

# MIMICIV\_hypercap\_EXT\_cohort

## Table of contents

<b>1</b>	<b>MIMIC-IV Hypercapnia Cohort — ICD Physiologic Thresholds (BigQuery)</b>	<b>2</b>
1.1	Pipeline stages (readable map) . . . . .	3
1.2	SQL registry (centralized query templates) . . . . .	3
<b>2</b>	<b>MIMIC-IV on BigQuery</b>	<b>4</b>
2.1	Environment Bootstrap & Smoke Test . . . . .	4
2.2	0. Prerequisites (one-time) . . . . .	4
2.3	Data Generation . . . . .	67
2.3.1	1) ICD cohort flags (hypercapnic respiratory failure) . . . . .	67
2.3.2	2) Blood gas inclusion thresholds — ANY PaCO <sub>2</sub> meeting cutoffs (LAB + POC) . . . . .	71
2.3.3	3) Cohort union (ICD thresholds) and <code>hadm_list</code> for downstream queries	75
2.3.4	4) First ABG and First VBG (LAB + POC, standardized to mmHg) . .	77
2.3.5	5) Demographics & outcomes . . . . .	86
2.3.6	6) NIH/OMB race & ethnicity (ED + Hospital) . . . . .	89
2.3.7	7) ED triage (linked to <code>hadm</code> ) and first ED vitals . . . . .	93
2.3.8	8) ICU meta (first ICU stay, LOS days) . . . . .	96
2.3.9	9) Ventilation flags (ICD procedures) . . . . .	97
2.3.10	10) Assemble final DataFrame . . . . .	101
2.3.11	11) Sanity checks . . . . .	113
2.3.12	PDF-ready long tables . . . . .	114
2.3.13	12) Save to Excel . . . . .	119
2.3.14	Create Annotation Dataset . . . . .	120
<b>3</b>	<b>ED-stay cohort expansion (timing, severity, comorbidity, outcomes)</b>	<b>122</b>
3.0.1	Phase 0 — Inventory & missing-field registry . . . . .	122
3.0.2	Phase 1 — ED encounter spine and ED enrichment (one row per ED stay)	126
3.0.3	Phase 2 — Hospital admission context and outcomes . . . . .	130
3.0.4	Phase 3 — ICU timing and LOS . . . . .	131
3.0.5	Phase 4 — ED longitudinal vitals (0–6h) . . . . .	132

3.0.6	Phase 5 — Robust lab discovery + gas panels . . . . .	135
3.0.7	Phase 5C — ICU POC blood gases (chartevents, optional) . . . . .	158
3.0.8	Phase 6 — BMI/anthropometrics (OMR) . . . . .	171
3.0.9	Phase 7 — ICD comorbidity flags . . . . .	178
3.0.10	Phase 8 — Timing phenotypes and derived bands . . . . .	182
3.1	QA & Data Fidelity . . . . .	187
3.2	Outputs (ED-stay cohort + long tables) . . . . .	206

## 1 MIMIC-IV Hypercapnia Cohort — ICD Physiologic Thresholds (BigQuery)

TODO:

[ ] want to add ED-rendered diagnoses - a flag to split off whether the hypercapnic respiratory failure ICD was rendered in the ED or during the hospital stay. [ ] are the installation instructions at the beginning of this notebook consistent with the way we intend the notebook to be used? is it also consistent with the README.md instructions?

**Goal:** Build an admissions-level tabular dataset that **enrolls** any hospital admission (`hadm_id`) meeting *any* of:

1. **ICD** codes for hypercapnic respiratory failure (legacy cohort).
2. **Any arterial blood gas** (LAB or POC) with **PaCO<sub>2</sub> ≥ 45.0 mmHg** anywhere during the episode.
3. **Any venous blood gas** (LAB or POC) with **PaCO<sub>2</sub> ≥ 50.0 mmHg** anywhere during the episode.

Then, keep all downstream columns/logic from the current workflow: - Per-code ICD indicators and an `any_hypercap_icd` flag. - Robust extraction of **first ABG** and **first VBG** (across LAB + POC) with standardized units (mmHg) and pairing logic. - Demographics/outcomes, NIH/OMB race & ethnicity, ED triage + first ED vitals, ICU meta (first stay + LOS), ventilation flags. - Sanity checks.

**Assumptions** - You already configured BigQuery auth (`gcloud auth application-default login`) and `.env` variables as in the previous notebook. - The PhysioNet hosting project is `physionet-data`. - Datasets exist (e.g., `mimiciv_3_1_hosp`, `mimiciv_3_1_icu`, and an ED dataset such as `mimiciv_ed`). This notebook auto-detects the ED dataset.

**Execution timing note:** To capture per-cell runtimes, enable cell execution timing in VS Code (Notebook: Show Cell Execution Time) or enable `ExecuteTime` in Jupyter. Then re-run and save to allow runtime summaries.

## 1.1 Pipeline stages (readable map)

1. **Stage A — Config & helpers:** dataset IDs, thresholds, shared helper functions.
2. **Stage B — Cohort spine:** ICD + gas thresholds → `hadm_list` / `ed_stay_list`.
3. **Stage C — Bulk pulls:** one query per table where possible.
4. **Stage D — Transforms:** panels, flags, timing, comorbidities, vitals.
5. **Stage E — QA checks:** deterministic assertions and range checks.
6. **Stage F — Outputs:** parquet + Excel exports.

## 1.2 SQL registry (centralized query templates)

All BigQuery SQL is defined in one place below, then referenced by name in subsequent cells.

```
# Purpose: Build ABG/VBG hypercapnia threshold flags from lab and ICU POC
↪ pCO2 measurements.
```

```
import sys
print(sys.executable)
```

```
# Central SQL registry (define all query templates here)
SQL = {}
```

```
def sql(name: str) -> str:
    if name not in SQL:
        raise KeyError(f"SQL template not found: {name}")
    return SQL[name]
```

```
# SQL templates (populated below in-place to keep notebook linear)
# Names: admit_sql, co2_thresholds_sql, cohort_icd_sql, counts_sql, demo_sql,
↪ ditems_sql, ed_counts_sql, ed_first_vitals_sql, ed_spine_sql,
↪ ed_to_icu_sql, ed_triage_sql, ed_vitals_sql, icd_sql, icu_meta_sql,
↪ icu_sql, labitems_sql, labs_sql, vent_chart_sql, vent_sql
```

```
/Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/Hypercapnia
CC-NLP/.venv/bin/python
```

**Rationale:** Define a reproducible, admission-level cohort that captures hypercapnia using complementary diagnostic (ICD) and physiologic (blood gas) criteria.

## 2 MIMIC-IV on BigQuery

### 2.1 Environment Bootstrap & Smoke Test

Purpose: make a clean, reproducible start on a new machine.

Outcome: verify auth, project config, and dataset access; provide a reusable BigQuery runner for the build notebook.

**Rationale:** Establish the BigQuery environment and dataset configuration so queries are consistent and reproducible across runs.

### 2.2 0. Prerequisites (one-time)

**Accounts & access** - PhysioNet access to MIMIC-IV on BigQuery; in BigQuery Console star project `physionet-data`. - A Google Cloud **Project ID** with BigQuery API enabled (this is your **billing** project).

**CLI & environment** - Google Cloud SDK (`gcloud`) installed and on `PATH`. - Python environment created with `uv` (see README) and Jupyter kernel selected. - A project-local `.env` with the variables below.

**.env variables**

```
MIMIC_BACKEND=bigquery
WORK_PROJECT=<your-billing-project-id>
BQ_PHYSIONET_PROJECT=physionet-data
BQ_DATASET_HOSP=mimiciv_3_1_hosp
BQ_DATASET_ICU=mimiciv_3_1_icu
BQ_DATASET_ED=mimiciv_ed
WORK_DIR=/path/to/Hypercap-CC-NLP
# GOOGLE_APPLICATION_CREDENTIALS=/path/to/service-account.json
```

**Command line quickstart**

```
brew install --cask google-cloud-sdk
gcloud init
gcloud auth application-default login
gcloud services enable bigquery.googleapis.com --project
↪ <your-billing-project-id>
ls -l ~/.config/gcloud/application_default_credentials.json
```

**Rationale:** Verify access and credentials up front to prevent silent failures later in the pipeline.

```

# Purpose: Set up project paths, environment variables, and BigQuery client
↳ connections for reproducible execution.

# --- Imports & environment
import os, re, json, math, textwrap, sys, hashlib
from pathlib import Path
from typing import Any, Mapping, Sequence

import numpy as np
import pandas as pd

from google.cloud import bigquery
from google.oauth2 import service_account
from dotenv import load_dotenv

load_dotenv()

WORK_DIR = Path(os.getenv("WORK_DIR", Path.cwd())).expanduser().resolve()
DATA_DIR = WORK_DIR / "MIMIC tabular data"
DATA_DIR.mkdir(parents=True, exist_ok=True)
SRC_DIR = WORK_DIR / "src"
if SRC_DIR.exists() and str(SRC_DIR) not in sys.path:
    sys.path.insert(0, str(SRC_DIR))

# Local helper functions (core cohort logic must remain embedded in this
↳ notebook)
OMR_RESULT_NAMES = ("bmi", "height", "weight")
OMR_OUTPUT_COLUMNS = (
    "bmi_closest_pre_ed",
    "height_closest_pre_ed",
    "weight_closest_pre_ed",
)

OMR_PROVENANCE_COLUMNS = (
    "anthro_timing_tier",
    "anthro_days_offset",
    "anthro_chartdate",
    "anthro_timing_uncertain",
    "anthro_source",
    "anthro_obstime",
    "anthro_hours_offset",
    "anthro_timing_basis",
)

```

```

OMR_TIMING_TIERS = ("pre_ed_365", "post_ed_365", "missing")

EXPECTED_STRUCTURAL_NULL_FIELDS = (
    "poc_abg_ph_uom",
    "poc_vbg_ph_uom",
    "poc_other_ph_uom",
)

PACO2_VALUE_UOM_PAIRS = (
    ("poc_abg_paco2", "poc_abg_paco2_uom"),
    ("poc_vbg_paco2", "poc_vbg_paco2_uom"),
    ("poc_other_paco2", "poc_other_paco2_uom"),
)

DEFAULT_VITALS_MODEL_RANGES: dict[str, tuple[float, float]] = {
    "ed_triage_hr": (20.0, 250.0),
    "ed_first_hr": (20.0, 250.0),
    "ed_triage_rr": (4.0, 80.0),
    "ed_first_rr": (4.0, 80.0),
}

DEFAULT_GAS_MODEL_RANGES: dict[str, tuple[float, float]] = {
    "first_ph": (6.8, 7.8),
    "first_pco2": (10.0, 200.0),
    "first_other_pco2": (10.0, 200.0),
    "first_lactate": (0.0, 30.0),
}

_NUMERIC_TOKEN_PATTERN = re.compile(r"(-?\d+(?:\.\d+)?)")
_ARTERIAL_HINT_PATTERN = re.compile(
    r"(arterial|abg|a[- ]?line|art line|\bart\b|\bartery\b)",
    re.I,
)

_VENOUS_HINT_PATTERN = re.compile(
    r"(venous|vbg|cvbg|mixed venous|central venous|\bven\b)",
    re.I,
)

_PO2_LABEL_DENY_PATTERN_DEFAULT = re.compile(
    r"(?:^[^a-z])p(?:a)?\s*o\s*2(?:[^\a-z]|$)|\bpo2\b|\bpao2\b|oxygen|o2\s*_
    ↪ (?:sat|saturation)|sao2|spo2|fio2|et\s*co2|end[-
    ↪ ]?tidal|t\s*co2|tco2|total\s*co2|hco3|bicar|v\s*co2|vco2|co2\s*(?:
    ↪ prod|production|elimin|elimination)",

```

```

        re.I,
    )

BLOOD_GAS_SPEC_PATH = WORK_DIR / "specs" / "blood_gas_itemids.json"

def _coerce_itemid_list(values: Sequence[Any], *, key_name: str) ->
    ↪ list[int]:
    cleaned: list[int] = []
    for raw in values:
        try:
            cleaned.append(int(raw))
        except (TypeError, ValueError) as exc:
            raise ValueError(f"Invalid itemid in {key_name}: {raw!r}") from
            ↪ exc
    return sorted(set(cleaned))

def _sql_int_array(values: Sequence[int], *, empty_sentinel: int =
    ↪ -999999999) -> str:
    """Return a BigQuery INT64 array literal from Python integers."""
    resolved = [int(v) for v in values]
    if not resolved:
        resolved = [int(empty_sentinel)]
    return "[" + ", ".join(str(v) for v in resolved) + "]"

def load_blood_gas_itemid_manifest(path: Path) -> dict[str, Any]:
    """Load and validate versioned blood-gas itemid manifest."""
    if not path.exists():
        raise FileNotFoundError(
            f"Blood-gas manifest missing at {path}. Add
            ↪ specs/blood_gas_itemids.json."
        )
    payload = json.loads(path.read_text())
    if not isinstance(payload, dict):
        raise ValueError("Blood-gas manifest must be a JSON object.")

    for section in ("lab", "icu", "denylist"):
        if section not in payload or not isinstance(payload[section], dict):
            raise ValueError(f"Blood-gas manifest missing required section
            ↪ '{section}'")

```

```

lab = payload["lab"]
icu = payload["icu"]
denylist = payload["denylist"]

lab_pco2 = _coerce_itemid_list(lab.get("pco2_itemids", []),
↪ key_name="lab.pco2_itemids")
lab_pco2_excluded = _coerce_itemid_list(
    lab.get("excluded_pco2_itemids", []),
    key_name="lab.excluded_pco2_itemids",
)
lab_ph = _coerce_itemid_list(lab.get("ph_itemids", []),
↪ key_name="lab.ph_itemids")
lab_hco3 = _coerce_itemid_list(lab.get("hco3_itemids", []),
↪ key_name="lab.hco3_itemids")
lab_spec = _coerce_itemid_list(
    lab.get("specimen_type_itemids", []),
    key_name="lab.specimen_type_itemids",
)
if not lab_pco2:
    raise ValueError("Blood-gas manifest requires at least one
↪ lab.pco2_itemids value.")
if not lab_ph:
    raise ValueError("Blood-gas manifest requires at least one
↪ lab.ph_itemids value.")
if not lab_hco3:
    raise ValueError("Blood-gas manifest requires at least one
↪ lab.hco3_itemids value.")
if not lab_spec:
    raise ValueError(
        "Blood-gas manifest requires at least one
↪ lab.specimen_type_itemids value."
    )

blood_fluid_terms = [str(term).strip().lower() for term in
↪ lab.get("blood_fluid_terms", ["blood"])]
blood_fluid_terms = sorted(set(term for term in blood_fluid_terms if
↪ term))
if not blood_fluid_terms:
    blood_fluid_terms = ["blood"]

icu_pco2 = _coerce_itemid_list(icu.get("pco2_itemids", []),
↪ key_name="icu.pco2_itemids")
icu_pco2_abg = _coerce_itemid_list(

```



```

        icu.get("pco2_abg_itemids", []),
        key_name="icu.pco2_abg_itemids",
    )
    icu_pco2_vbg = _coerce_itemid_list(
        icu.get("pco2_vbg_itemids", []),
        key_name="icu.pco2_vbg_itemids",
    )
    icu_pco2_excluded = _coerce_itemid_list(
        icu.get("excluded_pco2_itemids", []),
        key_name="icu.excluded_pco2_itemids",
    )
    icu_ph = _coerce_itemid_list(icu.get("ph_itemids", []),
↪ key_name="icu.ph_itemids")
    icu_hco3 = _coerce_itemid_list(icu.get("hco3_itemids", []),
↪ key_name="icu.hco3_itemids")
    icu_spec = _coerce_itemid_list(
        icu.get("specimen_type_itemids", []),
        key_name="icu.specimen_type_itemids",
    )
    if icu_pco2_excluded:
        excluded_set = set(icu_pco2_excluded)
        icu_pco2 = sorted(set(icu_pco2).difference(excluded_set))
        icu_pco2_abg = sorted(set(icu_pco2_abg).difference(excluded_set))
        icu_pco2_vbg = sorted(set(icu_pco2_vbg).difference(excluded_set))
    if icu_pco2_abg and not set(icu_pco2_abg).issubset(set(icu_pco2)):
        raise ValueError(
            "Blood-gas manifest icu.pco2_abg_itemids must be a subset of
↪ icu.pco2_itemids."
        )
    if icu_pco2_vbg and not set(icu_pco2_vbg).issubset(set(icu_pco2)):
        raise ValueError(
            "Blood-gas manifest icu.pco2_vbg_itemids must be a subset of
↪ icu.pco2_itemids."
        )

    deny_tokens = [
        str(token).strip().lower()
        for token in denylist.get("label_tokens", [])
        if str(token).strip()
    ]
    allow_pattern_fallback = bool(icu.get("allow_pattern_fallback", True))
    inclusion_mode = str(icu.get("inclusion_mode",
↪ "validated_only")).strip().lower()

```

```

if inclusion_mode not in {"validated_only", "disabled"}:
    raise ValueError(
        "Blood-gas manifest icu.inclusion_mode must be one of "
        "'{validated_only}', 'disabled'}.")
)

pco2_qc_payload = icu.get("pco2_qc", {})
if pco2_qc_payload is None:
    pco2_qc_payload = {}
if not isinstance(pco2_qc_payload, dict):
    raise ValueError("Blood-gas manifest icu.pco2_qc must be a JSON
        ↪ object.")
qc_allowed_uoms = [
    str(value).strip().lower()
    for value in pco2_qc_payload.get("allowed_uoms", ["", "mmhg", "kpa"])
]
qc_allowed_uoms = sorted(set(qc_allowed_uoms))
if not qc_allowed_uoms:
    qc_allowed_uoms = ["", "mmhg", "kpa"]
qc_label_include = [
    str(value).strip().lower()
    for value in pco2_qc_payload.get("label_must_include_tokens",
        ↪ ["pco2"])
    if str(value).strip()
]
qc_label_include = sorted(set(qc_label_include))
qc_label_exclude = [
    str(value).strip().lower()
    for value in pco2_qc_payload.get(
        "label_must_exclude_tokens",
        ["po2", "pao2", "oxygen", "sao2", "spo2", "fio2"],
    )
    if str(value).strip()
]
qc_label_exclude = sorted(set(qc_label_exclude))
qc_median_min = float(pco2_qc_payload.get("median_min", 45.0))
qc_median_max = float(pco2_qc_payload.get("median_max", 80.0))
if qc_median_min > qc_median_max:
    raise ValueError(
        "Blood-gas manifest icu.pco2_qc median_min must be <=
        ↪ median_max.")
)

```

```

manifest = {
    "version": str(payload.get("version", "UNSPECIFIED")),
    "lab": {
        "pco2_itemids": lab_pco2,
        "excluded_pco2_itemids": lab_pco2_excluded,
        "ph_itemids": lab_ph,
        "hco3_itemids": lab_hco3,
        "specimen_type_itemids": lab_spec,
        "blood_fluid_terms": blood_fluid_terms,
    },
    "icu": {
        "pco2_itemids": icu_pco2,
        "pco2_abg_itemids": icu_pco2_abg,
        "pco2_vbg_itemids": icu_pco2_vbg,
        "excluded_pco2_itemids": icu_pco2_excluded,
        "ph_itemids": icu_ph,
        "hco3_itemids": icu_hco3,
        "specimen_type_itemids": icu_spec,
        "allow_pattern_fallback": allow_pattern_fallback,
        "inclusion_mode": inclusion_mode,
        "pco2_qc": {
            "allowed_uoms": qc_allowed_uoms,
            "median_min": qc_median_min,
            "median_max": qc_median_max,
            "label_must_include_tokens": qc_label_include,
            "label_must_exclude_tokens": qc_label_exclude,
        },
    },
    "denylist": {"label_tokens": deny_tokens},
    "path": str(path),
}
return manifest

```

```

BLOOD_GAS_MANIFEST = load_blood_gas_itemid_manifest(BLOOD_GAS_SPEC_PATH)
LAB_PC02_ITEMIDS = BLOOD_GAS_MANIFEST["lab"]["pco2_itemids"]
LAB_EXCLUDED_PC02_ITEMIDS =
    ↪ BLOOD_GAS_MANIFEST["lab"]["excluded_pco2_itemids"]
if LAB_EXCLUDED_PC02_ITEMIDS:
    LAB_PC02_ITEMIDS = sorted(
        set(LAB_PC02_ITEMIDS).difference(set(LAB_EXCLUDED_PC02_ITEMIDS))
    )
if not LAB_PC02_ITEMIDS:

```

```

        raise ValueError(
            "All lab.pco2_itemids were excluded by lab.excluded_pco2_itemids. "
            "Update specs/blood_gas_itemids.json."
        )
LAB_PH_ITEMIDS = BLOOD_GAS_MANIFEST["lab"]["ph_itemids"]
LAB_HCO3_ITEMIDS = BLOOD_GAS_MANIFEST["lab"]["hco3_itemids"]
LAB_SPECIMEN_TYPE_ITEMIDS =
    ↪ BLOOD_GAS_MANIFEST["lab"]["specimen_type_itemids"]
LAB_BLOOD_FLUID_TERMS = BLOOD_GAS_MANIFEST["lab"]["blood_fluid_terms"]
ICU_PCO2_ITEMIDS = BLOOD_GAS_MANIFEST["icu"]["pco2_itemids"]
ICU_PCO2_ABG_ITEMIDS = BLOOD_GAS_MANIFEST["icu"]["pco2_abg_itemids"]
ICU_PCO2_VBG_ITEMIDS = BLOOD_GAS_MANIFEST["icu"]["pco2_vbg_itemids"]
ICU_EXCLUDED_PCO2_ITEMIDS =
    ↪ BLOOD_GAS_MANIFEST["icu"]["excluded_pco2_itemids"]
ICU_PH_ITEMIDS = BLOOD_GAS_MANIFEST["icu"]["ph_itemids"]
ICU_HCO3_ITEMIDS = BLOOD_GAS_MANIFEST["icu"]["hco3_itemids"]
ICU_SPECIMEN_TYPE_ITEMIDS =
    ↪ BLOOD_GAS_MANIFEST["icu"]["specimen_type_itemids"]
ICU_ALLOW_PATTERN_FALLBACK =
    ↪ bool(BLOOD_GAS_MANIFEST["icu"]["allow_pattern_fallback"])
ICU_INCLUSION_MODE =
    ↪ str(BLOOD_GAS_MANIFEST["icu"]["inclusion_mode"]).strip().lower()
ICU_PCO2_QC = BLOOD_GAS_MANIFEST["icu"]["pco2_qc"]
ICU_LABEL_DENY_TOKENS = BLOOD_GAS_MANIFEST["denylist"]["label_tokens"]
ICU_PCO2_QC_ALLOWED_UOMS = {
    str(value).strip().lower() for value in ICU_PCO2_QC["allowed_uoms"]
}
ICU_PCO2_QC_MEDIAN_MIN = float(ICU_PCO2_QC["median_min"])
ICU_PCO2_QC_MEDIAN_MAX = float(ICU_PCO2_QC["median_max"])
ICU_PCO2_QC_INCLUDE_TOKENS = [
    str(value).strip().lower()
    for value in ICU_PCO2_QC["label_must_include_tokens"]
]
ICU_PCO2_QC_EXCLUDE_TOKENS = [
    str(value).strip().lower()
    for value in ICU_PCO2_QC["label_must_exclude_tokens"]
]

LAB_PCO2_SQL_ARRAY = _sql_int_array(LAB_PCO2_ITEMIDS)
LAB_PH_SQL_ARRAY = _sql_int_array(LAB_PH_ITEMIDS)
LAB_HCO3_SQL_ARRAY = _sql_int_array(LAB_HCO3_ITEMIDS)
LAB_SPECIMEN_SQL_ARRAY = _sql_int_array(LAB_SPECIMEN_TYPE_ITEMIDS)
ICU_PCO2_SQL_ARRAY = _sql_int_array(ICU_PCO2_ITEMIDS)

```

```

ICU_PC02_ABG_SQL_ARRAY = _sql_int_array(ICU_PC02_ABG_ITEMIDS)
ICU_PC02_VBG_SQL_ARRAY = _sql_int_array(ICU_PC02_VBG_ITEMIDS)
ICU_PH_SQL_ARRAY = _sql_int_array(ICU_PH_ITEMIDS)
ICU_HCO3_SQL_ARRAY = _sql_int_array(ICU_HCO3_ITEMIDS)
ICU_SPECIMEN_SQL_ARRAY = _sql_int_array(ICU_SPECIMEN_TYPE_ITEMIDS)
LAB_BLOOD_FLUID_SQL_LIST = ", ".join(f"'{term}'" for term in
    ↪ LAB_BLOOD_FLUID_TERMS)
ICU_LABEL_DENY_REGEX = (
    "|".join(re.escape(token) for token in ICU_LABEL_DENY_TOKENS)
    if ICU_LABEL_DENY_TOKENS
    else _PO2_LABEL_DENY_PATTERN_DEFAULT.pattern
)

def _to_int64(series: pd.Series) -> pd.Series:
    """Return pandas nullable integer series for key columns."""
    return pd.to_numeric(series, errors="coerce").astype("Int64")

def prepare_omr_records(omr_df: pd.DataFrame) -> pd.DataFrame:
    """Normalize OMR records into a deterministic schema.

    Required source columns: ``subject_id``, ``chartdate``, ``result_name``,
    ``result_value``.

    Returns:
        DataFrame with columns:
        ``subject_id`` (Int64), ``chartdate_dt`` (datetime64),
        ``result_name`` (lowercase), and ``result_value_num`` (float).
    """
    required = {"subject_id", "chartdate", "result_name", "result_value"}
    missing = sorted(required.difference(omr_df.columns))
    if missing:
        raise KeyError(f"prepare_omr_records missing required columns:
            ↪ {missing}")

    prepared = omr_df.copy()
    prepared["subject_id"] = _to_int64(prepared["subject_id"])
    prepared["chartdate_dt"] = pd.to_datetime(prepared["chartdate"],
    ↪ errors="coerce")
    prepared["result_name"] = (
        prepared["result_name"].astype(str).str.strip().str.lower()
    )

```

```

prepared["result_value_num"] = pd.to_numeric(
    prepared["result_value"]
    .astype(str)
    .str.extract(_NUMERIC_TOKEN_PATTERN, expand=False),
    errors="coerce",
)

prepared =
↪ prepared.loc[prepared["result_name"].isin(OMR_RESULT_NAMES)].copy()
prepared = prepared.loc[
    prepared["subject_id"].notna() & prepared["chartdate_dt"].notna()
].copy()

return prepared[["subject_id", "chartdate_dt", "result_name",
↪ "result_value_num"]]

def attach_closest_pre_ed_omr(
    ed_df: pd.DataFrame,
    omr_df: pd.DataFrame,
    window_days: int = 365,
) -> tuple[pd.DataFrame, dict[str, Any]]:
    """Attach closest anthropometrics with tiered timing fallback.

    Args:
        ed_df: ED-stay grain dataframe with ``ed_stay_id``, ``subject_id``,
            and ``ed_intime``.
        omr_df: Prepared OMR records from ``prepare_omr_records``.
        window_days: Inclusion window in days before/after ED arrival.
    """

def _with_default_anthro_columns(frame: pd.DataFrame) -> pd.DataFrame:
    updated = frame.copy()
    for column_name in OMR_OUTPUT_COLUMNS:
        if column_name not in updated.columns:
            updated[column_name] = pd.NA
        if "anthro_timing_tier" not in updated.columns:
            updated["anthro_timing_tier"] = "missing"
        else:
            updated["anthro_timing_tier"] =
↪ updated["anthro_timing_tier"].fillna("missing")
            if "anthro_days_offset" not in updated.columns:

```

```

        updated["anthro_days_offset"] = pd.Series([pd.NA] * len(updated),
dtype="Int64")
    else:
        updated["anthro_days_offset"] = pd.to_numeric(
            updated["anthro_days_offset"], errors="coerce"
        ).astype("Int64")
    if "anthro_chartdate" not in updated.columns:
        updated["anthro_chartdate"] = pd.NaT
    else:
        updated["anthro_chartdate"] = pd.to_datetime(
            updated["anthro_chartdate"], errors="coerce"
        )
    if "anthro_timing_uncertain" not in updated.columns:
        updated["anthro_timing_uncertain"] = pd.Series(
            [pd.NA] * len(updated), dtype="boolean"
        )
    else:
        updated["anthro_timing_uncertain"] = updated[
            "anthro_timing_uncertain"
        ].astype("boolean")
    if "anthro_source" not in updated.columns:
        updated["anthro_source"] = "missing"
    else:
        updated["anthro_source"] = (
            updated["anthro_source"].astype(str).replace({"":
↪ "missing"}).fillna("missing")
        )
    if "anthro_obstime" not in updated.columns:
        updated["anthro_obstime"] = pd.NaT
    else:
        updated["anthro_obstime"] =
↪ pd.to_datetime(updated["anthro_obstime"], errors="coerce")
    if "anthro_hours_offset" not in updated.columns:
        updated["anthro_hours_offset"] = pd.NA
    updated["anthro_hours_offset"] = pd.to_numeric(
        updated["anthro_hours_offset"], errors="coerce"
    )
    if "anthro_timing_basis" not in updated.columns:
        updated["anthro_timing_basis"] = "missing"
    else:
        updated["anthro_timing_basis"] = (
            updated["anthro_timing_basis"].astype(str).replace({"":
↪ "missing"}).fillna("missing")

```

```

    )

    missing_mask = updated["anthro_timing_tier"].eq("missing")
    pre_mask = updated["anthro_timing_tier"].eq("pre_ed_365")
    post_mask = updated["anthro_timing_tier"].eq("post_ed_365")
    updated.loc[pre_mask, "anthro_timing_uncertain"] = False
    updated.loc[post_mask, "anthro_timing_uncertain"] = True
    updated.loc[missing_mask, "anthro_timing_uncertain"] = pd.NA
    updated.loc[pre_mask | post_mask, "anthro_source"] = "omr"
    updated.loc[missing_mask, "anthro_source"] = "missing"
    updated.loc[pre_mask, "anthro_timing_basis"] = "pre"
    updated.loc[post_mask, "anthro_timing_basis"] = "post"
    updated.loc[missing_mask, "anthro_timing_basis"] = "missing"
    updated.loc[missing_mask, "anthro_obstime"] = pd.NaT
    updated.loc[missing_mask, "anthro_hours_offset"] = pd.NA

    return updated

def _tier_counts(frame: pd.DataFrame) -> dict[str, int]:
    tier_series =
↪ frame["anthro_timing_tier"].astype(str).fillna("missing")
    return {
        "pre_ed_365": int((tier_series == "pre_ed_365").sum()),
        "post_ed_365": int((tier_series == "post_ed_365").sum()),
        "missing": int((tier_series == "missing").sum()),
    }

required_ed = {"ed_stay_id", "subject_id", "ed_intime"}
missing_ed = sorted(required_ed.difference(ed_df.columns))
if missing_ed:
    raise KeyError(f"attach_closest_pre_ed_omr missing ED columns:
↪ {missing_ed}")

required_omr = {"subject_id", "chartdate_dt", "result_name",
↪ "result_value_num"}
missing_omr = sorted(required_omr.difference(omr_df.columns))
if missing_omr:
    raise KeyError(f"attach_closest_pre_ed_omr missing OMR columns:
↪ {missing_omr}")

diagnostics: dict[str, Any] = {
    "window_days": int(window_days),
    "source_rows_prepared": int(len(omr_df)),

```



```

        "parsed_value_rows": int(omr_df["result_value_num"].notna().sum()),
        "ed_rows": int(len(ed_df)),
    }

    empty_window_diagnostics = {
        "nonnegative_candidate_rows": 0,
        "pre_window_candidate_rows": 0,
        "post_window_candidate_rows": 0,
        "closest_absolute_candidate_rows": 0,
        "within_window_candidate_rows": 0,
    }

    ed_norm = ed_df[["ed_stay_id", "subject_id", "ed_intime"]].copy()
    ed_norm["subject_id"] = _to_int64(ed_norm["subject_id"])
    ed_norm["ed_intime_dt"] = pd.to_datetime(ed_norm["ed_intime"],
    ↪ errors="coerce")
    ed_norm["ed_date_dt"] = ed_norm["ed_intime_dt"].dt.floor("D")
    ed_norm = ed_norm.loc[ed_norm["subject_id"].notna() &
    ↪ ed_norm["ed_date_dt"].notna()]

    if omr_df.empty or ed_norm.empty:
        updated = _with_default_anthro_columns(ed_df)
        tier_counts = _tier_counts(updated)
        total_rows = max(int(len(updated)), 1)
        diagnostics.update(
            {
                "ed_rows_eligible_for_join": int(len(ed_norm)),
                "subject_overlap_count": 0,
                "candidate_rows_after_subject_join": 0,
                **empty_window_diagnostics,
                "within_window_candidate_rows": 0,
                "eligible_ed_stays_with_candidates": 0,
                "attached_non_null_counts": {
                    column_name: int(updated[column_name].notna().sum())
                    for column_name in OMR_OUTPUT_COLUMNS
                },
                "attached_any_non_null_rows": int(
    ↪ updated[list(OMR_OUTPUT_COLUMNS)].notna().any(axis=1).sum()
                ),
                "selected_tier_counts": tier_counts,
                "selected_tier_rates": {

```

```

        key: float(value / total_rows) for key, value in
↪ tier_counts.items()
        },
        "timing_uncertain_count": int(
            updated["anthro_timing_uncertain"].fillna(False).sum()
        ),
    }
)
return updated, diagnostics

omr_pivot = (
    omr_df.pivot_table(
        index=["subject_id", "chartdate_dt"],
        columns="result_name",
        values="result_value_num",
        aggfunc="first",
    )
    .reset_index()
    .copy()
)

shared_subjects =
↪ set(ed_norm["subject_id"].dropna().astype(int)).intersection(
    set(omr_pivot["subject_id"].dropna().astype(int))
)
diagnostics["ed_rows_eligible_for_join"] = int(len(ed_norm))
diagnostics["subject_overlap_count"] = int(len(shared_subjects))

merged = ed_norm.merge(omr_pivot, on="subject_id", how="left")
diagnostics["candidate_rows_after_subject_join"] = int(
    merged["chartdate_dt"].notna().sum()
)

merged["days_before"] = (
    merged["ed_date_dt"] - pd.to_datetime(merged["chartdate_dt"],
↪ errors="coerce")
).dt.days
valid_days = merged["days_before"].dropna()
diagnostics["days_before_min"] = (
    int(valid_days.min()) if not valid_days.empty else None
)
diagnostics["days_before_max"] = (
    int(valid_days.max()) if not valid_days.empty else None
)

```

```

)
pre_window_mask = (
    merged["days_before"].notna()
    & merged["days_before"].ge(0)
    & merged["days_before"].le(window_days)
)
post_window_mask = (
    merged["days_before"].notna()
    & merged["days_before"].lt(0)
    & merged["days_before"].ge(-window_days)
)
abs_window_mask = (
    merged["days_before"].notna()
    & merged["days_before"].abs().le(window_days)
)

diagnostics["nonnegative_candidate_rows"] = int((merged["days_before"] >=
↪ 0).sum())
diagnostics["pre_window_candidate_rows"] = int(pre_window_mask.sum())
diagnostics["post_window_candidate_rows"] = int(post_window_mask.sum())
diagnostics["closest_absolute_candidate_rows"] =
↪ int(abs_window_mask.sum())

pre_candidates = merged.loc[pre_window_mask].copy()
post_candidates = merged.loc[post_window_mask].copy()
diagnostics["within_window_candidate_rows"] = int(len(pre_candidates))

selected_parts: list[pd.DataFrame] = []
selected_stays: set[Any] = set()

if not pre_candidates.empty:
    pre_selected = (
        pre_candidates.sort_values(
            ["ed_stay_id", "days_before", "chartdate_dt"],
            ascending=[True, True, False],
        )
        .groupby("ed_stay_id", as_index=False)
        .first()
    )
    pre_selected["anthro_timing_tier"] = "pre_ed_365"
    selected_parts.append(pre_selected)
    selected_stays.update(pre_selected["ed_stay_id"].tolist())

```

```

if not post_candidates.empty:
    post_candidates["days_after_abs"] =
↪ post_candidates["days_before"].abs()
    post_selected = (
        post_candidates.sort_values(
            ["ed_stay_id", "days_after_abs", "chartdate_dt"],
            ascending=[True, True, True],
        )
        .groupby("ed_stay_id", as_index=False)
        .first()
    )
    post_selected = post_selected.loc[
        ~post_selected["ed_stay_id"].isin(selected_stays)
    ].copy()
    if not post_selected.empty:
        post_selected["anthro_timing_tier"] = "post_ed_365"
        selected_parts.append(post_selected)

if selected_parts:
    selected = pd.concat(selected_parts, ignore_index=True)
    selected = selected.rename(
        columns={
            "bmi": "bmi_closest_pre_ed",
            "height": "height_closest_pre_ed",
            "weight": "weight_closest_pre_ed",
            "chartdate_dt": "anthro_chartdate",
            "days_before": "anthro_days_offset",
        }
    )
    selected["anthro_chartdate"] = pd.to_datetime(
        selected["anthro_chartdate"], errors="coerce"
    )
    selected["anthro_days_offset"] = pd.to_numeric(
        selected["anthro_days_offset"], errors="coerce"
    ).astype("Int64")
    selected["anthro_timing_uncertain"] =
↪ selected["anthro_timing_tier"].eq(
        "post_ed_365"
    )
    selected["anthro_source"] = "omr"
    selected["anthro_obstime"] = pd.to_datetime(
        selected["anthro_chartdate"], errors="coerce"
    )

