Gmail Michael Joyce <michaeljoyce217@gmail.com>

# diana

**Joyce, Mike** <Mike.Joyce@mercy.net> Tue, Feb 17, 2026 at 7:26 AM
To: Michael Joyce <michaeljoyce217@gmail.com>

Featurization_inference.py

```
# data/featurization_inference.py
# Acute Respiratory Failure Appeal Engine - Per-Case Data Preparation
#
# PER-CASE FEATURIZATION: Prepares all data needed for a single appeal:
# 1. Parse denial PDF → Extract text, account ID, payor, DRGs
# 2. Query clinical notes → ALL notes from 14 types from Epic Clarity
# 3. Extract clinical notes → LLM summarization of long notes
# 4. Query structured data → Labs, vitals, meds, diagnoses
# 5. Extract structured data → LLM summarization for acute respiratory failure evidence
# 6. Conflict detection → Flag discrepancies between notes and structured data
# 7. Write to case tables → Ready for inference.py to read
#
# OUTPUT: Case tables in Unity Catalog for inference.py
#
# Run on Databricks Runtime 15.4 LTS ML

# ============================================================================
# CELL 0: Install Dependencies (run this cell FIRST, then restart)
# ============================================================================
# IMPORTANT: Run this cell by itself, then run the rest of the notebook.
# After restart, the packages persist for the cluster session.
```

```
#
# Uncomment and run ONCE per cluster session:
# %pip install azure-ai-documentintelligence==1.0.2 openai python-docx
# dbutils.library.restartPython()


# ============================================================================
# CELL 1: Configuration
# ============================================================================
import os
import re
import json
import tiktoken
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor, as_completed
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, ArrayType, FloatType, BooleanType,
TimestampType


spark = SparkSession.builder.getOrCreate()


# ----------------------------------------------------------------------
# INPUT: Set the denial PDF to process
# ----------------------------------------------------------------------
# DENIAL_PDF_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
sample_denial_letters/4.4.25 drg DL SE UHC EXL EM (1).pdf" # Penny
# DENIAL_PDF_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
sample_denial_letters/KINWORTHY 060425 em (1).pdf" # Kinworthy
# DENIAL_PDF_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
sample_denial_letters/pp 10.21.25 drg RL Spfd UHC MedReview portal (2) (1).pdf" # Bell
DENIAL_PDF_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/sample_
denial_letters/Ellis, Albert RL (1).pdf" # Ellis
# DENIAL_PDF_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
```

sample_denial_letters/spfd Humana 020525 Sadler prepay RL pnc (1).pdf" # Saddler

```
# If account ID is known (production), set it here. Otherwise LLM will extract from PDF.
KNOWN_ACCOUNT_ID = None  # e.g., "12345678" or None to extract

# ----------------------------------------------------------------------------
# Processing Configuration
# ----------------------------------------------------------------------------
NOTE_EXTRACTION_THRESHOLD = 8000  # Chars - notes longer than this get LLM extraction
EMBEDDING_MODEL = "text-embedding-ada-002"

# ----------------------------------------------------------------------------
# Environment Setup
# ----------------------------------------------------------------------------
# NOTE: Data lives in prod catalog, but we write to our environment's catalog.
# This is intentional - we query from prod but can only write to our own env.
trgt_cat = os.environ.get('trgt_cat', 'dev')
spark.sql('USE CATALOG prod;')

# ----------------------------------------------------------------------------
# Output Tables (case data for inference.py to read)
# ----------------------------------------------------------------------------
CASE_DENIAL_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_denial"
CASE_CLINICAL_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_clinical"
CASE_STRUCTURED_SUMMARY_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_structured_summary"
CASE_CONFLICTS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_conflicts"

# Structured data tables (intermediate)
LABS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_labs"
VITALS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_vitals"
```

```
MEDS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_meds"
DIAGNOSIS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_diagnoses"
MERGED_TABLE = f"{trgt_cat}.fin_ds.creamsicle_structured_timeline"

print(f"Denial PDF: {DENIAL_PDF_PATH}")
print(f"Catalog: {trgt_cat}")

# ============================================================================
# CELL 2: Azure Credentials and Clients
# ============================================================================
from openai import AzureOpenAI
from azure.ai.documentintelligence import DocumentIntelligenceClient
from azure.ai.documentintelligence.models import AnalyzeDocumentRequest
from azure.core.credentials import AzureKeyCredential

# Load credentials
AZURE_OPENAI_KEY = dbutils.secrets.get(scope='idp_etl', key='az-openai-key1')
AZURE_OPENAI_ENDPOINT = dbutils.secrets.get(scope='idp_etl', key='az-openai-base')
AZURE_DOC_INTEL_KEY = dbutils.secrets.get(scope='idp_etl', key='az-aidcmntintel-key1')
AZURE_DOC_INTEL_ENDPOINT = dbutils.secrets.get(scope='idp_etl', key='az-aidcmntintel-endpoint')

# Initialize clients
openai_client = AzureOpenAI(
    api_key=AZURE_OPENAI_KEY,
    azure_endpoint=AZURE_OPENAI_ENDPOINT,
    api_version="2024-10-21"
)

doc_intel_client = DocumentIntelligenceClient(
```

```
    endpoint=AZURE_DOC_INTEL_ENDPOINT,
    credential=AzureKeyCredential(AZURE_DOC_INTEL_KEY)
)

print("Azure clients initialized")

# ============================================================================
# CELL 3: PDF Parsing and Denial Extraction Functions
# ============================================================================

def extract_text_from_pdf(file_path):
    """Extract text from PDF using Document Intelligence."""
    with open(file_path, 'rb') as f:
        document_bytes = f.read()

    poller = doc_intel_client.begin_analyze_document(
        model_id="prebuilt-layout",
        body=AnalyzeDocumentRequest(bytes_source=document_bytes),
    )
    result = poller.result()

    pages_text = []
    for page in result.pages:
        page_lines = [line.content for line in page.lines]
        pages_text.append("\n".join(page_lines))

    return pages_text

def generate_embedding(text):
    """
```

```
    Generate embedding vector for text using Azure OpenAI.
    Returns 1536-dimensional vector.
    """
    # Use the tokenizer for the specific model.
    # For text-embedding-ada-002, the encoding is 'cl100k_base'
    encoding = tiktoken.get_encoding("cl100k_base")

    # Define the model's token limit: 8192
    MODEL_TOKEN_LIMIT = 8192

    # Encode the text to get tokens
    tokens = encoding.encode(text)

    if len(tokens) > MODEL_TOKEN_LIMIT:
        print(f"  Warning: Text truncated from {len(tokens)} to {MODEL_TOKEN_
LIMIT} tokens for embedding")
        # Truncate tokens
        tokens = tokens[:MODEL_TOKEN_LIMIT]
        # Decode tokens back to text for the API call
        text = encoding.decode(tokens)

    response = openai_client.embeddings.create(
        model=EMBEDDING_MODEL,
        input=text
    )
    return response.data[0].embedding

DENIAL_PARSER_PROMPT = '''Extract key information from this denial letter.

# Denial Letter Text
```

{denial_text}

# Instructions

Find:

1. HOSPITAL ACCOUNT ID - starts with "H" followed by digits (e.g., H1234567890)

2. INSURANCE PAYOR - the company that sent this denial

3. ORIGINAL DRG - the DRG code the hospital billed (e.g., 189). ONLY if explicitly stated as a number.

4. PROPOSED DRG - the DRG the payor wants to change it to (e.g., 190). ONLY if explicitly stated as a number.

5. IS ARF RELATED - does this denial involve acute respiratory failure (ARF), chronic respiratory failure, acute-on-chronic respiratory failure, hypoxemic/hypoxic respiratory failure, hypercapnic respiratory failure, or related terms such as hypoxia or hypercapnia?

CRITICAL: For DRG codes, return NONE unless you see an actual 3-digit DRG number explicitly written in the letter.

Do NOT guess or infer DRG codes. If the letter just says "adjusted" or "changed" without specific numbers, return NONE.

Return ONLY these lines (no JSON):

ACCOUNT_ID: [H-prefixed number or NONE]

PAYOR: [insurance company name]

ORIGINAL_DRG: [3-digit code ONLY if explicitly stated, otherwise NONE]

PROPOSED_DRG: [3-digit code ONLY if explicitly stated, otherwise NONE]

IS_ARF: [YES or NO]"'

```
def transform_hsp_account_id(raw_id):
    """Transform HSP_ACCOUNT_ID from denial letter format to Clarity format."""
    if not raw_id:
        return None

    cleaned = str(raw_id).strip()
```

```
    if not cleaned.upper().startswith('H'):
        print(f"  Skipping non-H account ID: {cleaned}")
        return None

    cleaned = cleaned[1:]  # Remove H prefix

    if len(cleaned) > 2:
        cleaned = cleaned[:-2]  # Remove last 2 digits
    else:
        return None

    if cleaned and cleaned.isdigit():
        return cleaned

    return None

def extract_denial_info_llm(text):
    """Use LLM to extract denial info from text."""
    try:
        response = openai_client.chat.completions.create(
            model="gpt-4.1",
            messages=[
                {"role": "system", "content": "Extract information accurately. Return only the requested format."},
                {"role": "user", "content": DENIAL_PARSER_PROMPT.format(denial_text=text[:15000])}
            ],
            temperature=0,
            max_tokens=200
        )
```

```
raw = response.choices[0].message.content.strip()

result = {
    "hsp_account_id": None,
    "payor": "Unknown",
    "original_drg": None,
    "proposed_drg": None,
    "is_arf": False
}

for line in raw.split("\n"):
    line = line.strip()
    if ":" not in line:
        continue
    key, value = line.split(":", 1)
    key = key.strip().upper()
    value = value.strip()

    if key == "ACCOUNT_ID":
        if value and value.upper() != "NONE":
            result["hsp_account_id"] = transform_hsp_account_id(value)
    elif key == "PAYOR":
        result["payor"] = value if value else "Unknown"
    elif key == "ORIGINAL_DRG":
        if value and value.upper() != "NONE":
            result["original_drg"] = value
    elif key == "PROPOSED_DRG":
        if value and value.upper() != "NONE":
            result["proposed_drg"] = value
```

```
            elif key == "IS_ARF":
                result["is_arf"] = value.upper() == "YES"

        return result

    except Exception as e:
        print(f"  LLM extraction error: {e}")
        return {
            "hsp_account_id": None,
            "payor": "Unknown",
            "original_drg": None,
            "proposed_drg": None,
            "is_arf": False
        }

print("PDF parsing functions loaded")

# ============================================================================
# CELL 4: Clinical Notes Query Functions
# ============================================================================

NOTE_TYPE_MAP = {
    'H&P': ('hp_note_id', 'hp_note_csn_id', 'hp_note_text'),
    'Discharge Summary': ('discharge_summary_note_id', 'discharge_note_csn_id', '
discharge_summary_text'),
    'Procedures': ('procedures_note_id', 'procedures_note_csn_id', 'procedures_note_text'),
    'Progress Notes': ('progress_note_id', 'progress_note_csn_id', 'progress_note_text'),
    'Consults': ('consult_note_id', 'consult_note_csn_id', 'consult_note_text'),
    'ED Notes': ('ed_notes_id', 'ed_notes_csn_id', 'ed_notes_text'),
    'Initial Assessments': ('initial_assessment_id', 'initial_assessment_csn_id', 'initial_assessment_text'),
```

'ED Triage Notes': ('ed_triage_id', 'ed_triage_csn_id', 'ed_triage_text'),

'ED Provider Notes': ('ed_provider_note_id', 'ed_provider_note_csn_id', 'ed_provider_note_text'),

'Addendum Note': ('addendum_note_id', 'addendum_note_csn_id', 'addendum_note_text'),

'Hospital Course': ('hospital_course_id', 'hospital_course_csn_id', 'hospital_course_text'),

'Subjective & Objective': ('subjective_objective_id', 'subjective_objective_csn_id', 'subjective_objective_text'),

'Assessment & Plan Note': ('assessment_plan_id', 'assessment_plan_csn_id', ' assessment_plan_text'),

'Nursing Note': ('nursing_note_id', 'nursing_note_csn_id', 'nursing_note_text'),

'Code Documentation': ('code_documentation_id', 'code_documentation_csn_id', 'code_ documentation_text'),

'Anesthesia Preprocedure Evaluation': ('anesthesia_preprocedure_id', 'anesthesia_ preprocedure_csn_id', 'anesthesia_preprocedure_text'),

'Anesthesia Postprocedure Evaluation': ('anesthesia_postprocedure_id', ' anesthesia_postprocedure_csn_id', 'anesthesia_postprocedure_text'),

'H&P (View-Only)': ('hp_view_only_id', 'hp_view_only_csn_id', 'hp_view_only_text'),

'Interval H&P Note': ('interval_hp_note_id', 'interval_hp_note_csn_id', 'interval_hp_note_text'),

'Anesthesia Procedure Notes': ('anesthesia_procedure_id', 'anesthesia_procedure_csn_id', ' anesthesia_procedure_text'),

'L&D Delivery Note': ('ld_delivery_note_id', 'ld_delivery_note_csn_id', 'ld_delivery_note_text'),

'Pre-Procedure Assessment': ('pre_procedure_assessment_id', 'pre_procedure_ assessment_csn_id', 'pre_procedure_assessment_text'),

'Inpatient Medication Chart': ('inpatient_med_chart_id', 'inpatient_med_chart_csn_ id', 'inpatient_med_chart_text'),

'Hospice': ('hospice_id', 'hospice_csn_id', 'hospice_text'),

'Hospice Plan of Care': ('hospice_plan_of_care_id', 'hospice_plan_of_care_csn_id', 'hospice_plan_of_care_text'),

'Hospice Non-Covered': ('hospice_non_covered_id', 'hospice_non_covered_csn_id', ' hospice_non_covered_text'),

'OR Post-Procedure Note': ('or_post_procedure_id', 'or_post_procedure_csn_id', 'or_ post_procedure_text'),

'Peri-OP': ('peri_op_id', 'peri_op_csn_id', 'peri_op_text'),

'Treatment Plan': ('treatment_plan_id', 'treatment_plan_csn_id', 'treatment_plan_text'),

