일련의 호출을 수행하며, 단계별 오류를 감지하여 해당 레코드 처리 흐름을 조정. **인자**: record (dict, fields: FIID, GUBUN, LINE_INDEX, ATTACH_FILE, FILE_PATH), in_params (config). **반환**: 없음.
내부 기능:
 - **전처리 호출**: `run_pre_pre_process` -> 결과 영수증 이미지 리스트 확보 (또는 오류 리스트).
 - **전처리 결과 루프**: 여러 영수증 이미지 각각 OCR 처리. 전처리 단계에서 에러표시된 것은 skip.
 - **OCR 호출**: `run_azure_ocr` -> OCR dict 받음. 실패 시 RESULT_CODE 체크.
 - **OCR 실패 처리**: 실패시 error summary JSON 생성하여 파일 기록, DB insert 호출 (이상 로그 남기고 후처리 안 함).
 - **OCR 성공 처리**: OCR JSON 경로 설정 → 후처리 호출.
 - **후처리 호출**: `post_process_and_save` -> 결과 JSON (or fail JSON) 경로 받음.
 - **DB 저장 호출**: `insert_postprocessed_result` -> summary/items 테이블에 insert.
 - **오류 로깅**: try-catch로 예상치 못한 오류를 잡아 치명 로그 기록.
- **로그**: 시작/종료 로그, 단계별 경고/에러 로그. 이 함수는 파이프라인 각 모듈을 유기적으로 연결하며, 오류 상황에 따른 흐름 제어 (continue/skip)을 구현한 핵심 로직입니다.

• 데이터베이스 관련 함수 (in db_master):

- `query_data_by_date(in_params) -> list`: (Wrapper에서 사용) target_date에 해당하는 LDCOM_CARDFILE_LOG 레코드 목록 조회 ²³³. Input: in_params with target_date (optional, default to yesterday). Output: list of dict records (keys uppercase). Use in flow: provides records to wrapper for processing. If returns [] then stops.
- `insert_postprocessed_result(json_path, in_params) -> None`: (Wrapper에서 사용) 후처리 JSON 결과를 DB 두 테이블에 insert ³ ¹⁵⁷. Input: json_path of post JSON, in_params with

sqlalchemy_conn. Action: opens JSON, loads summary & items, executes two SQL INSERT statements (one for summary, loop for items) via text() & conn.execute, then commit. In wrapper flow: called for each processed receipt (success or fail JSON). Exception handling: inside catches exceptions and logs error, does not propagate. Hence, wrapper is oblivious to DB insert failure except logs.

- These are not in wrapper.py but heavily utilized. Documenting their usage context is essential as above.

• **로컬 모듈 import details:** wrapper uses `from pre_pre_process import run_pre_pre_process` but our file is `pre_process.py`. Possibly they named the file `pre_pre_process.py` in the codebase. We assume it's the same content as `pre_process.py`. For documentation, clarify if needed: The import suggests file name mismatch, which is likely a minor error or the file might have been originally named `pre_pre_process.py`. It's a detail not core to understanding, but if code is run as given, it would fail to import if file name is different. Possibly they changed file name but not import line.

- If truly an error, mention it as something to fix (rename import or file).

10. 로깅 및 예외 처리 전략:

- **로그 설정:** wrapper **main** sets up logging: - `logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s", handlers=[FileHandler(pipeline.log), StreamHandler()])` ⁹³ . - That means all loggers (root and named) with INFO or above go to pipeline.log and console. - Then `logger = logging.getLogger("WRAPPER")` and `logger.setLevel(logging.INFO)` ensures "WRAPPER" logger is at INFO (which it already was, might do if we wanted to allow debug sometimes). - This unified logging means we can see all steps from all modules in one log timeline. - **wrapper-specific logs:** - Starting run_wrapper: `[시작] run_wrapper` (Though snippet shows [113-L119] but not explicitly logging start; they'd likely do `logger.info` at entry but possibly omitted). - No data: "👉 처리할 데이터가 없습니다." INFO log with mailbox emoji to catch eye. - Starting processing count: logs number of records. - In threads: - `process_single_record` logs start/end for each record. - If any error not handled at step-level triggers `process_single_record` except: - It logs `[FATAL] 처리 중 오류 발생 - FIID=...` (error level, with stack). - 'FATAL' here meaning that entire record couldn't complete due to unforeseen error. This is serious as some steps may not complete for that record. - After thread tasks done, main thread logs "전체 파이프라인 완료" (with check mark). - Logging with emojis is used (, 👉) for user-friendliness in logs. - The logs from pre, doc, post, db, all integrated in time order in pipeline.log. E.g.:

```
2025-07-10 09:00:00 - INFO - [시작] run_wrapper
2025-07-10 09:00:00 - INFO - 총 5건 처리 시작
2025-07-10 09:00:00 - INFO - [시작] process_single_record - FIID=FIN123,
LINE_INDEX=1
2025-07-10 09:00:00 - INFO - [시작] run_pre_pre_process
...
2025-07-10 09:00:02 - INFO - [종료] run_pre_pre_process
2025-07-10 09:00:02 - INFO - [시작] run_azure_ocr
...
2025-07-10 09:00:03 - INFO - [종료] run_azure_ocr
2025-07-10 09:00:03 - INFO - [시작] post_process_and_save
...
2025-07-10 09:00:04 - INFO - [종료] post_process_and_save
```



```

2025-07-10 09:00:04 - INFO - [완료] DB 저장 - FIID=FIN123, RECEIPT_INDEX=1,
ITEMS_INSERTED=3
2025-07-10 09:00:04 - INFO - [종료] process_single_record - FIID=FIN123,
LINE_INDEX=1
...
2025-07-10 09:00:10 - INFO - 전체 파이프라인 완료
2025-07-10 09:00:10 - INFO - [종료] run_wrapper

```

(If errors occurred, they'd appear at relevant times.) - **예외 처리:** - The major try-except inside `process_single_record` covers everything except possibly DB insertion exceptions (caught inside `db_master`) and maybe subtle errors. - `run_wrapper` does not have its own try in code snippet, but **main** does not catch exceptions either. - But since all inner steps robustly catch their errors, the only likely unhandled exceptions might be: - DB connection lost or something outside of insert? `query_data_by_date` has try inside and returns [] on error so wrapper doesn't blow up (just logs and continues as 0 records). - DB insertion error: caught in `db_master` not raised, so wrapper misses it logically, just logs in `db_master`. - Memory issues or code mistakes: could propagate. - If any exception escapes a thread, `ThreadPool` as `_completed` will raise it when calling `future.result()`, but wrapper didn't catch around `future.result`, so it would bubble out of `run_wrapper`. That would mean `pipeline.log` might not show "완료". - Possibly they assume their excepts cover everything so not to bother catch at future. - For safety, one might wrap `future.result()` in try to log or something, but not done. - The **main** does not handle `run_wrapper` exceptions either, meaning if one not caught, script would crash with traceback. - However, since heavy exceptions (like DB connect fail) are handled in called functions (returns [], or error JSON etc), chance of crash is minimal. - Conclusion: wrapper tries to handle expected errors gracefully (skip or continue), and logs them rather than stopping the whole pipeline. Only unpredictable issues would stop it. - **오류 상황 대응 in design:** - If a record partially fails, they still commit partial success to DB for that record (like summary no items or error code). They do not attempt a retry in code, expecting maybe the human to re-run for that specific case if needed after investigating. - They skip entire record only if `pre_process` yields nothing to OCR at all (like if YOLO found nothing and attach file had only 1 image, then that record ends up not inserted in DB at all unless they consider skip vs fail differently). - Actually if YOLO none, we skip OCR and no DB insert for that receipt. The record goes silent (maybe no DB entry). - That might be an oversight, perhaps YOLO none should still produce a fail summary in DB? But they didn't do that: they just skip and not create anything for it. - For consistency, perhaps they rely on log and no DB entry signals missing data. - Or they treat YOLO none differently because maybe no receipt at all to log, in which case DB originally had record but no output. Possibly they'd want to mark it too. - Could be considered a minor issue - if YOLO fails, should we insert an `AZURE_ERR`-like record? (Though `AZURE` not used because didn't call `Azure`). - One approach: treat YOLO none as a "preprocess fail", could have added a code `E001` cause to DB. But they didn't code that. - So YOLO error `E002` or no detection `E001` just results in logs and skip. No DB flag. This might be something to adjust if needed (maybe maintenance note). - In contrast, `Azure` fail and post fail they do create fail JSON to insert. YOLO fail might have been considered at DB by adding a step similar. Could be oversight. - **로그 위치:** `pipeline.log` under logs folder. On each run, they open file in append mode "a". So multiple runs accumulate logs. Possibly they'd rotate externally or by schedule.