```

```

selected["anthro_hours_offset"] = pd.to_numeric(
    selected["anthro_days_offset"], errors="coerce"
) * 24.0
selected["anthro_timing_basis"] = selected["anthro_timing_tier"].map(
    {"pre_ed_365": "pre", "post_ed_365": "post"}
).fillna("missing")

updates = selected[
    [
        "ed_stay_id",
        *OMR_OUTPUT_COLUMNS,
        *OMR_PROVENANCE_COLUMNS,
    ]
].copy()
updated = ed_df.merge(updates, on="ed_stay_id", how="left")
else:
    updated = ed_df.copy()

updated = _with_default_anthro_columns(updated)

diagnostics["eligible_ed_stays_with_candidates"] = int(
    updated["anthro_timing_tier"].isin({"pre_ed_365",
↪ "post_ed_365"}).sum()
)
diagnostics["attached_non_null_counts"] = {
    column_name: int(updated[column_name].notna().sum())
    for column_name in OMR_OUTPUT_COLUMNS
}
diagnostics["attached_any_non_null_rows"] = int(
    updated[list(OMR_OUTPUT_COLUMNS)].notna().any(axis=1).sum()
)
tier_counts = _tier_counts(updated)
diagnostics["selected_tier_counts"] = tier_counts
total_rows = max(int(len(updated)), 1)
diagnostics["selected_tier_rates"] = {
    key: float(value / total_rows) for key, value in tier_counts.items()
}
diagnostics["timing_uncertain_count"] = int(
    updated["anthro_timing_uncertain"].fillna(False).sum()
)
diagnostics["anthro_source_counts"] = {
    str(key): int(value)
    for key, value in updated["anthro_source"].fillna("missing")
}

```

```

        .astype(str)
        .value_counts(dropna=False)
        .items()
    }

    return updated, diagnostics

def _infer_route_hint_text(value: object) -> str | None:
    text = "" if pd.isna(value) else str(value).strip().lower()
    if not text or text in {"nan", "none"}:
        return None
    if _ARTERIAL_HINT_PATTERN.search(text):
        return "arterial"
    if _VENOUS_HINT_PATTERN.search(text):
        return "venous"
    return None

def _resolve_route_hints(values: Sequence[str]) -> tuple[str | None, bool,
    ↪ int]:
    hints = [str(value).strip().lower() for value in values if
    ↪ str(value).strip()]
    arterial_n = sum(1 for value in hints if value == "arterial")
    venous_n = sum(1 for value in hints if value == "venous")
    conflict = arterial_n > 0 and venous_n > 0
    if conflict:
        return None, True, int(len(hints))
    if arterial_n > 0:
        return "arterial", False, int(len(hints))
    if venous_n > 0:
        return "venous", False, int(len(hints))
    return None, False, int(len(hints))

def _classify_specimen_type_text(value: object) -> str | None:
    text = "" if pd.isna(value) else str(value).strip().lower()
    if not text:
        return None
    if re.search(r"arter|(?:\b)art(?:\b|$)", text):
        return "arterial"
    if re.search(r"ven|mixed|central", text):
        return "venous"

```

```

return None

def assign_panel_source_from_specimen(
    panel_df: pd.DataFrame,
    labs_df: pd.DataFrame,
    *,
    specimen_source_itemids: Sequence[int] | None = None,
) -> tuple[pd.DataFrame, dict[str, Any]]:
    """Assign gas source using specimen-type rows only (deterministic core
    ↪ policy)."""
    required_panel = {"ed_stay_id", "specimen_id"}
    missing_panel = sorted(required_panel.difference(panel_df.columns))
    if missing_panel:
        raise KeyError(
            "assign_panel_source_from_specimen missing panel columns: "
            f"{missing_panel}"
        )
    required_labs = {"ed_stay_id", "specimen_id", "itemid"}
    missing_labs = sorted(required_labs.difference(labs_df.columns))
    if missing_labs:
        raise KeyError(
            "assign_panel_source_from_specimen missing labs columns: "
            f"{missing_labs}"
        )

    key_cols = ["ed_stay_id", "specimen_id"]
    panel = panel_df.copy()
    panel["source"] = "unknown"
    panel["source_inference_tier"] = "fallback_unknown"
    panel["source_hint_conflict"] = False
    panel["source_hint_count"] = pd.Series(0, index=panel.index,
    ↪ dtype="Int64")
    panel["source_hint_specimen_text"] = pd.Series(pd.NA, index=panel.index,
    ↪ dtype="string")

    specimen_ids = {
        int(value) for value in (specimen_source_itemids or []) if
        ↪ pd.notna(value)
    }
    if not specimen_ids or "value_text" not in labs_df.columns:
        diagnostics = summarize_gas_source(panel)
        diagnostics.update(

```

```

        {
            "mode": "specimen_only",
            "resolved_rows": int(panel["source"].isin({"arterial",
↪      "venous"}).sum()),
            "resolved_rate": float(panel["source"].isin({"arterial",
↪      "venous"}).mean())
            if len(panel)
            else 0.0,
            "unresolved_specimen_id_count": int(
                pd.to_numeric(panel["specimen_id"],
↪ errors="coerce").nunique(dropna=True)
            ),
            "unresolved_specimen_id_examples": [],
        }
    )
    return panel, diagnostics

labs = labs_df.copy()
labs["itemid"] = pd.to_numeric(labs["itemid"],
↪ errors="coerce").astype("Int64")
labs["specimen_id"] = pd.to_numeric(labs["specimen_id"],
↪ errors="coerce").astype("Int64")
labs = labs.loc[labs["itemid"].isin(specimen_ids) &
↪ labs["specimen_id"].notna()].copy()
if labs.empty:
    diagnostics = summarize_gas_source(panel)
    diagnostics.update(
        {
            "mode": "specimen_only",
            "resolved_rows": int(panel["source"].isin({"arterial",
↪      "venous"}).sum()),
            "resolved_rate": float(panel["source"].isin({"arterial",
↪      "venous"}).mean())
            if len(panel)
            else 0.0,
            "unresolved_specimen_id_count": int(
                pd.to_numeric(panel["specimen_id"],
↪ errors="coerce").nunique(dropna=True)
            ),
            "unresolved_specimen_id_examples": [],
        }
    )
    return panel, diagnostics

```



```

labs["specimen_text_raw"] = (
    labs["value_text"].astype("string").fillna("").str.strip()
)
labs["specimen_text_norm"] = labs["specimen_text_raw"].str.lower()
labs["specimen_route_hint"] =
↪ labs["specimen_text_norm"].map(_classify_specimen_type_text)

grouped = (
    labs.groupby(key_cols, dropna=False)
    .agg(
        route_hints=("specimen_route_hint", list),
        specimen_texts=("specimen_text_raw", list),
    )
    .reset_index()
)

mapping_rows: list[dict[str, Any]] = []
for _, grouped_row in grouped.iterrows():
    source, conflict, hint_count =
↪ _resolve_route_hints(grouped_row["route_hints"])
    specimen_texts = [
        str(value).strip()
        for value in grouped_row["specimen_texts"]
        if str(value).strip()
    ]
    specimen_text = specimen_texts[0] if specimen_texts else None
    mapping_rows.append(
        {
            "ed_stay_id": grouped_row["ed_stay_id"],
            "specimen_id": grouped_row["specimen_id"],
            "source": source if source is not None else "unknown",
            "source_inference_tier": "specimen_text" if source is not
↪ None else "fallback_unknown",
            "source_hint_conflict": bool(conflict),
            "source_hint_count": int(hint_count),
            "source_hint_specimen_text": specimen_text,
        }
    )

mapping = pd.DataFrame(mapping_rows)
if not mapping.empty:

```

```

    panel = panel.merge(mapping, on=key_cols, how="left", suffixes=("",
↪ "_specimen"))
    panel["source"] = (
        panel.get("source_specimen")
        .fillna(panel["source"])
        .astype("string")
        .fillna("unknown")
    )
    panel["source_inference_tier"] = (
        panel.get("source_inference_tier_specimen")
        .fillna(panel["source_inference_tier"])
        .astype("string")
        .fillna("fallback_unknown")
    )
    panel["source_hint_conflict"] = (
        panel.get("source_hint_conflict_specimen")
        .fillna(panel["source_hint_conflict"])
        .astype(bool)
    )
    panel["source_hint_count"] = pd.to_numeric(
        panel.get("source_hint_count_specimen").fillna(panel[
↪ "source_hint_count"]),
        errors="coerce",
    ).fillna(0).astype("Int64")
    panel["source_hint_specimen_text"] = (
        panel.get("source_hint_specimen_text_specimen")
        .fillna(panel["source_hint_specimen_text"])
        .astype("string")
    )
    panel = panel.drop(
        columns=[
            column_name
            for column_name in (
                "source_specimen",
                "source_inference_tier_specimen",
                "source_hint_conflict_specimen",
                "source_hint_count_specimen",
                "source_hint_specimen_text_specimen",
            )
            if column_name in panel.columns
        ]
    )

```

```

diagnostics = summarize_gas_source(panel)
unresolved = panel.loc[panel["source"].eq("unknown"),
↪ key_cols].drop_duplicates()
diagnostics.update(
    {
        "mode": "specimen_only",
        "resolved_rows": int(panel["source"].isin({"arterial",
↪ "venous"}).sum()),
        "resolved_rate": float(panel["source"].isin({"arterial",
↪ "venous"}).mean())
        if len(panel)
        else 0.0,
        "unresolved_specimen_id_count": int(
            pd.to_numeric(unresolved["specimen_id"],
↪ errors="coerce").nunique(dropna=True)
        ),
        "unresolved_specimen_id_examples": unresolved["specimen_id"]
            .dropna()
            .astype(int)
            .head(20)
            .tolist(),
    }
)
return panel, diagnostics

def infer_panel_gas_source_metadata(
    panel_df: pd.DataFrame,
    labs_df: pd.DataFrame,
    labitems_df: pd.DataFrame,
    *,
    specimen_source_itemids: Sequence[int] | None = None,
    pco2_itemids: Sequence[int] | None = None,
    mode: str = "metadata_only",
) -> tuple[pd.DataFrame, dict[str, Any]]:
    """Infer gas panel source from metadata/text hints with tier diagnostics.

    Tier precedence:
    1. Specimen/source row text (`value_text` from specimen/source
↪ itemids),
    2. `d_labitems` label/fluid hints,
    3. Panel co-occurrence hints (pCO2-labeled rows + free-text),
    4. Fallback ``unknown``.

```

```

"""
if mode != "metadata_only":
    raise ValueError(
        "infer_panel_gas_source_metadata supports only
        ↪ mode='metadata_only'."
    )

required_panel = {"ed_stay_id", "specimen_id"}
missing_panel = sorted(required_panel.difference(panel_df.columns))
if missing_panel:
    raise KeyError(
        "infer_panel_gas_source_metadata missing panel columns: "
        f"{missing_panel}"
    )
required_labs = {"ed_stay_id", "specimen_id", "itemid"}
missing_labs = sorted(required_labs.difference(labs_df.columns))
if missing_labs:
    raise KeyError(
        "infer_panel_gas_source_metadata missing labs columns: "
        f"{missing_labs}"
    )
required_labitems = {"itemid"}
missing_labitems =
↪ sorted(required_labitems.difference(labitems_df.columns))
if missing_labitems:
    raise KeyError(
        "infer_panel_gas_source_metadata missing labitems columns: "
        f"{missing_labitems}"
    )

key_cols = ["ed_stay_id", "specimen_id"]
panel = panel_df.copy()
panel["source"] = pd.Series(pd.NA, index=panel.index, dtype="string")
panel["source_inference_tier"] = "fallback_unknown"
panel["source_hint_conflict"] = pd.Series(False, index=panel.index,
↪ dtype="boolean")
↪ panel["source_hint_count"] = pd.Series(pd.NA, index=panel.index,
↪ dtype="Int64")

labs = labs_df.copy()
labs["itemid"] = pd.to_numeric(labs["itemid"],
↪ errors="coerce").astype("Int64")

```

```

    labs["specimen_id"] = pd.to_numeric(labs["specimen_id"],
↪ errors="coerce").astype(
        "Int64"
    )
    labs = labs.loc[labs["itemid"].notna() &
↪ labs["specimen_id"].notna()].copy()
    if labs.empty:
        panel["source"] = "unknown"
        diagnostics = summarize_gas_source(panel)
        diagnostics.update(
            {
                "mode": mode,
                "resolved_rows": 0,
                "resolved_rate": 0.0,
                "unresolved_specimen_id_count": 0,
                "unresolved_specimen_id_examples": [],
                "unresolved_value_text_top": {},
                "unresolved_label_top": {},
            }
        )
        return panel, diagnostics

    if "value_text" in labs.columns:
        labs["value_text_norm"] = (
            labs["value_text"]
            .astype("string")
            .fillna("")
            .str.strip()
            .str.lower()
        )
    else:
        labs["value_text_norm"] = ""

    labitems = labitems_df.copy()
    labitems["itemid"] = pd.to_numeric(labitems["itemid"],
↪ errors="coerce").astype(
        "Int64"
    )
    for text_col in ("label", "fluid"):
        if text_col not in labitems.columns:
            labitems[text_col] = ""
        labitems["label_norm"] =
↪ labitems["label"].astype("string").fillna("").str.strip()

```

```

labitems["fluid_norm"] =
↳ labitems["fluid"].astype("string").fillna("").str.strip()
labitems["item_route_hint"] = (
    (labitems["label_norm"] + " " + labitems["fluid_norm"])
    .str.strip()
    .map(_infer_route_hint_text)
)
item_meta = (
    labitems.dropna(subset=["itemid"])
    .drop_duplicates(subset=["itemid"])
    .set_index("itemid")[["label_norm", "fluid_norm", "item_route_hint"]]
)
labs = labs.merge(item_meta, left_on="itemid", right_index=True,
↳ how="left")
labs["item_route_hint"] = labs["item_route_hint"].astype("string")
labs["text_route_hint"] =
↳ labs["value_text_norm"].map(_infer_route_hint_text).astype(
    "string"
)
labs["label_fluid_text"] = (
    labs["label_norm"].fillna("").astype(str).str.strip()
    + " | "
    + labs["fluid_norm"].fillna("").astype(str).str.strip()
).str.strip(" |")

def _build_tier_mapping(
    frame: pd.DataFrame,
    *,
    hint_column: str,
    tier_name: str,
) -> pd.DataFrame:
    subset = frame[key_cols + [hint_column]].copy()
    subset = subset.loc[subset[hint_column].notna()].copy()
    if subset.empty:
        return pd.DataFrame(
            columns=[
                *key_cols,
                "tier_source",
                "tier_conflict",
                "tier_hint_count",
                "tier_name",
            ]
        )

```

```

grouped = (
    subset.groupby(key_cols, dropna=False)[hint_column]
    .agg(list)
    .reset_index(name="hints")
)
rows: list[dict[str, Any]] = []
for _, grouped_row in grouped.iterrows():
    source, conflict, hint_count =
↪ _resolve_route_hints(grouped_row["hints"])
    rows.append(
        {
            "ed_stay_id": grouped_row["ed_stay_id"],
            "specimen_id": grouped_row["specimen_id"],
            "tier_source": source,
            "tier_conflict": bool(conflict),
            "tier_hint_count": int(hint_count),
            "tier_name": tier_name,
        }
    )
resolved = pd.DataFrame(rows)
return resolved.loc[resolved["tier_source"]
↪ ].notna()].reset_index(drop=True)

specimen_source_itemid_set = {
    int(item) for item in (specimen_source_itemids or []) if
↪ pd.notna(item)
}
pco2_itemid_set = {int(item) for item in (pco2_itemids or []) if
↪ pd.notna(item)}

tier1 = _build_tier_mapping(
    labs.loc[labs["itemid"].astype("Int64")
↪ ].isin(specimen_source_itemid_set)],
    hint_column="text_route_hint",
    tier_name="specimen_text",
)
tier2 = _build_tier_mapping(
    labs,
    hint_column="item_route_hint",
    tier_name="label_fluid",
)
tier_cluster = pd.DataFrame(

```

```

        columns=[
            *key_cols,
            "tier_source",
            "tier_conflict",
            "tier_hint_count",
            "tier_name",
        ]
    )
    if "panel_time" in panel_df.columns:
        panel_times = panel_df[key_cols + ["panel_time"]].copy()
        panel_times["panel_time"] = pd.to_datetime(panel_times["panel_time"],
↪ errors="coerce")

    tier_seed = pd.concat([tier1, tier2], ignore_index=True)
    if not tier_seed.empty:
        tier_seed = (
            tier_seed[key_cols + ["tier_source"]]
            .dropna(subset=["tier_source"])
            .drop_duplicates(subset=key_cols)
        )
        panel_seed = panel_times.merge(tier_seed, on=key_cols,
↪ how="left")
        known = panel_seed.loc[panel_seed["tier_source"].notna()].copy()
        unknown = panel_seed.loc[panel_seed["tier_source"].isna()].copy()
        if not known.empty and not unknown.empty:
            pairs = unknown.merge(
                known[["ed_stay_id", "specimen_id", "panel_time",
↪ "tier_source"]],
                on="ed_stay_id",
                how="left",
                suffixes=("_unknown", "_known"),
            )
            pairs["delta_minutes"] = (
                (pairs["panel_time_known"] -
↪ pairs["panel_time_unknown"]).abs().dt.total_seconds()
                / 60.0
            )
            pairs = pairs.loc[pairs["delta_minutes"].notna() &
↪ pairs["delta_minutes"].le(60.0)].copy()
            if not pairs.empty:
                keys = ["ed_stay_id", "specimen_id_unknown"]
                min_delta =
↪ pairs.groupby(keys)["delta_minutes"].transform("min")

```



```

nearest =
↪ pairs.loc[pairs["delta_minutes"].eq(min_delta)].copy()
    conflict_stats = (
        nearest.groupby(keys)["tier_source_known"]
        .nunique(dropna=True)
        .reset_index(name="source_nunique")
    )
    nearest = nearest.merge(conflict_stats, on=keys,
↪ how="left")
    nearest =
↪ nearest.loc[nearest["source_nunique"].eq(1)].copy()
    if not nearest.empty:
        nearest = nearest.sort_values(keys +
↪ ["delta_minutes"])
        nearest = nearest.groupby(keys,
↪ as_index=False).first()
        tier_cluster = nearest.rename(
            columns={
                "specimen_id_unknown": "specimen_id",
                "tier_source_known": "tier_source",
            }
        )[
            ["ed_stay_id", "specimen_id", "tier_source"]
        ].copy()
        tier_cluster["tier_conflict"] = False
        tier_cluster["tier_hint_count"] = 1
        tier_cluster["tier_name"] = "cluster_inheritance"

tier3_candidates = labs.copy()
if pco2_itemid_set:
    tier3_candidates = tier3_candidates.loc[
        tier3_candidates["itemid"].astype("Int64").isin(pco2_itemid_set)
        | tier3_candidates["text_route_hint"].notna()
    ].copy()
    tier3_candidates["panel_route_hint"] =
↪ tier3_candidates["item_route_hint"].fillna(
        tier3_candidates["text_route_hint"]
    )
    tier3 = _build_tier_mapping(
        tier3_candidates,
        hint_column="panel_route_hint",
        tier_name="panel_cooccurrence",
    )

```

```

for tier_frame in (tier1, tier2, tier_cluster, tier3):
    if tier_frame.empty:
        continue
    panel = panel.merge(
        tier_frame[
            key_cols
            + ["tier_source", "tier_conflict", "tier_hint_count",
               ↪ "tier_name"]
        ],
        on=key_cols,
        how="left",
    )
    assign_mask = panel["source"].isna() & panel["tier_source"].notna()
    panel.loc[assign_mask, "source"] = panel.loc[assign_mask,
    ↪ "tier_source"]
    panel.loc[assign_mask, "source_inference_tier"] = panel.loc[
        assign_mask, "tier_name"
    ]
    selected_conflict = panel.loc[assign_mask,
    ↪ "tier_conflict"].astype("boolean")
    panel.loc[assign_mask, "source_hint_conflict"] =
    ↪ selected_conflict.fillna(False)
    panel.loc[assign_mask, "source_hint_count"] = pd.to_numeric(
        panel.loc[assign_mask, "tier_hint_count"], errors="coerce"
    ).astype("Int64")
    panel = panel.drop(
        columns=["tier_source", "tier_conflict", "tier_hint_count",
    ↪ "tier_name"]
    )

    panel["source"] = panel["source"].fillna("unknown").astype("string")
    panel.loc[panel["source"].eq("unknown"), "source_inference_tier"] =
    ↪ panel.loc[
        panel["source"].eq("unknown"), "source_inference_tier"
    ].replace({"": "fallback_unknown"}).fillna("fallback_unknown")
    panel["source_inference_tier"] =
    ↪ panel["source_inference_tier"].astype("string")
    panel["source_hint_conflict"] = (
        panel["source_hint_conflict"].fillna(False).astype("boolean")
    )

    diagnostics = summarize_gas_source(panel)

```

```

total = max(int(len(panel)), 1)
diagnostics["mode"] = mode
diagnostics["resolved_rows"] = int(
    panel["source"].isin({"arterial", "venous"}).sum()
)
diagnostics["resolved_rate"] = float(diagnostics["resolved_rows"] /
↪ total)

unresolved = panel.loc[panel["source"].eq("unknown"),
↪ key_cols].drop_duplicates()
diagnostics["unresolved_specimen_id_count"] =
↪ int(unresolved["specimen_id"].nunique())
diagnostics["unresolved_specimen_id_examples"] = (
    unresolved["specimen_id"].dropna().astype(int).head(20).tolist()
)
if unresolved.empty:
    diagnostics["unresolved_value_text_top"] = {}
    diagnostics["unresolved_label_top"] = {}
    return panel, diagnostics

unresolved_labs = labs.merge(unresolved, on=key_cols, how="inner")
unresolved_value_text = (
    unresolved_labs["value_text_norm"]
    .replace({"": pd.NA})
    .dropna()
    .astype(str)
    .value_counts()
    .head(15)
)
unresolved_label_text = (
    unresolved_labs["label_fluid_text"]
    .replace({"": pd.NA})
    .dropna()
    .astype(str)
    .value_counts()
    .head(15)
)
diagnostics["unresolved_value_text_top"] = {
    str(key): int(value) for key, value in unresolved_value_text.items()
}
diagnostics["unresolved_label_top"] = {
    str(key): int(value) for key, value in unresolved_label_text.items()
}

```

```

return panel, diagnostics

def summarize_gas_source(panel_df: pd.DataFrame) -> dict[str, Any]:
    """Summarize gas source composition from panel-level records."""
    total_rows = int(len(panel_df))
    if total_rows == 0:
        return {
            "panel_rows": 0,
            "source_present": bool("source" in panel_df.columns),
            "source_counts": {},
            "source_rates": {},
            "all_other_or_unknown": False,
            "tier_counts": {},
            "tier_rates": {},
        }

    if "source" not in panel_df.columns:
        return {
            "panel_rows": total_rows,
            "source_present": False,
            "source_counts": {},
            "source_rates": {},
            "all_other_or_unknown": True,
            "tier_counts": {},
            "tier_rates": {},
        }

    source = (
        panel_df["source"]
        .astype(str)
        .str.strip()
        .str.lower()
        .replace({"": "unknown", "nan": "unknown", "none": "unknown",
        ↪ "other": "unknown"})
        .fillna("unknown")
    )
    source_counts = {str(key): int(value) for key, value in
    ↪ source.value_counts(dropna=False).items()}
    source_rates = {key: float(value / total_rows) for key, value in
    ↪ source_counts.items()}
    all_other_or_unknown = bool(total_rows > 0 and
    ↪ set(source_counts.keys()).issubset({"unknown"}))

```

```

tier_counts: dict[str, int] = {}
tier_rates: dict[str, float] = {}
if "source_inference_tier" in panel_df.columns:
    tier = (
        panel_df["source_inference_tier"]
        .astype(str)
        .str.strip()
        .str.lower()
        .replace({"": "unknown", "nan": "unknown", "none": "unknown"})
        .fillna("unknown")
    )
    tier_counts = {
        str(key): int(value) for key, value in
        ↪ tier.value_counts(dropna=False).items()
    }
    tier_rates = {key: float(value / total_rows) for key, value in
    ↪ tier_counts.items()}

return {
    "panel_rows": total_rows,
    "source_present": True,
    "source_counts": source_counts,
    "source_rates": source_rates,
    "all_other_or_unknown": all_other_or_unknown,
    "tier_counts": tier_counts,
    "tier_rates": tier_rates,
}

def assert_gas_source_coverage(
    gas_source_audit: Mapping[str, Any],
    *,
    fail_on_all_other_source: bool = True,
) -> None:
    """Fail fast when source attribution collapses to all other/unknown."""
    if not fail_on_all_other_source:
        return
    if int(gas_source_audit.get("panel_rows", 0)) <= 0:
        return
    if bool(gas_source_audit.get("all_other_or_unknown", False)):
        raise ValueError(
            "Gas source attribution classified all panel rows as
            ↪ other/unknown. "

```

```

        "Set COHORT_FAIL_ON_ALL_OTHER_SOURCE=0 to bypass this guard."
    )

def build_gas_source_overlap_summary(ed_df: pd.DataFrame) -> pd.DataFrame:
    """Build ABG/VBG/OTHER overlap counts and percentages."""
    frame = ed_df.copy()
    abg = pd.to_numeric(frame.get("flag_abg_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int)
    vbg = pd.to_numeric(frame.get("flag_vbg_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int)
    other = pd.to_numeric(frame.get("flag_other_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int)

    labels = []
    for a, v, o in zip(abg.tolist(), vbg.tolist(), other.tolist(),
↪ strict=False):
        parts: list[str] = []
        if a == 1:
            parts.append("ABG")
        if v == 1:
            parts.append("VBG")
        if o == 1:
            parts.append("OTHER")
        labels.append("+".join(parts) if parts else "NO_GAS")

    counts = pd.Series(labels, dtype="string").value_counts(dropna=False,
↪ ).rename_axis("gas_overlap").reset_index(name="count")
    total = max(int(counts["count"].sum()), 1)
    counts["percent"] = counts["count"].astype(float) / total * 100.0
    counts["percent"] = counts["percent"].round(2)
    return counts.sort_values(["count", "gas_overlap"], ascending=[False,
↪ True]).reset_index(drop=True)

def add_gas_model_fields(
    ed_df: pd.DataFrame,
    *,
    ranges: Mapping[str, tuple[float, float]] | None = None,
) -> tuple[pd.DataFrame, pd.DataFrame]:
    """Create cleaned gas model fields and a field-level outlier audit."""
    resolved_ranges = dict(ranges or DEFAULT_GAS_MODEL_RANGES)
    updated = ed_df.copy()

```

```

audit_rows: list[dict[str, Any]] = []

for raw_column, (lower_bound, upper_bound) in resolved_ranges.items():
    if raw_column not in updated.columns:
        continue
    numeric = pd.to_numeric(updated[raw_column], errors="coerce")
    out_of_range = numeric.notna() & ((numeric < lower_bound) | (numeric
↪ > upper_bound))
    model_column = f"{raw_column}_model"
    outlier_flag_column = f"{raw_column}_outlier_flag"
    updated[model_column] = numeric.where(~out_of_range)
    updated[outlier_flag_column] = out_of_range.astype("boolean")
    nonnull_n = int(numeric.notna().sum())
    outlier_n = int(out_of_range.sum())
    examples = (
        numeric.loc[out_of_range]
        .round(4)
        .value_counts()
        .head(5)
        .index.astype(str)
        .tolist()
    )
    audit_rows.append(
        {
            "domain": "gas",
            "raw_column": raw_column,
            "model_column": model_column,
            "outlier_flag_column": outlier_flag_column,
            "lower_bound": float(lower_bound),
            "upper_bound": float(upper_bound),
            "nonnull_n": nonnull_n,
            "out_of_range_n": outlier_n,
            "out_of_range_pct": float(outlier_n / nonnull_n) if nonnull_n
↪ else 0.0,
            "example_outlier_values": "; ".join(examples),
        }
    )

audit =
↪ pd.DataFrame(audit_rows).sort_values("raw_column").reset_index(drop=True)
return updated, audit

```

```

def attach_charted_anthro_fallback(
    ed_df: pd.DataFrame,
    charted_df: pd.DataFrame,
    *,
    nearest_anytime: bool = True,
) -> tuple[pd.DataFrame, dict[str, Any]]:
    """Fill missing anthropometrics from charted records with nearest-time
    ↪ selection."""
    required_ed = {"ed_stay_id", "subject_id", "ed_intime"}
    missing_ed = sorted(required_ed.difference(ed_df.columns))
    if missing_ed:
        raise KeyError(f"attach_charted_anthro_fallback missing ED columns: "
            ↪ {missing_ed})

    required_charted = {"subject_id", "obs_time", "result_name",
    ↪ "result_value_num"}
    missing_charted = sorted(required_charted.difference(charted_df.columns))
    if missing_charted:
        raise KeyError(
            "attach_charted_anthro_fallback missing charted columns: "
            ↪ f"{missing_charted}"
        )

    updated = ed_df.copy()
    for output_column in OMR_OUTPUT_COLUMNS:
        if output_column not in updated.columns:
            updated[output_column] = pd.NA
    if "anthro_source" not in updated.columns:
        updated["anthro_source"] = "missing"
    else:
        updated["anthro_source"] = (
            updated["anthro_source"].astype(str).replace({"":
    ↪ "missing"}).fillna("missing")
        )
    if "anthro_obstime" not in updated.columns:
        updated["anthro_obstime"] = pd.NaT
    else:
        updated["anthro_obstime"] = pd.to_datetime(updated["anthro_obstime"],
    ↪ errors="coerce")
    if "anthro_hours_offset" not in updated.columns:
        updated["anthro_hours_offset"] = pd.NA
    updated["anthro_hours_offset"] =
    ↪ pd.to_numeric(updated["anthro_hours_offset"], errors="coerce")

```



```

if "anthro_timing_basis" not in updated.columns:
    updated["anthro_timing_basis"] = "missing"
if "anthro_timing_uncertain" not in updated.columns:
    updated["anthro_timing_uncertain"] = pd.Series([pd.NA] *
↪ len(updated), dtype="boolean")
else:
    updated["anthro_timing_uncertain"] =
↪ updated["anthro_timing_uncertain"].astype("boolean")

charted = charted_df.copy()
charted["subject_id"] = _to_int64(charthed["subject_id"])
charted["obs_time"] = pd.to_datetime(charthed["obs_time"],
↪ errors="coerce")
charted["result_name"] =
↪ charted["result_name"].astype(str).str.strip().str.lower()
charted["result_value_num"] = pd.to_numeric(charthed["result_value_num"],
↪ errors="coerce")
if "source" not in charted.columns:
    charted["source"] = "icu_charthed"
charted["source"] = charted["source"].astype(str).replace({"":
↪ "icu_charthed"}).fillna("icu_charthed")
charted = charted.loc[
    charted["subject_id"].notna()
    & charted["obs_time"].notna()
    & charted["result_name"].isin(OMR_RESULT_NAMES)
    & charted["result_value_num"].notna()
].copy()

diagnostics: dict[str, Any] = {
    "charted_rows_input": int(len(charthed_df)),
    "charted_rows_usable": int(len(charthed)),
    "nearest_anytime": bool(nearest_anytime),
    "filled_counts": {column: 0 for column in OMR_OUTPUT_COLUMNS},
    "rows_with_any_charthed_fill": 0,
}
if charted.empty:
    return updated, diagnostics

ed_norm = updated[["ed_stay_id", "subject_id", "ed_intime"]].copy()
ed_norm["subject_id"] = _to_int64(ed_norm["subject_id"])
ed_norm["ed_intime_dt"] = pd.to_datetime(ed_norm["ed_intime"],
↪ errors="coerce")

```

```

ed_norm = ed_norm.loc[ed_norm["subject_id"].notna() &
↪ ed_norm["ed_intime_dt"].notna()].copy()
if ed_norm.empty:
    return updated, diagnostics

merged = ed_norm.merge(charted, on="subject_id", how="inner")
if merged.empty:
    diagnostics["subject_overlap_count"] = 0
    return updated, diagnostics
diagnostics["subject_overlap_count"] =
↪ int(merged["subject_id"].nunique())
merged["hours_offset"] = (
    (merged["obs_time"] - merged["ed_intime_dt"]).dt.total_seconds() /
↪ 3600.0
)
merged["abs_hours_offset"] = merged["hours_offset"].abs()
if not nearest_anytime:
    merged = merged.loc[merged["abs_hours_offset"] <= 24.0].copy()

output_to_name = {
    "bmi_closest_pre_ed": "bmi",
    "height_closest_pre_ed": "height",
    "weight_closest_pre_ed": "weight",
}
any_fill_mask = pd.Series(False, index=updated.index)
for output_column, result_name in output_to_name.items():
    subset = merged.loc[merged["result_name"] == result_name].copy()
    if subset.empty:
        continue
    selected = (
        subset.sort_values(
            ["ed_stay_id", "abs_hours_offset", "obs_time"],
            ascending=[True, True, True],
        )
        .groupby("ed_stay_id", as_index=False)
        .first()
    )
    selected = selected.rename(
        columns={
            "result_value_num": f"{output_column}__candidate",
            "obs_time": f"{output_column}__obs_time",
            "hours_offset": f"{output_column}__hours_offset",
            "source": f"{output_column}__source",

```

```

    }
)
keep_columns = [
    "ed_stay_id",
    f"{output_column}__candidate",
    f"{output_column}__obs_time",
    f"{output_column}__hours_offset",
    f"{output_column}__source",
]
updated = updated.merge(selected[keep_columns], on="ed_stay_id",
↪ how="left")

    fill_mask = updated[output_column].isna() &
↪ updated[f"{output_column}__candidate"].notna()
    updated.loc[fill_mask, output_column] = updated.loc[fill_mask,
↪ f"{output_column}__candidate"]

    provenance_mask = fill_mask &
↪ updated["anthro_source"].isin({"missing", "nan"})
    updated.loc[provenance_mask, "anthro_source"] = updated.loc[
        provenance_mask, f"{output_column}__source"
    ]
    updated.loc[provenance_mask, "anthro_obstime"] = pd.to_datetime(
        updated.loc[provenance_mask, f"{output_column}__obs_time"],
↪ errors="coerce"
    )
    updated.loc[provenance_mask, "anthro_hours_offset"] = pd.to_numeric(
        updated.loc[provenance_mask, f"{output_column}__hours_offset"],
↪ errors="coerce"
    )
    updated.loc[provenance_mask, "anthro_timing_basis"] =
↪ "nearest_anytime"
    updated.loc[provenance_mask, "anthro_timing_uncertain"] = True

    any_fill_mask = any_fill_mask | fill_mask
    diagnostics["filled_counts"][output_column] = int(fill_mask.sum())

    drop_columns = [
        f"{output_column}__candidate",
        f"{output_column}__obs_time",
        f"{output_column}__hours_offset",
        f"{output_column}__source",
    ]

```

```

        updated = updated.drop(columns=drop_columns)

    diagnostics["rows_with_any_charted_fill"] = int(any_fill_mask.sum())
    diagnostics["fallback_source_counts"] = {
        str(key): int(value)
        for key, value in updated.loc[any_fill_mask, "anthro_source"]
        .fillna("missing")
        .astype(str)
        .value_counts(dropna=False)
        .items()
    }
    return updated, diagnostics

def build_anthro_coverage_audit(ed_df: pd.DataFrame) -> dict[str, Any]:
    """Summarize anthropometric coverage and provenance rates."""
    total_rows = max(int(len(ed_df)), 1)
    field_counts = {
        column: int(pd.to_numeric(ed_df[column],
        ↪ errors="coerce").notna().sum())
        for column in OMR_OUTPUT_COLUMNS
        if column in ed_df.columns
    }
    field_rates = {column: float(count / total_rows) for column, count in
    ↪ field_counts.items()}

    source_counts: dict[str, int] = {}
    source_rates: dict[str, float] = {}
    if "anthro_source" in ed_df.columns:
        source_counts = {
            str(key): int(value)
            for key, value in ed_df["anthro_source"]
            .fillna("missing")
            .astype(str)
            .value_counts(dropna=False)
            .items()
        }
        source_rates = {key: float(value / total_rows) for key, value in
        ↪ source_counts.items()}

    timing_basis_counts: dict[str, int] = {}
    timing_basis_rates: dict[str, float] = {}
    if "anthro_timing_basis" in ed_df.columns:

```

```

        timing_basis_counts = {
            str(key): int(value)
            for key, value in ed_df["anthro_timing_basis"]
                .fillna("missing")
                .astype(str)
                .value_counts(dropna=False)
                .items()
        }
        timing_basis_rates = {
            key: float(value / total_rows) for key, value in
↪ timing_basis_counts.items()
        }

    return {
        "row_count": int(len(ed_df)),
        "field_nonnull_counts": field_counts,
        "field_nonnull_rates": field_rates,
        "source_counts": source_counts,
        "source_rates": source_rates,
        "timing_basis_counts": timing_basis_counts,
        "timing_basis_rates": timing_basis_rates,
    }

```

```

def build_first_other_pco2_audit(ed_df: pd.DataFrame) -> pd.DataFrame:
    """Build route-stratified audit summary for first_other_pco2 values.