'Delivery': ('delivery_id', 'delivery_csn_id', 'delivery_text'),

```
    'Brief Op Note': ('brief_op_note_id', 'brief_op_note_csn_id', 'brief_op_note_text'),
    'Operative Report': ('operative_report_id', 'operative_report_csn_id', 'operative_report_text'),
    'Scanned Form': ('scanned_form_id', 'scanned_form_csn_id', 'scanned_form_text'),
    'Therapy Evaluation': ('therapy_evaluation_id', 'therapy_evaluation_csn_id', 'therapy_evaluation_text'),
    'Therapy Treatment': ('therapy_treatment_id', 'therapy_treatment_csn_id', 'therapy_treatment_text'),
    'Therapy Discharge': ('therapy_discharge_id', 'therapy_discharge_csn_id', 'therapy_discharge_text'),
    'Therapy Progress Note': ('therapy_progress_note_id', 'therapy_progress_note_csn_id',
 'therapy_progress_note_text'),
    'Wound Care': ('wound_care_id', 'wound_care_csn_id', 'wound_care_text'),
    'Anesthesia Post Evaluation': ('anesthesia_post_eval_id', 'anesthesia_post_eval_csn_id', '
anesthesia_post_eval_text'),
    'Query': ('query_id', 'query_csn_id', 'query_text'),
    'Anesthesia Post-Op Follow-up Note': ('anesthesia_postop_followup_id', '
anesthesia_postop_followup_csn_id', 'anesthesia_postop_followup_text'),
    'Anesthesia Handoff': ('anesthesia_handoff_id', 'anesthesia_handoff_csn_id', '
anesthesia_handoff_text'),
    'Anesthesia PAT Evaluation': ('anesthesia_pat_eval_id', 'anesthesia_pat_
eval_csn_id', 'anesthesia_pat_eval_text'),
    'Anesthesiology': ('anesthesiology_id', 'anesthesiology_csn_id', 'anesthesiology_text'),
    'ED Attestation Note': ('ed_attestation_note_id', 'ed_attestation_note_csn_id', 'ed_
attestation_note_text'),
    'ED Procedure Note': ('ed_procedure_note_id', 'ed_procedure_note_csn_id', 'ed_
procedure_note_text'),
    'ED Re-evaluation Note': ('ed_reeval_note_id', 'ed_reeval_note_csn_id', 'ed_reeval_note_text'),
    'CDU Provider Note': ('cdu_provider_note_id', 'cdu_provider_note_csn_id', 'cdu_provider_note_text'),
}


def query_clarity_for_account(account_id):
    """
    Query Clarity for clinical notes for a single account.
    Returns dict with patient info and ALL notes for each of 48 clinical note types.
    Notes are concatenated chronologically with timestamps.
    """
```

```
    print(f"  Querying Clarity for account {account_id}...")

    # Query 1: Get patient info
    patient_query = f"""
    SELECT ha.hsp_account_id, patient.pat_id, patient.pat_mrn_id,
        CONCAT(patient.pat_first_name, ' ', patient.pat_last_name) AS formatted_name,
        DATE_FORMAT(patient.birth_date, 'MM/dd/yyyy') AS formatted_birthdate,
        'Mercy Hospital' AS facility_name,
        DATEDIFF(ha.disch_date_time, DATE(ha.adm_date_time)) AS number_of_midnights,
        CONCAT(DATE_FORMAT(ha.adm_date_time, 'MM/dd/yyyy'), '-', DATE_FORMAT(ha.disch_
date_time, 'MM/dd/yyyy')) AS formatted_date_of_service
    FROM clarity_cur.hsp_account_enh ha
    INNER JOIN clarity_cur.patient_enh patient ON ha.pat_id = patient.pat_id
    WHERE ha.hsp_account_id = '{account_id}'
    """

    patient_rows = spark.sql(patient_query).collect()
    if not patient_rows:
        print(f"  WARNING: No patient found in Clarity for account {account_id}")
        return None

    clinical_data = patient_rows[0].asDict()
    print(f"  Found patient: {clinical_data.get('formatted_name', 'Unknown')}")

    # Query 2: Get ALL notes for this account (all encounters associated with this HSP_ACCOUNT_ID)
    notes_query = f"""
    SELECT
        nte.ip_note_type,
        nte.note_id,
        nte.note_csn_id,
```

```
        nte.contact_date,
        nte.ent_inst_local_dttm,
        CONCAT_WS('\\n', SORT_ARRAY(COLLECT_LIST(STRUCT(nte.line, nte.note_text))).note_
text) AS note_text
    FROM clarity_cur.pat_enc_hsp_har_enh peh
    INNER JOIN clarity_cur.hno_note_text_enh nte USING(pat_enc_csn_id)
    WHERE peh.hsp_account_id = '{account_id}'
        AND nte.ip_note_type IN (
            'Progress Notes', 'Consults', 'Procedures', 'H&P', 'Discharge Summary',
            'ED Notes', 'Initial Assessments', 'ED Triage Notes', 'ED Provider Notes',
            'Addendum Note', 'Hospital Course', 'Subjective & Objective',
            'Assessment & Plan Note', 'Nursing Note', 'Code Documentation',
            'Anesthesia Preprocedure Evaluation', 'Anesthesia Postprocedure Evaluation',
            'H&P (View-Only)', 'Interval H&P Note', 'Anesthesia Procedure Notes',
            'L&D Delivery Note', 'Pre-Procedure Assessment', 'Inpatient Medication Chart',
            'Hospice', 'Hospice Plan of Care', 'Hospice Non-Covered',
            'OR Post-Procedure Note', 'Peri-OP', 'Treatment Plan', 'Delivery',
            'Brief Op Note', 'Operative Report', 'Scanned Form',
            'Therapy Evaluation', 'Therapy Treatment',
            'Therapy Discharge', 'Therapy Progress Note', 'Wound Care',
            'Anesthesia Post Evaluation', 'Query', 'Anesthesia Post-Op Follow-up Note',
            'Anesthesia Handoff', 'Anesthesia PAT Evaluation', 'Anesthesiology',
            'ED Attestation Note', 'ED Procedure Note', 'ED Re-evaluation Note',
            'CDU Provider Note'
        )
    GROUP BY nte.ip_note_type, nte.note_id, nte.note_csn_id, nte.contact_date, nte.ent_inst_local_dttm
    ORDER BY nte.contact_date ASC, nte.ent_inst_local_dttm ASC
    """

    print(f"  Fetching clinical notes...")
```

```
notes_rows = spark.sql(notes_query).collect()
print(f"  Retrieved {len(notes_rows)} total notes")

# Group ALL notes by type (not just most recent)
notes_by_type = {}
for row in notes_rows:
    note_type = row['ip_note_type']
    if note_type not in notes_by_type:
        notes_by_type[note_type] = []
    notes_by_type[note_type].append(row)

# Report counts per type
for note_type, notes in notes_by_type.items():
    print(f"    {note_type}: {len(notes)} notes")

# Concatenate all notes of each type with timestamps
for note_type, (id_col, csn_col, text_col) in NOTE_TYPE_MAP.items():
    if note_type in notes_by_type:
        notes_list = notes_by_type[note_type]
        # Concatenate all notes with timestamps
        combined_text_parts = []
        note_ids = []
        csn_ids = []
        for row in notes_list:
            timestamp = row['ent_inst_local_dttm'] or row['contact_date'] or 'Unknown time'
            note_text = row['note_text'] if row['note_text'] else ''
            if note_text:
                combined_text_parts.append(f"[{timestamp}]\n{note_text}")
            if row['note_id']:
                note_ids.append(str(row['note_id']))
```

```
            if row['note_csn_id']:
                csn_ids.append(str(row['note_csn_id']))

        clinical_data[id_col] = ', '.join(note_ids) if note_ids else 'no id available'
        clinical_data[csn_col] = ', '.join(csn_ids) if csn_ids else 'no id available'
        clinical_data[text_col] = '\n\n---\n\n'.join(combined_text_parts) if combined_text_parts else 'No Note Available'
    else:
        clinical_data[id_col] = 'no id available'
        clinical_data[csn_col] = 'no id available'
        clinical_data[text_col] = 'No Note Available'

  return clinical_data

print("Clinical notes query functions loaded")

# ============================================================================
# CELL 5: Clinical Note Extraction (LLM)
# ============================================================================

NOTE_EXTRACTION_PROMPT = '''Extract clinically relevant information from this {note_type}.

CRITICAL: For EVERY piece of information you extract, include the associated date/time if available.
Format timestamps consistently as: MM/DD/YYYY HH:MM or MM/DD/YYYY if time not available.

# Clinical Note
{note_text}

# What to Extract (with timestamps)
```

## Oxygenation & Ventilation Parameters (PRIORITY - extract ALL available)

- PaO2 (partial pressure of oxygen)
- SaO2 (oxygen saturation from ABG)
- SpO2 (oxygen saturation via pulse oximetry)
- PaCO2 (partial pressure of carbon dioxide)
- FiO2 (fraction of inspired oxygen)
- P/F ratio (PaO2/FiO2) - if documented or calculable
- pH from ABG or VBG
- Oxygen delivery method and flow rate (nasal cannula, Venturi mask, non-rebreather, CPAP, BiPAP, mechanical ventilation)
- Ventilator settings if applicable (mode, tidal volume, PEEP, rate)

## Respiratory Assessment Findings

- Respiratory rate
- Work of breathing (accessory muscle use, retractions, nasal flaring)
- Breath sounds (wheezing, crackles, diminished, absent)
- Dyspnea severity or description
- Signs of respiratory distress (tachypnea > 20/min, bradypnea < 10/min, grunting, labored breathing)
- Mental status changes (confusion, somnolence, anxiety, restlessness)
- Cyanosis
- Documentation of "respiratory distress" or "acute respiratory distress"

## Baseline Respiratory Status

- Home oxygen use (flow rate, duration)
- Known chronic respiratory conditions (COPD, interstitial lung disease, etc.)
- Baseline ABG values if documented
- Prior intubation or ventilator dependence

## Laboratory & Diagnostic Findings

- ABG results (pH, PaO2, PaCO2, HCO3, SaO2) - with FiO2 at time of draw

- VBG results if ABG not available

- Chest X-ray or CT findings

- BNP/NT-proBNP (to differentiate cardiac vs. pulmonary etiology)

## Clinical Events

- Escalation of oxygen therapy (timing and levels)

- Initiation of CPAP, BiPAP, or mechanical ventilation

- ICU admission/transfer

- Respiratory therapy interventions

- Intubation or extubation

- Physician assessments mentioning respiratory failure, respiratory distress, hypoxia, hypercapnia, or respiratory insufficiency

## Underlying Etiology Indicators

- Pneumonia, COPD exacerbation, asthma, pulmonary edema, PE, aspiration

- Infection markers (WBC, procalcitonin, cultures)

- Cardiac findings if relevant to respiratory status

# Output Format

Return a structured summary with timestamps. Be thorough but concise.'''

```
def extract_clinical_data(note_text, note_type):
    """Extract clinically relevant data with timestamps from a long clinical note."""
    if not note_text or note_text == "No Note Available":
        return note_text

    if len(note_text) < NOTE_EXTRACTION_THRESHOLD:
        return note_text

    try:
```

```
        response = openai_client.chat.completions.create(
            model="gpt-4.1",
            messages=[
                {"role": "system", "content": "You are a clinical data extraction specialist. Extract relevant medical information with precise timestamps."},
                {"role": "user", "content": NOTE_EXTRACTION_PROMPT.format(note_type=note_type, note_text=note_text)}
            ],
            temperature=0,
            max_tokens=3000
        )

        extracted = response.choices[0].message.content.strip()
        print(f"    Extracted {note_type}: {len(note_text)} chars → {len(extracted)} chars")
        return extracted

    except Exception as e:
        print(f"    Warning: Extraction failed for {note_type}: {e}")
        return note_text[:NOTE_EXTRACTION_THRESHOLD] + "\n\n[Note truncated]"


def extract_notes_for_case(clinical_data):
    """Extract clinical data from all notes for a case."""
    note_types = {
        "discharge_summary": ("discharge_summary_text", "Discharge Summary"),
        "hp_note": ("hp_note_text", "History & Physical"),
        "procedures": ("procedures_note_text", "Procedures"),
        "progress_note": ("progress_note_text", "Progress Notes"),
        "consult_note": ("consult_note_text", "Consult Notes"),
        "ed_notes": ("ed_notes_text", "ED Notes"),
        "initial_assessment": ("initial_assessment_text", "Initial Assessments"),
```

```
    "ed_triage": ("ed_triage_text", "ED Triage Notes"),
    "ed_provider_note": ("ed_provider_note_text", "ED Provider Notes"),
    "addendum_note": ("addendum_note_text", "Addendum Note"),
    "hospital_course": ("hospital_course_text", "Hospital Course"),
    "subjective_objective": ("subjective_objective_text", "Subjective & Objective"),
    "assessment_plan": ("assessment_plan_text", "Assessment & Plan Note"),
    "nursing_note": ("nursing_note_text", "Nursing Note"),
    "code_documentation": ("code_documentation_text", "Code Documentation"),
    "anesthesia_preprocedure": ("anesthesia_preprocedure_text", "Anesthesia Preprocedure
Evaluation"),
    "anesthesia_postprocedure": ("anesthesia_postprocedure_text", "Anesthesia Postprocedure
Evaluation"),
    "hp_view_only": ("hp_view_only_text", "H&P (View-Only)"),
    "interval_hp_note": ("interval_hp_note_text", "Interval H&P Note"),
    "anesthesia_procedure": ("anesthesia_procedure_text", "Anesthesia Procedure Notes"),
    "ld_delivery_note": ("ld_delivery_note_text", "L&D Delivery Note"),
    "pre_procedure_assessment": ("pre_procedure_assessment_text", "Pre-Procedure Assessment"),
    "inpatient_med_chart": ("inpatient_med_chart_text", "Inpatient Medication Chart"),
    "hospice": ("hospice_text", "Hospice"),
    "hospice_plan_of_care": ("hospice_plan_of_care_text", "Hospice Plan of Care"),
    "hospice_non_covered": ("hospice_non_covered_text", "Hospice Non-Covered"),
    "or_post_procedure": ("or_post_procedure_text", "OR Post-Procedure Note"),
    "peri_op": ("peri_op_text", "Peri-OP"),
    "treatment_plan": ("treatment_plan_text", "Treatment Plan"),
    "delivery": ("delivery_text", "Delivery"),
    "brief_op_note": ("brief_op_note_text", "Brief Op Note"),
    "operative_report": ("operative_report_text", "Operative Report"),
    "scanned_form": ("scanned_form_text", "Scanned Form"),
    "therapy_evaluation": ("therapy_evaluation_text", "Therapy Evaluation"),
    "therapy_treatment": ("therapy_treatment_text", "Therapy Treatment"),
```

```
        "therapy_discharge": ("therapy_discharge_text", "Therapy Discharge"),
        "therapy_progress_note": ("therapy_progress_note_text", "Therapy Progress Note"),
        "wound_care": ("wound_care_text", "Wound Care"),
        "anesthesia_post_eval": ("anesthesia_post_eval_text", "Anesthesia Post Evaluation"),
        "query": ("query_text", "Query"),
        "anesthesia_postop_followup": ("anesthesia_postop_followup_text", "Anesthesia Post-Op Follow-up Note"),
        "anesthesia_handoff": ("anesthesia_handoff_text", "Anesthesia Handoff"),
        "anesthesia_pat_eval": ("anesthesia_pat_eval_text", "Anesthesia PAT Evaluation"),
        "anesthesiology": ("anesthesiology_text", "Anesthesiology"),
        "ed_attestation_note": ("ed_attestation_note_text", "ED Attestation Note"),
        "ed_procedure_note": ("ed_procedure_note_text", "ED Procedure Note"),
        "ed_reeval_note": ("ed_reeval_note_text", "ED Re-evaluation Note"),
        "cdu_provider_note": ("cdu_provider_note_text", "CDU Provider Note"),
    }

    extracted_notes = {}
    notes_to_extract = []

    for key, (col_name, display_name) in note_types.items():
        note_text = clinical_data.get(col_name, "No Note Available")
        if note_text and note_text != "No Note Available":
            notes_to_extract.append((key, col_name, display_name, note_text))
        else:
            extracted_notes[key] = "Not available"

    if notes_to_extract:
        print(f"  Extracting from {len(notes_to_extract)} notes in parallel...")
        with ThreadPoolExecutor(max_workers=len(notes_to_extract)) as executor:
            futures = {
```

```
            executor.submit(extract_clinical_data, note_text, display_name): key
            for key, col_name, display_name, note_text in notes_to_extract
        }
        for future in as_completed(futures):
            key = futures[future]
            extracted_notes[key] = future.result()

    return extracted_notes

print("Clinical note extraction functions loaded")

# ============================================================================
# CELL 6: Structured Data Query Functions
# ============================================================================

def create_target_encounter_view(account_id):
    """Create temp view for target encounter."""
    spark.sql(f"""
    CREATE OR REPLACE TEMP VIEW target_encounter AS
    SELECT
        peh.HSP_ACCOUNT_ID,
        peh.PAT_ENC_CSN_ID,
        peh.PAT_ID,
        peh.ADM_DATE_TIME AS ENCOUNTER_START,
        peh.DISCH_DATE_TIME AS ENCOUNTER_END
    FROM prod.clarity_cur.pat_enc_hsp_har_enh peh
    WHERE peh.HSP_ACCOUNT_ID = {account_id}
    """)
    spark.sql("CACHE TABLE target_encounter")
```

```
def query_labs(account_id):
    """Query all labs for account."""
    spark.sql(f"""
    CREATE OR REPLACE TABLE {LABS_TABLE} AS
    SELECT
        t.HSP_ACCOUNT_ID,
        t.PAT_ENC_CSN_ID,
        CAST(res_comp.COMP_VERIF_DTTM AS TIMESTAMP) AS EVENT_TIMESTAMP,
        cc.NAME AS LAB_NAME,
        CAST(REGEXP_REPLACE(res_comp.COMPONENT_VALUE, '>', '') AS STRING) AS lab_value,
        res_comp.component_units AS lab_units,
        zsab.NAME AS abnormal_flag
    FROM target_encounter t
    INNER JOIN prod.clarity.order_proc op ON t.PAT_ENC_CSN_ID = op.PAT_ENC_CSN_ID
    INNER JOIN prod.clarity.RES_DB_MAIN rdm ON rdm.RES_ORDER_ID = op.ORDER_PROC_ID
    INNER JOIN prod.clarity.res_components res_comp ON res_comp.result_id = rdm.result_id
    INNER JOIN prod.clarity.clarity_component cc ON cc.component_id = res_comp.component_id
    LEFT JOIN prod.clarity.zc_stat_abnorms zsab ON zsab.stat_abnorms_c = res_comp.
component_abn_c
    WHERE op.order_status_c = 5
      AND op.lab_status_c IN (3, 5)
      AND rdm.res_val_status_c = 9
      AND res_comp.COMPONENT_VALUE IS NOT NULL
      AND res_comp.COMPONENT_VALUE <> '-1'
    ORDER BY res_comp.COMP_VERIF_DTTM ASC
    """)
    count = spark.sql(f"SELECT COUNT(*) as cnt FROM {LABS_TABLE}").collect()[0]["cnt"]
    print(f"  Labs: {count} rows")


def query_vitals(account_id):
```

```
    """Query vitals for account."""
    spark.sql(f"""
    CREATE OR REPLACE TABLE {VITALS_TABLE} AS
    SELECT
        t.HSP_ACCOUNT_ID,
        t.PAT_ENC_CSN_ID,
        CAST(to_timestamp(substring(v.RECORDED_TIME, 1, 19), 'yyyy-MM-dd HH:mm:ss')
AS TIMESTAMP) AS EVENT_TIMESTAMP,
        v.FLO_MEAS_NAME AS VITAL_NAME,
        v.MEAS_VALUE AS vital_value
    FROM target_encounter t
    INNER JOIN prod.clarity_cur.ip_flwsht_rec_enh v ON t.PAT_ENC_CSN_ID = v.IP_DATA_
STORE_EPT_CSN
    WHERE v.FLO_MEAS_ID IN ('5', '6', '8', '9', '10', '11', '14')
      AND v.MEAS_VALUE IS NOT NULL
    ORDER BY EVENT_TIMESTAMP ASC
    """)
    count = spark.sql(f"SELECT COUNT(*) as cnt FROM {VITALS_TABLE}").collect()[0]["cnt"]
    print(f"  Vitals: {count} rows")


def query_meds(account_id):
    """Query medications for account."""
    spark.sql(f"""
    CREATE OR REPLACE TABLE {MEDS_TABLE} AS
    SELECT
        t.HSP_ACCOUNT_ID,
        t.PAT_ENC_CSN_ID,
        CAST(mar.TAKEN_TIME AS TIMESTAMP) AS EVENT_TIMESTAMP,
        om.SIMPLE_GENERIC_NAME AS MED_NAME,
        CAST(om.HV_DISCRETE_DOSE AS STRING) AS MED_DOSE,
```

```
        om.DOSE_UNIT AS MED_UNITS,
        mar.ROUTE AS MED_ROUTE,
        mar.ACTION AS ADMIN_ACTION
    FROM target_encounter t
    INNER JOIN prod.clarity_cur.order_med_enh om ON t.PAT_ENC_CSN_ID = om.PAT_ENC_CSN_ID
    INNER JOIN prod.clarity_cur.mar_admin_info_enh mar ON om.ORDER_MED_ID = mar.ORDER_
MED_ID
    WHERE mar.ACTION IN (
        'Given', 'Patient/Family Admin', 'Given-See Override',
        'Admin by Another Clinician (Comment)', 'New Bag', 'Bolus', 'Push',
        'Started by Another Clinician', 'Bag Switched', 'Clinic Sample Administered',
        'Applied', 'Feeding Started', 'Acknowledged', 'Contrast Given',
        'New Bag-See Override', 'Bolus from Bag'
    )
    ORDER BY EVENT_TIMESTAMP ASC
    """)
    count = spark.sql(f"SELECT COUNT(*) as cnt FROM {MEDS_TABLE}").collect()[0]["cnt"]
    print(f"  Medications: {count} rows")


def query_diagnoses(account_id):
    """
    Query diagnosis records for account (DX_NAME is the granular clinical description).
    All diagnoses include their timestamp - LLM decides relevance based on date.
    """
    spark.sql(f"""
    CREATE OR REPLACE TEMP VIEW encounter_diagnoses AS
    -- Outpatient encounter diagnoses
    SELECT te.HSP_ACCOUNT_ID, te.PAT_ENC_CSN_ID,
        dd.DX_ID,
        edg.DX_NAME,
```

```
    CAST(pe.CONTACT_DATE AS TIMESTAMP) AS EVENT_TIMESTAMP,
    'OUTPATIENT_ENC_DX' AS source
FROM target_encounter te
JOIN prod.clarity_cur.pat_enc_dx_enh dd ON dd.PAT_ENC_CSN_ID = te.PAT_ENC_CSN_ID
JOIN prod.clarity_cur.pat_enc_enh pe ON pe.PAT_ENC_CSN_ID = dd.PAT_ENC_CSN_ID
LEFT JOIN prod.clarity.clarity_edg edg ON dd.DX_ID = edg.DX_ID
WHERE dd.DX_ID IS NOT NULL
UNION ALL
-- Inpatient hospital account diagnoses
SELECT te.HSP_ACCOUNT_ID, te.PAT_ENC_CSN_ID,
    dx.DX_ID,
    edg.DX_NAME,
    CAST(ha.DISCH_DATE_TIME AS TIMESTAMP) AS EVENT_TIMESTAMP,
    'INPATIENT_ACCT_DX' AS source
FROM target_encounter te
JOIN prod.clarity_cur.hsp_acct_dx_list_enh dx ON dx.PAT_ID = te.PAT_ID
JOIN prod.clarity_cur.pat_enc_hsp_har_enh ha ON ha.HSP_ACCOUNT_ID = dx.HSP_ACCOUNT_ID
LEFT JOIN prod.clarity.clarity_edg edg ON dx.DX_ID = edg.DX_ID
WHERE dx.DX_ID IS NOT NULL
UNION ALL
-- Problem list history (uses HX fields directly)
SELECT te.HSP_ACCOUNT_ID, te.PAT_ENC_CSN_ID,
    phx.HX_PROBLEM_ID AS DX_ID,
    phx.HX_PROBLEM_DX_NAME AS DX_NAME,
    CAST(phx.HX_DATE_OF_ENTRY AS TIMESTAMP) AS EVENT_TIMESTAMP,
    'PROBLEM_LIST' AS source
FROM target_encounter te
JOIN prod.clarity_cur.problem_list_hx_enh phx ON phx.PAT_ID = te.PAT_ID
WHERE phx.HX_PROBLEM_ID IS NOT NULL AND phx.HX_STATUS = 'Active'
""")
```

```
    spark.sql(f"""
    CREATE OR REPLACE TABLE {DIAGNOSIS_TABLE} AS
    SELECT DISTINCT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, DX_ID, DX_NAME,
           EVENT_TIMESTAMP, source
    FROM encounter_diagnoses
    WHERE DX_NAME IS NOT NULL
    ORDER BY EVENT_TIMESTAMP ASC
    """)
    count = spark.sql(f"SELECT COUNT(*) as cnt FROM {DIAGNOSIS_TABLE}").collect()[0]["cnt"]
    print(f"  Diagnoses: {count} rows")


def create_merged_timeline(account_id):
    """Merge all structured data into chronological timeline."""
    spark.sql(f"""
    CREATE OR REPLACE TABLE {MERGED_TABLE} AS
    WITH RawEvents AS (
        SELECT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, 'LAB' AS event_type,
               CONCAT(LAB_NAME, ': ', lab_value, ' ', COALESCE(lab_units, ''),
                      CASE WHEN abnormal_flag IS NOT NULL THEN CONCAT(' (', abnormal_flag, ')') ELSE ''
END
               ) AS event_detail,
               CASE WHEN abnormal_flag IS NOT NULL THEN 1 ELSE 0 END AS is_abnormal
        FROM {LABS_TABLE}
        UNION ALL
        SELECT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, 'VITAL' AS event_type,
               CONCAT(VITAL_NAME, ': ', vital_value) AS event_detail, 0 AS is_abnormal
        FROM {VITALS_TABLE}
        UNION ALL
        SELECT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, 'MEDICATION' AS
```

```
event_type,
        CONCAT(MED_NAME, ' ', COALESCE(MED_DOSE, ''), ' ', COALESCE(MED_UNITS, ''),
            ' via ', COALESCE(MED_ROUTE, 'unknown'), ' - ', ADMIN_ACTION
        ) AS event_detail, 0 AS is_abnormal
    FROM {MEDS_TABLE}
    UNION ALL
    SELECT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, 'DIAGNOSIS' AS
event_type,
        COALESCE(DX_NAME, 'Unknown diagnosis') AS event_detail,
        0 AS is_abnormal
    FROM {DIAGNOSIS_TABLE}
    WHERE EVENT_TIMESTAMP IS NOT NULL
  ),
  Deduplicated AS (
    SELECT *, ROW_NUMBER() OVER (
      PARTITION BY HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, event_type,
event_detail
      ORDER BY EVENT_TIMESTAMP
    ) as rn
    FROM RawEvents
  )
  SELECT HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_TIMESTAMP, event_type, event_
detail, is_abnormal
  FROM Deduplicated WHERE rn = 1
  ORDER BY EVENT_TIMESTAMP ASC
  """)
  count = spark.sql(f"SELECT COUNT(*) as cnt FROM {MERGED_TABLE}").collect()[0]["cnt"]
  print(f"  Merged timeline: {count} rows")


print("Structured data query functions loaded")
```

```
# ============================================================================

# CELL 7: Structured Data Extraction (LLM)

# ============================================================================


STRUCTURED_DATA_EXTRACTION_PROMPT = '''You are a clinical data analyst
extracting acute respiratory failure-relevant information from structured EHR data.


**Context on Diagnosis Records:**

The diagnosis names are the granular clinical descriptions from Epic's diagnosis dictionary. Quote
these directly in appeals - they are the specific documented diagnoses. For example:

- "Acute respiratory failure with hypoxia"

- "Acute on chronic respiratory failure with hypercapnia"

- "Acute hypoxemic respiratory failure due to pneumonia"


Multiple diagnosis records may describe the same condition at different levels
of specificity. Use the most specific documented diagnosis that is supported by clinical evidence.


Diagnoses include timestamps - use these to understand if a condition is pre-existing (
before admission) or documented during the encounter.


**Your Task:**

Extract a focused summary of acute respiratory failure-relevant clinical data from this timeline. Prioritize:


1. **Oxygenation & Ventilation Parameters** (with timestamps and trends):
   - PaO2, SaO2, SpO2
   - PaCO2
   - FiO2 at time of measurement
   - P/F ratio (PaO2/FiO2) - calculated or documented
   - pH from ABG or VBG
   - Oxygen delivery method and flow rate
   - Ventilator settings if applicable (mode, tidal volume, PEEP, rate)
```