11. 테스트 방법 및 사용 예시:

- **개발 테스트 (main):** wrapper **main** expects a `Module_config_dex.toml` with DB details: - They open it and look for `[database]` or `[hana]` section with user, password, host, port ²³⁶. - After that, they create engine and connect. If config is correct and DB reachable, connection opens. If not, will raise (likely causing crash since not caught). - Then logging set, `in_params` prepared with example `target_date` "2025-07-09" (which presumably exists in `LDCOM_CARDFILE_LOG`). - They also fill `azure_endpoint` and key placeholders (which need actual values for `Azure` calls to succeed). - Directories: output,

preprocessed, etc (some not used by current code logic, as explained). - YOLO path to best.pt (this file must exist or YOLO call fails). - Then `run_wrapper(in_params)`. - After done, they close connection. - Running `python wrapper.py` thus tries to run entire pipeline. - If DB has data for `target_date` (or none, will log no data). - If it runs with actual YOLO and Azure connections, it will perform the full pipeline and populate DB. - This is integration test basically. It requires DB, YOLO model, and possibly Azure connectivity. Possibly in dev environment, they'd do smaller test with a known small input. - Alternatively, developer can simulate by replacing `run_pre_pre_process`, `run_azure_ocr` calls with dummy returns for test. But that isn't in code, they'd have to manually stub. - They provided **main** likely for actual usage (like run daily with config). - **Production usage:** Usually an external scheduler or script will call this wrapper (or import `run_wrapper` in bigger orchestrator). They might package it in an .exe or so, but code suggests it's run as Python script. - They have logs to monitor, and DB output for results. - If any error occurs not handled, they'd rely on log/traces to debug. - If they want to reprocess a particular date, they'd run wrapper with `target_date` set accordingly (maybe as CLI arg, but code doesn't parse CLI; they could modify config or wrapper code to accept date param). - **Monitoring common issues:** - If pipeline didn't run (no "파이프라인 완료" in logs), maybe crashed; need to see error logs. - Check DB row counts vs expected. - If needed to re-run a date, one can run `run_wrapper` again with that date if duplicates are handled or if process is idempotent. - But likely they'd not re-run same date to avoid duplicate DB entries, unless they'd clear previous and re-run. Possibly ID keys (FIID+INDEX) ensure duplicates not inserted if try, or they rely on uniqueness and insertion failing if duplicate (but they didn't handle unique constraints error). - Possibly FIID, LINE_INDEX, RECEIPT_INDEX combination might be primary key in summ table? If so, second run will error on duplicate key, which `insert_postprocessed_result` will catch and log error without raising, and pipeline moves on leaving older entry intact. That means re-run doesn't override, and error notifies duplicates exist, needing manual handling (like either change code to update if exists). - They removed Oracle code mentions, focusing on HANA. So ensure hdbcli installed and working in environment.