```

```

    This helper is intentionally tolerant for notebook QA execution: if the
    required fields are not present, it returns a sentinel audit row instead
↪ of
    raising.
    """
    columns = [
        "source",
        "count_nonnull",
        "mean",
        "median",
        "q25",
        "q75",
        "p95",
        "max",
        "pct_ge_80",
        "pct_ge_100",
    ]

```

```

        "pct_ge_150",
        "pct_eq_160",
        "top_values",
        "status",
        "missing_columns",
    ]
    required = {"first_other_pco2", "first_other_src"}
    missing = sorted(required.difference(ed_df.columns))
    if missing:
        return pd.DataFrame(
            [
                {
                    "source": "UNAVAILABLE",
                    "count_nonnull": 0,
                    "mean": None,
                    "median": None,
                    "q25": None,
                    "q75": None,
                    "p95": None,
                    "max": None,
                    "pct_ge_80": 0.0,
                    "pct_ge_100": 0.0,
                    "pct_ge_150": 0.0,
                    "pct_eq_160": 0.0,
                    "top_values": {},
                    "status": "missing_columns",
                    "missing_columns": ",".join(missing),
                }
            ],
            columns=columns,
        )

    frame = ed_df[["first_other_src", "first_other_pco2"]].copy()
    frame["first_other_src"] = (
        ↪ frame["first_other_src"].astype(str).str.strip().str.upper().replace({"":
        ↪ "UNKNOWN", "NAN": "UNKNOWN"})
    )
    frame["first_other_pco2"] = pd.to_numeric(frame["first_other_pco2"],
    ↪ errors="coerce")
    frame = frame.loc[frame["first_other_pco2"].notna()].copy()

    if frame.empty:

```

```

return pd.DataFrame(
    [
        {
            "source": "UNAVAILABLE",
            "count_nonnull": 0,
            "mean": None,
            "median": None,
            "q25": None,
            "q75": None,
            "p95": None,
            "max": None,
            "pct_ge_80": 0.0,
            "pct_ge_100": 0.0,
            "pct_ge_150": 0.0,
            "pct_eq_160": 0.0,
            "top_values": {},
            "status": "no_nonnull_values",
            "missing_columns": "",
        }
    ],
    columns=columns,
)

rows: list[dict[str, Any]] = []
for source_name, group in frame.groupby("first_other_src"):
    values = group["first_other_pco2"]
    rows.append(
        {
            "source": source_name,
            "count_nonnull": int(values.shape[0]),
            "mean": float(values.mean()),
            "median": float(values.median()),
            "q25": float(values.quantile(0.25)),
            "q75": float(values.quantile(0.75)),
            "p95": float(values.quantile(0.95)),
            "max": float(values.max()),
            "pct_ge_80": float((values >= 80).mean()),
            "pct_ge_100": float((values >= 100).mean()),
            "pct_ge_150": float((values >= 150).mean()),
            "pct_eq_160": float((values == 160).mean()),
            "top_values": {
                str(key): int(value)
                for key, value in values.value_counts().head(10).items()
            }
        }
    )

```

```

        },
        "status": "ok",
        "missing_columns": "",
    }
)
return pd.DataFrame(rows,
    ↪ columns=columns).sort_values("source").reset_index(drop=True)

def normalize_temperature_to_f(temp: pd.Series) -> pd.DataFrame:
    """Normalize mixed-unit temperature values to cleaned Fahrenheit output.

    Rules:
    - values in ``[20, 50]`` are treated as Celsius-like and converted to
    ↪ Fahrenheit,
    - all other numeric values are treated as Fahrenheit already,
    - cleaned output is valid only for Fahrenheit ``[50, 120]``.
    """
    numeric = pd.to_numeric(temp, errors="coerce")
    celsius_like = numeric.between(20.0, 50.0, inclusive="both")
    temp_f = numeric.where(~celsius_like, numeric * 9.0 / 5.0 + 32.0)
    out_of_range = ~temp_f.between(50.0, 120.0, inclusive="both")
    temp_f_clean = temp_f.where(~out_of_range)
    return pd.DataFrame(
        {
            "temp_f_clean": temp_f_clean,
            "temp_was_celsius_like": celsius_like.astype("boolean"),
            "temp_out_of_range": out_of_range.astype("boolean"),
        }
    )

def clean_pain_score(pain: pd.Series) -> pd.DataFrame:
    """Clean pain score with explicit sentinel handling."""
    numeric = pd.to_numeric(pain, errors="coerce")
    sentinel = numeric.eq(13)
    out_of_range = numeric.notna() & (~sentinel) & ((numeric < 0) | (numeric
    ↪ > 10))
    clean = numeric.where(~(sentinel | out_of_range))
    return pd.DataFrame(
        {
            "pain_clean": clean,
            "pain_is_sentinel_13": sentinel.astype("boolean"),
        }
    )

```



```

        "pain_out_of_range": out_of_range.astype("boolean"),
    }
)

def clean_bp(sbp: pd.Series, dbp: pd.Series) -> pd.DataFrame:
    """Range-clean SBP/DBP values with deterministic bounds."""
    sbp_num = pd.to_numeric(sbp, errors="coerce")
    dbp_num = pd.to_numeric(dbp, errors="coerce")
    sbp_out_of_range = sbp_num.notna() & ((sbp_num < 20) | (sbp_num > 300))
    dbp_out_of_range = dbp_num.notna() & ((dbp_num < 10) | (dbp_num > 200))
    return pd.DataFrame(
        {
            "sbp_clean": sbp_num.where(~sbp_out_of_range),
            "dbp_clean": dbp_num.where(~dbp_out_of_range),
            "sbp_out_of_range": sbp_out_of_range.astype("boolean"),
            "dbp_out_of_range": dbp_out_of_range.astype("boolean"),
        }
    )

def clean_o2sat(o2sat: pd.Series) -> pd.DataFrame:
    """Clean oxygen saturation and keep explicit >100 and zero flags."""
    numeric = pd.to_numeric(o2sat, errors="coerce")
    gt_100 = numeric > 100
    out_of_range = numeric.notna() & (numeric < 0)
    clean = numeric.where(~(gt_100 | out_of_range))
    return pd.DataFrame(
        {
            "o2sat_clean": clean,
            "o2sat_gt_100": gt_100.astype("boolean"),
            "o2sat_out_of_range": out_of_range.astype("boolean"),
            "o2sat_zero": numeric.eq(0).astype("boolean"),
        }
    )

def _series_distribution_stats(values: pd.Series) -> dict[str, Any]:
    """Return compact distribution metrics for audit CSV artifacts."""
    numeric = pd.to_numeric(values, errors="coerce")
    total_rows = int(len(numeric))
    non_missing = int(numeric.notna().sum())
    if non_missing == 0:

```

```

    return {
        "n_non_missing": 0,
        "missing_pct": 100.0 if total_rows else 0.0,
        "min": np.nan,
        "p1": np.nan,
        "median": np.nan,
        "mean": np.nan,
        "p99": np.nan,
        "max": np.nan,
    }
}

return {
    "n_non_missing": non_missing,
    "missing_pct": round(float((1.0 - non_missing / max(total_rows, 1)) *
↪ 100.0), 2),
    "min": float(numeric.min()),
    "p1": float(numeric.quantile(0.01)),
    "median": float(numeric.median()),
    "mean": float(numeric.mean()),
    "p99": float(numeric.quantile(0.99)),
    "max": float(numeric.max()),
}

def add_vitals_model_fields(
    ed_df: pd.DataFrame,
    *,
    ranges: Mapping[str, tuple[float, float]] | None = None,
) -> tuple[pd.DataFrame, pd.DataFrame]:
    """Create cleaned ED vitals fields and backward-compatible ``*_model``
↪ aliases."""
    updated = ed_df.copy()
    resolved_ranges = dict(ranges or DEFAULT_VITALS_MODEL_RANGES)
    audit_rows: list[dict[str, Any]] = []

    def _append_audit_row(
        *,
        raw_column: str,
        model_column: str,
        outlier_flag_column: str,
        lower_bound: float | None,
        upper_bound: float | None,
        numeric: pd.Series,
        outlier_flag: pd.Series,
    ) -> None:

```

```

) -> None:
    outlier_flag_numeric = outlier_flag.fillna(False).astype(bool)
    nonnull_n = int(numeric.notna().sum())
    outlier_n = int(outlier_flag_numeric.sum())
    examples = (
        numeric.loc[outlier_flag_numeric]
        .round(4)
        .value_counts()
        .head(5)
        .index.astype(str)
        .tolist()
    )
    audit_rows.append(
        {
            "domain": "vitals",
            "raw_column": raw_column,
            "model_column": model_column,
            "outlier_flag_column": outlier_flag_column,
            "lower_bound": lower_bound,
            "upper_bound": upper_bound,
            "nonnull_n": nonnull_n,
            "out_of_range_n": outlier_n,
            "out_of_range_pct": float(outlier_n / nonnull_n) if nonnull_n
            ↪ else 0.0,
            "example_outlier_values": "; ".join(examples),
        }
    )

for raw_column, (lower_bound, upper_bound) in resolved_ranges.items():
    if raw_column not in updated.columns:
        continue
    numeric = pd.to_numeric(updated[raw_column], errors="coerce")
    out_of_range = numeric.notna() & ((numeric < lower_bound) | (numeric
    ↪ > upper_bound))
    model_column = f"{raw_column}_model"
    outlier_flag_column = f"{raw_column}_outlier_flag"
    updated[model_column] = numeric.where(~out_of_range)
    updated[outlier_flag_column] = out_of_range.astype("boolean")
    _append_audit_row(
        raw_column=raw_column,
        model_column=model_column,
        outlier_flag_column=outlier_flag_column,
        lower_bound=float(lower_bound),

```

```

        upper_bound=float(upper_bound),
        numeric=numeric,
        outlier_flag=out_of_range,
    )

    for prefix in ("ed_triage", "ed_first"):
        temp_raw_col = f"{prefix}_temp"
        pain_raw_col = f"{prefix}_pain"
        sbp_raw_col = f"{prefix}_sbp"
        dbp_raw_col = f"{prefix}_dbp"
        o2sat_raw_col = f"{prefix}_o2sat"

        if temp_raw_col in updated.columns:
            temp_clean = normalize_temperature_to_f(updated[temp_raw_col])
            temp_clean_col = f"{prefix}_temp_f_clean"
            temp_celsius_flag_col = f"{prefix}_temp_was_celsius_like"
            temp_range_flag_col = f"{prefix}_temp_out_of_range"
            updated[temp_clean_col] = temp_clean["temp_f_clean"]
            updated[temp_celsius_flag_col] =
↪ temp_clean["temp_was_celsius_like"]
            updated[temp_range_flag_col] = temp_clean["temp_out_of_range"]
            updated[f"{prefix}_temp_f_model"] = updated[temp_clean_col]
            updated[f"{prefix}_temp_c_model"] =
↪ (updated[f"{prefix}_temp_f_model"] - 32.0) * (5.0 / 9.0)
            updated[f"{prefix}_temp_model"] =
↪ updated[f"{prefix}_temp_f_model"]
            updated[f"{prefix}_temp_outlier_flag"] =
↪ updated[temp_range_flag_col].astype("boolean")
            _append_audit_row(
                raw_column=temp_raw_col,
                model_column=f"{prefix}_temp_model",
                outlier_flag_column=f"{prefix}_temp_outlier_flag",
                lower_bound=50.0,
                upper_bound=120.0,
                numeric=pd.to_numeric(updated[temp_raw_col],
↪ errors="coerce"),
                outlier_flag=updated[temp_range_flag_col],
            )

        if pain_raw_col in updated.columns:
            pain_clean = clean_pain_score(updated[pain_raw_col])
            pain_clean_col = f"{prefix}_pain_clean"
            pain_sentinel_col = f"{prefix}_pain_is_sentinel_13"

```

```

    pain_range_col = f"{prefix}_pain_out_of_range"
    updated[pain_clean_col] = pain_clean["pain_clean"]
    updated[pain_sentinel_col] = pain_clean["pain_is_sentinel_13"]
    updated[pain_range_col] = pain_clean["pain_out_of_range"]
    updated[f"{prefix}_pain_model"] = updated[pain_clean_col]
    pain_outlier = updated[pain_sentinel_col].fillna(False) |
↪ updated[pain_range_col].fillna(False)
    updated[f"{prefix}_pain_outlier_flag"] =
↪ pain_outlier.astype("boolean")
    _append_audit_row(
        raw_column=pain_raw_col,
        model_column=f"{prefix}_pain_model",
        outlier_flag_column=f"{prefix}_pain_outlier_flag",
        lower_bound=0.0,
        upper_bound=10.0,
        numeric=pd.to_numeric(updated[pain_raw_col],
↪ errors="coerce"),
        outlier_flag=pain_outlier,
    )

if sbp_raw_col in updated.columns and dbp_raw_col in updated.columns:
    bp_clean = clean_bp(updated[sbp_raw_col], updated[dbp_raw_col])
    sbp_clean_col = f"{prefix}_sbp_clean"
    dbp_clean_col = f"{prefix}_dbp_clean"
    sbp_range_col = f"{prefix}_sbp_out_of_range"
    dbp_range_col = f"{prefix}_dbp_out_of_range"
    updated[sbp_clean_col] = bp_clean["sbp_clean"]
    updated[dbp_clean_col] = bp_clean["dbp_clean"]
    updated[sbp_range_col] = bp_clean["sbp_out_of_range"]
    updated[dbp_range_col] = bp_clean["dbp_out_of_range"]
    updated[f"{prefix}_sbp_model"] = updated[sbp_clean_col]
    updated[f"{prefix}_dbp_model"] = updated[dbp_clean_col]
    updated[f"{prefix}_sbp_outlier_flag"] =
↪ updated[sbp_range_col].astype("boolean")
    updated[f"{prefix}_dbp_outlier_flag"] =
↪ updated[dbp_range_col].astype("boolean")
    _append_audit_row(
        raw_column=sbp_raw_col,
        model_column=f"{prefix}_sbp_model",
        outlier_flag_column=f"{prefix}_sbp_outlier_flag",
        lower_bound=20.0,
        upper_bound=300.0,
        numeric=pd.to_numeric(updated[sbp_raw_col], errors="coerce"),

```

```

        outlier_flag=updated[dbp_range_col],
    )
    _append_audit_row(
        raw_column=dbp_raw_col,
        model_column=f"{prefix}_dbp_model",
        outlier_flag_column=f"{prefix}_dbp_outlier_flag",
        lower_bound=10.0,
        upper_bound=200.0,
        numeric=pd.to_numeric(updated[dbp_raw_col], errors="coerce"),
        outlier_flag=updated[dbp_range_col],
    )

if o2sat_raw_col in updated.columns:
    o2sat_clean = clean_o2sat(updated[o2sat_raw_col])
    o2sat_clean_col = f"{prefix}_o2sat_clean"
    o2sat_gt_100_col = f"{prefix}_o2sat_gt_100"
    o2sat_range_col = f"{prefix}_o2sat_out_of_range"
    o2sat_zero_col = f"{prefix}_o2sat_zero"
    updated[o2sat_clean_col] = o2sat_clean["o2sat_clean"]
    updated[o2sat_gt_100_col] = o2sat_clean["o2sat_gt_100"]
    updated[o2sat_range_col] = o2sat_clean["o2sat_out_of_range"]
    updated[o2sat_zero_col] = o2sat_clean["o2sat_zero"]
    updated[f"{prefix}_o2sat_model"] = updated[o2sat_clean_col]
    o2sat_outlier = updated[o2sat_gt_100_col].fillna(False) |
↪ updated[o2sat_range_col].fillna(False)
    updated[f"{prefix}_o2sat_outlier_flag"] =
↪ o2sat_outlier.astype("boolean")
    _append_audit_row(
        raw_column=o2sat_raw_col,
        model_column=f"{prefix}_o2sat_model",
        outlier_flag_column=f"{prefix}_o2sat_outlier_flag",
        lower_bound=0.0,
        upper_bound=100.0,
        numeric=pd.to_numeric(updated[o2sat_raw_col],
↪ errors="coerce"),
        outlier_flag=o2sat_outlier,
    )

audit =
↪ pd.DataFrame(audit_rows).sort_values("raw_column").reset_index(drop=True)
return updated, audit

```

```

def build_ed_vitals_audit_artifacts(
    ed_df: pd.DataFrame,
) -> tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """Build ED-vitals distribution, extreme-example, and raw-to-clean delta
    ↪ artifacts."""
    specs = [
        {
            "vital_name": "triage_temp",
            "raw_col": "ed_triage_temp",
            "clean_col": "ed_triage_temp_f_clean",
            "flags": ["ed_triage_temp_was_celsius_like",
                ↪ "ed_triage_temp_out_of_range"],
            "extra_cols": [],
        },
        {
            "vital_name": "first_temp",
            "raw_col": "ed_first_temp",
            "clean_col": "ed_first_temp_f_clean",
            "flags": ["ed_first_temp_was_celsius_like",
                ↪ "ed_first_temp_out_of_range"],
            "extra_cols": ["ed_first_vitals_time"],
        },
        {
            "vital_name": "triage_pain",
            "raw_col": "ed_triage_pain",
            "clean_col": "ed_triage_pain_clean",
            "flags": ["ed_triage_pain_is_sentinel_13",
                ↪ "ed_triage_pain_out_of_range"],
            "extra_cols": [],
        },
        {
            "vital_name": "first_pain",
            "raw_col": "ed_first_pain",
            "clean_col": "ed_first_pain_clean",
            "flags": ["ed_first_pain_is_sentinel_13",
                ↪ "ed_first_pain_out_of_range"],
            "extra_cols": ["ed_first_vitals_time"],
        },
        {
            "vital_name": "triage_sbp",
            "raw_col": "ed_triage_sbp",
            "clean_col": "ed_triage_sbp_clean",
            "flags": ["ed_triage_sbp_out_of_range"],
        },
    ]

```

```

        "extra_cols": [],
    },
    {
        "vital_name": "triage_dbp",
        "raw_col": "ed_triage_dbp",
        "clean_col": "ed_triage_dbp_clean",
        "flags": ["ed_triage_dbp_out_of_range"],
        "extra_cols": [],
    },
    {
        "vital_name": "first_sbp",
        "raw_col": "ed_first_sbp",
        "clean_col": "ed_first_sbp_clean",
        "flags": ["ed_first_sbp_out_of_range"],
        "extra_cols": ["ed_first_vitals_time"],
    },
    {
        "vital_name": "first_dbp",
        "raw_col": "ed_first_dbp",
        "clean_col": "ed_first_dbp_clean",
        "flags": ["ed_first_dbp_out_of_range"],
        "extra_cols": ["ed_first_vitals_time"],
    },
    {
        "vital_name": "triage_o2sat",
        "raw_col": "ed_triage_o2sat",
        "clean_col": "ed_triage_o2sat_clean",
        "flags": ["ed_triage_o2sat_gt_100",
            ↪ "ed_triage_o2sat_out_of_range", "ed_triage_o2sat_zero"],
        "extra_cols": [],
    },
    {
        "vital_name": "first_o2sat",
        "raw_col": "ed_first_o2sat",
        "clean_col": "ed_first_o2sat_clean",
        "flags": ["ed_first_o2sat_gt_100", "ed_first_o2sat_out_of_range",
            ↪ "ed_first_o2sat_zero"],
        "extra_cols": ["ed_first_vitals_time"],
    },
]

```

```

summary_rows: list[dict[str, Any]] = []
delta_rows: list[dict[str, Any]] = []

```



```

extreme_frames: list[pd.DataFrame] = []

for spec in specs:
    raw_col = spec["raw_col"]
    clean_col = spec["clean_col"]
    if raw_col not in ed_df.columns or clean_col not in ed_df.columns:
        continue

    raw = pd.to_numeric(ed_df[raw_col], errors="coerce")
    clean = pd.to_numeric(ed_df[clean_col], errors="coerce")
    raw_stats = _series_distribution_stats(raw)
    clean_stats = _series_distribution_stats(clean)
    row: dict[str, Any] = {
        "vital_name": spec["vital_name"],
        "raw_column": raw_col,
        "clean_column": clean_col,
        "raw_n_non_missing": raw_stats["n_non_missing"],
        "raw_missing_pct": raw_stats["missing_pct"],
        "raw_min": raw_stats["min"],
        "raw_p1": raw_stats["p1"],
        "raw_median": raw_stats["median"],
        "raw_mean": raw_stats["mean"],
        "raw_p99": raw_stats["p99"],
        "raw_max": raw_stats["max"],
        "clean_n_non_missing": clean_stats["n_non_missing"],
        "clean_missing_pct": clean_stats["missing_pct"],
        "clean_min": clean_stats["min"],
        "clean_p1": clean_stats["p1"],
        "clean_median": clean_stats["median"],
        "clean_mean": clean_stats["mean"],
        "clean_p99": clean_stats["p99"],
        "clean_max": clean_stats["max"],
    }

    delta_rows.append(
        {
            "vital_name": spec["vital_name"],
            "raw_column": raw_col,
            "clean_column": clean_col,
            "raw_n": int(raw.notna().sum()),
            "clean_n": int(clean.notna().sum()),
            "raw_removed_n": int(raw.notna().sum() -
                ↪ clean.notna().sum()),

```

```

        "raw_removed_pct": (
            float((raw.notna().sum() - clean.notna().sum()) /
                ↪ max(int(raw.notna().sum()), 1))
            if int(raw.notna().sum()) > 0
            else 0.0
        ),
        "converted_celsius_like_n": int(
            ed_df.get(spec["flags"][0], pd.Series(False,
↪ index=ed_df.index))
                .fillna(False)
                .astype(bool)
                .sum()
        )
        if "temp_was_celsius_like" in spec["flags"][0]
        else 0,
    }
)

available_flags = [flag for flag in spec["flags"] if flag in
↪ ed_df.columns]
for flag_col in available_flags:
    flag_series = ed_df[flag_col].fillna(False).astype(bool)
    row[f"{flag_col}_count"] = int(flag_series.sum())
    row[f"{flag_col}_pct"] = float(flag_series.mean()) if
↪ len(flag_series) else 0.0
    if not flag_series.any():
        continue
    include_cols = ["hadm_id", "ed_stay_id", raw_col, clean_col,
↪ flag_col]
    include_cols += [col for col in spec["extra_cols"] if col in
↪ ed_df.columns]
    include_cols = [col for col in include_cols if col in
↪ ed_df.columns]
    flagged = ed_df.loc[flag_series, include_cols].copy()
    flagged = flagged.rename(columns={raw_col: "raw_value",
↪ clean_col: "clean_value", flag_col: "flag_value"})
    low = flagged.sort_values("raw_value", ascending=True,
↪ na_position="last").head(5).copy()
    low["tail"] = "low"
    high = flagged.sort_values("raw_value", ascending=False,
↪ na_position="last").head(5).copy()
    high["tail"] = "high"

```

```

        extremes = pd.concat([low, high],
↪ ignore_index=True).drop_duplicates()
        extremes.insert(0, "vital_name", spec["vital_name"])
        extremes.insert(1, "raw_column", raw_col)
        extremes.insert(2, "clean_column", clean_col)
        extremes.insert(3, "flag_type", flag_col)
        extreme_frames.append(extremes)

    summary_rows.append(row)

distribution = pd.DataFrame(summary_rows)
delta = pd.DataFrame(delta_rows)
if extreme_frames:
    extremes = pd.concat(extreme_frames, ignore_index=True, sort=False)
else:
    extremes = pd.DataFrame(
        columns=[
            "vital_name",
            "raw_column",
            "clean_column",
            "flag_type",
            "hadm_id",
            "ed_stay_id",
            "raw_value",
            "clean_value",
            "flag_value",
            "tail",
        ]
    )
return distribution, extremes, delta

def evaluate_uom_expectations(ed_df: pd.DataFrame) -> dict[str, Any]:
    """Evaluate expected-null and value/uom consistency rules."""
    structural_nulls: dict[str, dict[str, Any]] = {}
    for field_name in EXPECTED_STRUCTURAL_NULL_FIELDS:
        if field_name not in ed_df.columns:
            continue
        structural_nulls[field_name] = {
            "present": True,
            "missing_n": int(ed_df[field_name].isna().sum()),
            "missing_pct": float(ed_df[field_name].isna().mean()),
            "all_null": bool(ed_df[field_name].isna().all()),
        }

```

```

    }

    paco2_checks: dict[str, dict[str, Any]] = {}
    for value_column, uom_column in PACO2_VALUE_UOM_PAIRS:
        if value_column not in ed_df.columns or uom_column not in
            ↪ ed_df.columns:
            paco2_checks[uom_column] = {
                "present": False,
                "reason": "missing_value_or_uom_column",
            }
            continue

        value_present = ed_df[value_column].notna()
        uom_lower = ed_df[uom_column].astype(str).str.strip().str.lower()
        missing_uom_with_value = int((value_present &
            ↪ ed_df[uom_column].isna()).sum())
        non_mmhg_uom_with_value = int(
            (value_present & ed_df[uom_column].notna() &
            ↪ uom_lower.ne("mmhg")).sum()
        )

        paco2_checks[uom_column] = {
            "present": True,
            "paired_value_column": value_column,
            "value_rows": int(value_present.sum()),
            "missing_uom_when_value_present": missing_uom_with_value,
            "non_mmhg_uom_when_value_present": non_mmhg_uom_with_value,
            "passes": bool(
                missing_uom_with_value == 0 and non_mmhg_uom_with_value == 0
            ),
        }

    return {
        "expected_structural_null_fields":
            ↪ list(EXPECTED_STRUCTURAL_NULL_FIELDS),
        "structural_null_checks": structural_nulls,
        "paco2_uom_checks": paco2_checks,
    }

def classify_missingness_expectations(
    ed_df: pd.DataFrame,
    target_fields: list[str],

```

```

*,
    expected_sparse_fields: set[str] | None = None,
) -> pd.DataFrame:
    """Classify field-level missingness into expected and unexpected
    ↪ categories."""
    rows: list[dict[str, Any]] = []
    total_rows = max(int(len(ed_df)), 1)
    expected_structural = set(EXPECTED_STRUCTURAL_NULL_FIELDS)
    expected_sparse = set(expected_sparse_fields or set())

    for field_name in target_fields:
        if field_name not in ed_df.columns:
            rows.append(
                {
                    "field": field_name,
                    "missing_n": int(len(ed_df)),
                    "missing_pct": 1.0,
                    "expectation": "missing_column",
                }
            )
            continue

        missing_n = int(ed_df[field_name].isna().sum())
        missing_pct = float(missing_n / total_rows)

        if field_name in expected_structural:
            expectation = "expected_structural_null"
        elif field_name in expected_sparse and missing_pct >= 1.0:
            expectation = "expected_sparse"
        elif missing_pct >= 1.0:
            expectation = "unexpected_full_null"
        elif missing_pct > 0.0:
            expectation = "conditional_sparse"
        else:
            expectation = "complete"

        rows.append(
            {
                "field": field_name,
                "missing_n": missing_n,
                "missing_pct": missing_pct,
                "expectation": expectation,
            }
        )

```

```

    )

    return pd.DataFrame(rows)

def render_latex_longtable(
    table_df: pd.DataFrame,
    *,
    caption: str,
    label: str,
    landscape: bool = False,
    index: bool = True,
) -> str:
    latex_text = table_df.to_latex(
        index=index,
        escape=False,
        longtable=True,
        caption=caption,
        label=label,
    )
    if landscape:
        latex_text = "\\begin{landscape}\\n" + latex_text +
↪ "\\n\\end{landscape}\\n"
    return latex_text

from hypercap_cc_nlp.pipeline_audit import collect_run_manifest
from hypercap_cc_nlp.pipeline_contracts import validate_cohort_contract,
↪ write_contract_report

# ---- Backend selection (we use BigQuery)
BACKEND = os.getenv("MIMIC_BACKEND", "bigquery").strip().lower()
assert BACKEND == "bigquery", "This notebook is BigQuery-specific."

WORK_PROJECT = os.getenv("WORK_PROJECT", "").strip() # your billing project
PHYS = os.getenv("BQ_PHYSIONET_PROJECT", "physionet-data").strip() # hosting
↪ project (read-only)

# Dataset preferences: resolved to accessible datasets in the next setup
↪ cell.
HOSP = os.getenv("BQ_DATASET_HOSP", "mimiciv_3_1_hosp").strip()
ICU = os.getenv("BQ_DATASET_ICU", "mimiciv_3_1_icu").strip()
ED = os.getenv("BQ_DATASET_ED", "").strip()

```

```

# BigQuery client
client = bigquery.Client(project=WORK_PROJECT)

print("Project:", WORK_PROJECT)
print("PhysioNet host:", PHYS)
print("HOSP (pref):", HOSP, "ICU (pref):", ICU, "ED (pref):", ED)
print("WORK_DIR:", WORK_DIR)

Project: mimic-hypercapnia
PhysioNet host: physionet-data
HOSP (pref): mimicy_3_1_hosp ICU (pref): mimicy_3_1_icu ED (pref): mimicy_ed
WORK_DIR: /Users/bloche/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Proj
CC-NLP

# Purpose: Define reusable BigQuery helpers and resolve dataset names across
↳ known naming variants.

from datetime import datetime, timezone

# Fail fast on long-running queries instead of hanging indefinitely in
↳ nbconvert.
BQ_QUERY_TIMEOUT_SECS = int(os.getenv("BQ_QUERY_TIMEOUT_SECS", "1800"))

# --- Helper: run SQL with optional named parameters
def run_sql_bq(sql: str, params: dict | None = None) -> pd.DataFrame:
    job_config = bigquery.QueryJobConfig()
    job_config.labels = {"pipeline": "hypercapcc", "notebook": "cohort"}
    if params:
        bq_params = []
        for k, v in params.items():
            if isinstance(v, (list, tuple, np.ndarray, pd.Series)):
                # BigQuery ARRAY<INT64> if all ints; else ARRAY<STRING>
                v_list = list(v)
                if all(isinstance(x, (int, np.integer)) for x in v_list):
                    bq_params.append(bigquery.ArrayQueryParameter(k, "INT64",
↳ list(map(int, v_list))))
                else:
                    bq_params.append(bigquery.ArrayQueryParameter(k,
↳ "STRING", list(map(str, v_list))))
            else:
                # scalar
                if isinstance(v, (int, np.integer)):

```

```

        bq_params.append(bigquery.ScalarQueryParameter(k,
↪ "INT64", int(v)))
        elif isinstance(v, float):
            bq_params.append(bigquery.ScalarQueryParameter(k,
↪ "FLOAT64", float(v)))
        else:
            bq_params.append(bigquery.ScalarQueryParameter(k,
↪ "STRING", str(v)))
        job_config.query_parameters = bq_params

    job = client.query(sql, job_config=job_config)
    try:
        result = job.result(timeout=BQ_QUERY_TIMEOUT_SECS)
    except Exception as exc:
        try:
            job.cancel()
        except Exception:
            pass
        raise RuntimeError(
            f"BigQuery query failed or timed out after
↪ {BQ_QUERY_TIMEOUT_SECS}s (job_id={job.job_id})."
        ) from exc

    try:
        return result.to_dataframe(create_bqstorage_client=True)
    except TypeError:
        return result.to_dataframe()

# --- Helper: test if a fully-qualified table exists and is accessible
def table_exists(fqtn: str) -> bool:
    try:
        _ = run_sql_bq(f"SELECT 1 FROM `{fqtn}` LIMIT 1")
        return True
    except Exception:
        return False

def resolve_dataset(preferred: str, candidates: list[str], probe_table: str,
↪ label: str) -> tuple[str, str]:
    # Keep preferred first, then try known aliases.
    ordered = []
    if preferred:
        ordered.append(preferred)
    for cand in candidates:

```



```

        if cand not in ordered:
            ordered.append(cand)

    for dataset in ordered:
        fqtn = f"{PHYS}.{dataset}.{probe_table}"
        if table_exists(fqtn):
            return dataset, fqtn

    raise RuntimeError(
        f"No accessible {label} dataset found for probe table
        ↳ '{probe_table}'. Tried: {ordered}"
    )

# Resolve HOSP/ICU/ED names so notebook runs across v3.1 naming variants.
HOSP, HOSP_PROBE = resolve_dataset(
    preferred=HOSP,
    candidates=["mimiciv_3_1_hosp", "mimiciv_v3_1_hosp", "mimiciv_hosp"],
    probe_table="admissions",
    label="HOSP",
)
ICU, ICU_PROBE = resolve_dataset(
    preferred=ICU,
    candidates=["mimiciv_3_1_icu", "mimiciv_v3_1_icu", "mimiciv_icu"],
    probe_table="icustays",
    label="ICU",
)
if not ED:
    ED = "mimiciv_ed"
ED, ED_PROBE = resolve_dataset(
    preferred=ED,
    candidates=["mimiciv_ed", "mimiciv_3_1_ed", "mimiciv_v3_1_ed"],
    probe_table="edstays",
    label="ED",
)

RUN_METADATA = {
    "run_utc": datetime.now(timezone.utc).isoformat(),
    "work_project": WORK_PROJECT,
    "physionet_project": PHYS,
    "datasets": {"hosp": HOSP, "icu": ICU, "ed": ED},
    "dataset_probes": {"hosp": HOSP_PROBE, "icu": ICU_PROBE, "ed": ED_PROBE},
    "notebook": "MIMICIV_hypercap_EXT_cohort.ipynb",
    "blood_gas_manifest_path": BLOOD_GAS_MANIFEST["path"],
}

```

```

    "blood_gas_manifest_version": BLOOD_GAS_MANIFEST["version"],
}

print("Resolved datasets:", RUN_METADATA["datasets"])
print("Dataset probes:", RUN_METADATA["dataset_probes"])

Resolved datasets: {'hosp': 'mimiciv_3_1_hosp', 'icu': 'mimiciv_3_1_icu', 'ed': 'mimiciv_ed'}
Dataset probes: {'hosp': 'physionet-data.mimiciv_3_1_hosp.admissions', 'icu': 'physionet-
data.mimiciv_3_1_icu.icustays', 'ed': 'physionet-data.mimiciv_ed.edstays'}

# Purpose: Create reusable data-quality and merge guardrail helpers to
↳ prevent silent join errors.

# --- Helper utilities for reproducibility and safe joins
def require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise KeyError(f"{name} missing columns: {missing}")

def assert_unique(df: pd.DataFrame, key: str, name: str) -> None:
    if df[key].duplicated().any():
        n = int(df[key].duplicated().sum())
        raise ValueError(f"{name} has {n} duplicate {key} values")

def safe_merge(left: pd.DataFrame, right: pd.DataFrame, on: list[str] | str,
↳ how: str, name: str) -> pd.DataFrame:
    # guard against accidental duplicate columns
    overlap = set(left.columns) & set(right.columns)
    if isinstance(on, str):
        on_cols = {on}
    else:
        on_cols = set(on)
    overlap = overlap - on_cols
    if overlap:
        raise ValueError(f"{name} merge would duplicate columns:
↳ {sorted(overlap)}")
    return left.merge(right, on=on, how=how)

def check_ranges(df: pd.DataFrame, ranges: dict[str, tuple[float, float]])
↳ -> pd.DataFrame:
    rows = []
    for col, (lo, hi) in ranges.items():

```

```

    if col not in df.columns:
        continue
    bad = df[col].notna() & ((df[col] < lo) | (df[col] > hi))
    rows.append({"col": col, "n_bad": int(bad.sum())})
return pd.DataFrame(rows)

```

## 2.3 Data Generation

### 2.3.1 1) ICD cohort flags (hypercapnic respiratory failure)

**Rationale:** Capture diagnosis-based hypercapnia from ED and hospital discharge codes to define a broad, clinically recognized cohort.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
 ↪ consistent across notebook stages.

```

# Target ICD codes (dotless, uppercase)
ICD10_CODES = ['J9602', 'J9612', 'J9622', 'J9692', 'E662']
ICD9_CODES = ['27803']

SQL["cohort_icd_sql"] = f"""
-- ICD-based cohort flags per admission
WITH target_codes AS (
  SELECT 'J9602' AS code, 10 AS ver UNION ALL
  SELECT 'J9612', 10 UNION ALL
  SELECT 'J9622', 10 UNION ALL
  SELECT 'J9692', 10 UNION ALL
  SELECT 'E662', 10 UNION ALL
  SELECT '27803', 9
),

-- Hospital ICDs restricted to target codes
hosp_dx AS (
  SELECT
    d.subject_id,
    d.hadm_id,
    UPPER(REPLACE(d.icd_code, '.', '')) AS code_norm,
    d.icd_version
  FROM `{PHYS}`.{HOSP}.diagnoses_icd` d
  JOIN target_codes t
    ON t.ver = d.icd_version AND t.code = UPPER(REPLACE(d.icd_code, '.', ''))
  WHERE d.hadm_id IS NOT NULL
),