2. **Respiratory Failure Diagnostic Criteria**:

- Hypoxic criteria: PaO2 < 60 mmHg, SpO2 < 91% on room air, P/F ratio < 300

- Hypoxic criteria on supplemental oxygen: Use P/F ratio < 300 to identify respiratory failure when patient is receiving oxygen therapy

- Hypercapnic criteria: PaCO2 > 50 mmHg with pH < 7.35

- Acute-on-chronic indicators: change ≥ 10 mmHg from baseline PaO2 or PaCO2; SpO2 < 91% on chronic oxygen administration

- Baseline respiratory status (home O2, chronic lung disease, prior ABGs)

3. **Clinical Trajectory**:

- When did patient meet respiratory failure criteria?

- Escalation of oxygen therapy (timing and levels)

- Initiation of CPAP, BiPAP, or mechanical ventilation

- Worst values and when they occurred

- Signs of respiratory distress (tachypnea > 20/min, bradypnea < 10/min, accessory muscle use, retractions, cyanosis)

- Mental status changes (confusion, somnolence, anxiety, restlessness)

4. **Underlying Etiology Indicators**:

- Pneumonia, COPD exacerbation, asthma, pulmonary edema, PE, aspiration

- Chest X-ray or CT findings

- Infection markers (WBC, procalcitonin, cultures)

- BNP/NT-proBNP (cardiac vs. pulmonary differentiation)

5. **Relevant Diagnoses** (with dates - note which are pre-existing vs new)

**Structured Timeline:**

{structured_timeline}

**Output Format:**

```
Provide a concise clinical summary (500-800 words) organized by the categories above, with specific timestamps and values. Flag any data gaps.'"


def extract_structured_data_summary(account_id):
    """Extract acute respiratory failure-relevant summary from structured data timeline."""
    print("  Extracting structured data summary...")

    # Get timeline data
    timeline_df = spark.sql(f"""
        SELECT EVENT_TIMESTAMP, event_type, event_detail, is_abnormal
        FROM {MERGED_TABLE}
        ORDER BY EVENT_TIMESTAMP
        LIMIT 500
    """)
    timeline_rows = timeline_df.collect()

    if not timeline_rows:
        return "No structured data available for this encounter."

    # Format timeline for LLM
    timeline_text = "\n".join([
        f"[{row['EVENT_TIMESTAMP']}] {row['event_type']}: {row['event_detail']}"
        + (" (ABNORMAL)" if row['is_abnormal'] else "")
        for row in timeline_rows
    ])

    try:
        response = openai_client.chat.completions.create(
            model="gpt-4.1",
            messages=[
```

```
            {"role": "system", "content": "You are a clinical data analyst specializing in acute respiratory failure cases."},
            {"role": "user", "content": STRUCTURED_DATA_EXTRACTION_PROMPT.format(structured_timeline=timeline_text)}
        ],
        temperature=0,
        max_tokens=2000
    )
    summary = response.choices[0].message.content.strip()
    print(f"  Structured data summary: {len(summary)} chars")
    return summary

  except Exception as e:
    print(f"  Warning: Structured data extraction failed: {e}")
    return f"Extraction failed. Raw timeline has {len(timeline_rows)} events."

print("Structured data extraction functions loaded")

# ============================================================================
# CELL 8: Conflict Detection
# ============================================================================

CONFLICT_DETECTION_PROMPT = '''You are comparing clinical documentation from two sources for the same patient encounter:

1. **PHYSICIAN NOTES** (primary source - clinical interpretation):
{notes_summary}

2. **STRUCTURED DATA** (objective measurements):
{structured_summary}
```

**Your Task:**

Identify any CONFLICTS where the physician notes say one thing but the structured data shows something different.

Examples of conflicts:

- Note says "SpO2 maintained above 94%" but vitals show SpO2 values below 91%

- Note says "patient on room air" but respiratory flowsheets show supplemental oxygen administration

- Note says "ABG normalized" but labs show PaO2 < 60 mmHg or PaCO2 > 50 mmHg with pH < 7.35

- Note says "no respiratory distress" but vitals show respiratory rate > 20 or < 10 breaths/min

- Note says "weaned to nasal cannula" but flowsheets show patient still on BiPAP or high-flow oxygen

- Note says "P/F ratio improved" but calculated P/F ratio remains < 300

- Note says "no oxygen requirement" but orders show active supplemental oxygen delivery

- Note says "patient alert and oriented" but documentation shows confusion, somnolence, or altered mental status

**Important:**

- Only flag CLEAR contradictions, not missing information

- Note the specific values from each source

- Consider timing - data from different times is not a conflict

**Output Format:**

If conflicts found, list each one:

CONFLICT 1: [Notes say X, but structured data shows Y]

CONFLICT 2: [Notes say X, but structured data shows Y]

If no conflicts: "NO CONFLICTS DETECTED"

Then provide:

RECOMMENDATION: [Brief guidance for CDI reviewer]"'

```
def detect_conflicts(notes_summary, structured_summary):
    """Detect conflicts between clinical notes and structured data."""
    print("  Running conflict detection...")

    if not notes_summary or not structured_summary:
        return {"conflicts": [], "recommendation": "Insufficient data for conflict detection"}

    try:
        response = openai_client.chat.completions.create(
            model="gpt-4.1",
            messages=[
                {"role": "system", "content": "You are a clinical documentation integrity specialist."},
                {"role": "user", "content": CONFLICT_DETECTION_PROMPT.format(
                    notes_summary=notes_summary[:8000],
                    structured_summary=structured_summary[:8000]
                )}
            ],
            temperature=0,
            max_tokens=1000
        )

        result_text = response.choices[0].message.content.strip()

        # Parse conflicts j- only caputer lines iwth actual content after "CONFLICT X"
        conflicts = []
        recommendation = ""

        lines = result_text.split("\n")
        for line in lines:
            line = line.strip()
```

```
            if line.startswith("CONFLICT"):
                # Check if there's actual content after "CONFLICT X:"
                if ":" in line:
                    after_colon = line.split(":", 1)[1].strip() if len(line.split(":", 1)) > 1 else ""
                    if after_colon: # Only add if there's content after the colon
                        conflicts.append(line)
            elif line.startswith("RECOMMENDATION:"):
                recommendation = line.replace("RECOMMENDATION:", "").strip()

        if "NO CONFLICTS DETECTED" in result_text:
            print("  No conflicts detected")
        else:
            print(f"  Found {len(conflicts)} conflicts")

        return {
            "conflicts": conflicts,
            "recommendation": recommendation,
            "raw_response": result_text
        }

    except Exception as e:
        print(f"  Warning: Conflict detection failed: {e}")
        return {"conflicts": [], "recommendation": f"Detection failed: {e}"}

print("Conflict detection functions loaded")

# ============================================================================
# CELL 9: Write to Case Tables
# ============================================================================
```

```
def write_case_denial_table(account_id, denial_text, denial_embedding, denial_info):
    """Write denial data to case table."""
    spark.sql(f"""
    CREATE TABLE IF NOT EXISTS {CASE_DENIAL_TABLE} (
        account_id STRING,
        denial_text STRING,
        denial_embedding ARRAY<FLOAT>,
        payor STRING,
        original_drg STRING,
        proposed_drg STRING,
        is_arf BOOLEAN,
        created_at TIMESTAMP
    ) USING DELTA
    """)

    record = [{
        "account_id": account_id,
        "denial_text": denial_text,
        "denial_embedding": denial_embedding,
        "payor": denial_info.get("payor", "Unknown"),
        "original_drg": denial_info.get("original_drg"),
        "proposed_drg": denial_info.get("proposed_drg"),
        "is_arf": denial_info.get("is_arf", False),
        "created_at": datetime.now()
    }]

    schema = StructType([
        StructField("account_id", StringType(), False),
        StructField("denial_text", StringType(), True),
```

```
        StructField("denial_embedding", ArrayType(FloatType()), True),
        StructField("payor", StringType(), True),
        StructField("original_drg", StringType(), True),
        StructField("proposed_drg", StringType(), True),
        StructField("is_arf", BooleanType(), True),
        StructField("created_at", TimestampType(), True)
    ])

    df = spark.createDataFrame(record, schema)
    df.write.format("delta").mode("overwrite").saveAsTable(CASE_DENIAL_TABLE)
    print(f"  Written to {CASE_DENIAL_TABLE}")

def write_case_clinical_table(account_id, clinical_data, extracted_notes):
    """Write clinical notes data to case table."""
    spark.sql(f"""
    CREATE TABLE IF NOT EXISTS {CASE_CLINICAL_TABLE} (
        account_id STRING,
        patient_name STRING,
        patient_dob STRING,
        facility_name STRING,
        date_of_service STRING,
        extracted_notes STRING,
        created_at TIMESTAMP
    ) USING DELTA
    """)

    record = [{
        "account_id": account_id,
        "patient_name": clinical_data.get("formatted_name", "Unknown"),
```

```
        "patient_dob": clinical_data.get("formatted_birthdate", ""),
        "facility_name": clinical_data.get("facility_name", "Mercy Hospital"),
        "date_of_service": clinical_data.get("formatted_date_of_service", ""),
        "extracted_notes": json.dumps(extracted_notes),
        "created_at": datetime.now()
    }]

    df = spark.createDataFrame(record)
    df.write.format("delta").mode("overwrite").saveAsTable(CASE_CLINICAL_TABLE)
    print(f"  Written to {CASE_CLINICAL_TABLE}")

def write_case_structured_summary_table(account_id, structured_summary):
    """Write structured data summary to case table."""
    spark.sql(f"""
    CREATE TABLE IF NOT EXISTS {CASE_STRUCTURED_SUMMARY_TABLE} (
        account_id STRING,
        structured_summary STRING,
        created_at TIMESTAMP
    ) USING DELTA
    """)

    record = [{
        "account_id": account_id,
        "structured_summary": structured_summary,
        "created_at": datetime.now()
    }]

    df = spark.createDataFrame(record)
    df.write.format("delta").mode("overwrite").saveAsTable(CASE_STRUCTURED_SUMMARY_TABLE)
    print(f"  Written to {CASE_STRUCTURED_SUMMARY_TABLE}")
```

```
def write_case_conflicts_table(account_id, conflicts_result):
    """Write conflicts data to case table."""
    spark.sql(f"""
    CREATE TABLE IF NOT EXISTS {CASE_CONFLICTS_TABLE} (
        account_id STRING,
        conflicts STRING,
        recommendation STRING,
        created_at TIMESTAMP
    ) USING DELTA
    """)

    record = [{
        "account_id": account_id,
        "conflicts": json.dumps(conflicts_result.get("conflicts", [])),
        "recommendation": conflicts_result.get("recommendation", ""),
        "created_at": datetime.now()
    }]

    df = spark.createDataFrame(record)
    df.write.format("delta").mode("overwrite").saveAsTable(CASE_CONFLICTS_TABLE)
    print(f"  Written to {CASE_CONFLICTS_TABLE}")

print("Case table write functions loaded")

# ============================================================================
# CELL 10: Main Processing Pipeline
# ============================================================================
print("\n" + "="*60)
```

```
print("FEATURIZATION - PER-CASE DATA PREPARATION")
print("="*60)

# Check input file exists
if not os.path.exists(DENIAL_PDF_PATH):
    print(f"\nERROR: Denial PDF not found: {DENIAL_PDF_PATH}")
    print("Set DENIAL_PDF_PATH to a valid PDF file path.")
else:
    print(f"\nInput: {os.path.basename(DENIAL_PDF_PATH)}")

    # ------------------------------------------------------------------
    # STEP 1: Parse denial PDF
    # ------------------------------------------------------------------
    print("\nStep 1: Parsing denial PDF...")
    pages = extract_text_from_pdf(DENIAL_PDF_PATH)
    denial_text = "\n\n".join(pages)
    denial_embedding = generate_embedding(denial_text)
    print(f"  Extracted {len(pages)} pages, {len(denial_text)} chars")
    print(f"  Generated embedding ({len(denial_embedding)} dims)")

    # ------------------------------------------------------------------
    # STEP 2: Extract denial info via LLM
    # ------------------------------------------------------------------
    print("\nStep 2: Extracting denial info...")
    denial_info = extract_denial_info_llm(denial_text[:15000])
    print(f"  Account ID: {denial_info['hsp_account_id'] or 'NOT FOUND'}")
    print(f"  Payor: {denial_info['payor']}")
    print(f"  DRG: {denial_info['original_drg']} → {denial_info['proposed_drg']}")
    print(f"  ARF: {denial_info['is_arf']}")
```

```
    # Use known account ID if provided
    account_id = KNOWN_ACCOUNT_ID or denial_info['hsp_account_id']

    if not account_id:
        print("\nERROR: No account ID found. Set KNOWN_ACCOUNT_ID or ensure denial
letter contains H-prefixed account number.")
    else:
        # ------------------------------------------------------------------
        # STEP 3: Query clinical notes
        # ------------------------------------------------------------------
        print(f"\nStep 3: Querying clinical notes for account {account_id}...")
        clinical_data = query_clarity_for_account(account_id)

        if clinical_data:
            # ------------------------------------------------------------------
            # STEP 4: Extract clinical notes
            # ------------------------------------------------------------------
            print("\nStep 4: Extracting clinical notes...")
            extracted_notes = extract_notes_for_case(clinical_data)

            # Create notes summary for conflict detection
            notes_summary = "\n\n".join([
                f"## {key}\n{value[:2000]}"
                for key, value in extracted_notes.items()
                if value and value != "Not available"
            ])

            # ------------------------------------------------------------------
            # STEP 5: Query structured data
            # ------------------------------------------------------------------
```

```
print(f"\nStep 5: Querying structured data for account {account_id}...")
create_target_encounter_view(account_id)
query_labs(account_id)
query_vitals(account_id)
query_meds(account_id)
query_diagnoses(account_id)
create_merged_timeline(account_id)

# ---------------------------------------------------------------------
# STEP 6: Extract structured data summary
# ---------------------------------------------------------------------
print("\nStep 6: Extracting structured data summary...")
structured_summary = extract_structured_data_summary(account_id)

# ---------------------------------------------------------------------
# STEP 7: Detect conflicts
# ---------------------------------------------------------------------
print("\nStep 7: Detecting conflicts...")
conflicts_result = detect_conflicts(notes_summary, structured_summary)

# ---------------------------------------------------------------------
# STEP 8: Write to case tables
# ---------------------------------------------------------------------
print("\nStep 8: Writing to case tables...")
write_case_denial_table(account_id, denial_text, denial_embedding, denial_info)
write_case_clinical_table(account_id, clinical_data, extracted_notes)
write_case_structured_summary_table(account_id, structured_summary)
write_case_conflicts_table(account_id, conflicts_result)
```

```
        # ----------------------------------------------------------------------
        # SUMMARY
        # ----------------------------------------------------------------------
        print("\n" + "="*60)
        print("FEATURIZATION COMPLETE")
        print("="*60)
        print(f"Account: {account_id}")
        print(f"Patient: {clinical_data.get('formatted_name', 'Unknown')}")
        print(f"Clinical notes extracted: {len([v for v in extracted_notes.values() if v != 'Not available'])}")
        print(f"Structured summary: {len(structured_summary)} chars")
        print(f"Conflicts: {len(conflicts_result.get('conflicts', []))}")
        if conflicts_result.get('conflicts'):
            print("\nConflicts found:")
            for conflict in conflicts_result['conflicts']:
                print(f"  - {conflict}")
        print(f"\nCase tables written - ready for inference.py")

    else:
        print("\nERROR: Could not retrieve clinical data from Clarity.")

print("\nFeaturization inference complete.")
```

featurization_train.py

```
# data/featurization_train.py
# Acute Respiratory Failure Appeal Engine - Knowledge Base Setup (One-Time)
#
# ONE-TIME SETUP: Ingests gold standard letters and Propel definitions
# into Unity Catalog tables. Run once, or when adding new gold letters.
```

```
#
# This is the "training" featurization - builds the knowledge base.
# For per-case data prep, see featurization_inference.py
#
# Run on Databricks Runtime 15.4 LTS ML

# ============================================================================
# CELL 1: Install Dependencies (run this cell FIRST, then restart)
# ============================================================================
# IMPORTANT: Run this cell by itself, then run the rest of the notebook.
# After restart, the packages persist for the cluster session.
#
# Uncomment and run ONCE per cluster session:
# %pip install azure-ai-documentintelligence==1.0.2 openai python-docx
# dbutils.library.restartPython()
#
# After restart completes, run Cell 2 onwards (leave this cell commented)

# ============================================================================
# CELL 2: Imports and Configuration
# ============================================================================
import os
import re
import uuid
import tiktoken
from datetime import datetime
from pyspark.sql import SparkSession
from pyspark.sql.types import (
    StructType, StructField, StringType, ArrayType, FloatType,
    TimestampType, MapType
```

```
)

spark = SparkSession.builder.getOrCreate()

# Configuration
EMBEDDING_MODEL = "text-embedding-ada-002"  # 1536 dimensions

# ============================================================================
# CELL 3: Environment Setup
# ============================================================================
trgt_cat = os.environ.get('trgt_cat', 'dev')
if trgt_cat == 'dev':
    spark.sql('USE CATALOG prod;')
else:
    spark.sql(f'USE CATALOG {trgt_cat};')

# Table names
GOLD_LETTERS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_gold_letters"
PROPEL_DATA_TABLE = f"{trgt_cat}.fin_ds.creamsicle_propel_data"

# Paths
GOLD_LETTERS_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
gold_standard_appeals/gold_standard_appeals_arf_only"
PROPEL_DATA_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/propel_data"

print(f"Catalog: {trgt_cat}")

# ============================================================================
# CELL 3B: Validation Checkpoints
# ============================================================================
```

```
def checkpoint_paths():
    """
    CHECKPOINT: Verify all required paths exist and contain expected files.
    Call this before running any ingestion.
    """
    print("\n" + "="*60)
    print("CHECKPOINT: Path Validation")
    print("="*60)

    all_valid = True

    # Check gold_standard_appeals
    if os.path.exists(GOLD_LETTERS_PATH):
        pdf_count = len([f for f in os.listdir(GOLD_LETTERS_PATH) if f.lower().endswith('.pdf')])
        docx_count = len([f for f in os.listdir(GOLD_LETTERS_PATH) if f.lower().endswith('.docx')])
        print(f"[OK] gold_standard_appeals: {pdf_count} PDFs, {docx_count} DOCX files")
        if pdf_count == 0:
            print("  [WARN] No PDF files found - gold letter ingestion will have nothing to process")
    else:
        print(f"[FAIL] gold_standard_appeals not found: {GOLD_LETTERS_PATH}")
        all_valid = False

    # Check propel_data
    if os.path.exists(PROPEL_DATA_PATH):
        pdf_count = len([f for f in os.listdir(PROPEL_DATA_PATH) if f.lower().endswith('.pdf')])
        print(f"[OK] propel_data: {pdf_count} PDFs")
        if pdf_count == 0:
            print("  [WARN] No PDF files found - propel ingestion will have nothing to process")
    else:
```

```
        print(f"[FAIL] propel_data not found: {PROPEL_DATA_PATH}")
        all_valid = False

    if all_valid:
        print("\n[CHECKPOINT PASSED] All paths valid")
    else:
        print("\n[CHECKPOINT FAILED] Fix path issues before proceeding")

    return all_valid

def checkpoint_table(table_name, expected_columns=None, min_rows=0):
    """
    CHECKPOINT: Verify a table exists and has expected structure.

    Args:
        table_name: Full table name (catalog.schema.table)
        expected_columns: List of column names that must exist
        min_rows: Minimum number of rows expected
    """
    print(f"\nCHECKPOINT: Validating {table_name}")

    try:
        # Check table exists and get count
        count = spark.sql(f"SELECT COUNT(*) as cnt FROM {table_name}").collect()[0]["cnt"]
        print(f"  Rows: {count}")

        if count < min_rows:
            print(f"  [WARN] Expected at least {min_rows} rows, found {count}")
            return False
```

```
        # Check columns if specified
        if expected_columns:
            df = spark.sql(f"SELECT * FROM {table_name} LIMIT 1")
            actual_columns = set(df.columns)
            missing = set(expected_columns) - actual_columns
            if missing:
                print(f"  [FAIL] Missing columns: {missing}")
                return False
            print(f"  [OK] All expected columns present")

        print(f"  [CHECKPOINT PASSED]")
        return True

    except Exception as e:
        print(f"  [FAIL] Could not validate table: {e}")
        return False


def checkpoint_gold_letters():
    """CHECKPOINT: Validate gold letters table after ingestion."""
    return checkpoint_table(
        GOLD_LETTERS_TABLE,
        expected_columns=["letter_id", "source_file", "payor", "denial_text", "rebuttal_text", "denial_
embedding"],
        min_rows=1
    )


def checkpoint_propel_data():
    """CHECKPOINT: Validate propel data table after ingestion."""
    return checkpoint_table(
        PROPEL_DATA_TABLE,
```

```
    expected_columns=["condition_name", "source_file", "definition_text", "definition_summary"],
    min_rows=1
)

# Run path checkpoint on load
checkpoint_paths()

# ============================================================================
# CELL 4: Azure Credentials
# ============================================================================
AZURE_OPENAI_KEY = dbutils.secrets.get(scope='idp_etl', key='az-openai-key1')
AZURE_OPENAI_ENDPOINT = dbutils.secrets.get(scope='idp_etl', key='az-openai-base')
AZURE_DOC_INTEL_KEY = dbutils.secrets.get(scope='idp_etl', key='az-aidcmntintel-key1')
AZURE_DOC_INTEL_ENDPOINT = dbutils.secrets.get(scope='idp_etl', key='az-aidcmntintel-endpoint')

print("Credentials loaded")

# ============================================================================
# CELL 5: Initialize Clients
# ============================================================================
from openai import AzureOpenAI
from azure.ai.documentintelligence import DocumentIntelligenceClient
from azure.ai.documentintelligence.models import AnalyzeDocumentRequest
from azure.core.credentials import AzureKeyCredential

openai_client = AzureOpenAI(
    api_key=AZURE_OPENAI_KEY,
    azure_endpoint=AZURE_OPENAI_ENDPOINT,
    api_version="2024-10-21"
```

```
)

doc_intel_client = DocumentIntelligenceClient(
    endpoint=AZURE_DOC_INTEL_ENDPOINT,
    credential=AzureKeyCredential(AZURE_DOC_INTEL_KEY)
)

print("Clients initialized")

# ============================================================================
# CELL 6: Core Parser Functions
# ============================================================================

def extract_text_from_pdf(file_path):
    """
    Extract text from PDF using Document Intelligence layout model.
    Returns list of strings, one per page.
    """
    with open(file_path, 'rb') as f:
        document_bytes = f.read()

    poller = doc_intel_client.begin_analyze_document(
        model_id="prebuilt-layout",
        body=AnalyzeDocumentRequest(bytes_source=document_bytes),
    )
    result = poller.result()

    pages_text = []
    for page in result.pages:
        page_lines = [line.content for line in page.lines]
```

```
        pages_text.append("\n".join(page_lines))

    return pages_text

def identify_denial_start(pages_text):
    """
    Identify which page the denial letter starts on in a gold standard letter.
    Gold letters contain appeal first, then the original denial attached.

    Returns (denial_start_page, payor_name) - 1-indexed page number.
    """
    # Insurance company names
    payor_patterns = [
        ("unitedhealth", "UnitedHealthcare"),
        ("united health", "UnitedHealthcare"),
        ("uhc", "UnitedHealthcare"),
        ("optum", "UnitedHealthcare"),
        ("aetna", "Aetna"),
        ("cigna", "Cigna"),
        ("evernorth", "Cigna"),
        ("humana", "Humana"),
        ("anthem", "Anthem"),
        ("elevance", "Anthem"),
        ("blue cross", "Blue Cross Blue Shield"),
        ("blue shield", "Blue Cross Blue Shield"),
        ("bcbs", "Blue Cross Blue Shield"),
        ("wellpoint", "Anthem"),
        ("kaiser", "Kaiser Permanente"),
        ("molina", "Molina Healthcare"),
```

```
        ("centene", "Centene"),
        ("ambetter", "Centene"),
        ("wellcare", "Centene"),
        ("healthnet", "Health Net"),
        ("medicaid", "Medicaid"),
        ("medicare", "Medicare"),
        ("essence", "Essence Healthcare")
    ]

    # Patterns for address and date (signs of a formal business letter header)
    address_pattern = re.compile(
        r'\b(AL|AK|AZ|AR|CA|CO|CT|DE|FL|GA|HI|ID|IL|IN|IA|KS|KY|LA|ME|MD|MA|MI|MN|MS|MO|
MT|NE|NV|NH|NJ|NM|NY|NC|ND|OH|OK|OR|PA|RI|SC|SD|TN|TX|UT|VT|VA|WA|WV|WI|WY)\s*\d{5}',
        re.IGNORECASE
    )
    date_pattern = re.compile(
        r'(\d{1,2}/\d{1,2}/\d{4})|((January|February|March|April|May|June|July|August|
September|October|November|December)\s+\d{1,2},?\s+\d{4})',
        re.IGNORECASE
    )

    # Skip page 1 - that's the appeal. Start from page 2.
    for i, page_text in enumerate(pages_text):
        if i == 0:
            continue

        first_lines = page_text.split("\n")[:15]
        header_text = "\n".join(first_lines)
        header_lower = header_text.lower()
```

```
    found_payor = None
    for pattern, payor_name in payor_patterns:
        if pattern in header_lower:
            found_payor = payor_name
            break

    if found_payor:
        has_address = bool(address_pattern.search(header_text))
        has_date = bool(date_pattern.search(header_text))

        if has_address or has_date:
            print(f"    Found denial: '{found_payor}' + {'address' if has_address else 'date'} on page {i+1}")
            return i + 1, found_payor
        else:
            print(f"    Found '{found_payor}' on page {i+1} (no address/date pattern)")
            return i + 1, found_payor

# Fallback: look for typical denial letter subject lines
denial_subject_phrases = [
    "claim review determination",
    "medical necessity review",
    "utilization review",
    "peer review determination",
    "clinical review",
    "prior authorization",
    "re: claim",
    "re: member",
    "re: patient",
]
```

```
    for i, page_text in enumerate(pages_text):
        if i == 0:
            continue

        first_lines = page_text.split("\n")[:15]
        header_lower = "\n".join(first_lines).lower()

        for phrase in denial_subject_phrases:
            if phrase in header_lower:
                print(f"    Found denial phrase '{phrase}' on page {i+1}")
                return i + 1, "Unknown"

    print("    WARNING: Could not identify denial start page")
    return len(pages_text) + 1, "Unknown"

def generate_embedding(text):
    """
    Generate embedding vector for text using Azure OpenAI.
    Returns 1536-dimensional vector.
    """
    # Use the tokenizer for the specific model.
    # For text-embedding-ada-002, the encoding is 'cl100k_base'
    encoding = tiktoken.get_encoding("cl100k_base")

    # Define the model's token limit: 8192
    MODEL_TOKEN_LIMIT = 8192

    # Encode the text to get tokens
    tokens = encoding.encode(text)
```

```
    if len(tokens) > MODEL_TOKEN_LIMIT:
        print(f"  Warning: Text truncated from {len(tokens)} to {MODEL_TOKEN_
LIMIT} tokens for embedding")
        # Truncate tokens
        tokens = tokens[:MODEL_TOKEN_LIMIT]
        # Decode tokens back to text for the API call
        text = encoding.decode(tokens)

    response = openai_client.embeddings.create(
        model=EMBEDDING_MODEL,
        input=text
    )
    return response.data[0].embedding


def parse_gold_letter_pdf(file_path):
    """
    Parse a gold standard letter PDF.
    Gold letters contain BOTH appeal (first) and denial (attached at end).
    Returns dict with: rebuttal_text (appeal), denial_text, denial_embedding, payor
    """
    pages = extract_text_from_pdf(file_path)
    denial_start_page, payor = identify_denial_start(pages)

    rebuttal_pages = pages[:denial_start_page - 1]
    denial_pages = pages[denial_start_page - 1:]

    rebuttal_text = "\n\n".join(rebuttal_pages) if rebuttal_pages else ""
    denial_text = "\n\n".join(denial_pages) if denial_pages else ""
```

```
    denial_embedding = generate_embedding(denial_text) if denial_text else None

    return {
        "rebuttal_text": rebuttal_text,
        "denial_text": denial_text,
        "denial_embedding": denial_embedding,
        "payor": payor,
        "denial_start_page": denial_start_page,
        "total_pages": len(pages)
    }

print("Parser functions loaded")

# ============================================================================
# CELL 7: Create Tables
# ============================================================================
# Gold Letters Table - stores past winning appeals with denial embeddings
create_gold_table_sql = f"""
CREATE TABLE IF NOT EXISTS {GOLD_LETTERS_TABLE} (
    letter_id STRING NOT NULL,
    source_file STRING NOT NULL,
    payor STRING,
    denial_text STRING,
    rebuttal_text STRING,
    denial_embedding ARRAY<FLOAT>,
    created_at TIMESTAMP,
    metadata MAP<STRING, STRING>
)
USING DELTA
"""
```

```
spark.sql(create_gold_table_sql)
print(f"Table {GOLD_LETTERS_TABLE} ready")

# Propel Data Table - stores official clinical definitions
create_propel_table_sql = f"""
CREATE TABLE IF NOT EXISTS {PROPEL_DATA_TABLE} (
    condition_name STRING NOT NULL,
    source_file STRING NOT NULL,
    definition_text STRING,
    definition_summary STRING,
    created_at TIMESTAMP
)
USING DELTA
COMMENT 'Official Propel clinical definitions for conditions'
"""
spark.sql(create_propel_table_sql)
print(f"Table {PROPEL_DATA_TABLE} ready")

# ############################################################################
# PART 1: GOLD STANDARD LETTER INGESTION
# ############################################################################

# ============================================================================
# CELL 8: Process Gold Standard Letters
# ============================================================================
RUN_GOLD_INGESTION = True

if RUN_GOLD_INGESTION:
    print("="*60)
```

```
print("GOLD STANDARD LETTER INGESTION")
print("="*60)

pdf_files = [f for f in os.listdir(GOLD_LETTERS_PATH) if f.lower().endswith('.pdf')]
print(f"Found {len(pdf_files)} PDF files")

gold_records = []

for i, pdf_file in enumerate(pdf_files):
    print(f"\nProcessing {i+1}/{len(pdf_files)}: {pdf_file}")
    file_path = os.path.join(GOLD_LETTERS_PATH, pdf_file)

    try:
        parsed = parse_gold_letter_pdf(file_path)

        print(f"  Pages: {parsed['total_pages']}, Denial starts: page {parsed['denial_start_page']}")
        print(f"  Appeal: {len(parsed['rebuttal_text'])} chars")
        print(f"  Denial: {len(parsed['denial_text'])} chars")
        print(f"  Payor: {parsed['payor']}")
        print(f"  Embedding: {len(parsed['denial_embedding'])} dims")

        record = {
            "letter_id": str(uuid.uuid4()),
            "source_file": pdf_file,
            "payor": parsed["payor"],
            "denial_text": parsed["denial_text"],
            "rebuttal_text": parsed["rebuttal_text"],
            "denial_embedding": parsed["denial_embedding"],
            "created_at": datetime.now(),
            "metadata": {
```

```
                "denial_start_page": str(parsed["denial_start_page"]),
                "total_pages": str(parsed["total_pages"])
            },
        }
        gold_records.append(record)
        print(f"  Record created: {record['letter_id']}")

    except Exception as e:
        print(f"  ERROR: {e}")
        continue

print(f"\nProcessed {len(gold_records)} gold standard letters")

if gold_records:
    schema = StructType([
        StructField("letter_id", StringType(), False),
        StructField("source_file", StringType(), False),
        StructField("payor", StringType(), True),
        StructField("denial_text", StringType(), True),
        StructField("rebuttal_text", StringType(), True),
        StructField("denial_embedding", ArrayType(FloatType()), True),
        StructField("created_at", TimestampType(), True),
        StructField("metadata", MapType(StringType(), StringType()), True),
    ])

    gold_df = spark.createDataFrame(gold_records, schema)
    gold_df.write.format("delta").mode("overwrite").saveAsTable(GOLD_LETTERS_TABLE)
    print(f"Wrote {len(gold_records)} records to {GOLD_LETTERS_TABLE}")
```

```
    count = spark.sql(f"SELECT COUNT(*) as cnt FROM {GOLD_LETTERS_TABLE}").collect()[0]["cnt"]
    print(f"Total records in gold letters table: {count}")

    checkpoint_gold_letters()

else:
    print("Gold ingestion skipped (set RUN_GOLD_INGESTION = True)")

# ############################################################################
# PART 2: PROPEL DATA INGESTION
# ############################################################################

# ============================================================================
# CELL 9: Propel Extraction Prompt and Function
# ============================================================================

PROPEL_EXTRACTION_PROMPT = """You are extracting the key clinical criteria from an
official Propel definition document.

Your task: Extract ONLY the essential diagnostic criteria that a physician would use to
determine if a patient meets the clinical definition.

OUTPUT FORMAT:
- Use clear, concise bullet points
- Include specific thresholds, values, and timeframes
- Preserve medical terminology exactly as written
- Include ALL required criteria (do not summarize away important details)
- Remove administrative content, references, and background explanations
- Target length: 500-1000 words (enough to capture all criteria, short enough for an LLM prompt)

CONDITION: {condition_name}
```

```
FULL PROPEL DOCUMENT:
{definition_text}

---

Extract the key clinical criteria below:'"

def extract_propel_summary(condition_name, definition_text):
    """
    Extract key clinical criteria from a Propel definition using LLM.
    Returns a concise summary suitable for inclusion in prompts.
    """
    if not definition_text or len(definition_text) < 100:
        return definition_text

    prompt = PROPEL_EXTRACTION_PROMPT.format(
        condition_name=condition_name.title(),
        definition_text=definition_text
    )

    try:
        response = openai_client.chat.completions.create(
            model="gpt-4.1",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.0,
            max_tokens=2000,
        )
        summary = response.choices[0].message.content.strip()
        return summary
```

```
    except Exception as e:
        print(f"  Warning: Extraction failed ({e}), using full text")
        return definition_text

# ============================================================================
# CELL 10: Process Propel Data Files
# ============================================================================
RUN_PROPEL_INGESTION = True

if RUN_PROPEL_INGESTION:
    print("\n" + "="*60)
    print("PROPEL DATA INGESTION")
    print("="*60)

    if os.path.exists(PROPEL_DATA_PATH):
        pdf_files = [f for f in os.listdir(PROPEL_DATA_PATH) if f.lower().endswith('.pdf')]
        print(f"Found {len(pdf_files)} PDF files in propel_data")

        propel_records = []

        for i, pdf_file in enumerate(pdf_files):
            print(f"\n[{i+1}/{len(pdf_files)}] Processing {pdf_file}")
            file_path = os.path.join(PROPEL_DATA_PATH, pdf_file)

            try:
                pages_text = extract_text_from_pdf(file_path)
                definition_text = "\n\n".join(pages_text)
                print(f"  Extracted {len(definition_text)} chars from {len(pages_text)} pages")
```

```
        # Derive condition name from filename
        base_name = os.path.splitext(pdf_file)[0].lower()
        if base_name.startswith("propel_"):
            condition_name = base_name[7:]
        else:
            condition_name = base_name
        print(f"  Condition: {condition_name}")

        print(f"  Extracting key criteria via LLM...")
        definition_summary = extract_propel_summary(condition_name, definition_text)
        print(f"  Summary: {len(definition_summary)} chars")

        propel_records.append({
            "condition_name": condition_name,
            "source_file": pdf_file,
            "definition_text": definition_text,
            "definition_summary": definition_summary,
            "created_at": datetime.now(),
        })

    except Exception as e:
        print(f"  ERROR: {e}")
        continue

if propel_records:
    propel_df = spark.createDataFrame(propel_records)
    propel_df.write.format("delta").mode("overwrite").saveAsTable(PROPEL_DATA_TABLE)
    print(f"\nWrote {len(propel_records)} records to {PROPEL_DATA_TABLE}")

count = spark.sql(f"SELECT COUNT(*) as cnt FROM {PROPEL_DATA_TABLE}").collect()[0]["cnt"]
```

```
        print(f"Total records in propel data table: {count}")

        checkpoint_propel_data()
    else:
        print(f"WARNING: Propel data path not found: {PROPEL_DATA_PATH}")

else:
    print("\nPropel ingestion skipped (set RUN_PROPEL_INGESTION = True)")

# ============================================================================
# CELL FINAL: Validation Summary
# ============================================================================
print("\n" + "="*60)
print("FINAL VALIDATION CHECKPOINTS")
print("="*60)

checkpoint_results = {}

print("\n--- Gold Letters Table ---")
checkpoint_results["gold_letters"] = checkpoint_gold_letters()

print("\n--- Propel Data Table ---")
checkpoint_results["propel_data"] = checkpoint_propel_data()

# Summary
print("\n" + "="*60)
print("CHECKPOINT SUMMARY")
print("="*60)
passed = sum(1 for v in checkpoint_results.values() if v)
```

```
total = len(checkpoint_results)
print(f"Passed: {passed}/{total}")

for name, result in checkpoint_results.items():
    status = "[OK]" if result else "[FAIL/WARN]"
    print(f"  {status} {name}")

if passed == total:
    print("\n[ALL CHECKPOINTS PASSED] Knowledge base ready for inference.py")
else:
    print("\n[CHECKPOINTS INCOMPLETE] Run ingestion steps as needed")

print("\nFeaturization complete.")
```