• 자주 발생하는 오류 및 해결 (wrapper level):

- DB Connection Failure: If `engine.connect()` fails (e.g., wrong credentials, DB down), the script will error out at **main** (traceback printed, no log because logging config is set after reading config but before connect? Actually they set up logging after engine created and connected. So if connect fails, it never sets up logging, so error will show on console only). To solve: check config, network, credentials. Possibly wrap connect in try and log friendly error.
- No Data to process: Not an error, just logs none and exits. Check if `target_date` is correct format and present in DB data. If needed to process older data, adjust `target_date` input.
- Threading Issues: If `max_workers` too high, many Azure calls could saturate network or cause Form Recognizer rate limits. In such case, Azure may start failing with throttle errors or degrade. If so, reduce `max_workers` or catch throttle and wait (not implemented).
- Memory Issues: If very large number of records (e.g., hundreds) processed concurrently, memory might spike due to multiple YOLO models in memory or many images loaded. Could set `max_workers` lower to throttle memory usage.
- One record consistently fails (like YOLO none or OCR fail): It will log as skip or error. If due to poor image, might need human to intervene, e.g., attach an alternate image, or adjust YOLO model. This pipeline by design doesn't attempt image enhancements or second tries except logging.
- Partial DB insert fails: As discussed, if DB insert had issues (like table constraints or connection lost mid-run but regained), some data might not be inserted. The code logs an error in `insert_postprocessed_result`. That record then ends processing normally in wrapper (no exception). The pipeline completeness log still prints. So one must inspect pipeline.log for any ERROR lines or DB warnings to catch such problems. Possibly implement a summary at end counting errors vs successes would be helpful (not done).

12. 유지보수 및 확장 고려사항:

- **에러 처리 일관성 & Robustness:** - 고려 추가: YOLO 탐지 실패(E001/E002)도 DB에 기록하는 방안. 현재 skip이라 DB에 안 남아, 나중에 분석 시 그 record는 아예 없는 것처럼 될 수 있음. 유지보수 시 `process_single_record` 에서 `"RESULT_CODE" in cropped` 발견하면, 대신 fail summary JSON 만들어 insert하는 로직을 추가하면 데이터 손실 없이 실패 원인 추적 가능. (e.g., treat E001/E002 similar to Azure error: set code "PRE_ERR"). - Insert 함수의 오류 처리를 wrapper에서 인지할 방법 (maybe set a flag or raise exception on DB failure). Currently pipeline could finish "성공" even though DB inserts might have failed. An enhancement could be to have `insert_postprocessed_result` raise on DB issues, and wrapper catch to handle or at least log clearly with FATAL. Or gather a list of failed DB inserts to notify at end. - Also, if needed, unify how exceptions propagate. For example, they intended `post_process` to raise, but changed to return. Could revisit and possibly unify by always returning a status rather than raising, for consistency. - **Configuration management:** - Instead of embedding config reading in `main`, consider CLI arguments for `target_date`, config file path, etc, for flexibility. Right now `target_date` can only be changed by editing `in_params` before `run_wrapper` (or config file default logic in `query_data_by_date` for yesterday). - Could add an argument parsing to override `target_date` easily (maybe parse `sys.argv`). - **Scalability:** - If volume of records per day grows, ensure `max_workers` size appropriate, or consider using `ProcessPoolExecutor` if CPU-bound (YOLO might become CPU-bound if heavy, though likely it uses GPU or at least PyTorch that might release GIL but still CPU heavy if no GPU). - Or pipeline segmentation: e.g. first fetch 100 records, process, then next 100 if memory is an issue. Now it fetches all for date at once. Could saturate memory if thousands of tasks in futures at once. Possibly limit chunk size. - **Alternate DB or multi-DB support:** Code has hints of Oracle support removal, but if needed to support Oracle or others, one could implement alternative query/insert functions or adapt connection string accordingly. For now, focusing on HANA. - **Logging improvement:** Possibly integrate log rotation for `pipeline.log` to avoid infinite growth. Also, unify logger naming usage (some modules used specific loggers like "AZURE_OCR", "PRE_PRE_PROCESS", but they didn't attach handlers specifically. Could either just use root logger or properly set propagate flags. In current state it's fine but a bit inconsistent). - **Enhancements to data pipeline:** - Possibly incorporate an acknowledgement or flag update in source DB after processing a record to mark it processed (to avoid reprocessing same records). Right now, every run with same date will attempt all records again. If repeated daily, maybe they intend one date's data processed once. But if need partial re-run, they'd specify `target_date`. - If incremental processing needed (like new records in near-real time), they'd need to adapt to not process duplicates or schedule differently (currently per date). - **Graceful shutdown:** If script receives interrupt (SIGINT), threads might not all be joined properly, depending on Python's handling. No special handling now. Usually not crucial unless user stops it mid-run. - **Extensibility to other document types:** The name "CARDFILE_LOG" and table names suggest corporate credit card receipt scanning. If extended to invoices or other docs, pipeline can be reused by swapping Azure model (like `prebuilt-invoice`) and adjusting `post_process` to invoice fields, YOLO maybe not needed if direct image. Possibly you could generalize wrapper to handle multiple document types by param. - **Parallelization:** - Right now per record parallel; note if one record has multiple receipts images, those are processed sequentially within that thread. If a record had 10 receipts (maybe `FILE_PATH` scanning multiple receipts), they process them one by one in that thread. Possibly parallelizing inside a single record is complex but maybe not needed if typically it's 1 or few receipts per record. - If each record only had one image usually, then `threads=4` means 4 images at a time. If performance needed, could up threads but watch Azure rate limit. - **Resource cleanup:** - They open one DB connection and share among threads. If any thread hits an error causing connection broken, others might fail too. Possibly better to have separate connections per thread (maybe by passing engine and doing `engine.connect()` inside each thread). But overhead for each maybe fine. Current approach is simpler but relies on driver thread-safety and one connection concurrency. - After finishing, they close connection. If an exception aborted `run_wrapper` prematurely, they might not reach close call. Possibly put `conn.close()` in a finally in `main` or use `with engine.connect()` as `conn` for auto closing. - **User notifications:** If desired, pipeline could email or alert if any errors (none in code, would rely on manual log check). If expanding, might integrate an alert if certain threshold of fails.