```

```

-- Hospital flags per admission
hosp_flags AS (
  SELECT
    subject_id, hadm_id,
    MAX(IF(icd_version=10 AND code_norm='J9602',1,0)) AS ICD10_J9602,
    MAX(IF(icd_version=10 AND code_norm='J9612',1,0)) AS ICD10_J9612,
    MAX(IF(icd_version=10 AND code_norm='J9622',1,0)) AS ICD10_J9622,
    MAX(IF(icd_version=10 AND code_norm='J9692',1,0)) AS ICD10_J9692,
    MAX(IF(icd_version=10 AND code_norm='E662', 1,0)) AS ICD10_E662,
    MAX(IF(icd_version=9 AND code_norm='27803',1,0)) AS ICD9_27803
  FROM hosp_dx
  GROUP BY subject_id, hadm_id
),

-- ED ICDs restricted to target codes (map to hadm via edstays)
ed_dx AS (
  SELECT
    s.subject_id,
    s.hadm_id,
    s.stay_id,
    s.intime AS ed_intime,
    UPPER(REPLACE(d.icd_code, '.', '')) AS code_norm,
    d.icd_version
  FROM `{PHYS}`.{ED}.diagnosis` d
  JOIN `{PHYS}`.{ED}.edstays` s
    ON s.subject_id = d.subject_id AND s.stay_id = d.stay_id
  JOIN target_codes t
    ON t.ver = d.icd_version AND t.code = UPPER(REPLACE(d.icd_code, '.', ''))
  WHERE s.hadm_id IS NOT NULL
),

-- ED flags per ED stay (so we can both: OR flags across stays and also pick
↳ earliest stay_id)
ed_flags_by_stay AS (
  SELECT
    subject_id, hadm_id, stay_id, MIN(ed_intime) AS ed_intime,
    MAX(IF(icd_version=10 AND code_norm='J9602',1,0)) AS ICD10_J9602,
    MAX(IF(icd_version=10 AND code_norm='J9612',1,0)) AS ICD10_J9612,
    MAX(IF(icd_version=10 AND code_norm='J9622',1,0)) AS ICD10_J9622,
    MAX(IF(icd_version=10 AND code_norm='J9692',1,0)) AS ICD10_J9692,
    MAX(IF(icd_version=10 AND code_norm='E662', 1,0)) AS ICD10_E662,
    MAX(IF(icd_version=9 AND code_norm='27803',1,0)) AS ICD9_27803

```

```

FROM ed_dx
GROUP BY subject_id, hadm_id, stay_id
),

-- OR the ED flags across all ED stays mapped to the same hadm
ed_flags_or AS (
SELECT
    subject_id, hadm_id,
    MAX(ICD10_J9602) AS ICD10_J9602,
    MAX(ICD10_J9612) AS ICD10_J9612,
    MAX(ICD10_J9622) AS ICD10_J9622,
    MAX(ICD10_J9692) AS ICD10_J9692,
    MAX(ICD10_E662 ) AS ICD10_E662,
    MAX(ICD9_27803) AS ICD9_27803
FROM ed_flags_by_stay
GROUP BY subject_id, hadm_id
),

-- Earliest ED stay_id per hadm (NO UNNEST of aggregates; use [OFFSET(0)])
ed_earliest AS (
SELECT
    subject_id,
    hadm_id,
    (ARRAY_AGG(STRUCT(stay_id, ed_intime) ORDER BY ed_intime LIMIT
↪ 1))[OFFSET(0)].stay_id AS stay_id
FROM ed_flags_by_stay
GROUP BY subject_id, hadm_id
),

-- Bring flags and earliest stay_id together
ed_by_hadm AS (
SELECT
    f.subject_id,
    f.hadm_id,
    e.stay_id,
    f.ICD10_J9602,
    f.ICD10_J9612,
    f.ICD10_J9622,
    f.ICD10_J9692,
    f.ICD10_E662,
    f.ICD9_27803
FROM ed_flags_or f
LEFT JOIN ed_earliest e

```

```

        USING (subject_id, hadm_id)
    ),

-- Combine ED and hospital flags at the admission level
combined AS (
    SELECT
        COALESCE(h.subject_id, e.subject_id) AS subject_id,
        COALESCE(h.hadm_id, e.hadm_id) AS hadm_id,
        GREATEST(IFNULL(h.ICD10_J9602,0), IFNULL(e.ICD10_J9602,0)) AS
↪ ICD10_J9602,
        GREATEST(IFNULL(h.ICD10_J9612,0), IFNULL(e.ICD10_J9612,0)) AS
↪ ICD10_J9612,
        GREATEST(IFNULL(h.ICD10_J9622,0), IFNULL(e.ICD10_J9622,0)) AS
↪ ICD10_J9622,
        GREATEST(IFNULL(h.ICD10_J9692,0), IFNULL(e.ICD10_J9692,0)) AS
↪ ICD10_J9692,
        GREATEST(IFNULL(h.ICD10_E662 ,0), IFNULL(e.ICD10_E662 ,0)) AS ICD10_E662,
        GREATEST(IFNULL(h.ICD9_27803,0), IFNULL(e.ICD9_27803,0)) AS ICD9_27803,

↪ IF((IFNULL(h.ICD10_J9602,0)+IFNULL(h.ICD10_J9612,0)+IFNULL(h.ICD10_J9622,0)+IFNULL(h.ICD
↪ > 0, 1, 0) AS any_hypercap_icd_hosp,

↪ IF((IFNULL(e.ICD10_J9602,0)+IFNULL(e.ICD10_J9612,0)+IFNULL(e.ICD10_J9622,0)+IFNULL(e.ICD
↪ > 0, 1, 0) AS any_hypercap_icd_ed
    FROM hosp_flags h
    FULL OUTER JOIN ed_by_hadm e
        ON h.hadm_id = e.hadm_id
)

SELECT
    subject_id, hadm_id,
    ICD10_J9602, ICD10_J9612, ICD10_J9622, ICD10_J9692, ICD10_E662, ICD9_27803,
    IF((ICD10_J9602+ICD10_J9612+ICD10_J9622+ICD10_J9692+ICD10_E662+ICD9_27803)
↪ > 0, 1, 0) AS any_hypercap_icd,
    any_hypercap_icd_hosp,
    any_hypercap_icd_ed,
    CASE
        WHEN any_hypercap_icd_hosp=1 AND any_hypercap_icd_ed=1 THEN 'ED+HOSP'
        WHEN any_hypercap_icd_ed=1 THEN 'ED'
        WHEN any_hypercap_icd_hosp=1 THEN 'HOSP'
        ELSE 'NONE'
    END AS icd_source
FROM combined

```

```
"""
```

```
cohort_icd = run_sql_bq(sql("cohort_icd_sql"))
print("ICD cohort admissions:", len(cohort_icd))
cohort_icd.head(3)
```

ICD cohort admissions: 4237

	subject_id	hadm_id	ICD10_J9602	ICD10_J9612	ICD10_J9622	ICD10_J9692	ICD10_E662	I
0	10485425	21207827	0	0	1	0	1	0
1	16826447	21830147	0	1	0	0	1	0
2	11482582	22869017	0	0	0	0	0	1

### 2.3.2 2) Blood gas inclusion thresholds — ANY PaCO<sub>2</sub> meeting cutoffs (LAB + POC)

**Rationale:** Identify physiologic hypercapnia using ABG/VBG thresholds, independent of diagnostic coding.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
↪ consistent across notebook stages.

```
SQL["co2_thresholds_sql"] = f"""
WITH hosp_raw AS (
  SELECT
    le.hadm_id,
    le.charttime,
    le.specimen_id,
    COALESCE(
      CAST(le.valuenum AS FLOAT64),
      SAFE_CAST(REGEXP_EXTRACT(LOWER(le.value), r'(-?\d+(?:\.\d+)?)') AS
↪ FLOAT64)
    ) AS val,
    LOWER(REPLACE(COALESCE(le.valueuom, ''), ' ', '')) AS uom_nospace,
    LOWER(COALESCE(di.label, '')) AS lbl,
    LOWER(COALESCE(di.fluid, '')) AS fluid_text
FROM `{PHYS}`.{HOSP}.labevents` le
JOIN `{PHYS}`.{HOSP}.d_labitems` di
  ON di.itemid = le.itemid
WHERE le.itemid IN UNNEST({LAB_PCO2_SQL_ARRAY})
  AND (le.valuenum IS NOT NULL OR le.value IS NOT NULL)
  AND LOWER(COALESCE(di.fluid, '')) IN ({LAB_BLOOD_FLUID_SQL_LIST})
```

```

),
hosp_specimen AS (
  SELECT
    le.specimen_id,
    LOWER(COALESCE(le.value, '')) AS specimen_type_text
  FROM `{PHYS}`.{HOSP}.labevents` le
  WHERE le.itemid IN UNNEST({LAB_SPECIMEN_SQL_ARRAY})
    AND le.specimen_id IS NOT NULL
    AND le.value IS NOT NULL
  QUALIFY ROW_NUMBER() OVER (
    PARTITION BY le.specimen_id
    ORDER BY le.charttime DESC
  ) = 1
),
hosp_pco2_clean AS (
  SELECT
    r.hadm_id,
    CASE
      WHEN REGEXP_CONTAINS(COALESCE(s.specimen_type_text, ''),
↪ r'arter|\\bart\\b') THEN 'arterial'
      WHEN REGEXP_CONTAINS(COALESCE(s.specimen_type_text, ''),
↪ r'ven|central|mixed') THEN 'venous'
      ELSE 'unknown'
    END AS site,
    r.charttime,
    CASE WHEN r.uom_nospace = 'kpa' THEN r.val * 7.50062 ELSE r.val END AS
↪ pco2_mmhg,
    'LAB' AS source_branch
  FROM hosp_raw r
  LEFT JOIN hosp_specimen s USING (specimen_id)
  WHERE r.val IS NOT NULL
    AND (CASE WHEN r.uom_nospace = 'kpa' THEN r.val * 7.50062 ELSE r.val END)
↪ BETWEEN 5 AND 200
),
icu_pco2_itemids AS (
  SELECT itemid, LOWER(label) AS lbl
  FROM `{PHYS}`.{ICU}.d_items`
  WHERE itemid IN UNNEST({ICU_PC02_SQL_ARRAY})
),
icu_pco2_raw AS (
  SELECT
    ie.hadm_id,
    ce.stay_id,

```



```

ce.itemid,
ce.charttime,
di.lbl,
LOWER(REPLACE(COALESCE(ce.valueuom, ''), ' ', '')) AS uom_nospace,
CASE
    WHEN ce.itemid IN UNNEST({ICU_PCO2_ABG_SQL_ARRAY}) THEN 'arterial'
    WHEN ce.itemid IN UNNEST({ICU_PCO2_VBG_SQL_ARRAY}) THEN 'venous'
    ELSE 'unknown'
END AS site_from_itemid,
COALESCE(
    CAST(ce.valuenum AS FLOAT64),
    SAFE_CAST(REGEXP_EXTRACT(LOWER(ce.value), r'(-?\d+(?:\.\d+)?)') AS
↪ FLOAT64)
) AS val
FROM `{PHYS}`.{ICU}.chartevents` ce
JOIN icu_pco2_itemids di
    ON di.itemid = ce.itemid
JOIN `{PHYS}`.{ICU}.icustays` ie
    ON ie.stay_id = ce.stay_id
WHERE ce.valuenum IS NOT NULL OR ce.value IS NOT NULL
),
icu_specimen AS (
    SELECT
        ce.stay_id,
        ce.charttime,
        LOWER(COALESCE(ce.value, '')) AS specimen_type_text
    FROM `{PHYS}`.{ICU}.chartevents` ce
    WHERE ce.itemid IN UNNEST({ICU_SPECIMEN_SQL_ARRAY})
        AND ce.value IS NOT NULL
),
icu_pco2_ranked AS (
    SELECT
        r.*,
        s.specimen_type_text,
        ROW_NUMBER() OVER (
            PARTITION BY r.hadm_id, r.stay_id, r.charttime
            ORDER BY ABS(TIMESTAMP_DIFF(r.charttime, s.charttime, SECOND))
        ) AS specimen_rank
    FROM icu_pco2_raw r
    LEFT JOIN icu_specimen s
        ON s.stay_id = r.stay_id
        AND ABS(TIMESTAMP_DIFF(r.charttime, s.charttime, MINUTE)) <= 120
),

```

```

icu_pco2_clean AS (
    SELECT
        hadm_id,
        CASE
            WHEN REGEXP_CONTAINS(COALESCE(specimen_type_text, ''),
↪ r'arter|\\bart\\b') THEN 'arterial'
            WHEN REGEXP_CONTAINS(COALESCE(specimen_type_text, ''),
↪ r'ven|central|mixed') THEN 'venous'
            WHEN site_from_itemid IN ('arterial', 'venous') THEN site_from_itemid
            ELSE 'unknown'
        END AS site,
        charttime,
        CASE WHEN uom_nospace = 'kpa' THEN val * 7.50062 ELSE val END AS
↪ pco2_mmhg,
        'POC' AS source_branch
    FROM icu_pco2_ranked
    WHERE val IS NOT NULL
        AND (specimen_rank = 1 OR specimen_type_text IS NULL)
        AND (CASE WHEN uom_nospace = 'kpa' THEN val * 7.50062 ELSE val END)
↪ BETWEEN 5 AND 200
),
all_pco2 AS (
    SELECT * FROM hosp_pco2_clean
    UNION ALL
    SELECT * FROM icu_pco2_clean
),
thresh AS (
    SELECT
        hadm_id,
        MAX(IF(site = 'arterial' AND pco2_mmhg >= 45.0, 1, 0)) AS
↪ abg_hypercap_threshold,
        MAX(IF(site = 'venous' AND pco2_mmhg >= 50.0, 1, 0)) AS
↪ vbg_hypercap_threshold,
        MAX(IF(site = 'unknown' AND pco2_mmhg >= 50.0, 1, 0)) AS
↪ unknown_hypercap_threshold,
        MAX(IF(site = 'unknown' AND source_branch = 'POC' AND pco2_mmhg >= 50.0,
↪ 1, 0)) AS poc_unknown_hypercap_threshold
    FROM all_pco2
    GROUP BY hadm_id
)
SELECT * FROM thresh
"""

```

```
co2_thresh = run_sql_bq(sql("co2_thresholds_sql"))
print("Admissions meeting thresholds:", len(co2_thresh))
co2_thresh.head(3)
```

Admissions meeting thresholds: 83568

	hadm_id	abg_hypercap_threshold	vbg_hypercap_threshold	unknown_hypercap_threshold	poc_u
0	20079585	1	1	1	0
1	20156947	1	0	1	0
2	20305613	0	0	0	0

### 2.3.3 3) Cohort union (ICD thresholds) and hadm\_list for downstream queries

**Rationale:** Combine ICD and physiologic routes to maximize sensitivity, then define a stable hadm\_id list for downstream joins.

```
# Purpose: Combine ICD and gas-threshold ascertainment routes to produce the
↳ final hadm inclusion list.
```

```
# Outer-join because thresholds can identify hadm_id with no ICD codes and
↳ vice versa
```

```
cohort_any = cohort_icd.merge(co2_thresh, how="outer", on="hadm_id")
```

```
# Fill missing flags with 0 where appropriate
```

```
icd_cols = ["ICD10_J9602", "ICD10_J9612", "ICD10_J9622", "ICD10_J9692", ]
↳ "ICD10_E662", "ICD9_27803", "any_hypercap_icd", "any_hypercap_icd_hosp", ]
↳ "any_hypercap_icd_ed"]
```

```
for c in icd_cols:
    if c in cohort_any.columns:
        cohort_any[c] = cohort_any[c].fillna(0).astype(int)
```

```
for c in [
    "abg_hypercap_threshold",
    "vbg_hypercap_threshold",
    "unknown_hypercap_threshold",
    "poc_unknown_hypercap_threshold",
]:
```

```
    if c in cohort_any.columns:
        cohort_any[c] = cohort_any[c].fillna(0).astype(int)
```

```

if "unknown_hypercap_threshold" in cohort_any.columns and
↳ "other_hypercap_threshold" not in cohort_any.columns:
    cohort_any["other_hypercap_threshold"] =
↳ cohort_any["unknown_hypercap_threshold"].astype(int)
if "poc_unknown_hypercap_threshold" in cohort_any.columns and
↳ "poc_other_quarantined_hypercap_threshold" not in cohort_any.columns:
    cohort_any["poc_other_quarantined_hypercap_threshold"] =
↳ cohort_any["poc_unknown_hypercap_threshold"].astype(int)

# Final enrollment flag
cohort_any["pco2_threshold_any"] = (
    (cohort_any["abg_hypercap_threshold"] == 1)
    | (cohort_any["vbg_hypercap_threshold"] == 1)
    | (cohort_any["unknown_hypercap_threshold"] == 1)
).astype(int)
cohort_any["enrolled_any"] = ((cohort_any["any_hypercap_icd"]==1) |
↳ (cohort_any["pco2_threshold_any"]==1)).astype(int)

print("ICD-only admissions      :",
↳ int((cohort_any["any_hypercap_icd"]==1).sum()))
print("Threshold-only admissions :",
↳ int(((cohort_any["pco2_threshold_any"]==1) &
↳ (cohort_any["any_hypercap_icd"]==0)).sum()))
print("Both ICD and threshold    :",
↳ int(((cohort_any["pco2_threshold_any"]==1) &
↳ (cohort_any["any_hypercap_icd"]==1)).sum()))
print("Total enrolled (union)    :",
↳ int((cohort_any["enrolled_any"]==1).sum()))
if "poc_unknown_hypercap_threshold" in cohort_any.columns:
    print(
        "POC UNKNOWN qualifying hadm count:",
        int(cohort_any["poc_unknown_hypercap_threshold"].sum()),
    )

# New hadm list used for the rest of the notebook
hadm_list = cohort_any.loc[cohort_any["enrolled_any"]==1,
↳ "hadm_id"].dropna().astype("int64").tolist()
len(hadm_list)

```

```

ICD-only admissions      : 4237
Threshold-only admissions : 36393
Both ICD and threshold    : 3310
Total enrolled (union)    : 40630

```

POC UNKNOWN qualifying hadm count: 0

40629

#### 2.3.4 4) First ABG and First VBG (LAB + POC, standardized to mmHg)

**Rationale:** Extract earliest ABG/VBG measurements to characterize baseline physiology with standardized units.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
→ consistent across notebook stages.

```
params = {"hadms": hadm_list}
```

```
bg_pairs_sql = rf"""
```

```
WITH hadms AS (SELECT hadm_id FROM UNNEST(@hadms) AS hadm_id),
```

```
/* ----- LAB (HOSP) ----- */
```

```
hosp_cand AS (
```

```
  SELECT
```

```
    le.subject_id, le.hadm_id, le.charttime, le.specimen_id, le.itemid,
```

```
    CAST(le.valuenum AS FLOAT64) AS val,
```

```
    LOWER(COALESCE(le.valueuom, '')) AS uom,
```

```
    LOWER(di.label) AS lbl,
```

```
    LOWER(COALESCE(di.fluid, '')) AS fl,
```

```
    LOWER(COALESCE(di.category, '')) AS cat
```

```
FROM `{PHYS}.{HOSP}.labevents` le
```

```
JOIN `{PHYS}.{HOSP}.d_labitems` di ON di.itemid = le.itemid
```

```
JOIN hadms h ON h.hadm_id = le.hadm_id
```

```
WHERE le.valuenum IS NOT NULL
```

```
  AND le.itemid IN UNNEST(ARRAY_CONCAT({LAB_PCO2_SQL_ARRAY},
```

```
→ {LAB_PH_SQL_ARRAY}))
```

```
  AND LOWER(COALESCE(di.fluid, '')) IN ({LAB_BLOOD_FLUID_SQL_LIST})
```

```
),
```

```
hosp_spec AS (
```

```
  SELECT le.specimen_id, LOWER(COALESCE(le.value, '')) AS spec_val
```

```
FROM `{PHYS}.{HOSP}.labevents` le
```

```
WHERE le.specimen_id IS NOT NULL
```

```
  AND le.itemid IN UNNEST({LAB_SPECIMEN_SQL_ARRAY})
```

```
QUALIFY ROW_NUMBER() OVER (
```

```
  PARTITION BY le.specimen_id
```

```
  ORDER BY le.charttime DESC
```

```

    ) = 1
),
hosp_class AS (
    SELECT
        c.hadm_id, c.charttime, c.specimen_id, c.itemid, c.val, c.uom, c.lbl,
        ↪ c.fl,
        CASE
            WHEN c.itemid IN UNNEST({LAB_PH_SQL_ARRAY}) THEN 'ph'
            WHEN c.itemid IN UNNEST({LAB_PC02_SQL_ARRAY}) THEN 'pco2'
            ELSE NULL
        END AS analyte,
        CASE
            WHEN REGEXP_CONTAINS(s.spec_val, r'arter') OR
            ↪ REGEXP_CONTAINS(s.spec_val, r'\bart\b') THEN 'arterial'
            WHEN REGEXP_CONTAINS(s.spec_val, r'ven|mixed|central') THEN 'venous'
            ELSE 'unknown'
        END AS site
    FROM hosp_cand c
    LEFT JOIN hosp_spec s USING (specimen_id)
),
hosp_pairs AS (
    SELECT
        hadm_id, specimen_id,
        MIN(charttime) AS sample_time,
        MAX(IF(analyte='ph', val, NULL)) AS ph,
        MAX(IF(analyte='pco2', val, NULL)) AS pco2_raw,
        (ARRAY_AGG(IF(analyte='pco2', uom, NULL) IGNORE NULLS LIMIT
        ↪ 1))[OFFSET(0)] AS pco2_uom,
        (ARRAY_AGG(IF(analyte='ph', uom, NULL) IGNORE NULLS LIMIT
        ↪ 1))[OFFSET(0)] AS ph_uom,
        (ARRAY_AGG(site IGNORE NULLS LIMIT 1))[OFFSET(0)] AS site
    FROM hosp_class
    GROUP BY hadm_id, specimen_id
    HAVING pco2_raw IS NOT NULL AND site IN ('arterial','venous','unknown')
),
hosp_pairs_std AS (
    SELECT
        hadm_id, specimen_id, sample_time, site,
        ph, ph_uom,
        CASE WHEN pco2_uom = 'kpa' THEN pco2_raw * 7.50062 ELSE pco2_raw END AS
        ↪ pco2_mmHg,
        'mmhg' AS pco2_uom_norm
    FROM hosp_pairs

```

```

WHERE (ph IS NULL OR (ph BETWEEN 6.3 AND 7.8))
      AND (pco2_raw IS NULL OR (CASE WHEN pco2_uom='kpa' THEN pco2_raw*7.50062
↪ ELSE pco2_raw END) BETWEEN 5 AND 200)
),
lab_abg AS (
  SELECT hadm_id,
         ph          AS lab_abg_ph,
         ph_uom      AS lab_abg_ph_uom,
         pco2_mmHg   AS lab_abg_paco2,
         'mmhg'      AS lab_abg_paco2_uom,
         sample_time AS lab_abg_time
  FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
        FROM hosp_pairs_std WHERE site='arterial') WHERE rn=1
),
lab_vbg AS (
  SELECT hadm_id,
         ph          AS lab_vbg_ph,
         ph_uom      AS lab_vbg_ph_uom,
         pco2_mmHg   AS lab_vbg_paco2,
         'mmhg'      AS lab_vbg_paco2_uom,
         sample_time AS lab_vbg_time
  FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
        FROM hosp_pairs_std WHERE site='venous') WHERE rn=1
),
lab_other AS (
  SELECT hadm_id,
         ph          AS lab_other_ph,
         ph_uom      AS lab_other_ph_uom,
         pco2_mmHg   AS lab_other_paco2,
         'mmhg'      AS lab_other_paco2_uom,
         sample_time AS lab_other_time
  FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
        FROM hosp_pairs_std WHERE site='unknown') WHERE rn=1
),

/* ----- POC (ICU) ----- */
icu_item_map AS (
  SELECT
    itemid,
    LOWER(label) AS lbl,

```

```

CASE
    WHEN itemid IN UNNEST({ICU_PH_SQL_ARRAY}) THEN 'ph'
    WHEN itemid IN UNNEST({ICU_PCO2_SQL_ARRAY}) THEN 'pco2'
    ELSE NULL
END AS analyte
FROM `{PHYS}`.{ICU}.d_items`
WHERE itemid IN UNNEST(ARRAY_CONCAT({ICU_PH_SQL_ARRAY},
↪ {ICU_PCO2_SQL_ARRAY}))
),
icu_spec AS (
    SELECT
        ce.stay_id,
        ce.charttime,
        LOWER(COALESCE(ce.value, '')) AS spec_val
    FROM `{PHYS}`.{ICU}.chartevents` ce
    WHERE ce.itemid IN UNNEST({ICU_SPECIMEN_SQL_ARRAY})
        AND ce.value IS NOT NULL
),
icu_raw AS (
    SELECT
        ie.hadm_id, ce.stay_id, ce.itemid, ce.charttime,
        di.lbl,
        di.analyte,
        CASE
            WHEN ce.itemid IN UNNEST({ICU_PCO2_ABG_SQL_ARRAY}) THEN 'arterial'
            WHEN ce.itemid IN UNNEST({ICU_PCO2_VBG_SQL_ARRAY}) THEN 'venous'
            ELSE 'unknown'
        END AS site_itemid,
        LOWER(REPLACE(COALESCE(ce.valueuom, ''), ' ', '')) AS uom_nospace,
        LOWER(COALESCE(ce.value, '')) AS valstr,
        COALESCE(
            CAST(ce.valuenum AS FLOAT64),
            SAFE_CAST(REGEXP_EXTRACT(LOWER(ce.value), r'(-?\d+(?:\.\d+)?)') AS
↪ FLOAT64)
        ) AS val
    FROM `{PHYS}`.{ICU}.chartevents` ce
    JOIN icu_item_map di ON di.itemid = ce.itemid
    JOIN `{PHYS}`.{ICU}.icustays` ie ON ie.stay_id = ce.stay_id
    JOIN hadms h ON h.hadm_id = ie.hadm_id
),
icu_cand_ranked AS (
    SELECT
        r.*,

```



```

        s.spec_val,
        ROW_NUMBER() OVER (
            PARTITION BY r.hadm_id, r.stay_id, r.itemid, r.charttime
            ORDER BY ABS(TIMESTAMP_DIFF(r.charttime, s.charttime, SECOND))
        ) AS spec_rank
    FROM icu_raw r
    LEFT JOIN icu_spec s
        ON s.stay_id = r.stay_id
        AND ABS(TIMESTAMP_DIFF(r.charttime, s.charttime, MINUTE)) <= 120
),
icu_cand AS (
    SELECT
        hadm_id, stay_id, itemid, charttime, lbl, analyte, uom_nospace, valstr,
        val,
        CASE
            WHEN REGEXP_CONTAINS(COALESCE(spec_val, ''), r'arter|\\bart\\b') THEN
                'arterial'
            WHEN REGEXP_CONTAINS(COALESCE(spec_val, ''), r'ven|central|mixed') THEN
                'venous'
            WHEN site_itemid IN ('arterial', 'venous') THEN site_itemid
            ELSE 'unknown'
        END AS site
    FROM icu_cand_ranked
    WHERE val IS NOT NULL
        AND analyte IN ('ph', 'pco2')
        AND (spec_rank = 1 OR spec_val IS NULL)
),
icu_ph AS (
    SELECT hadm_id, stay_id, charttime, val AS ph, site AS site_ph
    FROM icu_cand WHERE analyte='ph'
),
icu_co2 AS (
    SELECT hadm_id, stay_id, charttime, val AS pco2_raw, uom_nospace, valstr,
        site AS site_co2
    FROM icu_cand WHERE analyte='pco2'
),
icu_pair_win AS (
    SELECT
        p.hadm_id, p.stay_id,
        COALESCE(p.site_ph, c.site_co2) AS site,
        p.charttime AS ph_time, c.charttime AS co2_time,
        p.ph,
        CASE

```

```

        WHEN c.uom_nospace='kpa' THEN 'kpa'
        WHEN c.uom_nospace='mmhg' THEN 'mmhg'
        ELSE c.uom_nospace
    END AS pco2_uom_norm_raw,
    c.pco2_raw,
    ABS(TIMESTAMP_DIFF(c.charttime, p.charttime, SECOND)) AS dt_sec
FROM icu_ph p
JOIN icu_co2 c
    ON c.hadm_id = p.hadm_id
   AND c.stay_id = p.stay_id
   AND (COALESCE(p.site_ph, c.site_co2) IN ('arterial','venous','unknown'))
   AND ABS(TIMESTAMP_DIFF(c.charttime, p.charttime, MINUTE)) <= 10
QUALIFY ROW_NUMBER() OVER (
    PARTITION BY p.hadm_id, p.stay_id, p.charttime
    ORDER BY dt_sec
) = 1
),
icu_pairs_std AS (
    SELECT
        hadm_id, stay_id, site,
        LEAST(ph_time, co2_time) AS sample_time,
        ph,
        CAST(NULL AS STRING) AS ph_uom,           -- POC pH is unitless/null
        CASE WHEN pco2_uom_norm_raw='kpa' THEN pco2_raw*7.50062 ELSE pco2_raw END
        AS pco2_mmHg,
        'mmhg' AS pco2_uom_norm
    FROM icu_pair_win
    WHERE (ph BETWEEN 6.3 AND 7.8 OR ph IS NULL)
        AND (CASE WHEN pco2_uom_norm_raw='kpa' THEN pco2_raw*7.50062 ELSE
        ↪ pco2_raw END) BETWEEN 5 AND 200
),
icu_solo_pco2_std AS (
    SELECT
        hadm_id, stay_id, site_co2 AS site,
        charttime AS sample_time,
        CAST(NULL AS FLOAT64) AS ph,
        CAST(NULL AS STRING) AS ph_uom,           -- no pH here
        CASE WHEN uom_nospace='kpa' THEN pco2_raw*7.50062 ELSE pco2_raw END AS
        ↪ pco2_mmHg,
        'mmhg' AS pco2_uom_norm
    FROM icu_co2
    WHERE site_co2 IN ('arterial','venous','unknown')
        AND pco2_raw BETWEEN 5 AND 200

```

```

),
icu_all AS (
    SELECT * FROM icu_pairs_std
    UNION ALL
    SELECT * FROM icu_solo_pco2_std
),

poc_abg AS (
    SELECT hadm_id,
           ph          AS poc_abg_ph,
           ph_uom      AS poc_abg_ph_uom,
           pco2_mmHg   AS poc_abg_paco2,
           'mmhg'      AS poc_abg_paco2_uom,
           sample_time AS poc_abg_time
    FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
         FROM icu_all WHERE site='arterial') WHERE rn=1
),

poc_vbg AS (
    SELECT hadm_id,
           ph          AS poc_vbg_ph,
           ph_uom      AS poc_vbg_ph_uom,
           pco2_mmHg   AS poc_vbg_paco2,
           'mmhg'      AS poc_vbg_paco2_uom,
           sample_time AS poc_vbg_time
    FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
         FROM icu_all WHERE site='venous') WHERE rn=1
),

poc_other AS (
    SELECT hadm_id,
           ph          AS poc_other_ph,
           ph_uom      AS poc_other_ph_uom,
           pco2_mmHg   AS poc_other_paco2,
           'mmhg'      AS poc_other_paco2_uom,
           sample_time AS poc_other_time
    FROM (SELECT *, ROW_NUMBER() OVER (PARTITION BY hadm_id ORDER BY
↪ sample_time) rn
         FROM icu_all WHERE site='unknown') WHERE rn=1
)

/* ----- Final one row per hadm ----- */
SELECT

```

```

h.hadm_id,
-- LAB-ABG / LAB-VBG / LAB-OTHER
la.lab_abg_ph, la.lab_abg_ph_uom, la.lab_abg_paco2, la.lab_abg_paco2_uom,
↪ la.lab_abg_time,
lv.lab_vbg_ph, lv.lab_vbg_ph_uom, lv.lab_vbg_paco2, lv.lab_vbg_paco2_uom,
↪ lv.lab_vbg_time,
lo.lab_other_ph, lo.lab_other_ph_uom, lo.lab_other_paco2,
↪ lo.lab_other_paco2_uom, lo.lab_other_time,
-- POC-ABG / POC-VBG / POC-OTHER
pa.poc_abg_ph, pa.poc_abg_ph_uom, pa.poc_abg_paco2, pa.poc_abg_paco2_uom,
↪ pa.poc_abg_time,
pv.poc_vbg_ph, pv.poc_vbg_ph_uom, pv.poc_vbg_paco2, pv.poc_vbg_paco2_uom,
↪ pv.poc_vbg_time,
po.poc_other_ph, po.poc_other_ph_uom, po.poc_other_paco2,
↪ po.poc_other_paco2_uom, po.poc_other_time,
-- First ABG across LAB+POC
(SELECT AS STRUCT src, t, ph, pco2
  FROM (SELECT 'LAB' AS src, la.lab_abg_time AS t, la.lab_abg_ph AS ph,
↪ la.lab_abg_paco2 AS pco2
        UNION ALL
        SELECT 'POC', pa.poc_abg_time, pa.poc_abg_ph, pa.poc_abg_paco2)
  WHERE t IS NOT NULL
  ORDER BY t LIMIT 1) AS first_abg,
-- First VBG across LAB+POC
(SELECT AS STRUCT src, t, ph, pco2
  FROM (SELECT 'LAB' AS src, lv.lab_vbg_time AS t, lv.lab_vbg_ph AS ph,
↪ lv.lab_vbg_paco2 AS pco2
        UNION ALL
        SELECT 'POC', pv.poc_vbg_time, pv.poc_vbg_ph, pv.poc_vbg_paco2)
  WHERE t IS NOT NULL
  ORDER BY t LIMIT 1) AS first_vbg,
-- First UNKNOWN-source pCO2 across LAB+POC (definitive pCO2, unresolved
↪ sample type).
(SELECT AS STRUCT src, t, ph, pco2
  FROM (
    SELECT 'LAB_BG_UNKNOWN' AS src, lo.lab_other_time AS t,
↪ lo.lab_other_ph AS ph, lo.lab_other_paco2 AS pco2
    UNION ALL
    SELECT 'POC_BG_UNKNOWN', po.poc_other_time, po.poc_other_ph,
↪ po.poc_other_paco2
  )
  WHERE t IS NOT NULL
  ORDER BY t LIMIT 1) AS first_other

```

```

FROM hadms h
LEFT JOIN lab_abg la USING (hadm_id)
LEFT JOIN lab_vbg lv USING (hadm_id)
LEFT JOIN lab_other lo USING (hadm_id)
LEFT JOIN poc_abg pa USING (hadm_id)
LEFT JOIN poc_vbg pv USING (hadm_id)
LEFT JOIN poc_other po USING (hadm_id)
"""

bg_pairs = run_sql_bq(bg_pairs_sql, params)

# Flatten STRUCTs for first_abg, first_vbg, and first_other
for col in ["first_abg", "first_vbg", "first_other"]:
    if col in bg_pairs.columns:
        bg_pairs[f"{col}_src"] = bg_pairs[col].apply(lambda x: x.get("src"))
        ↪ if isinstance(x, dict) else None)
        bg_pairs[f"{col}_time"] = bg_pairs[col].apply(lambda x: x.get("t"))
        ↪ if isinstance(x, dict) else None)
        bg_pairs[f"{col}_ph"] = bg_pairs[col].apply(lambda x: x.get("ph"))
        ↪ if isinstance(x, dict) else None)
        bg_pairs[f"{col}_pco2"] = bg_pairs[col].apply(lambda x:
        ↪ x.get("pco2") if isinstance(x, dict) else None)
        bg_pairs = bg_pairs.drop(columns=[col])

if {"first_other_src", "poc_other_paco2"}.issubset(bg_pairs.columns):
    first_other_src_upper = (
        bg_pairs["first_other_src"].astype("string").str.upper().fillna("")
    )
    bg_pairs["first_other_src_detail"] = np.select(
        [
            first_other_src_upper.eq("LAB_BG_UNKNOWN"),
            first_other_src_upper.eq("POC_BG_UNKNOWN"),
            pd.to_numeric(bg_pairs["poc_other_paco2"],
        ↪ errors="coerce").notna(),
        ],
        [
            "lab_bg_unknown",
            "poc_bg_unknown",
            "poc_bg_unknown_available",
        ],
        default="missing",
    )
else:

```

```
bg_pairs["first_other_src_detail"] = "missing"

bg_pairs.head(3)
```

	hadm_id	lab_abg_ph	lab_abg_ph_uom	lab_abg_paco2	lab_abg_paco2_uom	lab_abg_time
0	20000147	7.41	units	35.0	mmhg	2121-08-30 17:38
1	20001305	NaN	None	NaN	None	NaT
2	20001494	7.39	units	45.0	mmhg	2125-10-26 11:27

### 2.3.5 5) Demographics & outcomes

**Rationale:** Build baseline covariates used for descriptive statistics and potential confounding adjustment.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
↳ consistent across notebook stages.

```
SQL["demo_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
SELECT
  a.hadm_id,
  a.subject_id,
  a.admittime,
  a.disctime,
  a.deathtime,
  a.admission_type,
  a.admission_location,
  a.discharge_location,
  a.insurance,
  -- LOS (days)
  TIMESTAMP_DIFF(a.disctime, a.admittime, HOUR) / 24.0 AS hosp_los_days,
  -- in-hospital death
  IF(a.deathtime IS NOT NULL, 1, 0) AS death_in_hosp,
  -- demographics
  p.gender,
  SAFE_CAST(ROUND(p.anchor_age + (EXTRACT(YEAR FROM a.admittime) -
↳ p.anchor_year), 1) AS FLOAT64) AS age_at_admit,
  -- 30-day all-cause mortality from admission
  IF(p.dod IS NOT NULL AND DATE_DIFF(DATE(p.dod), DATE(a.admittime), DAY)
↳ BETWEEN 0 AND 30, 1, 0) AS death_30d
```

```

FROM `{PHYS}`.{HOSP}.admissions` a
JOIN hadms h USING (hadm_id)
JOIN `{PHYS}`.{HOSP}.patients` p USING (subject_id)
"""
demo = run_sql_bq(sql("demo_sql"), {"hadms": hadm_list})
print("Demo rows:", len(demo))
demo.head(3)

```

Demo rows: 40629

	hadm_id	subject_id	admittime	disctime	deathtime	admission_type
0	23485217	10584718	2165-02-12 15:41:00	2165-03-06 08:20:00	2165-03-06 08:20:00	EW EMER.
1	27004173	10297948	2135-08-23 18:46:00	2135-09-08 15:50:00	NaT	URGENT
2	22661627	10032409	2130-01-12 18:42:00	2130-01-21 14:32:00	NaT	EW EMER.

```

# Purpose: Create reusable data-quality and merge guardrail helpers to
↳ prevent silent join errors.

# ==== Drop-in: safe merge utilities (one cell, run once) ====
import pandas as pd
from typing import Iterable, Optional, Literal

def _ensure_Int64(s: pd.Series) -> pd.Series:
    """Coerce to pandas nullable Int64 (preserves NA)."""
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def strip_subject_cols(fr: pd.DataFrame) -> pd.DataFrame:
    """Remove any subject_id-like columns from a frame (e.g., 'subject_id',
↳ 'Subject_ID')."""
    return fr.drop(columns=[c for c in fr.columns if
↳ c.lower().startswith("subject_id")],
                    errors="ignore")

def safe_merge_on_hadm(
    left: pd.DataFrame,
    right: pd.DataFrame,
    *,
    right_name: str,
    take: Optional[Iterable[str]] = None,
    order_by: Optional[Iterable[str]] = None,
    check_subject: Literal[False, "warn", "raise"] = False,

```

```

) -> pd.DataFrame:
    """
    Left-merge 'right' into 'left' on hadm_id, returning a copy of left with
    ↪ right's columns.
    - Dedupes right on hadm_id (optionally using order_by to pick the first
    ↪ row).
    - Optionally restricts right columns via `take`.
    - Optionally audits subject_id agreement before dropping subject_id from
    ↪ right.
    - Always strips subject_id-like columns from the right to prevent *_x/_y
    ↪ suffixes.
    - Raises if any *_x/_y suffixes still appear (indicates overlapping names
    ↪ besides hadm_id).
    """
    if "hadm_id" not in left.columns:
        raise KeyError(f"left frame lacks hadm_id before merging
        ↪ {right_name}")
    if "hadm_id" not in right.columns:
        raise KeyError(f"{right_name} lacks hadm_id")

    L = left.copy()
    R = right.copy()

    # Standardize dtypes of keys
    L["hadm_id"] = _ensure_Int64(L["hadm_id"])
    R["hadm_id"] = _ensure_Int64(R["hadm_id"])
    if "subject_id" in L.columns:
        L["subject_id"] = _ensure_Int64(L["subject_id"])
    if "subject_id" in R.columns:
        R["subject_id"] = _ensure_Int64(R["subject_id"])

    # Dedupe RIGHT by hadm_id (optionally order_by first)
    if order_by:
        R = (R.sort_values(list(order_by))
              .drop_duplicates(subset=["hadm_id"], keep="first"))
    else:
        R = R.drop_duplicates(subset=["hadm_id"], keep="first")

    # Optional subject_id consistency audit (before stripping)
    if check_subject and ("subject_id" in L.columns) and ("subject_id" in
    ↪ R.columns):
        # Join only on hadm_id where both sides have subject_id
        tmp = (L[["hadm_id", "subject_id"]]

```



```

        .merge(R[["hadm_id", "subject_id"]],
              on="hadm_id", how="inner", suffixes=("_L", "_R"))
    mism = (tmp["subject_id_L"].notna() & tmp["subject_id_R"].notna() &
           (tmp["subject_id_L"] != tmp["subject_id_R"]))
    n_mism = int(mism.sum())
    if n_mism > 0:
        sample_ids = tmp.loc[mism, "hadm_id"].head(10).tolist()
        msg = (f"[{right_name}] subject_id mismatch on {n_mism}"
        ↪  hadm_id(s). "
               f"Examples: {sample_ids}")
        if check_subject == "raise":
            raise ValueError(msg)
        else:
            print("WARNING:", msg)

    # Limit right columns (avoid accidental overlaps)
    if take is not None:
        keep = ["hadm_id"] + [c for c in take if c != "hadm_id"]
        R = R[[c for c in keep if c in R.columns]]

    # Always strip subject_id-like columns from right to prevent *_x/_y
    R = strip_subject_cols(R)

    # Final merge
    out = L.merge(R, on="hadm_id", how="left", suffixes=("", ""))

    # Guard: no suffixes should be present
    bad = [c for c in out.columns if c.endswith("_x") or c.endswith("_y")]
    if bad:
        raise RuntimeError(
            f"Merge with {right_name} produced suffixed columns {bad}. "
            "You likely have overlapping column names other than hadm_id."
        )
    return out

print("Safe merge helpers loaded.")