structured_data_ingestion.py

```
# data/structured_data_ingestion.py
# Structured Data Ingestion - Linear Script
#
# Run this notebook to gather structured data for a single account.
# Will be merged into featurization.py later.
#
# Run on Databricks Runtime 15.4 LTS ML

# ============================================================================
# CELL 1: Configuration
# ============================================================================
import os
from datetime import datetime
```

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

# ------------------------------------------------------------------------
# INPUT: Set the account ID to process
# ------------------------------------------------------------------------
# HSP_ACCOUNT_ID = 212200065422 # Penny
# HSP_ACCOUNT_ID = 21203218036 # Kinworthy
# HSP_ACCOUNT_ID = 41203238284 # Bell
HSP_ACCOUNT_ID = 53201656408 # Ellis
# HSP_ACCOUNT_ID = 41202436194 # Saddler

# ------------------------------------------------------------------------
# Environment Setup
# ------------------------------------------------------------------------
trgt_cat = os.environ.get('trgt_cat', 'dev')

spark.sql('USE CATALOG prod;')

# ------------------------------------------------------------------------
# SINGLE LOOKUP: Map HSP_ACCOUNT_ID → PAT_ENC_CSN_ID(s)
# This view is used by all downstream queries
# ------------------------------------------------------------------------
# ========================================================================
# STEP 1: Get the target encounter details (start/end times)
# ========================================================================

spark.sql(f"""
```

```
CREATE OR REPLACE TEMP VIEW target_encounter AS
SELECT
    peh.HSP_ACCOUNT_ID,
    peh.PAT_ENC_CSN_ID,
    peh.PAT_ID,
    peh.ADM_DATE_TIME AS ENCOUNTER_START,
    peh.DISCH_DATE_TIME AS ENCOUNTER_END
FROM prod.clarity_cur.pat_enc_hsp_har_enh peh
WHERE peh.HSP_ACCOUNT_ID = {HSP_ACCOUNT_ID}
""")

# Cache it since we'll hit it repeatedly
spark.sql("CACHE TABLE target_encounter")

# Table names
LABS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_labs"
VITALS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_vitals"
MEDS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_meds"
PROCEDURES_TABLE = f"{trgt_cat}.fin_ds.creamsicle_procedures"
ICD10_TABLE = f"{trgt_cat}.fin_ds.creamsicle_icd10"
MERGED_TABLE = f"{trgt_cat}.fin_ds.creamsicle_structured_timeline"
LLM_TIMELINE_TABLE = f"{trgt_cat}.fin_ds.creamsicle_llm_timeline_{HSP_ACCOUNT_ID}"

print(f"Account ID: {HSP_ACCOUNT_ID}")
print(f"Catalog: {trgt_cat}")
# ============================================================================
# STEP 2: Create Tables (run once)
# ============================================================================
CREATE_TABLES = True  # Set to True on first run
```

```
if CREATE_TABLES:
    spark.sql(f"""
        CREATE TABLE IF NOT EXISTS {LABS_TABLE} (
            HSP_ACCOUNT_ID BIGINT,
            PAT_ENC_CSN_ID BIGINT,
            EVENT_TIMESTAMP TIMESTAMP,
            LAB_NAME STRING,
            lab_value STRING,
            lab_units STRING,
            reference_range STRING,
            abnormal_flag STRING
        ) USING DELTA
    """)

    spark.sql(f"""
        CREATE TABLE IF NOT EXISTS {VITALS_TABLE} (
            HSP_ACCOUNT_ID BIGINT,
            PAT_ENC_CSN_ID BIGINT,
            EVENT_TIMESTAMP TIMESTAMP,
            VITAL_NAME STRING,
            vital_value STRING,
            vital_units STRING
        ) USING DELTA
    """)

    spark.sql(f"""
        CREATE TABLE IF NOT EXISTS {MEDS_TABLE} (
            HSP_ACCOUNT_ID BIGINT,
            PAT_ENC_CSN_ID BIGINT,
```

```
    EVENT_TIMESTAMP TIMESTAMP,
    MED_NAME STRING,
    MED_DOSE STRING,
    MED_UNITS STRING,
    MED_ROUTE STRING,
    ADMIN_ACTION STRING
  ) USING DELTA
""")

spark.sql(f"""
  CREATE TABLE IF NOT EXISTS {PROCEDURES_TABLE} (
    HSP_ACCOUNT_ID BIGINT,
    PAT_ENC_CSN_ID BIGINT,
    EVENT_TIMESTAMP TIMESTAMP,
    PROCEDURE_NAME STRING,
    PROCEDURE_CODE STRING
  ) USING DELTA
""")

spark.sql(f"""
  CREATE TABLE IF NOT EXISTS {ICD10_TABLE} (
    HSP_ACCOUNT_ID BIGINT,
    PAT_ENC_CSN_ID BIGINT,
    ICD10_CODE STRING,
    icd10_description STRING,
    TIMING_CATEGORY STRING,
    availability_time STRING,
    EVENT_TIMESTAMP TIMESTAMP,
    source STRING
  ) USING DELTA
```

```
    """)

    spark.sql(f"""
        CREATE TABLE IF NOT EXISTS {MERGED_TABLE} (
            HSP_ACCOUNT_ID BIGINT,
            PAT_ENC_CSN_ID BIGINT,
            EVENT_TIMESTAMP TIMESTAMP,
            event_type STRING,
            event_detail STRING
        ) USING DELTA
    """)

    spark.sql(f"""
        CREATE TABLE IF NOT EXISTS {LLM_TIMELINE_TABLE} (
            HSP_ACCOUNT_ID BIGINT,
            PAT_ENC_CSN_ID BIGINT,
            section_type STRING,
            sort_order INT,
            event_timestamp TIMESTAMP,
            content STRING
        ) USING DELTA
    """)

    print("Tables created")

# ============================================================================
# STEP 3: Query All Labs
# ============================================================================
print(f"\n{'='*60}")
```

```
print(f"QUERYING ALL LABS FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TABLE {LABS_TABLE} AS
SELECT
    t.HSP_ACCOUNT_ID AS HSP_ACCOUNT_ID,
    t.PAT_ENC_CSN_ID AS PAT_ENC_CSN_ID,
    CAST(res_comp.COMP_VERIF_DTTM AS TIMESTAMP) AS EVENT_TIMESTAMP,
    cc.NAME AS LAB_NAME,
    CAST(REGEXP_REPLACE(res_comp.COMPONENT_VALUE, '>', '') AS STRING) AS lab_value,
    res_comp.component_units AS lab_units,
    CAST(NULL AS STRING) AS reference_range, -- Explicitly cast NULL to STRING # new line
    -- NULL AS reference_range, # Original
    zsab.NAME AS abnormal_flag

FROM target_encounter t

INNER JOIN prod.clarity.order_proc op
    ON t.PAT_ENC_CSN_ID = op.PAT_ENC_CSN_ID