Overall, wrapper is crucial for integration and monitoring; modifications should be carefully tested end-to-end due to concurrency and external interactions.

1 97 109 110 119 120 186 190 Receipt data extraction - Document Intelligence - Azure AI services | Microsoft Learn

<https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/prebuilt/receipt?view=doc-intel-4.0.0>

2 3 157 182 183 213 222 231 233 234 259 260 db_master.py

file:///file-K84zghyNnvkUsGgHFREnU1

4 5 93 98 99 100 121 122 129 145 147 148 150 156 161 163 218 219 220 221 223 224 225 226 227 228 229 230 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 wrapper.py

file:///file-6sMGPV11f5v9yhGpFgGWrL

6 7 8 9 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 94 95

pre_process.py

file:///file-FskSSgPW8AVnX3sfAUjKip

10 26 96 Object Detection - Ultralytics YOLO Docs

<https://docs.ultralytics.com/tasks/detect/>

101 102 103 105 106 107 108 111 112 113 114 115 116 117 118 123 124 125 126 127 130 131 132 133 134 135 136 137 138 139 140 141 doc_process.py

file:///file-HLmAwrs4bSWHEh4FGCQosM

104 azure.ai.formrecognizer.DocumentAnalysisClient class | Microsoft Learn

<https://learn.microsoft.com/en-us/python/api/azure-ai-formrecognizer/azure.ai.formrecognizer.documentanalysisclient?view=azure-python>

128 142 143 144 146 149 151 152 153 154 155 158 159 160 162 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 184 185 187 188 189 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 214 215 216 217 post_process.py

file:///file-YEGAEsKyop27uiWnpsnGVZ

232 SQLAlchemy Dialect for SAP HANA - GitHub

<https://github.com/SAP/sqlalchemy-hana>