```

Safe merge helpers loaded.

### 2.3.6 6) NIH/OMB race & ethnicity (ED + Hospital)

**Rationale:** Harmonize race/ethnicity across sources using NIH/OMB categories for consistent reporting.

```

# Purpose: Define a reusable BigQuery execution helper so SQL calls are
↪ consistent across notebook stages.

race_eth_sql = rf"""
WITH hadms AS (
    SELECT x AS hadm_id
    FROM UNNEST(@hadms) AS x
),

-- Hospital admission "race" text
hosp AS (
    SELECT a.hadm_id, LOWER(TRIM(a.race)) AS race_hosp_raw
    FROM `{PHYS}`.{HOSP}.admissions` a
    JOIN hadms hm USING (hadm_id)
),

-- Earliest ED stay leading to the admission; take its "race" text if present
ed_first AS (
    SELECT
        e.hadm_id,
        (ARRAY_AGG(STRUCT(e.intime AS intime, LOWER(TRIM(e.race)) AS race_ed_raw)
            ORDER BY e.intime ASC LIMIT 1))[OFFSET(0)] AS pick
    FROM `{PHYS}`.{ED}.edstays` e
    JOIN hadms hm USING (hadm_id)
    GROUP BY e.hadm_id
),
ed AS (
    SELECT hadm_id, pick.race_ed_raw
    FROM ed_first
),

-- Combine ED + Hospital for maximum coverage
comb AS (
    SELECT
        hm.hadm_id,
        ho.race_hosp_raw,
        ed.race_ed_raw,
        TRIM(REGEXP_REPLACE(CONCAT(COALESCE(ho.race_hosp_raw,''), ' '),
↪ COALESCE(ed.race_ed_raw,'')), r'\s+', ' ')) AS race_text_any
    FROM hadms hm
    LEFT JOIN hosp ho USING (hadm_id)
    LEFT JOIN ed ed USING (hadm_id)
),

```

```

-- Tokenization to OMB families + Hispanic ethnicity
tok AS (
  SELECT
    hadm_id, race_hosp_raw, race_ed_raw, race_text_any,

    -- Ethnicity (Hispanic)
    REGEXP_CONTAINS(race_text_any, r'\b(hispanic|latinx|latino|latina)\b') AS
    ↪ is_hisp,

    -- Race families (use boundaries to reduce false positives)
    REGEXP_CONTAINS(race_text_any, r'american\s+indian|\balaska\b') AS
    ↪ is_aian,
    REGEXP_CONTAINS(race_text_any, r'\basian\b') AS is_asian,
    REGEXP_CONTAINS(race_text_any, r'\b(black|african\s+american)\b') AS
    ↪ is_black,
    REGEXP_CONTAINS(race_text_any, r'hawaiian|pacific\s+islander') AS
    ↪ is_nhopi,
    REGEXP_CONTAINS(race_text_any, r'\bwhite\b|caucasian') AS is_white,

    -- Unknown/other indicators
    REGEXP_CONTAINS(race_text_any,
    ↪ r'unknown|other|declined|unable|not\s+reported|missing|null') AS
    ↪ is_unknown_any,

    -- Multi-race hints
    REGEXP_CONTAINS(race_text_any,
    ↪ r'(two|2)\s+or\s+more|multi|biracial|multiracial') AS is_multi_hint
  FROM comb
),

-- Decide ethnicity per NIH
ethn AS (
  SELECT
    hadm_id, race_hosp_raw, race_ed_raw, race_text_any,
    CASE
      WHEN is_hisp THEN 'Hispanic or Latino'
      WHEN (race_text_any IS NULL OR race_text_any = '' OR is_unknown_any)
    ↪ THEN 'Unknown or Not Reported'
      ELSE 'Not Hispanic or Latino'
    END AS nih_ethnicity,
    (CAST(is_aian AS INT64) + CAST(is_asian AS INT64) + CAST(is_black AS
    ↪ INT64)

```

```

        + CAST(is_nhapi AS INT64) + CAST(is_white AS INT64)) AS race_hits,
        is_aian, is_asian, is_black, is_nhapi, is_white, is_multi_hint,
↪ is_unknown_any
    FROM tok
),

-- Decide race per NIH/OMB (1997)
race_assign AS (
    SELECT
        hadm_id, race_hosp_raw, race_ed_raw, race_text_any, nih_ethnicity,
        CASE
            WHEN race_hits >= 2 OR is_multi_hint THEN 'More than one race'
            WHEN is_aian THEN 'American Indian or Alaska Native'
            WHEN is_asian THEN 'Asian'
            WHEN is_black THEN 'Black or African American'
            WHEN is_nhapi THEN 'Native Hawaiian or Other Pacific Islander'
            WHEN is_white THEN 'White'
            WHEN is_unknown_any OR race_text_any IS NULL OR race_text_any = '' THEN
↪ 'Unknown or Not Reported'
            ELSE 'Unknown or Not Reported'
        END AS nih_race
    FROM ethn
)

SELECT hadm_id, race_hosp_raw, race_ed_raw, nih_race, nih_ethnicity
FROM race_assign
"""

race_eth = run_sql_bq(race_eth_sql, {"hadms": hadm_list})
print("Race/Eth rows:", len(race_eth))
race_eth.head(3)

```

Race/Eth rows: 40629

	hadm_id	race_hosp_raw	race_ed_raw	nih_race	nih_ethnicity
0	20000147	white - other european	None	White	Unknown or Not Reported
1	20001305	white	white	White	Not Hispanic or Latino
2	20001494	white	None	White	Not Hispanic or Latino

### 2.3.7 7) ED triage (linked to hadm) and first ED vitals

**Rationale:** Capture ED presentation features (vitals and chief complaint) for symptom and severity analyses.

```
# Purpose: Pull ED triage/first-vitals by loading ED tables once and
↳ filtering to cohort hadm_ids in pandas.

hadms_for_ed = set(
    pd.Series(hadm_list)
    .dropna()
    .astype("int64")
    .tolist()
)

# 1) ED stay map (full table), then restrict to cohort hadm_ids locally.
SQL["edmap_all_sql"] = f"""
SELECT stay_id, hadm_id, intime
FROM `{PHYS}`.{ED}.edstays`
WHERE hadm_id IS NOT NULL
"""

edmap_all = run_sql_bq(sql("edmap_all_sql"))
edmap_all["hadm_id"] = pd.to_numeric(edmap_all["hadm_id"],
↳ errors="coerce").astype("Int64")
edmap_all["stay_id"] = pd.to_numeric(edmap_all["stay_id"],
↳ errors="coerce").astype("Int64")
edmap_all["intime"] = pd.to_datetime(edmap_all["intime"], errors="coerce")

edmap = edmap_all[edmap_all["hadm_id"].isin(hadms_for_ed)].copy()
edmap = edmap.drop_duplicates(subset=["stay_id", "hadm_id"])
print("ED map rows (cohort):", len(edmap))

if edmap.empty:
    ed_triage = pd.DataFrame(columns=[
        "hadm_id", "ed_triage_temp", "ed_triage_hr", "ed_triage_rr",
        "ed_triage_o2sat", "ed_triage_sbp", "ed_triage_dbp",
        ↳ "ed_triage_pain",
        "ed_triage_acuity", "ed_triage_cc",
    ])
    ed_first = pd.DataFrame(columns=[
        "hadm_id", "ed_first_vitals_time", "ed_first_temp", "ed_first_hr",
        "ed_first_rr", "ed_first_o2sat", "ed_first_sbp", "ed_first_dbp",
        "ed_first_rhythm", "ed_first_pain",
```

```

    ])
else:
    # 2) ED triage (full table), then reduce to earliest ED stay per hadm.
    SQL["ed_triage_all_sql"] = f"""
    SELECT
        stay_id,
        temperature      AS ed_triage_temp,
        heartrate         AS ed_triage_hr,
        resprate          AS ed_triage_rr,
        o2sat             AS ed_triage_o2sat,
        sbp               AS ed_triage_sbp,
        dbp               AS ed_triage_dbp,
        pain              AS ed_triage_pain,
        acuity            AS ed_triage_acuity,
        chiefcomplaint AS ed_triage_cc
    FROM `{PHYS}`.{ED}.triage`
    """
    tri_all = run_sql_bq(sql("ed_triage_all_sql"))
    tri_all["stay_id"] = pd.to_numeric(tri_all["stay_id"],
    ↪ errors="coerce").astype("Int64")

    tri_merged = (
        edmap[["stay_id", "hadm_id", "intime"]]
        .merge(tri_all, on="stay_id", how="left")
    )
    ed_triage = (
        tri_merged
        .sort_values(["hadm_id", "intime"], na_position="last")
        .drop_duplicates(subset=["hadm_id"], keep="first")
        .drop(columns=["stay_id", "intime"])
        .reset_index(drop=True)
    )
    print("ED triage rows:", len(ed_triage))

    # 3) First vitals per stay across full ED vitals table, then reduce to
    ↪ earliest vitals-time per hadm.
    SQL["ed_first_vitals_all_sql"] = f"""
    WITH vs_ranked AS (
        SELECT
            v.stay_id,
            v.charttime,
            v.temperature,
            v.heartrate,

```

```

        v.resprate,
        v.o2sat,
        v.sbp,
        v.dbp,
        v.rhythm,
        v.pain,
        ROW_NUMBER() OVER (PARTITION BY v.stay_id ORDER BY v.charttime) AS rn
    FROM `{PHYS}.{ED}.vitalsign` v
)
SELECT
    stay_id,
    charttime      AS ed_first_vitals_time,
    temperature    AS ed_first_temp,
    heartrate      AS ed_first_hr,
    resprate       AS ed_first_rr,
    o2sat          AS ed_first_o2sat,
    sbp            AS ed_first_sbp,
    dbp            AS ed_first_dbp,
    rhythm         AS ed_first_rhythm,
    pain           AS ed_first_pain
FROM vs_ranked
WHERE rn = 1
"""

first_stay_all = run_sql_bq(sql("ed_first_vitals_all_sql"))
first_stay_all["stay_id"] = pd.to_numeric(first_stay_all["stay_id"],
↪ errors="coerce").astype("Int64")
↪ first_stay_all["ed_first_vitals_time"] =
↪ pd.to_datetime(first_stay_all["ed_first_vitals_time"], errors="coerce")

first_merged = (
    edmap[["stay_id", "hadm_id"]]
    .merge(first_stay_all, on="stay_id", how="left")
)
ed_first = (
    first_merged
    .sort_values(["hadm_id", "ed_first_vitals_time"], na_position="last")
    .drop_duplicates(subset=["hadm_id"], keep="first")
    .drop(columns=["stay_id"])
    .reset_index(drop=True)
)
print("ED first vitals rows:", len(ed_first))

```

ED map rows (cohort): 12038

ED triage rows: 12020  
ED first vitals rows: 12020

### 2.3.8 8) ICU meta (first ICU stay, LOS days)

**Rationale:** Summarize ICU exposure and length of stay to contextualize disease severity.

# Purpose: Build first-ICU metadata via full-table pull + local hadm filter  
↪ to avoid large ARRAY parameter stalls.

```
SQL["icu_all_sql"] = f"""
SELECT
    hadm_id,
    stay_id AS first_icu_stay_id,
    intime AS icu_intime,
    outtime AS icu_outtime
FROM `{PHYS}`.{ICU}.icustays`
WHERE hadm_id IS NOT NULL
"""

icu_all = run_sql_bq(sql("icu_all_sql"))
icu_all["hadm_id"] = pd.to_numeric(icu_all["hadm_id"],
    ↪ errors="coerce").astype("Int64")
icu_all["first_icu_stay_id"] = pd.to_numeric(icu_all["first_icu_stay_id"],
    ↪ errors="coerce").astype("Int64")
icu_all["icu_intime"] = pd.to_datetime(icu_all["icu_intime"],
    ↪ errors="coerce")
icu_all["icu_outtime"] = pd.to_datetime(icu_all["icu_outtime"],
    ↪ errors="coerce")

hadms_for_icu = set(pd.Series(hadm_list).dropna().astype("int64").tolist())
icu_all = icu_all[icu_all["hadm_id"].isin(hadms_for_icu)].copy()

icu_meta = (
    icu_all
    .sort_values(["hadm_id", "icu_intime", "first_icu_stay_id"],
    ↪ na_position="last")
    .drop_duplicates(subset=["hadm_id"], keep="first")
    .reset_index(drop=True)
)
icu_meta["icu_los_days"] = (
    (icu_meta["icu_outtime"] - icu_meta["icu_intime"]).dt.total_seconds() /
    ↪ 86400.0
)
```



```
print("ICU meta rows:", len(icu_meta))
```

```
ICU meta rows: 31033
```

### 2.3.9 9) Ventilation flags (ICD procedures)

**Rationale:** Identify IMV/NIV exposure as clinically relevant respiratory support indicators.

# Purpose: Build IMV/NIV ICD flags from targeted procedure codes, then filter  
↪ to cohort hadm\_ids locally.

```
SQL["vent_proc_sql"] = f"""
SELECT
    hadm_id,
    icd_version,
    REPLACE(icd_code, '.', '') AS code_norm
FROM `{PHYS}`.{HOSP}.procedures_icd`
WHERE
    (icd_version = 10 AND icd_code IN
    ↪ ('5A1935Z', '5A1945Z', '5A1955Z', 'OBH17EZ', 'OBH18EZ', '5A09357', '5A09457', '5A09557'))
    OR
    (icd_version = 9 AND REPLACE(icd_code, '.', '') IN
    ↪ ('9670', '9671', '9672', '9604', '9390', '9391', '9399'))
"""

vent_proc = run_sql_bq(sql("vent_proc_sql"))
vent_proc["hadm_id"] = pd.to_numeric(vent_proc["hadm_id"],
    ↪ errors="coerce").astype("Int64")
vent_proc["icd_version"] = pd.to_numeric(vent_proc["icd_version"],
    ↪ errors="coerce").astype("Int64")
vent_proc["code_norm"] = vent_proc["code_norm"].astype(str)

hadms_for_vent = set(pd.Series(hadm_list).dropna().astype("int64").tolist())
vent_proc = vent_proc[vent_proc["hadm_id"].isin(hadms_for_vent)].copy()

imv_codes_10 = {"5A1935Z", "5A1945Z", "5A1955Z", "OBH17EZ", "OBH18EZ"}
imv_codes_9 = {"9670", "9671", "9672", "9604"}
niv_codes_10 = {"5A09357", "5A09457", "5A09557"}
niv_codes_9 = {"9390", "9391", "9399"}

vent_proc["imv_hit"] = (
```

```

        ((vent_proc["icd_version"] == 10) &
↪ vent_proc["code_norm"].isin(imv_codes_10)) |
        ((vent_proc["icd_version"] == 9) &
↪ vent_proc["code_norm"].isin(imv_codes_9))
    )
vent_proc["niv_hit"] = (
    ((vent_proc["icd_version"] == 10) &
↪ vent_proc["code_norm"].isin(niv_codes_10)) |
    ((vent_proc["icd_version"] == 9) &
↪ vent_proc["code_norm"].isin(niv_codes_9))
)

vent = (
    vent_proc
    .groupby("hadm_id", as_index=False)
    .agg(
        imv_flag=("imv_hit", "max"),
        niv_flag=("niv_hit", "max"),
    )
)
vent["imv_flag"] = vent["imv_flag"].astype(int)
vent["niv_flag"] = vent["niv_flag"].astype(int)
vent["any_vent_flag"] = ((vent["imv_flag"] == 1) | (vent["niv_flag"] ==
↪ 1)).astype(int)

# Preserve one row per hadm in hadm_list even if no vent procedure codes were
↪ found.
vent = pd.DataFrame({"hadm_id": pd.Series(hadm_list,
↪ dtype="Int64"))).drop_duplicates().merge(vent, on="hadm_id", how="left")
vent[["imv_flag", "niv_flag", "any_vent_flag"]] = vent[["imv_flag",
↪ "niv_flag", "any_vent_flag"]].fillna(0).astype(int)

print("Vent rows:", len(vent))

```

Vent rows: 40629

and from charts

```

# Purpose: Extract earliest NIV/IMV charttimes using prefiltered ventilation
↪ itemids for better performance.

```

```

SQL["vent_chart_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x),

```

```

stays AS (
    SELECT DISTINCT hadm_id, stay_id
    FROM `{PHYS}`.{ICU}.icustays`
    WHERE hadm_id IN (SELECT hadm_id FROM hadms)
),

vent_itemids AS (
    SELECT itemid, LOWER(label) AS lbl
    FROM `{PHYS}`.{ICU}.d_items`
    WHERE REGEXP_CONTAINS(LOWER(label), r'(vent|ventilator|mode|bipap|bi[-
↵ ]?pap|cpap|nippv|niv|mask|ett|endotracheal)')
),

cand AS (
    SELECT
        s.hadm_id,
        ce.charttime,
        vi.lbl,
        LOWER(COALESCE(ce.value, '')) AS valstr
    FROM `{PHYS}`.{ICU}.chartevents` ce
    JOIN stays s ON s.stay_id = ce.stay_id
    JOIN vent_itemids vi ON vi.itemid = ce.itemid
),

flags AS (
    SELECT
        hadm_id,
        MIN(IF(
            REGEXP_CONTAINS(lbl, r'(non[- ]?invasive|niv|nippv|bipap|bi[-
↵ ]?pap|cpap)')
            OR REGEXP_CONTAINS(valstr, r'(non[- ]?invasive|niv|nippv|bipap|bi[-
↵ ]?pap|cpap)'),
            charttime, NULL)) AS first_niv_time,
        MIN(IF(
            REGEXP_CONTAINS(lbl, r'(invasive
↵ ventilation|endotracheal|ett|mech(|anical)? vent|ventilator
↵ mode|ac/|simv|prvc|aprvc|pcv|vcv|assist\s*control)')
            OR REGEXP_CONTAINS(valstr, r'(invasive
↵ ventilation|endotracheal|ett|ac/|simv|prvc|aprvc|pcv|vcv|assist\
↵ s*control)'),
            charttime, NULL)) AS first_imv_time,
        MAX(CASE

```

```

        WHEN REGEXP_CONTAINS(lbl, r'(non[- ]?invasive|niv|nippv|bipap|bi[-
↪ ]?pap|cpap)')
        OR REGEXP_CONTAINS(valstr, r'(non[-
↪ ]?invasive|niv|nippv|bipap|bi[- ]?pap|cpap)')
        THEN 1 ELSE 0 END) AS niv_chart_flag,
        MAX(CASE
            WHEN REGEXP_CONTAINS(lbl, r'(invasive
↪ ventilation|endotracheal|ett|mech(|anical)? vent|ventilator
↪ mode|ac/|simv|prvc|aprvc|pcv|vcv|assist\s*control)')
            OR REGEXP_CONTAINS(valstr, r'(invasive
↪ ventilation|endotracheal|ett|ac/|simv|prvc|aprvc|pcv|vcv|assist\_|
↪ s*control)')
            THEN 1 ELSE 0 END) AS imv_chart_flag
    FROM cand
    GROUP BY hadm_id
)

SELECT hadm_id, niv_chart_flag, imv_chart_flag, first_niv_time,
↪ first_imv_time
FROM flags
"""

try:
    vent_chart = run_sql_bq(sql("vent_chart_sql"), {"hadms": hadm_list})
except Exception as e:
    print("WARNING: vent chart extraction failed; falling back to ICD-only
↪ vent timing.", e)
    vent_chart = pd.DataFrame(columns=["hadm_id", "niv_chart_flag",
↪ "imv_chart_flag", "first_niv_time", "first_imv_time"])

# Purpose: Derive respiratory support flags and timing fields for IMV/NIV
↪ exposure.

# If your existing ICD-only result is called `vent`, rename for clarity:
vent_proc = vent.copy()

# Outer merge so we keep hadm_ids that appear in only one source
vent_combined = vent_proc.merge(vent_chart, on="hadm_id", how="outer")

# Fill missing with 0 before taking maxima
for c in ["imv_flag", "niv_flag", "any_vent_flag", "imv_chart_flag",
↪ "niv_chart_flag"]:
    if c in vent_combined.columns:

```

```

        vent_combined[c] = vent_combined[c].fillna(0).astype("Int64")

# Final "any-source" flags
vent_combined["imv_flag"] =
    ↪ vent_combined[["imv_flag", "imv_chart_flag"]].max(axis=1).astype("Int64")
vent_combined["niv_flag"] =
    ↪ vent_combined[["niv_flag", "niv_chart_flag"]].max(axis=1).astype("Int64")
vent_combined["any_vent_flag"] =
    ↪ vent_combined[["imv_flag", "niv_flag"]].max(axis=1).astype("Int64")

vent_combined = vent_combined[["hadm_id", "imv_flag", "niv_flag",
    ↪ "any_vent_flag", "first_imv_time", "first_niv_time"]]
print("After combining ICD + chart signals:",
      "\nIMV=1:", int((vent_combined["imv_flag"]==1).sum()),
      "\nNIV=1:", int((vent_combined["niv_flag"]==1).sum()))

```

After combining ICD + chart signals:

IMV=1: 25082

NIV=1: 16756

### 2.3.10 10) Assemble final DataFrame

**Rationale:** Merge all derived features into a single analytic table keyed by `hadm_id`.

```

# Purpose: Extract and standardize first ABG/VBG physiology fields for
    ↪ baseline characterization.

# Canonical base (carries authoritative subject_id)
df = demo.copy()

# Cohort flags / thresholds / labs / etc.
df = safe_merge_on_hadm(df, cohort_any, right_name="cohort_any",
    ↪ check_subject="warn")
df = safe_merge_on_hadm(df, bg_pairs, right_name="bg_pairs")
df = safe_merge_on_hadm(df, race_eth, right_name="race_eth")
df = safe_merge_on_hadm(df, ed_triage, right_name="ed_triage")
df = safe_merge_on_hadm(df, ed_first, right_name="ed_first")
df = safe_merge_on_hadm(df, icu_meta, right_name="icu_meta")
df = safe_merge_on_hadm(df, vent_combined, right_name="vent_combined")

# Anchor to first ED presentation (per admission)
if "ed_intime_first" in globals():

```

```

    df = safe_merge_on_hadm(df, ed_intime_first,
↪   right_name="ed_intime_first")

# Derived timing: first NIV/IMV relative to ED presentation
if "ed_intime_first" in df.columns and "first_imv_time" in df.columns:
    df["dt_first_imv_hours"] = (df["first_imv_time"] -
↪   df["ed_intime_first"]).dt.total_seconds() / 3600.0
if "ed_intime_first" in df.columns and "first_niv_time" in df.columns:
    df["dt_first_niv_hours"] = (df["first_niv_time"] -
↪   df["ed_intime_first"]).dt.total_seconds() / 3600.0

# ABG/VBG before IMV (hadm-level)
if {"first_abg_time", "first_imv_time"}.issubset(df.columns):
    df["abg_before_imv"] = (
        df["first_abg_time"].notna() & df["first_imv_time"].notna() &
        (df["first_abg_time"] < df["first_imv_time"])
    ).astype("Int64")
if {"first_vbg_time", "first_imv_time"}.issubset(df.columns):
    df["vbg_before_imv"] = (
        df["first_vbg_time"].notna() & df["first_imv_time"].notna() &
        (df["first_vbg_time"] < df["first_imv_time"])
    ).astype("Int64")

# Canonical hadm-level threshold normalization.
# This consolidates explicit threshold columns with parsed ABG/VBG/other pCO2
↪   fields
# so later merges cannot silently zero-out source-specific thresholds.
def _num_series(frame: pd.DataFrame, col: str) -> pd.Series:
    if col in frame.columns:
        return pd.to_numeric(frame[col], errors="coerce")
    return pd.Series(np.nan, index=frame.index)

def _any_ge(frame: pd.DataFrame, cols: list[str], threshold: float) ->
↪   pd.Series:
    avail = [c for c in cols if c in frame.columns]
    if not avail:
        return pd.Series(0, index=frame.index, dtype="Int64")
    mat = pd.concat([_num_series(frame, c) for c in avail], axis=1)
    return mat.ge(threshold).any(axis=1).astype("Int64")

```

```

existing_abg = _num_series(df,
    ↪ "abg_hypercap_threshold").fillna(0).astype(int)
existing_vbg = _num_series(df,
    ↪ "vbg_hypercap_threshold").fillna(0).astype(int)
existing_unknown = _num_series(
    df,
    "unknown_hypercap_threshold" if "unknown_hypercap_threshold" in
    ↪ df.columns else "other_hypercap_threshold",
).fillna(0).astype(int)

abg_from_values = _any_ge(df, ["lab_abg_paco2", "poc_abg_paco2",
    ↪ "first_abg_pco2"], 45.0)
vbg_from_values = _any_ge(df, ["lab_vbg_paco2", "poc_vbg_paco2",
    ↪ "first_vbg_pco2"], 50.0)
unknown_from_values = _any_ge(df, ["first_other_pco2"], 50.0)

df["abg_hypercap_threshold"] = ((existing_abg == 1) | (abg_from_values ==
    ↪ 1)).astype("Int64")
df["vbg_hypercap_threshold"] = ((existing_vbg == 1) | (vbg_from_values ==
    ↪ 1)).astype("Int64")
df["unknown_hypercap_threshold"] = (
    (existing_unknown == 1) | (unknown_from_values == 1)
).astype("Int64")
df["other_hypercap_threshold"] =
    ↪ df["unknown_hypercap_threshold"].astype("Int64")

df["pco2_threshold_any"] = (
    (df["abg_hypercap_threshold"] == 1)
    | (df["vbg_hypercap_threshold"] == 1)
    | (df["unknown_hypercap_threshold"] == 1)
).astype("Int64")

if "any_hypercap_icd" in df.columns:
    df["enrolled_any"] = ((pd.to_numeric(df["any_hypercap_icd"],
    ↪ errors="coerce").fillna(0).astype(int) == 1) | (df["pco2_threshold_any"]
    ↪ == 1)).astype("Int64")

print(
    "Threshold sanity (hadm-level):",
    "ABG=", int(df["abg_hypercap_threshold"].fillna(0).sum()),
    "VBG=", int(df["vbg_hypercap_threshold"].fillna(0).sum()),
    "UNKNOWN=", int(df["unknown_hypercap_threshold"].fillna(0).sum()),
    "ANY=", int(df["pco2_threshold_any"].fillna(0).sum()),

```

```

)
if "poc_unknown_hypercaps_threshold" in df.columns:
    poc_unknown_n = int(
        pd.to_numeric(df["poc_unknown_hypercaps_threshold"], errors="coerce")
        .fillna(0)
        .astype(int)
        .sum()
    )
    print("POC UNKNOWN qualifying hadm count:", poc_unknown_n)

# Harmonize NIH race/ethnicity into a collapsed reporting label at
↳ cohort-build time.
RACE_NIH_MAP = {
    "WHITE": "WHITE",
    "BLACK OR AFRICAN AMERICAN": "BLACK",
    "ASIAN": "ASIAN",
    "AMERICAN INDIAN OR ALASKA NATIVE": "AI/AN",
    "NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER": "NH/PI",
    "MORE THAN ONE RACE": "MULTIRACIAL",
    "UNKNOWN OR NOT REPORTED": "UNKNOWN",
    "OTHER": "OTHER",
    "UNKNOWN": "UNKNOWN",
}

ETH_NIH_MAP = {
    "HISPANIC OR LATINO": "HISPANIC",
    "NOT HISPANIC OR LATINO": "NOT HISPANIC",
    "UNKNOWN OR NOT REPORTED": "UNKNOWN",
    "UNKNOWN": "UNKNOWN",
}

LABEL_MAP = {
    "WHITE": "Non-Hispanic White",
    "BLACK": "Non-Hispanic Black",
    "ASIAN": "Non-Hispanic Asian",
    "AI/AN": "Non-Hispanic American Indian/Alaska Native",
    "NH/PI": "Non-Hispanic Native Hawaiian/Pacific Islander",
    "MULTIRACIAL": "Non-Hispanic Multiracial/Other",
    "OTHER": "Unknown/Other",
    "UNKNOWN": "Unknown/Other",
}

```



```

def add_collapsed_race_eth(
    frame: pd.DataFrame,
    race_col: str = "nih_race",
    eth_col: str = "nih_ethnicity",
    out_col: str = "race_eth_collapsed",
) -> pd.DataFrame:
    """Collapse NIH race/ethnicity fields into publication-friendly
    ↪ categories."""
    if race_col not in frame.columns or eth_col not in frame.columns:
        return frame

    def _std(value):
        if pd.isna(value):
            return None
        return re.sub(r"\s+", " ", str(value).strip()).upper()

    race_std = frame[race_col].map(lambda value:
    ↪ RACE_NIH_MAP.get(_std(value), "OTHER"))
    eth_std = frame[eth_col].map(lambda value: ETH_NIH_MAP.get(_std(value),
    ↪ "UNKNOWN"))

    collapsed = []
    for race_value, eth_value in zip(race_std, eth_std):
        if eth_value == "HISPANIC":
            collapsed.append("Hispanic/Latino")
        else:
            collapsed.append(LABEL_MAP.get(race_value, "Unknown/Other"))

    category_order = [
        "Hispanic/Latino",
        "Non-Hispanic White",
        "Non-Hispanic Black",
        "Non-Hispanic Asian",
        "Non-Hispanic American Indian/Alaska Native",
        "Non-Hispanic Native Hawaiian/Pacific Islander",
        "Non-Hispanic Multiracial/Other",
        "Unknown/Other",
    ]

    frame[out_col] = pd.Categorical(collapsed, categories=category_order,
    ↪ ordered=True)
    frame["nih_race_std"] = race_std
    frame["nih_ethnicity_std"] = eth_std

```

```

    return frame

df = add_collapsed_race_eth(df)

print("Final df rows:", len(df), "cols:", len(df.columns))

# Safety checks
assert "subject_id" in df.columns, "subject_id missing from final df"
assert not any(c.endswith("_x") or c.endswith("_y") for c in df.columns),
    ↪ "Found suffixed columns"
print("Final df rows:", len(df), "cols:", len(df.columns))
df.head(3)

Threshold sanity (hadm-level): ABG= 30737 VBG= 14324 UNKNOWN= 4148 ANY= 39702
POC UNKNOWN qualifying hadm count: 0
Final df rows: 40629 cols: 111
Final df rows: 40629 cols: 111

```

	hadm_id	subject_id	admittime	disctime	deathtime	admission_type
0	23485217	10584718	2165-02-12 15:41:00	2165-03-06 08:20:00	2165-03-06 08:20:00	EW EMER.
1	27004173	10297948	2135-08-23 18:46:00	2135-09-08 15:50:00	NaT	URGENT
2	22661627	10032409	2130-01-12 18:42:00	2130-01-21 14:32:00	NaT	EW EMER.

```

# Purpose: Define a reusable BigQuery execution helper so SQL calls are
    ↪ consistent across notebook stages.

# --- Cohort flow counts (ED / ICU / blood gas / hypercapnia / CC) ---

# 1) Dataset-level ED counts
SQL["ed_counts_sql"] = f"""
SELECT
    COUNT(*) AS total_ed_encounters,
    COUNTIF(hadm_id IS NOT NULL) AS ed_encounters_with_hadm
FROM `{PHYS}`.{ED}.edstays`
"""

SQL["ed_to_icu_sql"] = f"""
SELECT
    COUNT(DISTINCT e.hadm_id) AS ed_to_icu_hadm,
    COUNT(DISTINCT e.stay_id) AS ed_to_icu_edstays

```

```

FROM `{PHYS}`.{ED}.edstays` e
JOIN `{PHYS}`.{ICU}.icustays` i USING (hadm_id)
WHERE e.hadm_id IS NOT NULL
"""

try:
    ed_counts = run_sql_bq(sql("ed_counts_sql"))
    ed_to_icu = run_sql_bq(sql("ed_to_icu_sql"))
except Exception as e:
    print("Warning: ED/ICU counts query failed:", e)
    ed_counts = None
    ed_to_icu = None

# 2) Cohort-level counts (admission-level)
cohort_union = int((cohort_any["enrolled_any"] == 1).sum()) if "cohort_any"
    ↪ in globals() and "enrolled_any" in cohort_any.columns else len(hadm_list)
cohort_df_n = len(df)

# Any blood gas present (ABG/VBG, LAB/POC)
co2_cols = [c for c in [
    "lab_abg_paco2", "lab_vbg_paco2", "poc_abg_paco2", "poc_vbg_paco2"
] if c in df.columns]
any_bg = int(df[co2_cols].notna().any(axis=1).sum()) if co2_cols else None

# Hypercapnia thresholds and ICD
icd_count = int((df["any_hypercap_icd"] == 1).sum()) if "any_hypercap_icd"
    ↪ in df.columns else None
threshold_count = int((df["pco2_threshold_any"] == 1).sum()) if
    ↪ "pco2_threshold_any" in df.columns else None

# ED chief complaint missing / present (within cohort)
if "ed_triage_cc" in df.columns:
    mask_cc_present = df["ed_triage_cc"].notna() &
    ↪ (df["ed_triage_cc"].astype(str).str.strip() != "")
    cc_present = int(mask_cc_present.sum())
    cc_missing = int((~mask_cc_present).sum())
else:
    cc_present = None
    cc_missing = None

# ED→ICU within cohort (admissions with ED triage data and ICU stay)
if "first_icu_stay_id" in df.columns and "ed_triage_cc" in df.columns:

```

```

        cohort_ed_to_icu = int((df["first_icu_stay_id"].notna() &
↪ mask_cc_present).sum())
    else:
        cohort_ed_to_icu = None

rows = []
if ed_counts is not None:
    rows.append({"step": "Total ED encounters (edstays)", "count":
↪ int(ed_counts.loc[0, "total_ed_encounters"]), "scope": "All ED dataset"})
    rows.append({"step": "ED encounters with hadm_id", "count":
↪ int(ed_counts.loc[0, "ed_encounters_with_hadm"]), "scope": "All ED
↪ dataset"})
if ed_to_icu is not None:
    rows.append({"step": "ED→ICU admissions (distinct hadm_id)", "count":
↪ int(ed_to_icu.loc[0, "ed_to_icu_hadm"]), "scope": "All ED+ICU"})
    rows.append({"step": "ED→ICU ED-stays (distinct stay_id)", "count":
↪ int(ed_to_icu.loc[0, "ed_to_icu_edstays"]), "scope": "All ED+ICU"})

rows.append({"step": "Cohort admissions (union ICD thresholds)", "count":
↪ cohort_union, "scope": "Cohort"})
rows.append({"step": "Cohort admissions after merges (df rows)", "count":
↪ cohort_df_n, "scope": "Cohort"})
if any_bg is not None:
    rows.append({"step": "Cohort with any ABG/VBG (LAB or POC)", "count":
↪ any_bg, "scope": "Cohort"})
if threshold_count is not None:
    rows.append({"step": "Cohort meeting hypercapnia thresholds", "count":
↪ threshold_count, "scope": "Cohort"})
if icd_count is not None:
    rows.append({"step": "Cohort meeting ICD code criteria", "count":
↪ icd_count, "scope": "Cohort"})
if cc_present is not None:
    rows.append({"step": "Cohort with ED chief complaint present", "count":
↪ cc_present, "scope": "Cohort"})
if cc_missing is not None:
    rows.append({"step": "Cohort excluded for missing ED chief complaint",
↪ "count": cc_missing, "scope": "Cohort"})
if cohort_ed_to_icu is not None:
    rows.append({"step": "Cohort ED→ICU (ED CC present + ICU stay)", "count":
↪ cohort_ed_to_icu, "scope": "Cohort"})

flow_counts = pd.DataFrame(rows)
flow_counts

```

	step	count	scope
0	Total ED encounters (edstays)	425087	All ED dataset
1	ED encounters with hadm_id	203016	All ED dataset
2	ED→ICU admissions (distinct hadm_id)	31862	All ED+ICU
3	ED→ICU ED-stays (distinct stay_id)	31916	All ED+ICU
4	Cohort admissions (union ICD thresholds)	40630	Cohort
5	Cohort admissions after merges (df rows)	40629	Cohort
6	Cohort with any ABG/VBG (LAB or POC)	39478	Cohort
7	Cohort meeting hypercapnia thresholds	39702	Cohort
8	Cohort meeting ICD code criteria	4237	Cohort
9	Cohort with ED chief complaint present	12020	Cohort
10	Cohort excluded for missing ED chief complaint	28609	Cohort
11	Cohort ED→ICU (ED CC present + ICU stay)	9286	Cohort

```
# Purpose: Quantify overlap between ascertainment routes so non-exclusive
↳ cohorts are explicit.
```

```
# --- Ascertainment overlap counts (ABG/VBG/ICD) ---
```

```
required = ["abg_hypercap_threshold", "vbg_hypercap_threshold",
↳ "other_hypercap_threshold", "any_hypercap_icd"]
missing = [c for c in required if c not in df.columns]
if missing:
    raise KeyError(f"Missing required columns for overlap counts: {missing}")

abg = pd.to_numeric(df["abg_hypercap_threshold"],
↳ errors="coerce").fillna(0).astype(int)
vbg = pd.to_numeric(df["vbg_hypercap_threshold"],
↳ errors="coerce").fillna(0).astype(int)
other = pd.to_numeric(df["other_hypercap_threshold"],
↳ errors="coerce").fillna(0).astype(int)

gas_any = (
    pd.to_numeric(df.get("pco2_threshold_any", None), errors="coerce")
    if "pco2_threshold_any" in df.columns else (abg | vbg | other)
)
if hasattr(gas_any, "fillna"):
    gas_any = gas_any.fillna(0).astype(int)
else:
    gas_any = gas_any.astype(int)
```

```

icd = pd.to_numeric(df["any_hypercap_icd"],
    ↪ errors="coerce").fillna(0).astype(int)

total_n = len(df)
ngas = int((gas_any == 1).sum())

abg_vbg_overlap = pd.DataFrame([
    {"group": "ABG-only", "count": int(((abg==1) & (vbg==0) &
    ↪ (other==0)).sum())},
    {"group": "VBG-only", "count": int(((vbg==1) & (abg==0) &
    ↪ (other==0)).sum())},
    {"group": "Other-only", "count": int(((other==1) & (abg==0) &
    ↪ (vbg==0)).sum())},
    {"group": "Mixed (>=2 routes)", "count": int((((abg + vbg + other) >=
    ↪ 2).sum()))},
])
if ngas > 0:
    abg_vbg_overlap["pct_of_gas"] = (abg_vbg_overlap["count"] / ngas *
    ↪ 100).round(1)
else:
    abg_vbg_overlap["pct_of_gas"] = 0.0
abg_vbg_overlap["pct_of_cohort"] = (abg_vbg_overlap["count"] / max(total_n,1)
    ↪ * 100).round(1)

icd_gas_overlap = pd.DataFrame([
    {"group": "ICD+Gas", "count": int(((icd==1) & (gas_any==1)).sum())},
    {"group": "ICD-only", "count": int(((icd==1) & (gas_any==0)).sum())},
    {"group": "Gas-only", "count": int(((icd==0) & (gas_any==1)).sum())},
    {"group": "Neither", "count": int(((icd==0) & (gas_any==0)).sum())},
])
icd_gas_overlap["pct_of_cohort"] = (icd_gas_overlap["count"] / max(total_n,1)
    ↪ * 100).round(1)

print("ABG/VBG/Other overlap (among gas-positive):")
print(abg_vbg_overlap.to_string(index=False))
print("ICD vs Gas overlap (cohort-level):")
print(icd_gas_overlap.to_string(index=False))

```

ABG/VBG/Other overlap (among gas-positive):

group	count	pct_of_gas	pct_of_cohort
ABG-only	23145	58.3	57.0
VBG-only	7562	19.0	18.6
Other-only	839	2.1	2.1

Mixed (>=2 routes)	8156	20.5	20.1
--------------------	------	------	------

ICD vs Gas overlap (cohort-level):

group	count	pct_of_cohort
ICD+Gas	3310	8.1
ICD-only	927	2.3
Gas-only	36392	89.6
Neither	0	0.0

```
# Purpose: Capture ED presentation context (chief complaint, acuity, and
↳ early vitals) at the ED-stay level.
```

```
# --- Missingness summary (chief complaint, race/ethnicity, ED triage/vitals)
↳ ---
```

```
import numpy as np
```

```
summary_rows = []
```

```
# Chief complaint missingness (ED triage CC)
if "ed_triage_cc" in df.columns:
    cc_present = df["ed_triage_cc"].notna() &
    ↳ (df["ed_triage_cc"].astype(str).str.strip() != "")
    summary_rows.append({
        "variable": "ed_triage_cc_present",
        "missing_n": int((~cc_present).sum()),
        "missing_pct": float((~cc_present).mean())
    })
```

```
# Race/Ethnicity missingness (NIH categories + raw sources)
unknown_tokens = {
    "unknown or not reported",
    "unknown",
    "not reported",
    "missing",
    "declined",
    "unable"
}
```

```
def _missing_rate(series):
    if series is None:
        return None, None
    s = series.astype(str).str.strip()
```

```

        is_missing = series.isna() | (s == "") |
        ↪ s.str.lower().isin(unknown_tokens)
        return int(is_missing.sum()), float(is_missing.mean())

for col in ["nih_race", "nih_ethnicity", "race_hosp_raw", "race_ed_raw"]:
    if col in df.columns:
        m_n, m_p = _missing_rate(df[col])
        summary_rows.append({
            "variable": col,
            "missing_n": m_n,
            "missing_pct": m_p
        })

missing_summary = pd.DataFrame(summary_rows)
print("Missingness summary (key variables):")
missing_summary

# ED triage + first ED vitals missingness
triage_cols = [c for c in df.columns if c.startswith("ed_triage_")]
first_cols = [c for c in df.columns if c.startswith("ed_first_")]

vital_cols = triage_cols + first_cols
if vital_cols:
    miss_tbl = (
        pd.DataFrame({"variable": vital_cols})
        .assign(
            missing_n=lambda d: [int(df[c].isna().sum()) for c in
        ↪ d["variable"]],
            missing_pct=lambda d: [float(df[c].isna().mean()) for c in
        ↪ d["variable"]]
        )
        .sort_values("missing_pct", ascending=False)
    )
    print("Missingness summary (ED triage + first ED vitals):")
    miss_tbl
else:
    miss_tbl = pd.DataFrame(columns=["variable", "missing_n", "missing_pct"])

Missingness summary (key variables):
Missingness summary (ED triage + first ED vitals):

```



### 2.3.11 11) Sanity checks

**Rationale:** Run QC checks to validate units, flags, and basic data integrity before export.

# Purpose: Derive respiratory support flags and timing fields for IMV/NIV  
↪ exposure.

# ICU LOS negative?

```
if {"icu_los_days", "first_icu_stay_id"}.issubset(df.columns):  
    neg_los = int((df["icu_los_days"] < 0).fillna(False).sum())  
    print("Negative ICU LOS rows:", neg_los)
```

# Vent flags consistency

```
vent_cols = {"imv_flag", "niv_flag", "any_vent_flag"}  
if vent_cols.issubset(df.columns):  
    any_calc = ((df["imv_flag"]==1) |  
    ↪ (df["niv_flag"]==1)).fillna(False).astype(int)  
    any_flag = pd.to_numeric(df["any_vent_flag"],  
    ↪ errors="coerce").fillna(0).astype(int)  
    mism = int((any_calc != any_flag).sum())  
    print("any_vent_flag mismatches vs (imv|niv):", mism)
```

# UOMs: expect mmhg only

```
uom_cols = [c for c in df.columns if c.endswith("_paco2_uom")]  
for c in uom_cols:  
    vals = sorted(pd.Series(df[c]).dropna().astype(str).str.lower().str_|  
    ↪ .strip().unique().tolist())  
    print(c, vals)
```

# ABG/VBG coverage QC

```
def qc_pair(df, ph_col, co2_col, label, ph_lo=6.3, ph_hi=7.8, co2_lo=5,  
    ↪ co2_hi=200):  
    ph = pd.to_numeric(df.get(ph_col), errors="coerce")  
    co2 = pd.to_numeric(df.get(co2_col), errors="coerce")  
    return {  
        "pair": label,  
        "present_any": int(((ph.notna()) | (co2.notna()))).sum(),  
        "present_both": int(((ph.notna()) & (co2.notna()))).sum(),  
        "only_ph": int(((ph.notna()) & (~co2.notna()))).sum(),  
        "only_pco2": int(((co2.notna()) & (~ph.notna()))).sum(),  
        "ph_oob": int((((ph < ph_lo) | (ph > ph_hi)) &  
    ↪ ph.notna()).sum()),
```

```

        "pco2_oob":      int((((co2 < co2_lo) | (co2 > co2_hi)) &
        ↪ co2.notna()).sum()),
    }

qc = pd.DataFrame([
    qc_pair(df, "lab_abg_ph", "lab_abg_paco2", "LAB ABG"),
    qc_pair(df, "lab_vbg_ph", "lab_vbg_paco2", "LAB VBG"),
    qc_pair(df, "poc_abg_ph", "poc_abg_paco2", "POC ABG"),
    qc_pair(df, "poc_vbg_ph", "poc_vbg_paco2", "POC VBG"),
])
qc

```

```

Negative ICU LOS rows: 0
any_vent_flag mismatches vs (imv|niv): 0
lab_abg_paco2_uom ['mmhg']
lab_vbg_paco2_uom ['mmhg']
lab_other_paco2_uom ['mmhg']
poc_abg_paco2_uom ['mmhg']
poc_vbg_paco2_uom ['mmhg']
poc_other_paco2_uom []

```

	pair	present_any	present_both	only_ph	only_pco2	ph_oob	pco2_oob
0	LAB ABG	32306	32299	0	7	0	0
1	LAB VBG	17909	17904	0	5	0	0
2	POC ABG	25820	0	0	25820	0	0
3	POC VBG	11844	0	0	11844	0	0

### 2.3.12 PDF-ready long tables

```

print(
    render_latex_longtable(
        flow_counts,
        caption=f"Cohort flow and key denominator checkpoints
    ↪ (N={len(df):,}).",
        label="tab:cohort_flow_counts",
        index=False,
    )
)
print(
    render_latex_longtable(

```

```

        abg_vbg_overlap,
        caption="ABG/VBG/OTHER overlap among gas-positive encounters.",
        label="tab:gas_overlap_abg_vbg_other",
        index=False,
    )
)
print(
    render_latex_longtable(
        icd_gas_overlap,
        caption="ICD versus gas ascertainment overlap in the full cohort.",
        label="tab:icd_gas_overlap",
        index=False,
    )
)
print(
    render_latex_longtable(
        missing_summary,
        caption="Missingness summary for key cohort variables.",
        label="tab:key_missingness",
        index=False,
    )
)
if "miss_tbl" in globals() and not miss_tbl.empty:
    print(
        render_latex_longtable(
            miss_tbl,
            caption="ED triage and first ED vital-sign missingness summary.",
            label="tab:ed_vital_missingness",
            landscape=True,
            index=False,
        )
    )
print(
    render_latex_longtable(
        qc,
        caption="ABG/VBG pH and pCO2 plausibility quality checks.",
        label="tab:abg_vbg_qc",
        landscape=True,
        index=False,
    )
)

\begin{longtable}{lrl}

```

```

\caption{Cohort flow and key denominator checkpoints (N=40,629).} \label{tab:cohort_flow_cou
\toprule
step & count & scope \\
\midrule
\endfirsthead
\caption[]{}{Cohort flow and key denominator checkpoints (N=40,629).} \\
\toprule
step & count & scope \\
\midrule
\endhead
\midrule
\multicolumn{3}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
Total ED encounters (edstays) & 425087 & All ED dataset \\
ED encounters with hadm_id & 203016 & All ED dataset \\
ED→ICU admissions (distinct hadm_id) & 31862 & All ED+ICU \\
ED→ICU ED-stays (distinct stay_id) & 31916 & All ED+ICU \\
Cohort admissions (union ICD thresholds) & 40630 & Cohort \\
Cohort admissions after merges (df rows) & 40629 & Cohort \\
Cohort with any ABG/VBG (LAB or POC) & 39478 & Cohort \\
Cohort meeting hypercapnia thresholds & 39702 & Cohort \\
Cohort meeting ICD code criteria & 4237 & Cohort \\
Cohort with ED chief complaint present & 12020 & Cohort \\
Cohort excluded for missing ED chief complaint & 28609 & Cohort \\
Cohort ED→ICU (ED CC present + ICU stay) & 9286 & Cohort \\
\end{longtable}

\begin{longtable}{lrrr}
\caption{ABG/VBG/OTHER overlap among gas-positive encounters.} \label{tab:gas_overlap_abg_vbg
\toprule
group & count & pct_of_gas & pct_of_cohort \\
\midrule
\endfirsthead
\caption[]{}{ABG/VBG/OTHER overlap among gas-positive encounters.} \\
\toprule
group & count & pct_of_gas & pct_of_cohort \\
\midrule
\endhead
\midrule
\multicolumn{4}{r}{Continued on next page} \\

```

```

\midrule
\endfoot
\bottomrule
\endlastfoot
ABG-only & 23145 & 58.300000 & 57.000000 \\
VBG-only & 7562 & 19.000000 & 18.600000 \\
Other-only & 839 & 2.100000 & 2.100000 \\
Mixed ( $\geq 2$  routes) & 8156 & 20.500000 & 20.100000 \\
\end{longtable}

\begin{longtable}{lrr}
\caption{ICD versus gas ascertainment overlap in the full cohort.} \label{tab:icd_gas_overlap}
\toprule
group & count & pct_of_cohort \\
\midrule
\endfirsthead
\caption[]{}{ICD versus gas ascertainment overlap in the full cohort.} \\
\toprule
group & count & pct_of_cohort \\
\midrule
\endhead
\midrule
\multicolumn{3}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
ICD+Gas & 3310 & 8.100000 \\
ICD-only & 927 & 2.300000 \\
Gas-only & 36392 & 89.600000 \\
Neither & 0 & 0.000000 \\
\end{longtable}

\begin{longtable}{lrr}
\caption{Missingness summary for key cohort variables.} \label{tab:key_missingness} \\
\toprule
variable & missing_n & missing_pct \\
\midrule
\endfirsthead
\caption[]{}{Missingness summary for key cohort variables.} \\
\toprule
variable & missing_n & missing_pct \\
\midrule

```

```

\endhead
\midrule
\multicolumn{3}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
ed_triage_cc_present & 28609 & 0.704152 \\
nih_race & 7607 & 0.187231 \\
nih_ethnicity & 7105 & 0.174875 \\
race_hosp_raw & 3436 & 0.084570 \\
race_ed_raw & 29836 & 0.734352 \\
\end{longtable}

\begin{landscape}\n\begin{longtable}{lrr}
\caption{ED triage and first ED vital-sign missingness summary.} \label{tab:ed_vital_missing}
\toprule
variable & missing_n & missing_pct \\
\midrule
\endfirsthead
\caption[]{}{ED triage and first ED vital-sign missingness summary.} \\
\toprule
variable & missing_n & missing_pct \\
\midrule
\endhead
\midrule
\multicolumn{3}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
ed_first_rhythm & 39958 & 0.983485 \\
ed_first_temp & 33646 & 0.828128 \\
ed_triage_temp & 32206 & 0.792685 \\
ed_triage_rr & 31891 & 0.784932 \\
ed_triage_o2sat & 31855 & 0.784046 \\
ed_triage_dbp & 31799 & 0.782668 \\
ed_triage_sbp & 31758 & 0.781658 \\
ed_triage_hr & 31725 & 0.780846 \\
ed_first_o2sat & 31588 & 0.777474 \\
ed_first_dbp & 31345 & 0.771493 \\
ed_first_sbp & 31345 & 0.771493 \\
ed_first_rr & 31268 & 0.769598 \\

```

```

ed_first_hr & 31149 & 0.766669 \\
ed_first_pain & 30956 & 0.761919 \\
ed_triage_pain & 30460 & 0.749711 \\
ed_triage_acuity & 29971 & 0.737675 \\
ed_first_vitals_time & 29671 & 0.730291 \\
ed_triage_cc & 28609 & 0.704152 \\
\end{longtable}
\n\end{landscape}\n
\begin{landscape}\n\begin{longtable}{lrrrrrr}
\caption{ABG/VBG pH and pCO2 plausibility quality checks.} \label{tab:abg_vbg_qc} \\
\toprule
pair & present_any & present_both & only_ph & only_pco2 & ph_oob & pco2_oob \\
\midrule
\endfirsthead
\caption[]{}{ABG/VBG pH and pCO2 plausibility quality checks.} \\
\toprule
pair & present_any & present_both & only_ph & only_pco2 & ph_oob & pco2_oob \\
\midrule
\endhead
\midrule
\multicolumn{7}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
LAB ABG & 32306 & 32299 & 0 & 7 & 0 & 0 \\
LAB VBG & 17909 & 17904 & 0 & 5 & 0 & 0 \\
POC ABG & 25820 & 0 & 0 & 25820 & 0 & 0 \\
POC VBG & 11844 & 0 & 0 & 11844 & 0 & 0 \\
\end{longtable}
\n\end{landscape}\n

```

### 2.3.13 12) Save to Excel

**Rationale:** Persist cohort outputs for downstream annotation and NLP analyses.

# Purpose: Optional archive export of the full hadm-level cohort workbook.

```
from datetime import datetime
```

```
WRITE_ARCHIVE_XLSX_EXPORTS = os.getenv("WRITE_ARCHIVE_XLSX_EXPORTS", "0") ==
    ↪ "1"
```

```

if WRITE_ARCHIVE_XLSX_EXPORTS:
    prior_runs_dir = DATA_DIR / "prior runs"
    prior_runs_dir.mkdir(parents=True, exist_ok=True)
    out_path = prior_runs_dir / f"mimic_hypercaps_EXT_bq_abg_vbg_{datetime.
↪ now().strftime('%Y%m%d_%H%M%S')}.xlsx"
    with pd.ExcelWriter(out_path, engine="openpyxl") as xw:
        df.to_excel(xw, sheet_name="cohort", index=False)
        try:
            qc.to_excel(xw, sheet_name="qc_abg_vbg", index=False)
        except Exception:
            pass
    out_path
else:
    print("Skipping archive hadm-level workbook export (set
↪ WRITE_ARCHIVE_XLSX_EXPORTS=1 to enable).")

```

Skipping archive hadm-level workbook export (set WRITE\_ARCHIVE\_XLSX\_EXPORTS=1 to enable).

### 2.3.14 Create Annotation Dataset

**Rationale:** Create ED chief-complaint subsets and a reproducible sample for manual annotation.

```

# Purpose: Capture ED presentation context (chief complaint, acuity, and
↪ early vitals) at the ED-stay level.

# ---- Extra exports: (1) ED chief-complaint only; (2) random sample of 160
↪ patients ----
from datetime import datetime

# Dated artifacts go to archive folder (optional)
prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
WRITE_ARCHIVE_XLSX_EXPORTS = os.getenv("WRITE_ARCHIVE_XLSX_EXPORTS", "0") ==
↪ "1"

# 1) Filter to rows with a non-empty ED chief complaint
if "ed_triage_cc" not in df.columns:
    raise KeyError(
        "Column 'ed_triage_cc' not found in df. "

```



```

        "Ensure the ED triage merge cell ran earlier."
    )

mask_cc = df["ed_triage_cc"].notna() &
    ↪ (df["ed_triage_cc"].astype(str).str.strip() != "")
df_cc = df.loc[mask_cc].copy()

print(f"ED-CC present rows: {len(df_cc)} of {len(df)} "
      f"({len(df_cc) / max(len(df),1)}:.1%) of cohort.")

# Save ED-CC-only cohort only when archive exports are enabled
if WRITE_ARCHIVE_XLSX_EXPORTS:
    out_path_cc = prior_runs_dir /
    ↪ f"mimic_hypercap_EXT_EDcc_only_bq_abg_vbg_{timestamp}.xlsx"
    with pd.ExcelWriter(out_path_cc, engine="openpyxl") as xw:
        df_cc.to_excel(xw, sheet_name="cohort_cc_only", index=False)
        try:
            qc.to_excel(xw, sheet_name="qc_abg_vbg", index=False)
        except Exception:
            pass
    print("Saved:", out_path_cc)

# 2) Random sample of n = 160 patients (distinct subject_id), one row per
    ↪ patient
if "subject_id" not in df_cc.columns:
    raise KeyError("Column 'subject_id' missing; cannot sample by patient.")

# Make a one-row-per-patient frame by earliest admission
if "admittime" in df_cc.columns:
    df_cc_one = (
        df_cc.sort_values(["subject_id", "admittime"])
        .groupby("subject_id", as_index=False)
        .head(1)
    )
else:
    # Fallback if admittime not present: choose the smallest hadm_id per
    ↪ patient
    df_cc_one = (
        df_cc.sort_values(["subject_id", "hadm_id"])
        .groupby("subject_id", as_index=False)
        .head(1)
    )

```

```

N = 160
n_avail = len(df_cc_one)
n_take = min(N, n_avail)
if n_avail < N:
    print(f"Warning: only {n_avail} unique patients with ED chief complaint;
    ↪ sampling all of them.")

RANDOM_SEED = 42
df_cc_sample = df_cc_one.sample(n=n_take, random_state=RANDOM_SEED)

# Save the sample only when archive exports are enabled
if WRITE_ARCHIVE_XLSX_EXPORTS:
    out_path_cc_sample = prior_runs_dir /
    ↪ f"mimic_hypercap_EXT_EDcc_sample{n_take}_bq_abg_vbg_{timestamp}.xlsx"
    with pd.ExcelWriter(out_path_cc_sample, engine="openpyxl") as xw:
        df_cc_sample.to_excel(xw, sheet_name="cohort_cc_sample", index=False)
        try:
            qc.to_excel(xw, sheet_name="qc_abg_vbg", index=False)
        except Exception:
            pass
    print("Saved:", out_path_cc_sample)
else:
    print("Skipping ED-CC archive exports (set WRITE_ARCHIVE_XLSX_EXPORTS=1
    ↪ to enable).")

```

ED-CC present rows: 12020 of 40629 (29.6% of cohort).

Skipping ED-CC archive exports (set WRITE\_ARCHIVE\_XLSX\_EXPORTS=1 to enable).

### 3 ED-stay cohort expansion (timing, severity, comorbidity, outcomes)

**Rationale:** Build a one-row-per-ED-stay analytic extract with time-anchored gas phenotypes, key comorbidities, and outcomes.

#### 3.0.1 Phase 0 — Inventory & missing-field registry

**Rationale:** Detect which fields already exist and only add missing fields to avoid redundant extraction or join explosions.

```

# Purpose: Inventory available fields and identify missing target variables
↪ before enrichment steps.

```

```

from pathlib import Path
import json
import pandas as pd

# Identify current cohort dataframe (admission-level)
if 'df' not in globals():
    raise NameError("Expected admission-level df to exist before inventory
        ↪ step.")

ED_KEY = "ed_stay_id" # target key for ED-level cohort

print("Current admission-level df:", df.shape)
print("Current columns count:", len(df.columns))
if ED_KEY in df.columns:
    print("ED stay unique count:", int(df[ED_KEY].nunique()))
else:
    print("ED stay unique count: ED_KEY not in columns")

# Persist columns snapshot
cols_out = WORK_DIR / "current_columns.json"
cols_out.write_text(json.dumps(sorted(df.columns), indent=2))
print("Wrote:", cols_out)

# Persist ED-stay columns snapshot (if ed_df exists)
if "ed_df" in globals():
    ed_cols_out = WORK_DIR / "ed_columns.json"
    ed_cols_out.write_text(json.dumps(sorted(ed_df.columns), indent=2))
    print("Wrote:", ed_cols_out)

Current admission-level df: (40629, 111)
Current columns count: 111
ED stay unique count: ED_KEY not in columns
Wrote: /Users/bloche/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/current_columns.json

# Purpose: Capture ED presentation context (chief complaint, acuity, and
    ↪ early vitals) at the ED-stay level.

# Target field registry
TARGET_RAW_FIELDS = {

```

```

"ed_edstays": [
    "ed_stay_id", "subject_id", "hadm_id", "ed_intime", "ed_outtime", ]
    ↪ "ed_intime_first",
    "arrival_transport", "disposition", "ed_gender", "ed_race",
],
"ed_triage": [
    "ed_triage_temp", "ed_triage_hr", "ed_triage_rr", "ed_triage_o2sat",
    "ed_triage_sbp", "ed_triage_dbp", "ed_triage_pain", "ed_triage_acuity", ]
    ↪ "ed_triage_cc",
],
"ed_vitals_first": [
    "ed_first_vitals_time", "ed_first_temp", "ed_first_hr", "ed_first_rr",
    "ed_first_o2sat", "ed_first_sbp", "ed_first_dbp", "ed_first_rhythm", ]
    ↪ "ed_first_pain",
],
"admissions": [
    "admittime", "disctime", "deathtime", "hospital_expire_flag",
    "admission_type", "admission_location", "discharge_location",
    "insurance", "language", "marital_status", "hosp_race",
],
"icu": [
    "icu_stay_id", "icu_intime_first", "icu_outtime_last", "icu_los_total", ]
    ↪ "n_icu_stays",
    "first_careunit", "last_careunit",
],
"labs_gas": [
    "first_gas_time", "first_pco2", "first_ph", "first_hco3", ]
    ↪ "first_lactate",
    "max_pco2_0_6h", "min_ph_0_6h", "max_pco2_0_24h", "min_ph_0_24h",
    "flag_abg_hypercapnia", "flag_vbg_hypercapnia", ]
    ↪ "flag_unknown_hypercapnia", "flag_other_hypercapnia", ]
    ↪ "flag_any_gas_hypercapnia",
    "gas_source_unknown_rate", "gas_source_other_rate", ]
    ↪ "poc_inclusion_enabled", "poc_inclusion_reason",
    "first_gas_specimen_type", "first_gas_specimen_present", ]
    ↪ "first_gas_pco2_itemid", "first_gas_pco2_fluid",
    "co2_other_is_blood_asserted", "first_gas_anchor_has_pco2", ]
    ↪ "first_gas_anchor_source_validated",
    "dt_first_imv_hours", "dt_first_niv_hours", "first_other_time",
],
"omr": [
    "bmi_closest_pre_ed", "height_closest_pre_ed", "weight_closest_pre_ed",

```

```

        "bmi_closest_pre_ed_model", "height_closest_pre_ed_model",
        ↪ "weight_closest_pre_ed_model",
        "bmi_outlier_flag", "height_outlier_flag", "weight_outlier_flag",
        "anthro_timing_tier", "anthro_days_offset", "anthro_chartdate",
        ↪ "anthro_timing_uncertain",
        "anthro_source", "anthro_obstime", "anthro_hours_offset",
        ↪ "anthro_timing_basis",
    ],
    "dx_flags": [
        "flag_copd", "flag_osa_ohs", "flag_chf", "flag_neuromuscular",
        "flag_opioid_substance", "flag_pneumonia",
    ],
    "timing": [
        "dt_first_qualifying_gas_hours", "presenting_hypercapnia",
        ↪ "late_hypercapnia",
        "qualifying_gas_observed", "presenting_hypercapnia_tri",
        ↪ "late_hypercapnia_tri", "hypercap_timing_class",
        "dt_first_imv_hours", "dt_first_niv_hours", "abg_before_imv",
        ↪ "vbg_before_imv",
        "dt_first_imv_hours_model", "dt_first_niv_hours_model",
        "imv_time_outside_window_flag", "niv_time_outside_window_flag",
        "hospital_los_negative_flag", "admittime_before_ed_intime_flag",
        ↪ "dischtime_before_admittime_flag", "time_integrity_any",
        "ph_band", "hco3_band", "lactate_band",
    ],
}

TARGET_DERIVED_FIELDS = [
    "hospital_los_hours", "hospital_los_hours_model", "in_hospital_death",
]

TARGET_FIELDS = sorted({c for v in TARGET_RAW_FIELDS.values() for c in v} |
    ↪ set(TARGET_DERIVED_FIELDS))

# Grouped missing report
missing_by_group = {}
for group, cols in TARGET_RAW_FIELDS.items():
    missing_by_group[group] = [c for c in cols if c not in df.columns]

missing_derived = [c for c in TARGET_DERIVED_FIELDS if c not in df.columns]
missing_by_group["derived"] = missing_derived

missing_flat = [c for cols in missing_by_group.values() for c in cols]

```

```
print("Missing fields total:", len(missing_flat))
for group, cols in missing_by_group.items():
    if cols:
        print(f"- {group}: {cols}")
```

Missing fields total: 92

```
- ed_edstays: ['ed_stay_id', 'ed_intime', 'ed_outtime', 'ed_intime_first', 'arrival_transport']
- admissions: ['hospital_expire_flag', 'language', 'marital_status', 'hosp_race']
- icu: ['icu_stay_id', 'icu_intime_first', 'icu_outtime_last', 'icu_los_total', 'n_icu_stays']
- labs_gas: ['first_gas_time', 'first_pco2', 'first_ph', 'first_hco3', 'first_lactate', 'max_ph']
- omr: ['bmi_closest_pre_ed', 'height_closest_pre_ed', 'weight_closest_pre_ed', 'bmi_closest_post_ed', 'height_closest_post_ed', 'weight_closest_post_ed']
- dx_flags: ['flag_copd', 'flag_osa_ohs', 'flag_chf', 'flag_neuromuscular', 'flag_opioid_substance']
- timing: ['dt_first_qualifying_gas_hours', 'presenting_hypercapnia', 'late_hypercapnia', 'q10_hypercapnia']
- derived: ['hospital_los_hours', 'hospital_los_hours_model', 'in_hospital_death']
```

### 3.0.2 Phase 1 — ED encounter spine and ED enrichment (one row per ED stay)

**Rationale:** Create a dedicated ED-stay-level cohort with ED-specific attributes to avoid mixing admission- and ED-level grains.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
↪ consistent across notebook stages.