INNER JOIN prod.clarity.RES_DB_MAIN rdm
    ON rdm.RES_ORDER_ID = op.ORDER_PROC_ID

INNER JOIN prod.clarity.res_components res_comp
    ON res_comp.result_id = rdm.result_id

INNER JOIN prod.clarity.clarity_component cc
    ON cc.component_id = res_comp.component_id
```

```
LEFT JOIN prod.clarity.zc_stat_abnorms zsab
    ON zsab.stat_abnorms_c = res_comp.component_abn_c

WHERE op.order_status_c = 5
  AND op.lab_status_c IN (3, 5)
  AND rdm.res_val_status_c = 9
  AND res_comp.COMPONENT_VALUE IS NOT NULL
  AND res_comp.COMPONENT_VALUE <> '-1'

ORDER BY res_comp.COMP_VERIF_DTTM ASC
""")

print(f"All labs written to {LABS_TABLE}")

# ============================================================================
# STEP 4: Query Vitals
# ============================================================================
print(f"\n{'='*60}")
print(f"QUERYING VITALS FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TABLE {VITALS_TABLE} AS
SELECT
    t.HSP_ACCOUNT_ID AS HSP_ACCOUNT_ID,
    t.PAT_ENC_CSN_ID AS PAT_ENC_CSN_ID,
    CAST(to_timestamp(substring(v.RECORDED_TIME, 1, 19), 'yyyy-MM-dd HH:mm:ss')
AS TIMESTAMP) AS EVENT_TIMESTAMP,
    v.FLO_MEAS_NAME AS VITAL_NAME,
    v.MEAS_VALUE AS vital_value
```

```
    -- Removed vital_units from here
FROM target_encounter t
INNER JOIN prod.clarity_cur.ip_flwsht_rec_enh v
    ON t.PAT_ENC_CSN_ID = v.IP_DATA_STORE_EPT_CSN
WHERE v.FLO_MEAS_ID IN ('5', '6', '8', '9', '10', '11', '14')
  AND v.MEAS_VALUE IS NOT NULL
ORDER BY EVENT_TIMESTAMP ASC
""")

print(f"Vitals written to {VITALS_TABLE}")

# ============================================================================
# STEP 5: Query Medications
# ============================================================================
print(f"\n{'='*60}")
print(f"QUERYING ALL MEDICATIONS FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TABLE {MEDS_TABLE} AS
SELECT
    t.HSP_ACCOUNT_ID AS HSP_ACCOUNT_ID,
    t.PAT_ENC_CSN_ID AS PAT_ENC_CSN_ID,
    CAST(mar.TAKEN_TIME AS TIMESTAMP) AS EVENT_TIMESTAMP,
    om.SIMPLE_GENERIC_NAME AS MED_NAME,
    CAST(om.HV_DISCRETE_DOSE AS STRING) AS MED_DOSE,
    om.DOSE_UNIT AS MED_UNITS,
    mar.ROUTE AS MED_ROUTE,
    mar.ACTION AS ADMIN_ACTION
```

```
FROM target_encounter t

INNER JOIN prod.clarity_cur.order_med_enh om
    ON t.PAT_ENC_CSN_ID = om.PAT_ENC_CSN_ID

INNER JOIN prod.clarity_cur.mar_admin_info_enh mar
    ON om.ORDER_MED_ID = mar.ORDER_MED_ID

WHERE mar.ACTION IN (
    'Given',
    'Patient/Family Admin',
    'Given-See Override',
    'Admin by Another Clinician (Comment)',
    'New Bag',
    'Bolus',
    'Push',
    'Started by Another Clinician',
    'Bag Switched',
    'Clinic Sample Administered',
    'Applied',
    'Feeding Started',
    'Acknowledged',
    'Contrast Given',
    'New Bag-See Override',
    'Bolus from Bag'
)

ORDER BY EVENT_TIMESTAMP ASC
""")
```

```
print(f"All medications written to {MEDS_TABLE}")

# ============================================================================
# STEP 6: Query Procedures
# ============================================================================
print(f"\n{'='*60}")
print(f"QUERYING PROCEDURES FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TABLE {PROCEDURES_TABLE} AS
SELECT
    t.HSP_ACCOUNT_ID AS HSP_ACCOUNT_ID,
    t.PAT_ENC_CSN_ID AS PAT_ENC_CSN_ID,
    CAST(op.PROC_START_TIME AS TIMESTAMP) AS EVENT_TIMESTAMP,
    op.PROC_NAME AS PROCEDURE_NAME,
    CAST(op.ORDER_PROC_ID AS STRING) AS PROCEDURE_CODE

FROM target_encounter t

INNER JOIN prod.clarity_cur.order_proc_enh op
    ON t.PAT_ENC_CSN_ID = op.PAT_ENC_CSN_ID

WHERE op.PROC_NAME IN (
    'DIET NPO',
    'DIET TUBE FEEDING',
    'DIET CALORIE CONTROLLED',
    'DIET COMPLEX CARB',
```

```
    'DIET KETOGENIC',
    'DIET DIABETIC'
)
AND op.ORDER_STATUS_C = 5
AND op.PROC_START_TIME IS NOT NULL

ORDER BY EVENT_TIMESTAMP ASC
""")

print(f"Procedures written to {PROCEDURES_TABLE}")

# ============================================================================
# STEP 7: Query ICD-10 Codes
# ============================================================================
print(f"\n{'='*60}")
print(f"QUERYING ALL ICD-10 CODES FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TEMP VIEW encounter_icd10_codes AS

-- SOURCE 1: Outpatient encounter diagnoses
SELECT
    te.HSP_ACCOUNT_ID,
    te.PAT_ENC_CSN_ID,
    dd.ICD10_CODE,
    edg.DX_NAME AS icd10_description,
    CAST(pe.CONTACT_DATE AS TIMESTAMP) AS EVENT_TIMESTAMP,
    CASE
        WHEN pe.CONTACT_DATE < te.ENCOUNTER_START THEN 'BEFORE'
```

```
        WHEN pe.CONTACT_DATE >= te.ENCOUNTER_START
            AND (te.ENCOUNTER_END IS NULL OR pe.CONTACT_DATE <= te.ENCOUNTER_
END) THEN 'DURING'
        ELSE 'AFTER'
    END AS TIMING_CATEGORY,
    'OUTPATIENT_ENC_DX' AS source
FROM target_encounter te
JOIN prod.clarity_cur.pat_enc_dx_enh dd
    ON dd.PAT_ENC_CSN_ID = te.PAT_ENC_CSN_ID
JOIN prod.clarity_cur.pat_enc_enh pe
    ON pe.PAT_ENC_CSN_ID = dd.PAT_ENC_CSN_ID
LEFT JOIN prod.clarity.clarity_edg edg
    ON dd.DX_ID = edg.DX_ID
WHERE dd.ICD10_CODE IS NOT NULL

UNION ALL

-- SOURCE 2: Inpatient hospital account diagnoses
SELECT
    te.HSP_ACCOUNT_ID,
    te.PAT_ENC_CSN_ID,
    dx.CODE AS ICD10_CODE,
    edg.DX_NAME AS icd10_description,
    CAST(ha.DISCH_DATE_TIME AS TIMESTAMP) AS EVENT_TIMESTAMP,
    CASE
        WHEN ha.DISCH_DATE_TIME < te.ENCOUNTER_START THEN 'BEFORE'
        WHEN ha.DISCH_DATE_TIME >= te.ENCOUNTER_START
            AND (te.ENCOUNTER_END IS NULL OR ha.DISCH_DATE_TIME <= te.
ENCOUNTER_END) THEN 'DURING'
        ELSE 'AFTER'
```

```
    END AS TIMING_CATEGORY,
    'INPATIENT_ACCT_DX' AS source
FROM target_encounter te
JOIN prod.clarity_cur.hsp_acct_dx_list_enh dx
    ON dx.PAT_ID = te.PAT_ID
JOIN prod.clarity_cur.pat_enc_hsp_har_enh ha
    ON ha.HSP_ACCOUNT_ID = dx.HSP_ACCOUNT_ID
LEFT JOIN prod.clarity.clarity_edg edg
    ON dx.DX_ID = edg.DX_ID
WHERE dx.CODE IS NOT NULL

UNION ALL

-- SOURCE 3: Problem list history
SELECT
    te.HSP_ACCOUNT_ID,
    te.PAT_ENC_CSN_ID,
    phx.HX_PROBLEM_ICD10_CODE AS ICD10_CODE,
    phx.HX_PROBLEM_DX_NAME AS icd10_description,  -- Use the name already in the table
    CAST(phx.HX_DATE_OF_ENTRY AS TIMESTAMP) AS EVENT_TIMESTAMP,
    CASE
        WHEN phx.HX_DATE_OF_ENTRY < te.ENCOUNTER_START THEN 'BEFORE'
        WHEN phx.HX_DATE_OF_ENTRY >= te.ENCOUNTER_START
            AND (te.ENCOUNTER_END IS NULL OR phx.HX_DATE_OF_ENTRY <= te.
ENCOUNTER_END) THEN 'DURING'
        ELSE 'AFTER'
    END AS TIMING_CATEGORY,
    'PROBLEM_LIST' AS source
FROM target_encounter te
JOIN prod.clarity_cur.problem_list_hx_enh phx
```

```
    ON phx.PAT_ID = te.PAT_ID
WHERE phx.HX_PROBLEM_ICD10_CODE IS NOT NULL
  AND phx.HX_STATUS = 'Active'
""")

# Final output - exclude AFTER, format timing appropriately
spark.sql(f"""
CREATE OR REPLACE TABLE {ICD10_TABLE} AS
SELECT
    HSP_ACCOUNT_ID,
    PAT_ENC_CSN_ID,
    ICD10_CODE,
    icd10_description,
    TIMING_CATEGORY,
    CASE
        WHEN TIMING_CATEGORY = 'BEFORE' THEN 'Available at encounter start'
        WHEN TIMING_CATEGORY = 'DURING' THEN CAST(EVENT_TIMESTAMP AS STRING)
    END AS availability_time,
    EVENT_TIMESTAMP,
    source
FROM encounter_icd10_codes
WHERE TIMING_CATEGORY != 'AFTER'
ORDER BY
    TIMING_CATEGORY DESC,
    EVENT_TIMESTAMP ASC
""")

print(f"All ICD-10 codes written to {ICD10_TABLE}")

# ============================================================================
```

```
# STEP 8: Merge All Events into Chronological Timeline (with deduplication)
# ============================================================================
print(f"\n{'='*60}")
print(f"CREATING UNIFIED TIMELINE FOR {HSP_ACCOUNT_ID} WITH DEDUPLICATION")
print(f"{'='*60}")

spark.sql(f"""
CREATE OR REPLACE TABLE {MERGED_TABLE} AS
WITH RawMergedEvents AS (
    -- Labs
    SELECT
        HSP_ACCOUNT_ID,
        PAT_ENC_CSN_ID,
        EVENT_TIMESTAMP,
        'LAB' AS event_type,
        CONCAT(
            LAB_NAME, ': ', lab_value, ' ', COALESCE(lab_units, ''),
            CASE WHEN abnormal_flag IS NOT NULL THEN CONCAT(' (', abnormal_flag, ')') ELSE '' END
        ) AS event_detail,
        CASE WHEN abnormal_flag IS NOT NULL THEN 1 ELSE 0 END AS is_abnormal
    FROM {LABS_TABLE}

    UNION ALL

    -- Vitals
    SELECT
        HSP_ACCOUNT_ID,
        PAT_ENC_CSN_ID,
        EVENT_TIMESTAMP,
```

```
    'VITAL' AS event_type,
    CONCAT(VITAL_NAME, ': ', vital_value) AS event_detail,
    0 AS is_abnormal
FROM {VITALS_TABLE}

UNION ALL

-- Medications
SELECT
    HSP_ACCOUNT_ID,
    PAT_ENC_CSN_ID,
    EVENT_TIMESTAMP,
    'MEDICATION' AS event_type,
    CONCAT(
        MED_NAME, ' ', COALESCE(MED_DOSE, ''), ' ', COALESCE(MED_UNITS, ''),
        ' via ', COALESCE(MED_ROUTE, 'unknown route'),
        ' - ', ADMIN_ACTION
    ) AS event_detail,
    0 AS is_abnormal
FROM {MEDS_TABLE}

UNION ALL

-- Procedures
SELECT
    HSP_ACCOUNT_ID,
    PAT_ENC_CSN_ID,
    EVENT_TIMESTAMP,
    'PROCEDURE' AS event_type,
    PROCEDURE_NAME AS event_detail,
```

```
        0 AS is_abnormal
    FROM {PROCEDURES_TABLE}

    UNION ALL

    -- ICD-10 codes (only those DURING encounter with timestamps)
    SELECT
        HSP_ACCOUNT_ID,
        PAT_ENC_CSN_ID,
        EVENT_TIMESTAMP,
        'DIAGNOSIS' AS event_type,
        CONCAT(
            ICD10_CODE, ' - ', COALESCE(icd10_description, 'Unknown'),
            ' (', source, ')'
        ) AS event_detail,
        0 AS is_abnormal
    FROM {ICD10_TABLE}
    WHERE TIMING_CATEGORY = 'DURING'
      AND EVENT_TIMESTAMP IS NOT NULL
),
DeduplicatedEvents AS (
    SELECT
        *,
        ROW_NUMBER() OVER (PARTITION BY HSP_ACCOUNT_ID, PAT_ENC_CSN_ID, EVENT_
TIMESTAMP, event_type, event_detail ORDER BY EVENT_TIMESTAMP) as rn
    FROM RawMergedEvents
)
SELECT
    HSP_ACCOUNT_ID,
    PAT_ENC_CSN_ID,
```

```
    EVENT_TIMESTAMP,
    event_type,
    event_detail,
    is_abnormal
FROM DeduplicatedEvents
WHERE rn = 1
ORDER BY EVENT_TIMESTAMP ASC
""")

print(f"Unified timeline (deduplicated) written to {MERGED_TABLE}")

# ============================================================================
# STEP 9: Create LLM-Ready Timeline Table (with all pre-existing diagnoses and their original timestamps)
# ============================================================================

spark.sql(f"""
CREATE OR REPLACE TABLE {LLM_TIMELINE_TABLE} AS
WITH encounter_context AS (
    SELECT
        HSP_ACCOUNT_ID,
        PAT_ENC_CSN_ID,
        ENCOUNTER_START,
        ENCOUNTER_END,
        'ENCOUNTER_INFO' AS section_type,
        0 AS sort_order,
        NULL AS event_timestamp,
        CONCAT(
            'Account ID: ', HSP_ACCOUNT_ID, '\n',
            'Encounter: ', PAT_ENC_CSN_ID, '\n',
```

```
            'Admission: ', ENCOUNTER_START, '\n',
            'Discharge: ', COALESCE(CAST(ENCOUNTER_END AS STRING), 'Still admitted')
        ) AS content
    FROM target_encounter
),

pre_existing_dx AS (
    SELECT
        icd.HSP_ACCOUNT_ID,
        icd.PAT_ENC_CSN_ID,
        'PRE_EXISTING_DX' AS section_type,
        1 AS sort_order,
        -- Use the original event timestamp from the ICD10 table
        icd.EVENT_TIMESTAMP AS event_timestamp,
        CONCAT(
            ICD10_CODE, ': ',
            COALESCE(icd.icd10_description, 'Unknown'),
            ' (', source, ')'
        ) AS content
    FROM {ICD10_TABLE} icd
    -- We still join to target_encounter to ensure we're only looking at diagnoses
    -- relevant to the specific encounter's patient, though ENCOUNTER_START isn't
used for filtering here.
    JOIN target_encounter te
        ON icd.PAT_ENC_CSN_ID = te.PAT_ENC_CSN_ID
    WHERE icd.TIMING_CATEGORY = 'BEFORE'
    -- Removed the recency filter here to include all 'BEFORE' diagnoses
),

timeline_events AS (
```

```
    SELECT
      m.HSP_ACCOUNT_ID,
      m.PAT_ENC_CSN_ID,
      'TIMELINE_EVENT' AS section_type,
      2 AS sort_order,
      m.EVENT_TIMESTAMP AS event_timestamp,
      CONCAT(
        '[', CAST(m.EVENT_TIMESTAMP AS STRING), '] ',
        UPPER(m.event_type), ': ',
        m.event_detail
      ) AS content
    FROM {MERGED_TABLE} m
    -- IMPORTANT: Keep this exclusion filter to prevent duplication.
    -- Diagnoses originally 'BEFORE' are now handled by the pre_existing_dx CTE.
    WHERE NOT (m.event_type = 'DIAGNOSIS' AND m.event_detail LIKE '% - Pre-existing%')
)

SELECT
    HSP_ACCOUNT_ID,
    PAT_ENC_CSN_ID,
    section_type,
    sort_order,
    event_timestamp,
    content
FROM encounter_context

UNION ALL

SELECT * FROM pre_existing_dx
```

```
UNION ALL

SELECT * FROM timeline_events

ORDER BY sort_order, event_timestamp NULLS FIRST
""")

print(f"LLM-ready timeline (with all pre-existing diagnoses) written to {LLM_TIMELINE_TABLE}")



# ============================================================================
# STEP 10: View Results
# ============================================================================
print(f"\n{'='*60}")
print(f"RESULTS FOR {HSP_ACCOUNT_ID}")
print(f"{'='*60}")

# Uncomment to view results:
display(spark.sql(f"SELECT * FROM dev.fin_ds.creamsicle_llm_timeline_{HSP_ACCOUNT_ID}
ORDER BY EVENT_TIMESTAMP"))

print("Done.")
```

inference.py

```
# model/inference.py
# Acute Respiratory Failure Appeal Engine - Appeal Letter Generation
#
```

```
# GENERATION PIPELINE (reads prepared data from case tables):
# 1. Load knowledge base (gold letters, Propel definitions written by featurization_train.py)
# 2. Read case data from tables (written by featurization_inference.py)
# 3. Vector search → Find best matching gold letter
# 4. Generate appeal letter → Using prepared clinical data
# 5. Assess strength → Evaluate against criteria
# 6. Export to DOCX → With conflicts appendix if applicable
#
# PREREQUISITE: Run featurization_inference.py first to prepare case data.
#
# Run on Databricks Runtime 15.4 LTS ML


# ============================================================================
# CELL 0: Install Dependencies (run this cell FIRST, then restart)
# ============================================================================
# IMPORTANT: Run this cell by itself, then run the rest of the notebook.
# After restart, the packages persist for the cluster session.
#
# Uncomment and run ONCE per cluster session:
# %pip install azure-ai-documentintelligence==1.0.2 openai python-docx
# dbutils.library.restartPython()


# ============================================================================
# CELL 1: Configuration
# ============================================================================
import os
import math
import json
import re
from datetime import datetime
```

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

# ------------------------------------------------------------------------
# Processing Configuration
# ------------------------------------------------------------------------
SCOPE_FILTER = "arf"
ARF_DRG_CODES = ["189", "190", "191", "207", "208"]
MATCH_SCORE_THRESHOLD = 0.7

# Default template path
DEFAULT_TEMPLATE_PATH = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/
gold_standard_appeals/gold_standard_appeals_arf_only/default_respiratory_failure_template.docx"

# Output configuration
EXPORT_TO_DOCX = True
DOCX_OUTPUT_BASE = "/Workspace/Repos/dimo6213@mercy.net/creamsicle/utils/outputs"

# ------------------------------------------------------------------------
# Environment Setup
# ------------------------------------------------------------------------
# NOTE: Data lives in prod catalog, but we write to our environment's catalog.
# This is intentional - we query from prod but can only write to our own env.
trgt_cat = os.environ.get('trgt_cat', 'dev')
spark.sql('USE CATALOG prod;')

# Knowledge base tables
GOLD_LETTERS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_gold_letters"
PROPEL_DATA_TABLE = f"{trgt_cat}.fin_ds.creamsicle_propel_data"
```

```
# Case data tables (written by featurization_inference.py)
CASE_DENIAL_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_denial"
CASE_CLINICAL_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_clinical"
CASE_STRUCTURED_SUMMARY_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_structured_summary"
CASE_CONFLICTS_TABLE = f"{trgt_cat}.fin_ds.creamsicle_case_conflicts"

print(f"Catalog: {trgt_cat}")

# ============================================================================
# CELL 2: Azure Credentials and Client
# ============================================================================
from openai import AzureOpenAI

AZURE_OPENAI_KEY = dbutils.secrets.get(scope='idp_etl', key='az-openai-key1')
AZURE_OPENAI_ENDPOINT = dbutils.secrets.get(scope='idp_etl', key='az-openai-base')

openai_client = AzureOpenAI(
    api_key=AZURE_OPENAI_KEY,
    azure_endpoint=AZURE_OPENAI_ENDPOINT,
    api_version="2024-10-21"
)

print("Azure OpenAI client initialized")

# ============================================================================
# CELL 3: Load Knowledge Base
# ============================================================================
print("\n" + "="*60)
```

```
print("LOADING KNOWLEDGE BASE")
print("="*60)

# Load gold letters
print("\nLoading gold standard letters...")
gold_letters_cache = []
try:
    gold_letters_df = spark.sql(f"""
        SELECT letter_id, source_file, payor, denial_text, rebuttal_text, denial_embedding, metadata
        FROM {GOLD_LETTERS_TABLE}
    """)
    gold_letters = gold_letters_df.collect()

    gold_letters_cache = [
        {
            "letter_id": row["letter_id"],
            "source_file": row["source_file"],
            "payor": row["payor"],
            "denial_text": row["denial_text"],
            "appeal_text": row["rebuttal_text"],
            "denial_embedding": list(row["denial_embedding"]) if row["denial_embedding"] else None,
            "metadata": dict(row["metadata"]) if row["metadata"] else {},
        }
        for row in gold_letters
    ]
    print(f"  Loaded {len(gold_letters_cache)} gold standard letters")
except Exception as e:
    print(f"  Warning: Could not load gold letters: {e}")

# Load default template
```

```
print("\nLoading default template...")
default_template = None
try:
    if os.path.exists(DEFAULT_TEMPLATE_PATH):
        from docx import Document as DocxDocument
        doc = DocxDocument(DEFAULT_TEMPLATE_PATH)
        template_text = "\n\n".join([para.text for para in doc.paragraphs if para.text.strip()])

        default_template = {
            "letter_id": "default_template",
            "source_file": os.path.basename(DEFAULT_TEMPLATE_PATH),
            "payor": "Generic",
            "denial_text": None,
            "appeal_text": template_text,
            "denial_embedding": None,
            "metadata": {"is_default_template": "true"},
        }
        print(f"  Loaded default template: {len(template_text)} chars")
    else:
        print(f"  Warning: Default template not found at {DEFAULT_TEMPLATE_PATH}")
except Exception as e:
    print(f"  Warning: Could not load default template: {e}")

# Load Propel definitions
print("\nLoading Propel clinical definitions...")
propel_definitions = {}
try:
    propel_df = spark.sql(f"""
        SELECT condition_name, definition_summary, definition_text
```

```
        FROM {PROPEL_DATA_TABLE}
    """)
    for row in propel_df.collect():
        definition = row["definition_summary"] or row["definition_text"]
        propel_definitions[row["condition_name"]] = definition
        print(f"  {row['condition_name']}: {len(definition)} chars")
except Exception as e:
    print(f"  Warning: Could not load propel definitions: {e}")

# ============================================================================
# CELL 4: Load Case Data from Tables
# ============================================================================
print("\n" + "="*60)
print("LOADING CASE DATA FROM TABLES")
print("="*60)

def load_case_data():
    """Load all case data from tables written by featurization_inference.py."""
    case_data = {}

    # Load denial data
    print("\nLoading denial data...")
    try:
        denial_row = spark.sql(f"SELECT * FROM {CASE_DENIAL_TABLE}").collect()
        if denial_row:
            row = denial_row[0]
            case_data["account_id"] = row["account_id"]
            case_data["denial_text"] = row["denial_text"]
            case_data["denial_embedding"] = list(row["denial_embedding"]) if row["
denial_embedding"] else None
```

```
            case_data["payor"] = row["payor"]
            case_data["original_drg"] = row["original_drg"]
            case_data["proposed_drg"] = row["proposed_drg"]
            case_data["is_arf"] = row["is_arf"]
            print(f"  Account: {case_data['account_id']}")
            print(f"  Payor: {case_data['payor']}")
            print(f"  ARF: {case_data['is_arf']}")
        else:
            print("  ERROR: No denial data found. Run featurization_inference.py first.")
            return None
    except Exception as e:
        print(f"  ERROR: Could not load denial data: {e}")
        return None

    # Load clinical data
    print("\nLoading clinical data...")
    try:
        clinical_row = spark.sql(f"SELECT * FROM {CASE_CLINICAL_TABLE}").collect()
        if clinical_row:
            row = clinical_row[0]
            case_data["patient_name"] = row["patient_name"]
            case_data["patient_dob"] = row["patient_dob"]
            case_data["facility_name"] = row["facility_name"]
            case_data["date_of_service"] = row["date_of_service"]
            case_data["extracted_notes"] = json.loads(row["extracted_notes"]) if row["extracted_notes"]
else {}
            print(f"  Patient: {case_data['patient_name']}")
            print(f"  Notes loaded: {len([v for v in case_data['extracted_notes'].values() if v != 'Not
available'])}")
        else:
```

```
            print("  WARNING: No clinical data found")
            case_data["patient_name"] = "Unknown"
            case_data["patient_dob"] = ""
            case_data["facility_name"] = "Mercy Hospital"
            case_data["date_of_service"] = ""
            case_data["extracted_notes"] = {}
    except Exception as e:
        print(f"  WARNING: Could not load clinical data: {e}")
        case_data["extracted_notes"] = {}

    # Load structured summary
    print("\nLoading structured data summary...")
    try:
        structured_row = spark.sql(f"SELECT * FROM {CASE_STRUCTURED_SUMMARY_TABLE}
").collect()
        if structured_row:
            case_data["structured_summary"] = structured_row[0]["structured_summary"]
            print(f"  Summary: {len(case_data['structured_summary'])} chars")
        else:
            print("  WARNING: No structured summary found")
            case_data["structured_summary"] = "No structured data available."
    except Exception as e:
        print(f"  WARNING: Could not load structured summary: {e}")
        case_data["structured_summary"] = "No structured data available."

    # Load conflicts
    print("\nLoading conflicts...")
    try:
        conflicts_row = spark.sql(f"SELECT * FROM {CASE_CONFLICTS_TABLE}").collect()
        if conflicts_row:
```

```
            row = conflicts_row[0]
            conflicts_list = json.loads(row["conflicts"]) if row["conflicts"] else []
            case_data["conflicts"] = {
                "conflicts": conflicts_list,
                "recommendation": row["recommendation"] or ""
            }
            print(f"  Conflicts: {len(conflicts_list)}")
        else:
            print("  No conflicts data found")
            case_data["conflicts"] = {"conflicts": [], "recommendation": ""}
    except Exception as e:
        print(f"  WARNING: Could not load conflicts: {e}")
        case_data["conflicts"] = {"conflicts": [], "recommendation": ""}

    return case_data

# Load case data
case_data = load_case_data()

if not case_data:
    print("\n" + "="*60)
    print("ERROR: Case data not found. Run featurization_inference.py first.")
    print("="*60)
else:
    # ============================================================
    # CELL 5: Vector Search Function
    # ============================================================
```

```
    def find_best_gold_letter(denial_embedding):
        """Find the best matching gold letter using cosine similarity."""
        if not gold_letters_cache or not denial_embedding:
            return None, 0.0

        best_score = 0.0
        best_match = None

        for letter in gold_letters_cache:
            if letter["denial_embedding"]:
                vec1 = denial_embedding
                vec2 = letter["denial_embedding"]
                dot_product = sum(a * b for a, b in zip(vec1, vec2))
                norm1 = math.sqrt(sum(a * a for a in vec1))
                norm2 = math.sqrt(sum(b * b for b in vec2))
                if norm1 > 0 and norm2 > 0:
                    similarity = dot_product / (norm1 * norm2)
                    if similarity > best_score:
                        best_score = similarity
                        best_match = letter

        if best_match and best_score >= MATCH_SCORE_THRESHOLD:
            return best_match, best_score
        elif default_template:
            return default_template, 0.0
        else:
            return None, 0.0

    # ================================================================
    ======================
```

```
# CELL 5B: Rebuttal Templates

# ======================================================
======================


# Rebuttal for denials requiring CONSECUTIVE/SEQUENTIAL SpO2 readings

CONSECUTIVE_SPO2_REBUTTAL = '''Apply ONLY if the denial requires "consecutive," "
sequential," or "back-to-back" SpO2 readings.


Rebuttal points:

1. No CMS, ICD-10-CM, or Coding Clinic requirement mandates multiple sequential
SpO2 readings for acute hypoxemic respiratory failure.

2. Consecutive reading requirements are proprietary payor criteria—not nationally
recognized standards.

3. A single documented SpO2 <91% on room air with clinical context and
treatment response supports the diagnosis.


Cite ALL relevant timestamped low SpO2 readings, even if not consecutive.'''


# Rebuttal for denials requiring PERSISTENT/CONTINUOUS symptoms

PERSISTENT_SYMPTOMS_REBUTTAL = '''Apply ONLY if the denial argues symptoms
were not "persistent," "continuous," or "sustained."


Rebuttal points:

1. Acute respiratory failure is defined by onset and severity at presentation—
not by unchanged symptoms throughout the stay.

2. Improvement with treatment confirms appropriate intervention, not absence of the condition.

3. Per CMS/Coding Clinic, a diagnosis is reportable when documented by the
provider and supported by clinical indicators and treatment at presentation.


Cite documentation of respiratory distress at presentation and interventions required.'''


# Rebuttal for denials imposing proprietary clinical thresholds generally

PROPRIETARY_CRITERIA_REBUTTAL = '''Apply if the denial imposes clinical
thresholds beyond provider documentation and nationally recognized standards.
```

Rebuttal points:

1. DRG validation confirms clinical support for documented diagnoses—it does not substitute proprietary payor thresholds for physician judgment.

2. Per CMS/AHIMA/Coding Clinic, a diagnosis is reportable when (a) documented by the provider and (b) clinically supported by patient-specific indicators and treatment.

3. Internal payor criteria exceeding CMS standards are not valid grounds for DRG reassignment.'"

# ========== ==========

# CELL 6: Writer Prompt

# ========== ==========

WRITER_PROMPT = '"You are a clinical coding expert writing a DRG validation appeal letter for Mercy Hospital.

# CRITICAL: Source of Truth for Clinical Criteria and References

The ONLY authoritative source for clinical definitions, diagnostic criteria, and approved references is the **PROPEL CRITERIA** section below.

When writing the appeal letter:

- Use clinical definitions and diagnostic thresholds ONLY as defined in the Propel document.

- You may cite references from the Propel document's reference list.

- DO NOT list or comment on the payor's cited references unless directly refuting a specific clinical claim.

The denial letter should be used to:

- Extract payor address, reviewer name, and claim numbers

- Understand the payor's specific arguments (so you can refute them)

# CONDITIONAL REBUTTALS - Apply ONLY if the denial matches the specific scenario

## Rebuttal A: Consecutive/Sequential SpO2 Readings

{consecutive_spo2_rebuttal}

## Rebuttal B: Persistent/Continuous Symptoms Requirement

{persistent_symptoms_rebuttal}

## Rebuttal C: Proprietary Clinical Criteria (General)

{proprietary_criteria_rebuttal}

**IMPORTANT**: Read the denial carefully. Apply ONLY the rebuttal(s) that match the payor's actual argument. Do not apply rebuttals for arguments the payor did not make.

# Original Denial Letter

{denial_letter_text}

# Clinical Notes (PRIMARY EVIDENCE - from physician documentation)

## Discharge Summary

{discharge_summary}

## H&P Note

{hp_note}

## Progress Notes

{progress_note}

## Consult Notes

{consult_note}

## ED Notes

{ed_notes}

## Initial Assessments

{initial_assessment}

## ED Triage Notes

{ed_triage}

## ED Provider Notes

{ed_provider_note}

## Addendum Note

{addendum_note}

## Hospital Course

{hospital_course}

## Subjective & Objective

{subjective_objective}

## Assessment & Plan Note

{assessment_plan}

## Nursing Note

{nursing_note}

## Code Documentation

{code_documentation}

# Structured Data Summary (SUPPORTING EVIDENCE - from labs, vitals, meds)

{structured_data_summary}

# PROPEL CRITERIA (AUTHORITATIVE SOURCE - USE THIS ONLY)

{clinical_definition_section}

# Gold Standard Letter

{gold_letter_section}

# Patient Information

Name: {patient_name}

DOB: {patient_dob}

Hospital Account #: {hsp_account_id}

Date of Service: {date_of_service}

Facility: {facility_name}

Original DRG: {original_drg}

Proposed DRG: {proposed_drg}

Payor: {payor}

# Instructions

{gold_letter_instructions}

1. READ THE DENIAL LETTER carefully. Identify the payor's SPECIFIC arguments. Apply ONLY the rebuttals that directly address those arguments—do not include rebuttals for claims the payor did not make.

2. ADDRESS EACH DENIAL ARGUMENT - quote the payer's argument, then refute using ONLY the Propel criteria. Do not list or critique the payor's references unless necessary to refute a specific clinical claim.

3. CITE CLINICAL EVIDENCE from provider notes FIRST, then structured data

4. INCLUDE TIMESTAMPS with clinical values

5. QUANTIFY ACUTE RESPIRATORY FAILURE using diagnostic criteria from the Propel guidelines when available:

- Reference specific values: PaO2, SpO2, PaCO2, pH, FiO2, P/F ratio

- Hypoxic criteria: PaO2 < 60 mmHg, SpO2 < 91% on room air, P/F ratio < 300

- Hypercapnic criteria: PaCO2 > 50 mmHg with pH < 7.35

- Acute-on-chronic indicators: ≥ 10 mmHg change from baseline PaO2 or PaCO2

- Document oxygen delivery method and escalation of respiratory support

6. Follow the Mercy Hospital template structure exactly

# Formatting Requirements

- DO NOT include a "Summary Table of Key Clinical Data" or any similar summary table at the end of the letter. All clinical data should be presented within the body of the letter where it is relevant to the argument. Do not repeat it in a table format at the end.

Return ONLY the letter text.'''

# ==========

# CELL 7: Assessment Functions

# ==========

ASSESSMENT_PROMPT = '''You are evaluating the strength of an acute respiratory failure DRG appeal letter.

═══ PROPEL RESPIRATORY FAILURE CRITERIA (AUTHORITATIVE SOURCE - USE THIS ONLY) ═══

{propel_definition}

═══ DENIAL LETTER ═══

{denial_text}

═══ GOLD LETTER TEMPLATE USED ═══

{gold_letter_text}

═══ AVAILABLE CLINICAL EVIDENCE ═══

{extracted_clinical_data}

═══ STRUCTURED DATA SUMMARY ═══

{structured_summary}

═══ GENERATED APPEAL LETTER ═══

{generated_letter}

═══ EVALUATION INSTRUCTIONS ═══

IMPORTANT: The Propel Respiratory Failure Criteria above is the ONLY authoritative source for clinical definitions and approved references.