# ED stay spine (rename stay\_id to ed\_stay\_id)

```
SQL["ed_spine_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
SELECT
    s.stay_id AS ed_stay_id,
    s.subject_id,
    s.hadm_id,
    s.intime AS ed_intime,
    s.outtime AS ed_outtime,
    s.arrival_transport,
    s.disposition,
    s.gender AS ed_gender,
    s.race AS ed_race
FROM `{PHYS}`.{ED}.edstays` s
JOIN hadms h ON h.hadm_id = s.hadm_id
"""
```

```
ed_spine = run_sql_bq(sql("ed_spine_sql"), {"hadms": hadm_list})
```

```

print("ED spine rows:", len(ed_spine), "unique ed_stay_id:",
      ↪ ed_spine["ed_stay_id"].nunique())

# ensure uniqueness
if ed_spine["ed_stay_id"].nunique() != len(ed_spine):
    raise ValueError("ed_stay_id not unique in ED spine")

# First ED presentation time per admission
ed_intime_first = (
    ed_spine.groupby("hadm_id", as_index=False)["ed_intime"]
    .min()
    .rename(columns={"ed_intime": "ed_intime_first"})
)

# Start ED-level df
ed_df = ed_spine.copy()
ed_df = ed_df.merge(ed_intime_first, on="hadm_id", how="left")

ED spine rows: 12038 unique ed_stay_id: 12038

# Purpose: Define a reusable BigQuery execution helper so SQL calls are
↪ consistent across notebook stages.

# ED triage and first ED vitals (reuse existing logic if present; otherwise
↪ join)

def _needs_cols(df, cols):
    return (df is None) or any(c not in df.columns for c in cols)

# Use existing ed_triage / ed_first if already in memory from earlier cells
try:
    _ = ed_triage
except NameError:
    ed_triage = None

try:
    _ = ed_first
except NameError:
    ed_first = None

# Force re-query if required keys are missing
if _needs_cols(locals().get('ed_triage', None), ['ed_stay_id', 'hadm_id']):

```

```

    ed_triage = None
if _needs_cols(locals().get('ed_first', None), ['ed_stay_id']):
    ed_first = None

# If missing, prefer in-memory tables loaded earlier in the notebook.
# This avoids re-running expensive ED queries during nbconvert.
if ed_triage is None:
    if 'edmap' in globals() and 'tri_all' in globals():
        ed_triage = (
            edmap[['stay_id', 'hadm_id']]
            .merge(tri_all, on='stay_id', how='left')
            .rename(columns={'stay_id': 'ed_stay_id'})
        )
        print('ED triage rows (from in-memory tables):', len(ed_triage))
    else:
        ed_triage_sql = f"""
        WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
        SELECT
            s.stay_id AS ed_stay_id,
            s.hadm_id,
            t.temperature      AS ed_triage_temp,
            t.heartrate        AS ed_triage_hr,
            t.resprate         AS ed_triage_rr,
            t.o2sat            AS ed_triage_o2sat,
            t.sbp              AS ed_triage_sbp,
            t.dbp              AS ed_triage_dbp,
            t.pain             AS ed_triage_pain,
            t.acuity           AS ed_triage_acuity,
            t.chiefcomplaint AS ed_triage_cc
        FROM `{PHYS}`.{ED}.edstays` s
        JOIN hadms h ON h.hadm_id = s.hadm_id
        LEFT JOIN `{PHYS}`.{ED}.triage` t ON t.stay_id = s.stay_id
        """
        ed_triage = run_sql_bq(ed_triage_sql, {'hadms': hadm_list})
        print('ED triage rows (fallback query):', len(ed_triage))

if ed_first is None:
    if 'edmap' in globals() and 'first_stay_all' in globals():
        ed_first = (
            edmap[['stay_id', 'hadm_id']]
            .merge(first_stay_all, on='stay_id', how='left')
            .rename(columns={'stay_id': 'ed_stay_id'})
        )

```



```

    print('ED first vitals rows (from in-memory tables):', len(ed_first))
else:
    ed_first_vitals_sql = f"""
    WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x),
    edmap AS (
        SELECT stay_id, hadm_id
        FROM `{PHYS}`.{ED}.edstays`
        WHERE hadm_id IN (SELECT hadm_id FROM hadms)
    ),
    vs AS (
        SELECT stay_id, charttime, temperature, heartrate, resprate, o2sat,
        ↪ sbp, dbp, rhythm, pain
        FROM `{PHYS}`.{ED}.vitalsign`
    ),
    first_vs AS (
        SELECT
            v.stay_id,
            (ARRAY_AGG(STRUCT(v.charttime, v.temperature, v.heartrate,
        ↪ v.resprate, v.o2sat, v.sbp, v.dbp, v.rhythm, v.pain)
                        ORDER BY v.charttime LIMIT 1))[OFFSET(0)] AS pick
        FROM vs v
        JOIN edmap m USING (stay_id)
        GROUP BY v.stay_id
    )
    SELECT
        f.stay_id AS ed_stay_id,
        m.hadm_id,
        pick.charttime AS ed_first_vitals_time,
        pick.temperature AS ed_first_temp,
        pick.heartrate AS ed_first_hr,
        pick.resprate AS ed_first_rr,
        pick.o2sat AS ed_first_o2sat,
        pick.sbp AS ed_first_sbp,
        pick.dbp AS ed_first_dbp,
        pick.rhythm AS ed_first_rhythm,
        pick.pain AS ed_first_pain
    FROM first_vs f
    JOIN edmap m ON m.stay_id = f.stay_id
    """

    ed_first = run_sql_bq(ed_first_vitals_sql, {'hadms': hadm_list})
    print('ED first vitals rows (fallback query):', len(ed_first))

# Debug columns before merge

```

```

print('ed_triage cols:', list(ed_triage.columns))
print('ed_first cols:', list(ed_first.columns))
print('ed_df cols:', list(ed_df.columns))

if 'ed_stay_id' not in ed_df.columns:
    raise KeyError('ed_df missing ed_stay_id; ensure ED spine cell ran.')

# Merge ED triage + vitals onto ed_df with available keys
merge_keys_triage = [k for k in ["ed_stay_id", "hadm_id"] if k in
    ↪ ed_df.columns and k in ed_triage.columns]
if not merge_keys_triage:
    raise KeyError("No common keys between ed_df and ed_triage")
ed_df = ed_df.merge(ed_triage, on=merge_keys_triage, how="left")
merge_keys_first = [k for k in ["ed_stay_id", "hadm_id"] if k in
    ↪ ed_df.columns and k in ed_first.columns]
if not merge_keys_first:
    raise KeyError("No common keys between ed_df and ed_first")
ed_df = ed_df.merge(ed_first, on=merge_keys_first, how="left")

ED triage rows (from in-memory tables): 12038
ED first vitals rows (from in-memory tables): 12038
ed_triage cols: ['ed_stay_id', 'hadm_id', 'ed_triage_temp', 'ed_triage_hr', 'ed_triage_rr',
ed_first cols: ['ed_stay_id', 'hadm_id', 'ed_first_vitals_time', 'ed_first_temp', 'ed_first_l
ed_df cols: ['ed_stay_id', 'subject_id', 'hadm_id', 'ed_intime', 'ed_outtime', 'arrival_trans

```

### 3.0.3 Phase 2 — Hospital admission context and outcomes

**Rationale:** Add admission-level outcomes and demographics for downstream stratification and analysis.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
 ↪ consistent across notebook stages.

```

# Admissions fields
SQL["admit_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
SELECT
    a.hadm_id,
    a.admittime,
    a.dischtime,
    a.deathtime,
    a.hospital_expire_flag,

```

```

a.admission_type,
a.admission_location,
a.discharge_location,
a.insurance,
a.language,
a.marital_status,
a.race AS hosp_race
FROM `{PHYS}`.{HOSP}.admissions` a
JOIN hadms h USING (hadm_id)
"""

admit = run_sql_bq(sql("admit_sql"), {"hadms": hadm_list})
print("Admissions rows:", len(admit))

ed_df = ed_df.merge(admit, on="hadm_id", how="left")

# Merge ventilation flags/times (hadm-level)
if "vent_combined" in globals():
    ed_df = ed_df.merge(vent_combined, on="hadm_id", how="left")

# Derived outcomes
ed_df["hospital_los_hours"] = (ed_df["disctime"] -
    ↪ ed_df["admittime"]).dt.total_seconds() / 3600.0
ed_df["in_hospital_death"] = ((ed_df["hospital_expire_flag"] == 1) |
    ↪ ed_df["deathtime"].notna()).astype("int64")

# Concordance check
discord = (
    ((ed_df["hospital_expire_flag"] == 1) & ed_df["deathtime"].isna()) |
    ((ed_df["hospital_expire_flag"] == 0) & ed_df["deathtime"].notna())
)
print("Admissions discordance (expire_flag vs deathtime):",
    ↪ int(discord.sum()))

Admissions rows: 40629
Admissions discordance (expire_flag vs deathtime): 3

```

### 3.0.4 Phase 3 — ICU timing and LOS

**Rationale:** Capture ICU exposure, timing, and total LOS for severity stratification.

# Purpose: Define a reusable BigQuery execution helper so SQL calls are  
 ↪ consistent across notebook stages.

```

# ICU stays (aggregate per hadm)
SQL["icu_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
SELECT
    i.hadm_id,
    i.stay_id AS icu_stay_id,
    i.intime,
    i.outtime,
    i.los,
    i.first_careunit,
    i.last_careunit
FROM `{PHYS}`.{ICU}.icustays` i
JOIN hadms h USING (hadm_id)
"""

icu = run_sql_bq(sql("icu_sql"), {"hadms": hadm_list})
print("ICU stay rows:", len(icu))

if len(icu) > 0:
    icu_agg = (
        icu.sort_values(["hadm_id", "intime"]).groupby("hadm_id",
↪ as_index=False)
        .agg(
            icu_intime_first=("intime", "min"),
            icu_outtime_last=("outtime", "max"),
            icu_los_total=("los", "sum"),
            n_icu_stays=("icu_stay_id", "nunique"),
            first_careunit=("first_careunit", "first"),
            last_careunit=("last_careunit", "last"),
        )
    )
    ed_df = ed_df.merge(icu_agg, on="hadm_id", how="left")
else:
    print("No ICU stays found for cohort.")

```

ICU stay rows: 36753

### 3.0.5 Phase 4 — ED longitudinal vitals (0–6h)

**Rationale:** Summarize early ED vitals for severity phenotyping.

```

# Purpose: Build ED vitals features robustly without a single large BigQuery
↪ join that can stall.

# ED vitals long + aggregates (0-6h)

# Build the ED stay map from in-memory cohort tables to keep query scope
↪ tight and deterministic.
if "ed_df" not in globals():
    raise NameError("ed_df is required before ED vitals extraction")

required_cols = ["ed_stay_id", "hadm_id", "ed_intime"]
missing_required = [c for c in required_cols if c not in ed_df.columns]
if missing_required:
    raise KeyError(f"ed_df missing required columns for ED vitals extraction:
↪ {missing_required}")

edmap_local = (
    ed_df[required_cols]
    .dropna(subset=["ed_stay_id"])
    .drop_duplicates(subset=["ed_stay_id"])
    .copy()
)
edmap_local["ed_stay_id"] = pd.to_numeric(edmap_local["ed_stay_id"],
↪ errors="coerce")
edmap_local = edmap_local.dropna(subset=["ed_stay_id"])
edmap_local["ed_stay_id"] = edmap_local["ed_stay_id"].astype(int)

stay_ids = sorted(edmap_local["ed_stay_id"].unique().tolist())
print("ED stays for vitals pull:", len(stay_ids))

SQL["ed_vitals_sql"] = f"""
SELECT
    stay_id AS ed_stay_id,
    charttime,
    temperature,
    heartrate,
    resprate,
    o2sat,
    sbp,
    dbp,
    rhythm,
    pain
FROM `{PHYS}.{ED}.vitalsign`

```

```

WHERE stay_id IN UNNEST(@stay_ids)
"""

# Query in chunks so we avoid one very large parameter payload/job.
chunk_size = 5000
vitals_chunks = []
for i in range(0, len(stay_ids), chunk_size):
    chunk = stay_ids[i:i + chunk_size]
    part = run_sql_bq(sql("ed_vitals_sql"), {"stay_ids": chunk})
    if len(part) > 0:
        part["ed_stay_id"] = pd.to_numeric(part["ed_stay_id"],
↪ errors="coerce")
        part = part.dropna(subset=["ed_stay_id"])
        part["ed_stay_id"] = part["ed_stay_id"].astype(int)
        vitals_chunks.append(part)
    print(f"ED vitals chunk {i // chunk_size + 1}: rows={len(part)}")

if vitals_chunks:
    ed_vitals_long = pd.concat(vitals_chunks, ignore_index=True)
else:
    ed_vitals_long = pd.DataFrame(
        columns=[
            "ed_stay_id", "charttime", "temperature", "heartrate",
            ↪ "resprate",
            "o2sat", "sbp", "dbp", "rhythm", "pain"
        ]
    )

ed_vitals_long = ed_vitals_long.merge(edmap_local, on="ed_stay_id",
↪ how="inner")
print("ED vitals long rows:", len(ed_vitals_long))

# Window filter: 0-6h from ED intime
ed_vitals_long["dt_hours"] = (ed_vitals_long["charttime"] -
↪ ed_vitals_long["ed_intime"]).dt.total_seconds() / 3600.0
in_6h = ed_vitals_long["dt_hours"].between(0, 6, inclusive="both")

agg = (
    ed_vitals_long.loc[in_6h]
    .groupby("ed_stay_id", as_index=False)
    .agg(
        max_heartrate_0_6h=("heartrate", "max"),
        max_resprate_0_6h=("resprate", "max"),

```

```

        min_o2sat_0_6h=("o2sat", "min"),
        min_sbp_0_6h=("sbp", "min"),
        n_vitals_0_6h=("charttime", "count"),
    )
)

ed_df = ed_df.merge(agg, on="ed_stay_id", how="left")

# Range warnings (report only, do not drop)
range_checks = {
    "heartrate": (0, 300),
    "resprate": (0, 80),
    "o2sat": (0, 100),
    "sbp": (0, 300),
}
for col, (lo, hi) in range_checks.items():
    bad = ed_vitals_long[col].notna() & (~ed_vitals_long[col].between(lo,
↪ hi))
    if bad.any():
        print(f"Warning: {col} out of range count:", int(bad.sum()))

```

```

ED stays for vitals pull: 12038
ED vitals chunk 1: rows=30885
ED vitals chunk 2: rows=31757
ED vitals chunk 3: rows=12365
ED vitals long rows: 75007
Warning: heartrate out of range count: 1
Warning: resprate out of range count: 2
Warning: o2sat out of range count: 1

```

### 3.0.6 Phase 5 — Robust lab discovery + gas panels

**Rationale:** Capture blood gas and key chemistry labs with label-robust item discovery and unit normalization.

```

# Purpose: Discover lab item IDs robustly and record cohort-aware item
↪ frequency for auditability.

```

```

import re
import json

```

```

# Discover itemids from d_labitems

```

```

SQL["labitems_sql"] = f"""
SELECT itemid, label, fluid, category
FROM `{PHYS}`.{HOSP}.d_labitems`
"""

labitems = run_sql_bq(sql("labitems_sql"))

patterns = {
    "gas_pco2": re.compile(r"\bp\s*co2\b|pco2|pco ", re.I),
    "gas_ph": re.compile(r"\bph\b", re.I),
    "gas_hco3": re.compile(r"hco3|bicarbonate", re.I),
    "gas_lactate": re.compile(r"lactate", re.I),
    "gas_specimen": re.compile(r"specimen|source|type", re.I),
    "chem_creatinine": re.compile(r"creatinine", re.I),
    "chem_sodium": re.compile(r"\bsodium\b", re.I),
    "chem_chloride": re.compile(r"\bchloride\b", re.I),
    "chem_total_co2": re.compile(r"carbon dioxide|total co2|\bco2\b", re.I),
    "cbc_hemoglobin": re.compile(r"hemoglobin", re.I),
}

# Category filters limit false matches (for example chemistry CO2 vs
#   ↪ blood-gas pCO2)
cat_gas = re.compile(r"blood\s*gas|blood gas|arterial|venous", re.I)
cat_chem = re.compile(r"chemistry|chem|blood", re.I)
cat_cbc = re.compile(r"hematology|cbc", re.I)

manifest_lab_itemid_sets: dict[str, set[int]] = {
    "gas_pco2": set(LAB_PC02_ITEMIDS),
    "gas_ph": set(LAB_PH_ITEMIDS),
    "gas_hco3": set(LAB_HCO3_ITEMIDS),
    "gas_specimen": set(LAB_SPECIMEN_TYPE_ITEMIDS),
}

manifest_missing_rows: list[dict[str, Any]] = []
for group_name, itemids in manifest_lab_itemid_sets.items():
    if not itemids:
        continue
    present_itemids = set(
        pd.to_numeric(
            labitems.loc[labitems["itemid"].isin(itemids), "itemid"],
            errors="coerce",
        )
    )
    .dropna()
    .astype(int)

```



```

        .tolist()
    )
    missing_itemids = sorted(itemids.difference(present_itemids))
    if missing_itemids:
        manifest_missing_rows.append(
            {
                "group_name": group_name,
                "missing_itemids": ";\n".join(str(value) for value in
                    ↪ missing_itemids),
                "status": "missing_from_d_labitems",
            }
        )

if manifest_missing_rows:
    raise ValueError(
        "Manifest lab itemids were not found in d_labitems: "
        f"{manifest_missing_rows}. Update specs/blood_gas_itemids.json."
    )

matches = {}
for name, pat in patterns.items():
    if name in manifest_lab_itemid_sets:
        itemids = manifest_lab_itemid_sets[name]
        dfm = labitems.loc[labitems["itemid"].isin(itemids)].copy()
    else:
        dfm = labitems.copy()
        dfm = dfm[dfm["label"].str.contains(pat, na=False)]
        if name.startswith("gas_"):
            dfm = dfm[dfm["category"].str.contains(cat_gas, na=False)]
        elif name.startswith("chem_"):
            dfm = dfm[dfm["category"].str.contains(cat_chem, na=False)]
        elif name.startswith("cbc_"):
            dfm = dfm[dfm["category"].str.contains(cat_cbc, na=False)]
        matches[name] = dfm[["itemid", "label", "category"]]

# Build lab_item_map with cohort counts. Run hadm counts in chunks to avoid
    ↪ one oversized query.
itemids_all = sorted({int(i) for dfm in matches.values() for i in
    ↪ dfm["itemid"].tolist()})

SQL["counts_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x)
SELECT itemid, COUNT(*) AS n

```

```

FROM `{PHYS}`.{HOSP}.labevents`
WHERE hadm_id IN (SELECT hadm_id FROM hadms)
  AND itemid IN UNNEST(@itemids)
GROUP BY itemid
"""

counts_cols = ["itemid", "n"]
if itemids_all and len(hadm_list) > 0:
    hadm_chunk_size = 10000
    count_parts = []
    for i in range(0, len(hadm_list), hadm_chunk_size):
        hadm_chunk = hadm_list[i:i + hadm_chunk_size]
        part = run_sql_bq(sql("counts_sql"), {"hadms": hadm_chunk, "itemids":
↪ itemids_all})
        if len(part) > 0:
            count_parts.append(part)
            print(f"Lab item count chunk {i // hadm_chunk_size + 1}:
↪ rows={len(part)}")

    if count_parts:
        counts = (
            pd.concat(count_parts, ignore_index=True)
            .groupby("itemid", as_index=False)["n"]
            .sum()
        )
    else:
        counts = pd.DataFrame(columns=counts_cols)
else:
    counts = pd.DataFrame(columns=counts_cols)

lab_item_map = {}
for name, dfm in matches.items():
    tmp = dfm.merge(counts, on="itemid", how="left").fillna({"n": 0})
    lab_item_map[name] = {
        "pattern": patterns[name].pattern,
        "items": tmp.sort_values("n",
↪ ascending=False).to_dict(orient="records"),
    }

lab_item_map_path = WORK_DIR / "lab_item_map.json"
lab_item_map_path.write_text(json.dumps(lab_item_map, indent=2))
print("Wrote:", lab_item_map_path)

```

```

manifest_lab_audit_rows: list[dict[str, Any]] = []
for analyte_name, itemids in manifest_lab_itemid_sets.items():
    if not itemids:
        continue
    frame = labitems.loc[labitems["itemid"].isin(itemids), ["itemid",
↪ "label", "category", "fluid"]].copy()
    present_ids = set(frame["itemid"].astype(int).tolist()) if not
↪ frame.empty else set()
    frame = frame.merge(counts, on="itemid", how="left").fillna({"n": 0})
    record_map = {
        int(record["itemid"]): record for record in
        ↪ frame.to_dict(orient="records")
    }
    for itemid_value in sorted(itemids):
        record = record_map.get(int(itemid_value), {})
        manifest_lab_audit_rows.append(
            {
                "source_branch": "LAB",
                "analyte": analyte_name,
                "itemid": int(itemid_value),
                "label": record.get("label"),
                "category": record.get("category"),
                "fluid": record.get("fluid"),
                "cohort_row_count": int(record.get("n", 0)),
                "manifest_version": BLOOD_GAS_MANIFEST["version"],
                "present_in_dictionary": int(itemid_value in present_ids),
            }
        )

blood_gas_itemid_manifest_audit = pd.DataFrame(manifest_lab_audit_rows)
missing_manifest_rows = blood_gas_itemid_manifest_audit.loc[
    pd.to_numeric(
        blood_gas_itemid_manifest_audit.get("present_in_dictionary"),
        ↪ errors="coerce"
    ).fillna(0).astype(int).eq(0)
].copy()
if not missing_manifest_rows.empty:
    missing_repr = ", ".join(
        f"{row.source_branch}:{row.analyte}:{int(row.itemid)}"
        for row in missing_manifest_rows.itertuples(index=False)
    )
    raise ValueError(
        "Blood-gas manifest itemids missing from dictionary tables: "

```

```

        f"{missing_repr}"
    )

Lab item count chunk 1: rows=38
Lab item count chunk 2: rows=38
Lab item count chunk 3: rows=36
Lab item count chunk 4: rows=37
Lab item count chunk 5: rows=24
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/lab_item_map.json

# Purpose: Extract cohort labs and apply ED-time windows in pandas for
↳ predictable runtime.

# Extract labevents and apply ED 0-24h windows locally

# Assemble itemid lists
itemid_sets = {k: [int(x["itemid"]) for x in v["items"]] for k, v in
↳ lab_item_map.items()}
all_itemids = sorted({i for v in itemid_sets.values() for i in v})

if "ed_df" not in globals():
    raise NameError("ed_df is required before lab window extraction")

ed_windows = (
    ed_df[["ed_stay_id", "hadm_id", "ed_intime"]]
    .dropna(subset=["ed_stay_id", "hadm_id", "ed_intime"])
    .drop_duplicates()
    .copy()
)
ed_windows["hadm_id"] = pd.to_numeric(ed_windows["hadm_id"],
↳ errors="coerce").astype("Int64")
ed_windows = ed_windows.dropna(subset=["hadm_id"])
ed_windows["hadm_id"] = ed_windows["hadm_id"].astype(int)

SQL["labs_sql"] = f"""
SELECT
    subject_id,
    hadm_id,
    itemid,
    charttime,
    specimen_id,

```

```

        valuenum,
        valueuom,
        value AS value_text
FROM `{PHYS}`.{HOSP}.labevents`
WHERE hadm_id IN UNNEST(@hadms)
      AND itemid IN UNNEST(@itemids)
"""

# Run in hadm chunks; merge to ED windows in pandas to keep each query
↪ lightweight.
labs_parts = []
if all_itemids and len(hadm_list) > 0:
    hadm_chunk_size = 5000
    for i in range(0, len(hadm_list), hadm_chunk_size):
        hadm_chunk = hadm_list[i:i + hadm_chunk_size]
        part = run_sql_bq(sql("labs_sql"), {"hadms": hadm_chunk, "itemids":
↪ all_itemids})
        raw_n = len(part)

        if raw_n > 0:
            part["hadm_id"] = pd.to_numeric(part["hadm_id"], errors="coerce")
            part = part.dropna(subset=["hadm_id"])
            part["hadm_id"] = part["hadm_id"].astype(int)

            part = part.merge(ed_windows, on="hadm_id", how="inner")
            in_window = (
                (part["charttime"] >= part["ed_intime"]) &
                (part["charttime"] <= (part["ed_intime"] +
↪ pd.Timedelta(hours=24)))
            )
            part = part.loc[in_window, [
                "ed_stay_id", "subject_id", "hadm_id", "itemid", "charttime",
                ↪ "specimen_id", "valuenum", "valueuom", "value_text"
            ]]
            if len(part) > 0:
                labs_parts.append(part)

        print(f"Labs window chunk {i // hadm_chunk_size + 1}: raw={raw_n},
↪ kept={len(part) if raw_n > 0 else 0}")

if labs_parts:
    labs_long = pd.concat(labs_parts, ignore_index=True)
else:

```

```

labs_long = pd.DataFrame(
    columns=["ed_stay_id", "subject_id", "hadm_id", "itemid",
↪  "charttime", "specimen_id", "valuenum", "valueuom", "value_text"]
)

print("Labs long rows:", len(labs_long))

# Unit audit for pCO2
pco2_ids = itemid_sets.get("gas_pco2", [])
unit_audit = (
    labs_long.loc[labs_long["itemid"].isin(pco2_ids)]
    .groupby("valueuom", dropna=False)
    .size().reset_index(name="n")
)
unit_audit_path = WORK_DIR / "lab_unit_audit.csv"
unit_audit.to_csv(unit_audit_path, index=False)
print("Wrote:", unit_audit_path)

Labs window chunk 1: raw=640998, kept=25845
Labs window chunk 2: raw=646475, kept=26480
Labs window chunk 3: raw=652356, kept=26977
Labs window chunk 4: raw=646769, kept=25471
Labs window chunk 5: raw=656528, kept=25060
Labs window chunk 6: raw=672434, kept=26325
Labs window chunk 7: raw=651644, kept=27073
Labs window chunk 8: raw=632006, kept=24831
Labs window chunk 9: raw=84710, kept=3286
Labs long rows: 211348
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/lab_unit_audit.csv

# Purpose: Reconstruct gas panels and normalize units before ED-stay level
↪  aggregation.

# Reconstruct gas panels by specimen_id within ED stay

pco2_ids = set(itemid_sets.get("gas_pco2", []))
ph_ids = set(itemid_sets.get("gas_ph", []))
hco3_ids = set(itemid_sets.get("gas_hco3", []))
lact_ids = set(itemid_sets.get("gas_lactate", []))

labs = labs_long.copy()

```

```

if "fluid" in labitems.columns:
    item_fluid = (
        labitems[["itemid", "fluid"]]
        .drop_duplicates(subset=["itemid"])
        .copy()
    )
    labs = labs.merge(item_fluid, on="itemid", how="left")
else:
    labs["fluid"] = pd.NA

# Convert pCO2 kPa to mmHg when needed
is_kpa = labs["valueuom"].astype(str).str.lower().str.contains("kpa",
    ↪ na=False)
mask_pco2 = labs["itemid"].isin(pco2_ids)
if mask_pco2.any() and is_kpa.any():
    labs.loc[mask_pco2 & is_kpa, "valuenum"] = labs.loc[mask_pco2 & is_kpa,
    ↪ "valuenum"] * 7.50062
    labs.loc[mask_pco2 & is_kpa, "valueuom"] = "mmHg"

# panel by specimen_id
panel = (
    labs.groupby(["ed_stay_id", "specimen_id"], as_index=False)
    .agg(panel_time=("charttime", "min"))
)

# attach analytes

def pick_analyte(
    df: pd.DataFrame,
    ids: set[int],
    name: str,
    *,
    include_item_meta: bool = False,
) -> pd.DataFrame:
    cols = ["ed_stay_id", "specimen_id", "charttime", "valuenum", "itemid",
    ↪ "fluid"]
    tmp = df.loc[df["itemid"].isin(ids), cols].copy()
    tmp["valuenum"] = pd.to_numeric(tmp["valuenum"], errors="coerce")
    tmp = tmp.loc[tmp["valuenum"].notna()].copy()
    if tmp.empty:
        base_cols = ["ed_stay_id", "specimen_id", name]
        if include_item_meta:

```

```

        base_cols += [f"{name}_itemid", f"{name}_fluid"]
    return pd.DataFrame(columns=base_cols)
tmp = tmp.sort_values(
    ["ed_stay_id", "specimen_id", "charttime", "itemid"], kind="stable"
)
first = tmp.groupby(["ed_stay_id", "specimen_id"],
    ↪ as_index=False).first()
first = first.rename(columns={"valuenum": name})
if include_item_meta:
    first = first.rename(
        columns={"itemid": f"{name}_itemid", "fluid": f"{name}_fluid"}
    )
    keep_cols = ["ed_stay_id", "specimen_id", name, f"{name}_itemid",
    ↪ f"{name}_fluid"]
else:
    keep_cols = ["ed_stay_id", "specimen_id", name]
return first[keep_cols]

if pco2_ids:
    panel = panel.merge(
        pick_analyte(
            labs,
            pco2_ids,
            "pco2",
            include_item_meta=True,
        ),
        on=["ed_stay_id", "specimen_id"],
        how="left",
    )
if ph_ids:
    panel = panel.merge(pick_analyte(labs, ph_ids, "ph"),
    ↪ on=["ed_stay_id", "specimen_id"], how="left")
if hco3_ids:
    panel = panel.merge(pick_analyte(labs, hco3_ids, "hco3"),
    ↪ on=["ed_stay_id", "specimen_id"], how="left")
if lact_ids:
    panel = panel.merge(pick_analyte(labs, lact_ids, "lactate"),
    ↪ on=["ed_stay_id", "specimen_id"], how="left")

# First pCO2-qualified panel per ED stay (gas anchor must include pCO2)
panel_anchor_candidates = panel.loc[
    pd.to_numeric(panel.get("pco2"), errors="coerce").notna()
].copy()

```



```

if panel_anchor_candidates.empty:
    first_panel = panel.head(0).copy()
else:
    first_panel = (
        panel_anchor_candidates.sort_values(["ed_stay_id", "panel_time"])
        .groupby("ed_stay_id", as_index=False)
        .first()
    )

# 0-6h and 0-24h extrema
panel = panel.merge(ed_df[["ed_stay_id", "ed_intime"]], on="ed_stay_id",
    ↪ how="left")
panel["dt_hours"] = (panel["panel_time"] -
    ↪ panel["ed_intime"]).dt.total_seconds() / 3600.0

# Ensure expected panel columns exist even if analyte is absent
for col in ["pco2", "ph", "hco3", "lactate"]:
    if col not in panel.columns:
        panel[col] = pd.NA
for col in ["pco2_itemid", "pco2_fluid"]:
    if col not in panel.columns:
        panel[col] = pd.NA

p06 = panel.loc[panel["dt_hours"].between(0,6, inclusive="both")]
p24 = panel.loc[panel["dt_hours"].between(0,24, inclusive="both")]

agg06 = p06.groupby("ed_stay_id",
    ↪ as_index=False).agg(max_pco2_0_6h=("pco2", "max"),
    ↪ min_ph_0_6h=("ph", "min"))
agg24 = p24.groupby("ed_stay_id",
    ↪ as_index=False).agg(max_pco2_0_24h=("pco2", "max"),
    ↪ min_ph_0_24h=("ph", "min"))

first_panel_export_cols = [
    col
    for col in ["ed_stay_id", "panel_time", "pco2", "ph", "hco3", "lactate"]
    if col in first_panel.columns
]
if first_panel_export_cols:
    ed_df = ed_df.merge(
        first_panel[first_panel_export_cols].rename(
            columns={
                "panel_time": "first_gas_time",

```

```

        "pco2": "first_pco2",
        "ph": "first_ph",
        "hco3": "first_hco3",
        "lactate": "first_lactate",
    }
),
on="ed_stay_id",
how="left",
)
else:
    ed_df["first_gas_time"] = pd.NaT
    ed_df["first_pco2"] = pd.NA
    ed_df["first_ph"] = pd.NA
    ed_df["first_hco3"] = pd.NA
    ed_df["first_lactate"] = pd.NA

# Prioritize LAB blood-gas bicarbonate aligned to pCO2 anchor panel, then
↪ fallback
# to nearest LAB bicarbonate around first_gas_time when same-panel value is
↪ missing.
if "first_hco3_source" not in ed_df.columns:
    ed_df["first_hco3_source"] = "missing"

panel_hco3_present = pd.to_numeric(ed_df.get("first_hco3"),
↪ errors="coerce").notna()
ed_df.loc[panel_hco3_present, "first_hco3_source"] =
↪ "lab_panel_same_specimen"

lab_hco3_long = pd.DataFrame(columns=["ed_stay_id", "charttime", "valuenum"])
if hco3_ids and "first_gas_time" in ed_df.columns:
    lab_hco3_long = labs.loc[
        labs["itemid"].isin(hco3_ids),
        ["ed_stay_id", "charttime", "valuenum"],
    ].copy()
    if not lab_hco3_long.empty:
        lab_hco3_long["valuenum"] = pd.to_numeric(lab_hco3_long["valuenum"],
↪ errors="coerce")
        lab_hco3_long =
↪ lab_hco3_long.loc[lab_hco3_long["valuenum"].notna()].copy()
        anchor_missing = ed_df.loc[
            pd.to_numeric(ed_df["first_hco3"], errors="coerce").isna()
            & pd.to_datetime(ed_df["first_gas_time"],
↪ errors="coerce").notna(),

```

```

        ["ed_stay_id", "first_gas_time"],
    ].copy()
    if not anchor_missing.empty and not lab_hco3_long.empty:
        fallback_candidates = anchor_missing.merge(
            lab_hco3_long,
            on="ed_stay_id",
            how="left",
        )
        fallback_candidates["first_gas_time"] = pd.to_datetime(
            fallback_candidates["first_gas_time"], errors="coerce"
        )
        fallback_candidates["charttime"] = pd.to_datetime(
            fallback_candidates["charttime"], errors="coerce"
        )
        fallback_candidates["dt_abs_hours"] = (
            (fallback_candidates["charttime"] -
↪ fallback_candidates["first_gas_time"])
            .dt.total_seconds()
            .abs()
            / 3600.0
        )
        fallback_candidates = fallback_candidates.loc[
            fallback_candidates["dt_abs_hours"].notna()
        ].copy()
        if not fallback_candidates.empty:
            nearest_hco3 = (
                fallback_candidates.sort_values(
                    ["ed_stay_id", "dt_abs_hours", "charttime"],
                    kind="stable",
                )
                .groupby("ed_stay_id", as_index=False)
                .first()
                .rename(columns={"valuenum": "first_hco3_fallback_lab"})
            )
            ed_df = ed_df.merge(
                nearest_hco3[["ed_stay_id", "first_hco3_fallback_lab"]],
                on="ed_stay_id",
                how="left",
            )
            fill_mask = (
                pd.to_numeric(ed_df["first_hco3"],
↪ errors="coerce").isna()

```

```

        & pd.to_numeric(ed_df["first_hco3_fallback_lab"],
        ↪ errors="coerce").notna()
    )
    ed_df.loc[fill_mask, "first_hco3"] = ed_df.loc[
        fill_mask, "first_hco3_fallback_lab"
    ]
    ed_df.loc[fill_mask, "first_hco3_source"] =
    ↪ "lab_nearest_to_anchor"
    ed_df = ed_df.drop(columns=["first_hco3_fallback_lab"])

if hco3_ids and not lab_hco3_long.empty:
    remaining_missing = pd.to_numeric(ed_df["first_hco3"],
    ↪ errors="coerce").isna()
    if remaining_missing.any():
        first_hco3_in_window = (
            lab_hco3_long.sort_values(["ed_stay_id", "charttime"],
            ↪ kind="stable")
            .groupby("ed_stay_id", as_index=False)
            .first()
            .rename(columns={"valuenum": "first_hco3_fallback_window"})
        )
        ed_df = ed_df.merge(
            first_hco3_in_window[["ed_stay_id",
            ↪ "first_hco3_fallback_window"]],
            on="ed_stay_id",
            how="left",
        )
        window_fill_mask = (
            pd.to_numeric(ed_df["first_hco3"], errors="coerce").isna()
            & pd.to_numeric(ed_df["first_hco3_fallback_window"],
            ↪ errors="coerce").notna()
        )
        ed_df.loc[window_fill_mask, "first_hco3"] = ed_df.loc[
            window_fill_mask, "first_hco3_fallback_window"
        ]
        ed_df.loc[window_fill_mask, "first_hco3_source"] = "lab_first_0_24h"
        ed_df = ed_df.drop(columns=["first_hco3_fallback_window"])

ed_df["first_hco3_source"] = np.where(
    pd.to_numeric(ed_df["first_hco3"], errors="coerce").notna(),
    ed_df["first_hco3_source"].astype("string").fillna("missing"),
    "missing",
)

```

```

ed_df["first_gas_anchor_has_pco2"] = ed_df["first_gas_time"].notna()
ed_df["first_gas_anchor_source_validated"] = False

ed_df = ed_df.merge(agg06, on="ed_stay_id", how="left")
ed_df = ed_df.merge(agg24, on="ed_stay_id", how="left")

# Purpose: Reconstruct gas panels and normalize units before ED-stay level
↳ aggregation.

# ABG vs VBG classification (specimen-type rows only for core cohort
↳ behavior)
panel, gas_source_audit = assign_panel_source_from_specimen(
    panel,
    labs,
    specimen_source_itemids=itemid_sets.get("gas_specimen", []),
)

# Keep heuristic route diagnostics for QA transparency only (non-core
↳ behavior).
heuristic_gas_source_audit: dict[str, Any] | None = None
heuristic_mode = os.getenv("COHORT_GAS_SOURCE_INFERENCE_MODE",
↳ "metadata_only").strip().lower()
try:
    _, heuristic_gas_source_audit = infer_panel_gas_source_metadata(
        panel_df=panel.copy(),
        labs_df=labs,
        labitems_df=labitems,
        specimen_source_itemids=itemid_sets.get("gas_specimen", []),
        pco2_itemids=itemid_sets.get("gas_pco2", []),
        mode=heuristic_mode,
    )
except Exception as exc:
    heuristic_gas_source_audit = {
        "status": "error",
        "message": f"heuristic_diagnostics_failed: {exc}",
    }
gas_source_audit["heuristic_reference"] = heuristic_gas_source_audit

fail_on_all_other_source = os.getenv("COHORT_FAIL_ON_ALL_OTHER_SOURCE",
↳ "1").strip() == "1"
if gas_source_audit["all_other_or_unknown"]:
    if fail_on_all_other_source:

```

```

        assert_gas_source_coverage(
            gas_source_audit,
            fail_on_all_other_source=fail_on_all_other_source,
        )
    else:
        print(
            "WARNING: Gas source attribution collapsed to other/unknown; "
            "continuing because COHORT_FAIL_ON_ALL_OTHER_SOURCE=0."
        )

warn_other_rate_threshold = float(os.getenv("COHORT_WARN_OTHER_RATE",
    ↪ "0.50"))
fail_other_rate_raw = os.getenv("COHORT_FAIL_OTHER_RATE", "").strip()
fail_other_rate_threshold = float(fail_other_rate_raw) if fail_other_rate_raw
    ↪ else None
other_source_rate = float(gas_source_audit.get("source_rates",
    ↪ {}).get("unknown", 0.0))

if fail_other_rate_threshold is not None and other_source_rate >=
    ↪ fail_other_rate_threshold:
    raise ValueError(
        "gas_source_other_rate exceeded configured fail threshold "
        f"{fail_other_rate_threshold:.3f}: {other_source_rate:.4f}."
    )
if other_source_rate >= warn_other_rate_threshold:
    print(
        "WARNING: gas_source_other_rate exceeded warning threshold "
        f"{warn_other_rate_threshold:.3f}: {other_source_rate:.4f}."
    )

print("Gas source rates:", gas_source_audit.get("source_rates", {}))
print("Gas source inference tiers:", gas_source_audit.get("tier_rates", {}))
print("Unresolved specimen IDs:",
    ↪ gas_source_audit.get("unresolved_specimen_id_count", 0))

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
out_date = datetime.now().strftime("%Y-%m-%d")
gas_source_audit_path = prior_runs_dir / f"{out_date} gas_source_audit.json"

reduction_target = float(os.getenv("COHORT_OTHER_RELATIVE_REDUCTION_MIN",
    ↪ "0.10"))
baseline_other_rate = None

```

```

baseline_audit_path = None
baseline_mode_mismatch_path = None
current_source_mode = str(gas_source_audit.get("mode", "")).strip().lower()
for candidate in sorted(prior_runs_dir.glob("* gas_source_audit.json"),
↪ reverse=True):
    if candidate.resolve() == gas_source_audit_path.resolve():
        continue
    try:
        baseline_payload = json.loads(candidate.read_text())
        baseline_mode = str(baseline_payload.get("mode", "")).strip().lower()
        if current_source_mode and baseline_mode and baseline_mode !=
↪ current_source_mode:
            baseline_mode_mismatch_path = candidate
            continue
        baseline_rates = baseline_payload.get("source_rates", {})
        baseline_other_rate = float(
            baseline_rates.get(
                "unknown",
                baseline_rates.get("other", float("nan")),
            )
        )
        if math.isnan(baseline_other_rate):
            baseline_other_rate = None
            continue
        baseline_audit_path = candidate
        break
    except Exception:
        continue

if baseline_other_rate is None or baseline_other_rate <= 0:
    gas_source_audit["baseline_other_rate"] = None
    gas_source_audit["baseline_path"] = str(baseline_audit_path) if
↪ baseline_audit_path else None
    gas_source_audit["baseline_mode_mismatch_path"] = (
        str(baseline_mode_mismatch_path) if baseline_mode_mismatch_path else
↪ None
    )
    gas_source_audit["relative_reduction_vs_baseline"] = None
    print(
        "WARNING: Unable to evaluate gas_source_other_rate relative reduction
↪ "
        "because no valid baseline gas_source_audit was found for the current
↪ source mode."
    )