- References cited in the appeal letter are acceptable ONLY if they appear in the Propel document's reference list.

- If the appeal letter cites a reference from the payor's denial letter that is NOT in the Propel document, flag this as a source fidelity error.

- If the appeal letter cites a reference that appears in BOTH the Propel document and the denial letter, this is acceptable.

═══ CONDITIONAL REBUTTALS (for evaluation) ═══

The appeal letter should apply rebuttals ONLY when they match the payor's actual argument:

## Rebuttal A: Consecutive/Sequential SpO2 Readings

{consecutive_spo2_rebuttal}

## Rebuttal B: Persistent/Continuous Symptoms Requirement

{persistent_symptoms_rebuttal}

## Rebuttal C: Proprietary Clinical Criteria (General)

{proprietary_criteria_rebuttal}

If the denial includes one of these arguments and the appeal letter fails to rebut it appropriately, flag this as a deficiency. If the appeal letter applies a rebuttal for an argument the

payor did NOT make, flag this as an error.

Evaluate this appeal letter and provide:

1. OVERALL SCORE (1-10) and RATING (LOW for 1-4, MODERATE for 5-7, HIGH for 8-10)

2. SUMMARY (2-3 sentences)

3. DETAILED BREAKDOWN with scores and specific findings

Evaluation Criteria:

- **Source Fidelity**: Did the letter use ONLY Propel-approved criteria and references? Did it appropriately reject proprietary payor criteria when applicable? Did it avoid listing/critiquing payor references unnecessarily?

- **Propel Criteria Alignment**: Are the Propel diagnostic criteria correctly applied and cited?

- **Argument Structure**: Does the letter systematically address and refute each payor argument? Were rebuttals applied appropriately (only when matching the denial's actual claims)?

- **Evidence Quality**: Is clinical evidence from notes and structured data properly cited with timestamps?

- **Formatting Compliance**: The letter should NOT contain a "Summary Table of Key Clinical Data" or similar summary table at the end. If such a table is present, flag it as a formatting error.

Return ONLY valid JSON in this format:

{{

"overall_score": <1-10>,

"overall_rating": "<LOW|MODERATE|HIGH>",

"summary": "<2-3 sentence summary>",

"source_fidelity": {{

"score": <1-10>,

"findings": [

{{"status": "<correct|incorrect>", "item": "<description>"}}

]

}},

```
    "propel_criteria": {{
        "score": <1-10>,
        "findings": [
        {{"status": "<present|could_strengthen|missing>", "item": "<description>"}}
        ]
    }},
    "argument_structure": {{
        "score": <1-10>,
        "findings": [
        {{"status": "<present|could_strengthen|missing>", "item": "<description>"}}
        ]
    }},
    "evidence_quality": {{
        "clinical_notes": {{"score": <1-10>, "findings": [...]}},
        "structured_data": {{"score": <1-10>, "findings": [...]}}
    }},
    "formatting_compliance": {{
        "score": <1-10>,
        "findings": [
        {{"status": "<compliant|non_compliant>", "item": "<description>"}}
        ]
    }}
    }}"'

    def assess_appeal_strength(generated_letter, propel_definition, denial_text,
                               extracted_notes, gold_letter_text, structured_summary,
                               consecutive_spo2_rebuttal, persistent_symptoms_rebuttal,
                               proprietary_criteria_rebuttal):
        """Assess the strength of a generated appeal letter."""
        print("  Running strength assessment...")
```

```
        # Format extracted notes
        notes_summary = []
        for note_type, content in extracted_notes.items():
            if content and content != "Not available":
                truncated = content[:2000] + "..." if len(content) > 2000 else content
                notes_summary.append(f"## {note_type}\n{truncated}")
        extracted_clinical_data = "\n\n".join(notes_summary) if notes_summary else "No clinical notes available"


        try:
            response = openai_client.chat.completions.create(
                model="gpt-4.1",
                messages=[
                    {"role": "system", "content": "You are a clinical appeal quality assessor. Return only valid JSON."},
                    {"role": "user", "content": ASSESSMENT_PROMPT.format(
                        propel_definition=propel_definition or "Propel criteria not available",
                        denial_text=denial_text[:5000] if denial_text else "Denial text not available",
                        gold_letter_text=gold_letter_text[:3000] if gold_letter_text else "No gold letter template used",
                        extracted_clinical_data=extracted_clinical_data,
                        structured_summary=structured_summary[:3000] if structured_summary else "No structured data",
                        generated_letter=generated_letter,
                        consecutive_spo2_rebuttal=consecutive_spo2_rebuttal,
                        persistent_symptoms_rebuttal=persistent_symptoms_rebuttal,
                        proprietary_criteria_rebuttal=proprietary_criteria_rebuttal
                    )}
                ],
                temperature=0,
```

```
            max_tokens=2000
        )

        raw_response = response.choices[0].message.content.strip()

        # Parse JSON
        if raw_response.startswith("```"):
            raw_response = raw_response.split("```")[1]
            if raw_response.startswith("json"):
                raw_response = raw_response[4:]
            raw_response = raw_response.strip()

        assessment = json.loads(raw_response)

        if "overall_score" in assessment:
            assessment["overall_score"] = max(1, min(10, int(assessment["overall_score"])))

        print(f"  Assessment complete: {assessment.get('overall_score', '?')}/10 - {assessment.get('
overall_rating', '?')}")
        return assessment

    except json.JSONDecodeError as e:
        print(f"  Warning: Assessment JSON parse error: {e}")
        return None
    except Exception as e:
        print(f"  Warning: Assessment failed: {e}")
        return None

def format_assessment_for_docx(assessment):
    """Format assessment dict into text for DOCX output."""
```

```
    if not assessment:
        return "Assessment unavailable\n\nPlease review letter manually."

    status_symbols = {"present": "✓", "could_strengthen": "△", "missing": "✗"}
    lines = []

    lines.append(f"Overall Strength: {assessment.get('overall_score', '?')}/10 -
{assessment.get('overall_rating', '?')}")
    lines.append("")
    lines.append(f"Summary: {assessment.get('summary', 'No summary available')}")
    lines.append("")
    lines.append("Detailed Breakdown:")
    lines.append("─" * 55)

    # Propel Criteria
    propel = assessment.get("propel_criteria", {})
    lines.append(f"PROPEL CRITERIA: {propel.get('score', '?')}/10")
    for finding in propel.get("findings", []):
        symbol = status_symbols.get(finding.get("status", ""), "?")
        lines.append(f"  {symbol} {finding.get('item', '')}")

    # Argument Structure
    argument = assessment.get("argument_structure", {})
    lines.append(f"\nARGUMENT STRUCTURE: {argument.get('score', '?')}/10")
    for finding in argument.get("findings", []):
        symbol = status_symbols.get(finding.get("status", ""), "?")
        lines.append(f"  {symbol} {finding.get('item', '')}")

    # Evidence Quality
    evidence = assessment.get("evidence_quality", {})
```

```
    clinical = evidence.get("clinical_notes", {})
    structured = evidence.get("structured_data", {})
    lines.append(f"\nEVIDENCE - Clinical Notes: {clinical.get('score', '?')}/10")
    for finding in clinical.get("findings", []):
        symbol = status_symbols.get(finding.get("status", ""), "?")
        lines.append(f"  {symbol} {finding.get('item', '')}")
    lines.append(f"\nEVIDENCE - Structured Data: {structured.get('score', '?')}/10")
    for finding in structured.get("findings", []):
        symbol = status_symbols.get(finding.get("status", ""), "?")
        lines.append(f"  {symbol} {finding.get('item', '')}")

    lines.append("—" * 55)
    return "\n".join(lines)

# ============================================================
# CELL 8: Main Processing Pipeline
# ============================================================
print("\n" + "="*60)
print("INFERENCE - APPEAL LETTER GENERATION")
print("="*60)

account_id = case_data["account_id"]
extracted_notes = case_data["extracted_notes"]
structured_summary = case_data["structured_summary"]
conflicts_result = case_data["conflicts"]

# ------------------------------------------------------------------
# STEP 1: Find best gold letter match
```

```
    # ------------------------------------------------------------------------
    print("\nStep 1: Finding best gold letter match...")
    gold_letter, gold_letter_score = find_best_gold_letter(case_data["denial_embedding"])
    if gold_letter:
        is_default = gold_letter.get("metadata", {}).get("is_default_template") == "true"
        if is_default:
            print(f"  Match: {gold_letter['source_file']} (score: N/A - default template)")
        else:
            print(f"  Match: {gold_letter['source_file']} (score: {gold_letter_score:.3f})")
    else:
        print("  No match found")

    # ------------------------------------------------------------------------
    # STEP 2: Generate appeal letter
    # ------------------------------------------------------------------------
    print("\nStep 2: Generating appeal letter...")

    # Build gold letter section
    if gold_letter:
        is_default = gold_letter.get("metadata", {}).get("is_default_template") == "true"
        if is_default:
            gold_letter_section = f"## APPEAL TEMPLATE\n{gold_letter['appeal_text']}"
            gold_letter_instructions = "**NOTE: Using default template as structural guide.**\n"
        else:
            gold_letter_section = f"## WINNING APPEAL (Score: {gold_letter_score:.
3f})\nPayor: {gold_letter.get('payor')}\n\n{gold_letter['appeal_text']}"
            gold_letter_instructions = "**CRITICAL: Learn from this winning appeal.**\n"
    else:
        gold_letter_section = "No template available."
        gold_letter_instructions = ""
```

```
    # Clinical definition
    if case_data["is_arf"] and "respiratory_failure" in propel_definitions:
        clinical_definition_section = f"## OFFICIAL ACUTE RESPIRATORY FAILURE
DEFINITION\n{propel_definitions['respiratory_failure']}"
    else:
        clinical_definition_section = "No specific definition loaded."
    print(f"================================================
============================================================
{clinical_definition_section}")


    # Build prompt
    writer_prompt = WRITER_PROMPT.format(
        consecutive_spo2_rebuttal=CONSECUTIVE_SPO2_REBUTTAL,
        persistent_symptoms_rebuttal=PERSISTENT_SYMPTOMS_REBUTTAL,
        proprietary_criteria_rebuttal=PROPRIETARY_CRITERIA_REBUTTAL,
        denial_letter_text=case_data["denial_text"],
        discharge_summary=extracted_notes.get("discharge_summary", "Not available"),
        hp_note=extracted_notes.get("hp_note", "Not available"),
        progress_note=extracted_notes.get("progress_note", "Not available"),
        consult_note=extracted_notes.get("consult_note", "Not available"),
        ed_notes=extracted_notes.get("ed_notes", "Not available"),
        initial_assessment=extracted_notes.get("initial_assessment", "Not available"),
        ed_triage=extracted_notes.get("ed_triage", "Not available"),
        ed_provider_note=extracted_notes.get("ed_provider_note", "Not available"),
        addendum_note=extracted_notes.get("addendum_note", "Not available"),
        hospital_course=extracted_notes.get("hospital_course", "Not available"),
        subjective_objective=extracted_notes.get("subjective_objective", "Not available"),
        assessment_plan=extracted_notes.get("assessment_plan", "Not available"),
        nursing_note=extracted_notes.get("nursing_note", "Not available"),
        code_documentation=extracted_notes.get("code_documentation", "Not available"),
```

```
    structured_data_summary=structured_summary,
    clinical_definition_section=clinical_definition_section,
    gold_letter_section=gold_letter_section,
    gold_letter_instructions=gold_letter_instructions,
    patient_name=case_data.get("patient_name", ""),
    patient_dob=case_data.get("patient_dob", ""),
    hsp_account_id=account_id,
    date_of_service=case_data.get("date_of_service", ""),
    facility_name=case_data.get("facility_name", "Mercy Hospital"),
    original_drg=case_data.get("original_drg") or "Unknown",
    proposed_drg=case_data.get("proposed_drg") or "Unknown",
    payor=case_data.get("payor", "Unknown"),
)

# Generate letter
writer_response = openai_client.chat.completions.create(
    model="gpt-4.1",
    messages=[
        {"role": "system", "content": "You are a clinical coding expert writing DRG appeal letters."},
        {"role": "user", "content": writer_prompt}
    ],
    temperature=0.2,
    max_tokens=4000
)

letter_text = writer_response.choices[0].message.content.strip()
print(f"  Generated {len(letter_text)} character letter")

# ----------------------------------------------------------------------
# STEP 3: Assess appeal strength
```

```
# ---------------------------------------------------------------------
print("\nStep 3: Assessing appeal strength...")
propel_def = propel_definitions.get("respiratory_failure") if case_data["is_arf"] else None
gold_text = gold_letter.get("appeal_text", "") if gold_letter else ""

assessment = assess_appeal_strength(
    letter_text, propel_def, case_data["denial_text"],
    extracted_notes, gold_text, structured_summary,
    CONSECUTIVE_SPO2_REBUTTAL,
    PERSISTENT_SYMPTOMS_REBUTTAL,
    PROPRIETARY_CRITERIA_REBUTTAL
)

# ---------------------------------------------------------------------
# STEP 4: Export to DOCX
# ---------------------------------------------------------------------
if EXPORT_TO_DOCX:
    print("\nStep 4: Exporting to DOCX...")
    from docx import Document
    from docx.shared import Pt

    def add_markdown_paragraph(doc, text):
        """Add paragraph with markdown bold converted to Word bold."""
        p = doc.add_paragraph()
        parts = re.split(r'(\*\*.*?\*\*)', text)
        for part in parts:
            if part.startswith('**') and part.endswith('**'):
                run = p.add_run(part[2:-2])
                run.bold = True
```

```
            else:
                p.add_run(part)
        return p

    os.makedirs(DOCX_OUTPUT_BASE, exist_ok=True)

    doc = Document()
    doc.add_heading('Appeal Letter', level=1)

    # Metadata
    meta = doc.add_paragraph()
    meta.add_run("Generated: ").bold = True
    meta.add_run(f"{datetime.now().strftime('%Y-%m-%d %H:%M')}\n")
    meta.add_run("Patient: ").bold = True
    meta.add_run(f"{case_data.get('patient_name', 'Unknown')}\n")
    meta.add_run("Payor: ").bold = True
    meta.add_run(f"{case_data.get('payor', 'Unknown')}\n")
    meta.add_run("Gold Letter: ").bold = True
    is_default = gold_letter.get("metadata", {}).get("is_default_template") == "true" if gold_letter else
False
    if is_default:
        meta.add_run(f"{gold_letter['source_file'] if gold_letter else 'None'} (score: N/A)\n")
    else:
        meta.add_run(f"{gold_letter['source_file'] if gold_letter else 'None'} (score: {gold_letter_score:.3f}
)\n")

    # Assessment section
    doc.add_paragraph("=" * 55)
    header = doc.add_paragraph()
    header.add_run("APPEAL STRENGTH ASSESSMENT (Internal Review Only)").bold = True
```

```
    doc.add_paragraph("═" * 55)

    for line in format_assessment_for_docx(assessment).split('\n'):
        p = doc.add_paragraph(line)
        p.paragraph_format.space_after = Pt(0)

    doc.add_paragraph("═" * 55)

    # Conflicts appendix (if any)
    if conflicts_result.get('conflicts'):
        doc.add_paragraph()
        conflict_header = doc.add_paragraph()
        conflict_header.add_run("⚠️ CONFLICTS DETECTED - REQUIRES CDI REVIEW").bold = True
        doc.add_paragraph("─" * 55)
        for conflict in conflicts_result['conflicts']:
            p = doc.add_paragraph(conflict)
            p.paragraph_format.space_after = Pt(4)
        if conflicts_result.get('recommendation'):
            doc.add_paragraph()
            rec = doc.add_paragraph()
            rec.add_run("Recommendation: ").bold = True
            rec.add_run(conflicts_result['recommendation'])
        doc.add_paragraph("─" * 55)

    doc.add_paragraph()

    # Letter content
    for paragraph in letter_text.split('\n\n'):
        if paragraph.strip():
```

```
            p = add_markdown_paragraph(doc, paragraph.strip())
            p.paragraph_format.space_after = Pt(12)

        # Save
        patient_name = case_data.get('patient_name', 'Unknown')
        safe_name = "".join(c for c in patient_name if c.isalnum() or c in (' ', '-', '_')).strip()
        filename = f"{account_id}_{safe_name}_appeal.docx"
        filepath = os.path.join(DOCX_OUTPUT_BASE, filename)
        doc.save(filepath)

        print(f"  Saved: {filepath}")

    # ---------------------------------------------------------------------
    # SUMMARY
    # ---------------------------------------------------------------------
    print("\n" + "="*60)
    print("INFERENCE COMPLETE")
    print("="*60)
    print(f"Patient: {case_data.get('patient_name', 'Unknown')}")
    print(f"Account: {account_id}")
    print(f"Letter length: {len(letter_text)} chars")
    if assessment:
        print(f"Strength: {assessment.get('overall_score', '?')}/10 - {assessment.get('overall_rating', '?')}")
    if conflicts_result.get('conflicts'):
        print(f"Conflicts: {len(conflicts_result['conflicts'])} (see appendix in DOCX)")
    if EXPORT_TO_DOCX:
        print(f"Output: {filepath}")

print("\nInference complete.")
```

**Michael (Mike) Joyce, PhD**

Senior Data Scientist
Enterprise Data & Analytics

**Mercy**
14528 S Outer 40, Suite 100 | Chesterfield, MO 63017
Mobile: 512-736-5440



---

**image001.png**
7K View Download