```

```

    )
else:
    relative_reduction = (baseline_other_rate - other_source_rate) /
    ↪ baseline_other_rate
    gas_source_audit["baseline_other_rate"] = baseline_other_rate
    gas_source_audit["baseline_path"] = str(baseline_audit_path)
    gas_source_audit["baseline_mode_mismatch_path"] = (
        str(baseline_mode_mismatch_path) if baseline_mode_mismatch_path else
        ↪ None
    )
    gas_source_audit["relative_reduction_vs_baseline"] = relative_reduction
    gas_source_audit["relative_reduction_target"] = reduction_target
    print(
        "Gas source OTHER relative reduction vs baseline: "
        f"{relative_reduction:.4f} (target={reduction_target:.4f})"
    )
    if relative_reduction < reduction_target:
        message = (
            "gas_source_other_rate relative reduction target not met: "
            f"baseline={baseline_other_rate:.4f},
            ↪ current={other_source_rate:.4f}, "
            f"reduction={relative_reduction:.4f},
            ↪ target={reduction_target:.4f}."
        )
        contract_mode = os.getenv("PIPELINE_CONTRACT_MODE",
        ↪ "fail").strip().lower()
        if contract_mode == "fail":
            raise ValueError(message)
        print(f"WARNING: {message}")

gas_source_audit_path.write_text(json.dumps(gas_source_audit, indent=2))
print("Wrote:", gas_source_audit_path)

panel["pco2_source_branch"] = "LAB"
panel_anchor = panel.loc[pd.to_numeric(panel["pco2"],
    ↪ errors="coerce").notna()].copy()
if panel_anchor.empty:
    first_anchor = ed_df[["ed_stay_id"]].drop_duplicates().copy()
    first_anchor["first_gas_time"] = pd.NaT
    first_anchor["first_pco2"] = pd.NA
    first_anchor["first_ph"] = pd.NA
    first_anchor["first_hco3_anchor"] = pd.NA
    first_anchor["first_lactate"] = pd.NA

```



```

first_anchor["first_gas_specimen_type"] = pd.NA
first_anchor["first_gas_specimen_present"] = False
first_anchor["first_gas_pco2_itemid"] = pd.NA
first_anchor["first_gas_pco2_fluid"] = pd.NA
first_anchor["co2_other_is_blood_asserted"] = False
first_anchor["first_gas_anchor_has_pco2"] = False
first_anchor["first_gas_anchor_source_validated"] = False
else:
    first_anchor = (
        panel_anchor.sort_values(["ed_stay_id", "panel_time"])
        .groupby("ed_stay_id", as_index=False)
        .first()
    )
    first_anchor = first_anchor.rename(
        columns={
            "panel_time": "first_gas_time",
            "pco2": "first_pco2",
            "ph": "first_ph",
            "hco3": "first_hco3_anchor",
            "lactate": "first_lactate",
            "source_hint_specimen_text": "first_gas_specimen_type",
            "pco2_itemid": "first_gas_pco2_itemid",
            "pco2_fluid": "first_gas_pco2_fluid",
        }
    )
    first_anchor["first_gas_specimen_present"] = (
        first_anchor["first_gas_specimen_type"]
        .astype("string")
        .fillna("")
        .str.strip()
        .ne("")
    )
    first_anchor["co2_other_is_blood_asserted"] = (
        first_anchor["source"].astype("string").str.lower().eq("unknown")
        & first_anchor["first_gas_pco2_fluid"]
        .astype("string")
        .str.strip()
        .str.lower()
        .isin(LAB_BLOOD_FLUID_TERMS)
    )
    first_anchor["first_gas_anchor_has_pco2"] = True
    first_anchor["first_gas_anchor_source_validated"] =
↪ first_anchor["source"].isin(

```

```

        {"arterial", "venous", "unknown"}
    )

anchor_merge_cols = [
    "ed_stay_id",
    "first_gas_time",
    "first_pco2",
    "first_ph",
    "first_hco3_anchor",
    "first_lactate",
    "first_gas_specimen_type",
    "first_gas_specimen_present",
    "first_gas_pco2_itemid",
    "first_gas_pco2_fluid",
    "co2_other_is_blood_asserted",
    "first_gas_anchor_has_pco2",
    "first_gas_anchor_source_validated",
]
drop_existing_anchor_cols = [
    col
    for col in anchor_merge_cols
    if col in ed_df.columns and col not in {"ed_stay_id",
        ↪ "first_hco3_anchor"}
]
if drop_existing_anchor_cols:
    ed_df = ed_df.drop(columns=drop_existing_anchor_cols)
ed_df = ed_df.merge(first_anchor[anchor_merge_cols], on="ed_stay_id",
    ↪ how="left")
if "first_hco3_source" not in ed_df.columns:
    ed_df["first_hco3_source"] = "missing"
if "first_hco3" not in ed_df.columns:
    ed_df["first_hco3"] = pd.NA
hco3_anchor_numeric = pd.to_numeric(ed_df.get("first_hco3_anchor"),
    ↪ errors="coerce")
hco3_existing_numeric = pd.to_numeric(ed_df.get("first_hco3"),
    ↪ errors="coerce")
hco3_fill_from_anchor = hco3_existing_numeric.isna() &
    ↪ hco3_anchor_numeric.notna()
ed_df.loc[hco3_fill_from_anchor, "first_hco3"] =
    ↪ hco3_anchor_numeric.loc[hco3_fill_from_anchor]
ed_df.loc[hco3_fill_from_anchor, "first_hco3_source"] =
    ↪ "lab_panel_same_specimen"
ed_df["first_hco3_source"] = np.where(

```

```

    pd.to_numeric(ed_df["first_hco3"], errors="coerce").notna(),
    ed_df["first_hco3_source"].astype("string").fillna("missing"),
    "missing",
)
if "first_hco3_anchor" in ed_df.columns:
    ed_df = ed_df.drop(columns=["first_hco3_anchor"])
ed_df["first_gas_anchor_has_pco2"] = (
    ed_df["first_gas_anchor_has_pco2"].fillna(False).astype(bool)
)
ed_df["first_gas_anchor_source_validated"] = (
    ed_df["first_gas_anchor_source_validated"].fillna(False).astype(bool)
)
ed_df["first_gas_specimen_present"] = (
    ed_df["first_gas_specimen_present"].fillna(False).astype(bool)
)
ed_df["co2_other_is_blood_asserted"] = (
    ed_df["co2_other_is_blood_asserted"].fillna(False).astype(bool)
)

# flags
panel["flag_abg_hypercapnia"] = ((panel["source"]=="arterial") &
    ↪ (panel["pco2"]>=45)).astype(int)
panel["flag_vbg_hypercapnia"] = ((panel["source"]=="venous") &
    ↪ (panel["pco2"]>=50)).astype(int)
panel["flag_unknown_hypercapnia"] = ((panel["source"]=="unknown") &
    ↪ (panel["pco2"]>=50)).astype(int)
panel["flag_other_hypercapnia"] =
    ↪ panel["flag_unknown_hypercapnia"].astype(int)
panel["flag_any_gas_hypercapnia"] = (
    (panel["flag_abg_hypercapnia"] == 1)
    | (panel["flag_vbg_hypercapnia"] == 1)
    | (panel["flag_unknown_hypercapnia"] == 1)
).astype(int)

# per-stay unknown source rate
if len(panel) > 0:
    unk_rate = (
        panel.assign(_unk=(panel["source"].fillna("unknown") == "unknown"))
        .groupby("ed_stay_id", as_index=False)[ "_unk"].mean()
        .rename(columns={"_unk": "gas_source_unknown_rate"})
    )
else:
    unk_rate = panel[["ed_stay_id"]].drop_duplicates()

```

```

unk_rate["gas_source_unknown_rate"] = 1.0

if len(panel) > 0:
    panel_for_diag = panel.copy()
    panel_for_diag["source_inference_tier"] = (
        panel_for_diag["source_inference_tier"]
        .astype("string")
        .fillna("fallback_unknown")
    )
    panel_for_diag["source_hint_conflict"] = (
        panel_for_diag["source_hint_conflict"]
        .fillna(False)
        .astype(bool)
    )
    panel_for_diag["source"] = (
        ↪ panel_for_diag["source"].astype("string").fillna("unknown").str.lower()
    )
    panel_for_diag["source_resolved"] = panel_for_diag["source"].isin(
        {"arterial", "venous"}
    )
    panel_for_diag["tier_specimen_text"] =
    ↪ panel_for_diag["source_inference_tier"].eq(
        "specimen_text"
    )
    panel_for_diag["tier_label_fluid"] =
    ↪ panel_for_diag["source_inference_tier"].eq(
        "label_fluid"
    )
    panel_for_diag["tier_cluster_inheritance"] = panel_for_diag[
        "source_inference_tier"
    ].eq("cluster_inheritance")
    panel_for_diag["tier_panel_cooccurrence"] = panel_for_diag[
        "source_inference_tier"
    ].eq("panel_cooccurrence")
    panel_for_diag["tier_fallback_other"] =
    ↪ panel_for_diag["source_inference_tier"].eq(
        "fallback_unknown"
    )

def _tier_mode(series: pd.Series) -> str:
    mode_values = series.mode(dropna=True)
    if mode_values.empty:

```

```

        return "fallback_unknown"
    return str(mode_values.iloc[0])

source_diag = (
    panel_for_diag.groupby("ed_stay_id", as_index=False)
    .agg(
        gas_source_inference_primary_tier=("source_inference_tier",
↪ _tier_mode),
        gas_source_hint_conflict_rate=("source_hint_conflict", "mean"),
        gas_source_resolved_rate=("source_resolved", "mean"),
        gas_source_tier_specimen_text_rate=("tier_specimen_text",
↪ "mean"),
        gas_source_tier_label_fluid_rate=("tier_label_fluid", "mean"),

↪ gas_source_tier_cluster_inheritance_rate=("tier_cluster_inheritance",
↪ "mean"),

↪ gas_source_tier_panel_cooccurrence_rate=("tier_panel_cooccurrence",
↪ "mean"),
        gas_source_tier_fallback_other_rate=("tier_fallback_other",
↪ "mean"),
    )
    .copy()
)
else:
    source_diag = ed_df[["ed_stay_id"]].drop_duplicates().copy()
    source_diag["gas_source_inference_primary_tier"] = "fallback_unknown"
    source_diag["gas_source_hint_conflict_rate"] = 0.0
    source_diag["gas_source_resolved_rate"] = 0.0
    source_diag["gas_source_tier_specimen_text_rate"] = 0.0
    source_diag["gas_source_tier_label_fluid_rate"] = 0.0
    source_diag["gas_source_tier_cluster_inheritance_rate"] = 0.0
    source_diag["gas_source_tier_panel_cooccurrence_rate"] = 0.0
    source_diag["gas_source_tier_fallback_other_rate"] = 1.0

# collapse to ED stay flags
flags = panel.groupby("ed_stay_id", as_index=False).agg(
    flag_abg_hypercapnia=("flag_abg_hypercapnia", "max"),
    flag_vbg_hypercapnia=("flag_vbg_hypercapnia", "max"),
    flag_unknown_hypercapnia=("flag_unknown_hypercapnia", "max"),
    flag_other_hypercapnia=("flag_other_hypercapnia", "max"),
    flag_any_gas_hypercapnia=("flag_any_gas_hypercapnia", "max"),
)

```

```

ed_df = ed_df.merge(flags, on="ed_stay_id", how="left")
ed_df = ed_df.merge(unk_rate, on="ed_stay_id", how="left")
ed_df = ed_df.merge(source_diag, on="ed_stay_id", how="left")
ed_df["gas_source_unknown_rate"] =
    ↪ ed_df["gas_source_unknown_rate"].fillna(1.0)
ed_df["gas_source_other_rate"] = ed_df["gas_source_unknown_rate"]
ed_df["flag_other_hypercapnia"] = (
    pd.to_numeric(ed_df.get("flag_unknown_hypercapnia"), errors="coerce")
    .fillna(pd.to_numeric(ed_df.get("flag_other_hypercapnia"),
    ↪ errors="coerce"))
    .fillna(0)
    .astype("Int64")
)

```

WARNING: gas\_source\_other\_rate exceeded warning threshold 0.500: 0.6626.

Gas source rates: {'unknown': 0.6626423339262455, 'arterial': 0.20497485959187103, 'venous':

Gas source inference tiers: {'fallback\_unknown': 0.6626423339262455, 'specimen\_text': 0.3373

Unresolved specimen IDs: 47279

Gas source OTHER relative reduction vs baseline: 0.1407 (target=0.1000)

Wrote: /Users/bloque/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project

CC-NLP/MIMIC tabular data/prior runs/2026-02-20 gas\_source\_audit.json

### 3.0.7 Phase 5C — ICU POC blood gases (chartevents, optional)

**Rationale:** Capture ICU point-of-care gases if central lab labevents miss early measurements.

```

# Purpose: Define a reusable BigQuery execution helper so SQL calls are
    ↪ consistent across notebook stages.

```

```

# Discover ICU POC itemids from d_items

```

```

SQL["ditems_sql"] = f"""
SELECT itemid, label, category
FROM `{PHYS}`.{ICU}.d_items`
"""

```

```

ditems = run_sql_bq(sql("ditems_sql"))

```

```

icu_patterns = {
    "pco2": re.compile(r"(?:^[^a-z])p(?:a)?\s*co(?:2| )(?:[^\s]|$)", re.I),
    "ph": re.compile(r"\bph\b", re.I),
}

```

```

    "hco3": re.compile(r"hco3|bicarbonate", re.I),
    "lactate": re.compile(r"lactate", re.I),
    "specimen": re.compile(r"specimen|source|type", re.I),
}

icu_cat = re.compile(r"blood\s*gas|blood gas|resp|arterial|venous", re.I)
icu_pco2_exclusion = re.compile(ICU_LABEL_DENY_REGEX, re.I)

icu_matches = {}
manifest_icu_itemids = {
    "pco2": set(ICU_PCO2_ITEMIDS),
    "ph": set(ICU_PH_ITEMIDS),
    "hco3": set(ICU_HCO3_ITEMIDS),
    "specimen": set(ICU_SPECIMEN_TYPE_ITEMIDS),
}
for name, pat in icu_patterns.items():
    manifest_ids = manifest_icu_itemids.get(name, set())
    if manifest_ids:
        dfm = ditems.loc[ditems["itemid"].isin(manifest_ids)].copy()
    else:
        dfm = pd.DataFrame(columns=["itemid", "label", "category"])

    allow_fallback_for_analyte = ICU_ALLOW_PATTERN_FALLBACK and name !=
    ↪ "hco3"
    if dfm.empty and allow_fallback_for_analyte:
        dfm = ditems[ditems["label"].str.contains(pat, na=False)].copy()
        if name in {"pco2", "ph", "specimen"}:
            dfm = dfm[dfm["category"].str.contains(icu_cat, na=False)]

    if name == "pco2" and len(dfm) > 0:
        dfm = dfm[~dfm["label"].str.contains(icu_pco2_exclusion, na=False)]
    icu_matches[name] = dfm[["itemid", "label", "category"]]

icu_itemid_map_frames = [
    frame.assign(analyte=name)
    for name, frame in icu_matches.items()
    if not frame.empty
]
if not icu_itemid_map_frames:
    icu_itemid_map = pd.DataFrame(columns=["analyte", "itemid", "label",
    ↪ "category"])
else:
    icu_itemid_map = pd.concat(icu_itemid_map_frames, ignore_index=True)

```

```

icu_itemid_map = (
    icu_itemid_map[["analyte", "itemid", "label", "category"]]
    .drop_duplicates()
    .sort_values(["analyte", "itemid"], kind="stable")
    .reset_index(drop=True)
)

icu_itemids = sorted({int(i) for dfm in icu_matches.values() for i in
    ↪ dfm["itemid"].tolist()})
print("ICU POC candidate itemids:", len(icu_itemids))
if not icu_itemid_map.empty:
    print("ICU POC itemid map preview:")
    print(icu_itemid_map.head(20).to_string(index=False))

audit_date = pd.Timestamp.now().strftime("%Y-%m-%d")
prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
icu_itemid_map_path = prior_runs_dir / f"{audit_date} icu_poc_itemid_map.csv"
icu_itemid_map.to_csv(icu_itemid_map_path, index=False)
print("Wrote:", icu_itemid_map_path)

manifest_icu_audit_rows: list[dict[str, Any]] = []
for analyte_name, itemids in manifest_icu_itemids.items():
    if not itemids:
        continue
    frame = ditems.loc[ditems["itemid"].isin(itemids), ["itemid", "label",
    ↪ "category"]].copy()
    present_ids = set(frame["itemid"].astype(int).tolist()) if not
    ↪ frame.empty else set()
    for itemid in sorted(itemids):
        row = {
            "source_branch": "ICU",
            "analyte": analyte_name,
            "itemid": int(itemid),
            "label": None,
            "category": None,
            "fluid": None,
            "cohort_row_count": 0,
            "manifest_version": BLOOD_GAS_MANIFEST["version"],
            "present_in_dictionary": int(itemid in present_ids),
        }
        if itemid in present_ids:
            record = frame.loc[frame["itemid"].eq(itemid)].iloc[0]

```



```

        row["label"] = record.get("label")
        row["category"] = record.get("category")
        manifest_icu_audit_rows.append(row)

if "blood_gas_itemid_manifest_audit" in globals() and not
↳ blood_gas_itemid_manifest_audit.empty:
    manifest_audit_combined = pd.concat(
        [blood_gas_itemid_manifest_audit,
↳ pd.DataFrame(manifest_icu_audit_rows)],
        ignore_index=True,
        sort=False,
    )
else:
    manifest_audit_combined = pd.DataFrame(manifest_icu_audit_rows)

blood_gas_itemid_manifest_audit = manifest_audit_combined
manifest_missing = blood_gas_itemid_manifest_audit.loc[
    pd.to_numeric(
        blood_gas_itemid_manifest_audit.get("present_in_dictionary"),
↳ errors="coerce"
    ).fillna(1).astype(int).eq(0)
].copy()
if not manifest_missing.empty:
    missing_repr = ", ".join(
        f"{row.source_branch}:{row.analyte}:{int(row.itemid)}"
        for row in manifest_missing.itertuples(index=False)
    )
    raise ValueError(
        "Blood-gas manifest references itemids not present in runtime
↳ dictionaries: "
        f"{missing_repr}"
    )
blood_gas_itemid_manifest_audit_path = prior_runs_dir / f"{audit_date}"
↳ blood_gas_itemid_manifest_audit.csv"
blood_gas_itemid_manifest_audit.to_csv(blood_gas_itemid_manifest_audit_path,
↳ index=False)
print("Wrote:", blood_gas_itemid_manifest_audit_path)

```

ICU POC candidate itemids: 2

ICU POC itemid map preview:

analyte	itemid	label	category
pco2	220235	Arterial CO2 Pressure	Labs
pco2	226062	Venous CO2 Pressure	Labs

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/CC-NLP/MIMIC tabular data/prior runs/2026-02-20 icu\_poc\_itemid\_map.csv  
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/CC-NLP/MIMIC tabular data/prior runs/2026-02-20 blood\_gas\_itemid\_manifest\_audit.csv

```
# Purpose: Define a reusable BigQuery execution helper so SQL calls are
↳ consistent across notebook stages.
```

```
# Extract ICU chartevents within ED 0-24h window for cohort ICU stays
```

```
if len(icu_itemids) == 0:
    icu_poc_long = pd.DataFrame()
    print("No ICU POC itemids found.")
else:
    # ensure icu stay ids available
    if 'icu' not in globals():
        raise NameError("ICU stays table 'icu' not found; run ICU phase
↳ first.")

    icu_stays = icu[["icu_stay_id", "hadm_id", "intime"]].copy()
    icu_stays = icu_stays.dropna(subset=["icu_stay_id"])

    icu_poc_sql = f"""
WITH icu_stays AS (
    SELECT stay_id AS icu_stay_id, hadm_id
    FROM `{PHYS}`.{ICU}.icustays`
    WHERE stay_id IN UNNEST(@icu_stay_ids)
),
eds AS (
    SELECT stay_id AS ed_stay_id, hadm_id, intime AS ed_intime
    FROM `{PHYS}`.{ED}.edstays`
    WHERE hadm_id IN (SELECT hadm_id FROM icu_stays)
)
SELECT
    s.icu_stay_id,
    e.ed_stay_id,
    e.hadm_id,
    ce.charttime,
    ce.itemid,
    ce.valuenum,
    ce.valueuom
FROM `{PHYS}`.{ICU}.chartevents` ce
JOIN icu_stays s ON s.icu_stay_id = ce.stay_id
```

```

JOIN eds e ON e.hadm_id = s.hadm_id
WHERE ce.itemid IN UNNEST(@itemids)
      AND ce.charttime BETWEEN e.ed_intime AND TIMESTAMP_ADD(e.ed_intime,
↪ INTERVAL 24 HOUR)
      """"

icu_poc_long = run_sql_bq(icu_poc_sql, {"icu_stay_ids":
↪ icu_stays["icu_stay_id"].astype(int).tolist(), "itemids": icu_itemids})
print("ICU POC long rows:", len(icu_poc_long))

# Build panels by 5-minute bins per ICU stay
if len(icu_poc_long) > 0:
    icu_poc_long["time_bin"] = icu_poc_long["charttime"].dt.floor("5min")

    def pick_from_ids(df, ids, name):
        tmp = df.loc[df["itemid"].isin(ids),
↪ ["icu_stay_id", "time_bin", "valuenum"]]
        tmp = tmp.rename(columns={"valuenum": name})
        return tmp.groupby(["icu_stay_id", "time_bin"],
↪ as_index=False).first()

    pco2_ids = set(icu_matches.get("pco2", pd.DataFrame()).get("itemid",
↪ []).tolist())
    ph_ids = set(icu_matches.get("ph", pd.DataFrame()).get("itemid",
↪ []).tolist())
    hco3_ids = set(icu_matches.get("hco3", pd.DataFrame()).get("itemid",
↪ []).tolist())
    lact_ids = set(icu_matches.get("lactate", pd.DataFrame()).get("itemid",
↪ []).tolist())

    panel_poc = (
        icu_poc_long.groupby(["icu_stay_id", "time_bin"], as_index=False)
        .agg(panel_time=("charttime", "min"))
    )
    if pco2_ids:
        panel_poc = panel_poc.merge(pick_from_ids(icu_poc_long, pco2_ids,
↪ "pco2"), on=["icu_stay_id", "time_bin"], how="left")
    if ph_ids:
        panel_poc = panel_poc.merge(pick_from_ids(icu_poc_long, ph_ids,
↪ "ph"), on=["icu_stay_id", "time_bin"], how="left")
    if hco3_ids:
        panel_poc = panel_poc.merge(pick_from_ids(icu_poc_long, hco3_ids,
↪ "hco3"), on=["icu_stay_id", "time_bin"], how="left")

```

```

if lact_ids:
    panel_poc = panel_poc.merge(pick_from_ids(icu_poc_long, lact_ids,
↪ "lactate"), on=["icu_stay_id", "time_bin"], how="left")

# map to ED stay via hadm_id
panel_poc = panel_poc.merge(icu[["icu_stay_id", "hadm_id"]],
↪ on="icu_stay_id", how="left")
panel_poc = panel_poc.merge(ed_df[["ed_stay_id", "hadm_id", "ed_intime"]],
↪ on="hadm_id", how="left")
panel_poc["dt_hours"] = (panel_poc["panel_time"] -
↪ panel_poc["ed_intime"]).dt.total_seconds() / 3600.0

icu_itemid_usage = (
    icu_poc_long.groupby("itemid", as_index=False)
        .size()
        .rename(columns={"size": "row_count"})
)
icu_itemid_usage["itemid"] = pd.to_numeric(
    icu_itemid_usage["itemid"], errors="coerce"
).astype("Int64")
if not icu_itemid_map.empty:
    icu_itemid_usage = icu_itemid_usage.merge(
        icu_itemid_map[["analyte", "itemid", "label",
↪ "category"]].drop_duplicates(),
        on="itemid",
        how="left",
    )
icu_itemid_usage = icu_itemid_usage.sort_values(
    ["row_count", "itemid"], ascending=[False, True], kind="stable"
).reset_index(drop=True)
icu_itemid_usage_path = prior_runs_dir / f"{audit_date}"
↪ icu_poc_itemid_usage.csv"
icu_itemid_usage.to_csv(icu_itemid_usage_path, index=False)
print("Wrote:", icu_itemid_usage_path)
if "blood_gas_itemid_manifest_audit" in globals() and not
↪ blood_gas_itemid_manifest_audit.empty:
    usage_merge = icu_itemid_usage[["itemid", "row_count"]].copy()
    usage_merge["itemid"] = pd.to_numeric(usage_merge["itemid"],
↪ errors="coerce").astype("Int64")
    manifest_audit = blood_gas_itemid_manifest_audit.copy()
    manifest_audit["itemid"] = pd.to_numeric(manifest_audit["itemid"],
↪ errors="coerce").astype("Int64")
    manifest_audit = manifest_audit.merge(

```

```

        usage_merge.rename(columns={"row_count": "icu_usage_row_count"}),
        on="itemid",
        how="left",
    )
    if "cohort_row_count" in manifest_audit.columns:
        manifest_audit["cohort_row_count"] =
↪ manifest_audit["cohort_row_count"].fillna(0).astype(int)
        manifest_audit["cohort_row_count"] = (
            manifest_audit["cohort_row_count"] +
↪ manifest_audit["icu_usage_row_count"].fillna(0).astype(int)
        )
        manifest_audit = manifest_audit.drop(columns=["icu_usage_row_count"])
        blood_gas_itemid_manifest_audit = manifest_audit

↪ blood_gas_itemid_manifest_audit.to_csv(blood_gas_itemid_manifest_audit_path,
↪ index=False)
        print("Updated:", blood_gas_itemid_manifest_audit_path)
    else:
        panel_poc = pd.DataFrame()

# Ensure expected panel_poc columns exist even if analyte is absent
for col in ["pco2", "ph", "hco3", "lactate"]:
    if col not in panel_poc.columns:
        panel_poc[col] = pd.NA

poc_inclusion_enabled = False
poc_inclusion_reason = "disabled_manifest_or_qc_fail"
pco2_itemid_qc_rows: list[dict[str, Any]] = []

if len(icu_poc_long) > 0 and pco2_ids:
    pco2_long =
↪ icu_poc_long.loc[icu_poc_long["itemid"].isin(pco2_ids)].copy()
    if not pco2_long.empty:
        pco2_long["label"] = pco2_long["itemid"].map(
            icu_itemid_map.set_index("itemid")["label"] if not
↪ icu_itemid_map.empty else {}
        )
        pco2_long["category"] = pco2_long["itemid"].map(
            icu_itemid_map.set_index("itemid")["category"] if not
↪ icu_itemid_map.empty else {}
        )
        pco2_long["valuenum"] = pd.to_numeric(pco2_long["valuenum"],
↪ errors="coerce")

```

```

pco2_long["valueuom_norm"] = (
↪ pco2_long["valueuom"].astype("string").fillna("").str.strip().str.lower()
    )
    is_kpa = pco2_long["valueuom_norm"].eq("kpa")
    pco2_long.loc[is_kpa, "valuenum"] = pco2_long.loc[is_kpa, "valuenum"]
↪ * 7.50062
    pco2_long["pco2_mmhg"] = pco2_long["valuenum"]

    for itemid_value, group in pco2_long.groupby("itemid", dropna=False):
        label_text =
↪ str(group["label"].dropna().astype(str).head(1).tolist()[0]) if
↪ group["label"].notna().any() else ""
        category_text =
↪ str(group["category"].dropna().astype(str).head(1).tolist()[0]) if
↪ group["category"].notna().any() else ""
        label_text_l = label_text.lower()
        if int(itemid_value) in set(ICU_PC02_ABG_ITEMIDS):
            sample_type_hint = "ABG"
        elif int(itemid_value) in set(ICU_PC02_VBG_ITEMIDS):
            sample_type_hint = "VBG"
        else:
            sample_type_hint = "UNKNOWN"
        value_units =
↪ sorted(set(group["valueuom_norm"].dropna().astype(str).tolist()))
        unit_ok = set(value_units).issubset(ICU_PC02_QC_ALLOWED_UOMS)
        include_ok = any(token in label_text_l for token in
↪ ICU_PC02_QC_INCLUDE_TOKENS)
        exclude_hit = any(token in label_text_l for token in
↪ ICU_PC02_QC_EXCLUDE_TOKENS)
        numeric = pd.to_numeric(group["pco2_mmhg"],
↪ errors="coerce").dropna()
        median_value = float(numeric.median()) if len(numeric) else
↪ np.nan
        median_ok = bool(
            len(numeric)
            and ICU_PC02_QC_MEDIAN_MIN <= median_value <=
↪ ICU_PC02_QC_MEDIAN_MAX
        )
        contamination_flag = bool((not include_ok) or exclude_hit or
↪ (not unit_ok))

        pco2_itemid_qc_rows.append(

```

```

        {
            "itemid": int(itemid_value),
            "label": label_text,
            "category": category_text,
            "source_branch": "POC",
            "sample_type_hint": sample_type_hint,
            "row_count": int(len(group)),
            "non_null_count": int(len(numeric)),
            "units_observed": ";".join(value_units),
            "p50_mmhg": median_value if len(numeric) else np.nan,
            "p95_mmhg": float(numeric.quantile(0.95)) if len(numeric)
                ↪ else np.nan,
            "max_mmhg": float(numeric.max()) if len(numeric) else
                ↪ np.nan,
            "label_include_ok": bool(include_ok),
            "label_exclude_hit": bool(exclude_hit),
            "unit_ok": bool(unit_ok),
            "median_in_expected_range": bool(median_ok),
            "contamination_flag": contamination_flag,
            "manifest_whitelisted": int(itemid_value in
                ↪ set(ICU_PC02_ITEMIDS)),
        }
    )

pco2_itemid_qc_columns = [
    "itemid",
    "label",
    "category",
    "source_branch",
    "sample_type_hint",
    "row_count",
    "non_null_count",
    "units_observed",
    "p50_mmhg",
    "p95_mmhg",
    "max_mmhg",
    "label_include_ok",
    "label_exclude_hit",
    "unit_ok",
    "median_in_expected_range",
    "contamination_flag",
    "manifest_whitelisted",
]

```

```

pco2_itemid_qc_audit = pd.DataFrame(pco2_itemid_qc_rows)
for column_name in pco2_itemid_qc_columns:
    if column_name not in pco2_itemid_qc_audit.columns:
        pco2_itemid_qc_audit[column_name] = pd.NA
pco2_itemid_qc_audit = pco2_itemid_qc_audit[pco2_itemid_qc_columns]
pco2_itemid_qc_audit_path = prior_runs_dir / f"{audit_date}"
    ↪ pco2_itemid_qc_audit.csv"
pco2_itemid_qc_audit.to_csv(pco2_itemid_qc_audit_path, index=False)
print("Wrote:", pco2_itemid_qc_audit_path)

if ICU_INCLUSION_MODE == "validated_only":
    if ICU_PC02_ITEMIDS and not pco2_itemid_qc_audit.empty:
        whitelist_mask =
    ↪ pco2_itemid_qc_audit["itemid"].isin(set(ICU_PC02_ITEMIDS))
        whitelist_rows = pco2_itemid_qc_audit.loc[whitelist_mask].copy()
        if (
            len(whitelist_rows) == len(set(ICU_PC02_ITEMIDS))
            and whitelist_rows["contamination_flag"].eq(False).all()
            and whitelist_rows["median_in_expected_range"].eq(True).all()
        ):
            poc_inclusion_enabled = True
            poc_inclusion_reason = "validated_itemids_passed_qc"
        else:
            poc_inclusion_reason = "validated_itemids_failed_qc"
        elif not ICU_PC02_ITEMIDS:
            poc_inclusion_reason = "manifest_has_no_icu_pco2_itemids"
        else:
            poc_inclusion_reason = "missing_itemid_qc_rows"
    else:
        poc_inclusion_reason = "icu_inclusion_mode_disabled"

print(
    "POC inclusion status:",
    {"enabled": int(poc_inclusion_enabled), "reason": poc_inclusion_reason},
)

poc_flags = pd.DataFrame(columns=["ed_stay_id",
    ↪ "flag_any_gas_hypercapnia_poc"])
if len(panel_poc) > 0 and poc_inclusion_enabled:
    p24_poc = panel_poc.loc[panel_poc["dt_hours"].between(0,24,
    ↪ inclusive="both")].copy()
    p24_poc.loc[:, "flag_any_gas_hypercapnia_poc"] = (p24_poc["pco2"] >=
    ↪ 50).astype("Int64")

```



```

poc_flags = p24_poc.groupby("ed_stay_id", as_index=False).agg(
    flag_any_gas_hypercapnia_poc=("flag_any_gas_hypercapnia_poc", "max")
)

# Avoid duplicate column on re-run
if "flag_any_gas_hypercapnia_poc" in ed_df.columns:
    ed_df = ed_df.drop(columns=["flag_any_gas_hypercapnia_poc"])

if len(poc_flags) > 0:
    ed_df = ed_df.merge(poc_flags, on="ed_stay_id", how="left")
else:
    ed_df["flag_any_gas_hypercapnia_poc"] = 0
ed_df["flag_any_gas_hypercapnia_poc"] = (
    pd.to_numeric(ed_df["flag_any_gas_hypercapnia_poc"], errors="coerce")
    .fillna(0)
    .astype("Int64")
)

ed_df["poc_inclusion_enabled"] = bool(poc_inclusion_enabled)
ed_df["poc_inclusion_reason"] = str(poc_inclusion_reason)

# Optional fallback for first_hco3 from ICU POC only when explicit manifest
# ↪ itemids are provided.
if ICU_HCO3_ITEMIDS and len(panel_poc) > 0:
    poc_hco3_candidates = panel_poc.loc[
        panel_poc["dt_hours"].between(0, 24, inclusive="both")
    ].copy()
    poc_hco3_candidates["hco3"] = pd.to_numeric(poc_hco3_candidates["hco3"],
    ↪ errors="coerce")
    poc_hco3_candidates =
    ↪ poc_hco3_candidates.loc[poc_hco3_candidates["hco3"].notna()].copy()
    if not poc_hco3_candidates.empty:
        first_poc_hco3 = (
            poc_hco3_candidates.sort_values(
                ["ed_stay_id", "dt_hours", "panel_time"],
                kind="stable",
            )
            .groupby("ed_stay_id", as_index=False)
            .first()[["ed_stay_id", "hco3"]]
            .rename(columns={"hco3": "first_hco3_poc_fallback"})
        )
    ed_df = ed_df.merge(first_poc_hco3, on="ed_stay_id", how="left")

```

```

        hco3_missing_mask = pd.to_numeric(ed_df.get("first_hco3"),
↪ errors="coerce").isna()
        hco3_fill_mask = hco3_missing_mask & pd.to_numeric(
            ed_df["first_hco3_poc_fallback"], errors="coerce"
        ).notna()
        ed_df.loc[hco3_fill_mask, "first_hco3"] = ed_df.loc[
            hco3_fill_mask, "first_hco3_poc_fallback"
        ]
        if "first_hco3_source" not in ed_df.columns:
            ed_df["first_hco3_source"] = "missing"
        ed_df.loc[hco3_fill_mask, "first_hco3_source"] =
↪ "poc_explicit_itemid_fallback"
        ed_df = ed_df.drop(columns=["first_hco3_poc_fallback"])

# incremental yield
base = ed_df.get("flag_any_gas_hypercapnia", pd.Series(0,
↪ index=ed_df.index)).fillna(0).astype(int)
poc = ed_df.get("flag_any_gas_hypercapnia_poc", pd.Series(0,
↪ index=ed_df.index)).fillna(0).astype(int)
inc = ((base == 0) & (poc == 1)).sum()
print("ICU POC incremental hypercapnia cases (ED stays):", int(inc))

# optional export
if len(panel_poc) > 0:
    from datetime import datetime

    prior_runs_dir = DATA_DIR / "prior runs"
    prior_runs_dir.mkdir(parents=True, exist_ok=True)
    poc_path = prior_runs_dir /
↪ f"gas_panels_poc_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
    panel_poc.to_parquet(poc_path, index=False)
    print("Wrote:", poc_path)

```

ICU POC long rows: 20532

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/

CC-NLP/MIMIC tabular data/prior runs/2026-02-20 icu\_poc\_itemid\_usage.csv

Updated: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/

CC-NLP/MIMIC tabular data/prior runs/2026-02-20 blood\_gas\_itemid\_manifest\_audit.csv

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/

CC-NLP/MIMIC tabular data/prior runs/2026-02-20 pco2\_itemid\_qc\_audit.csv

POC inclusion status: {'enabled': 0, 'reason': 'validated\_itemids\_failed\_qc'}

ICU POC incremental hypercapnia cases (ED stays): 0

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/CC-NLP/MIMIC tabular data/prior runs/gas\_panels\_poc\_20260220\_122955.parquet

### 3.0.8 Phase 6 — BMI/anthropometrics (OMR)

**Rationale:** Add BMI/height/weight closest to ED presentation when available.

# Purpose: Attach closest anthropometrics with pre-ED preference and bounded  
↪ post-ED fallback.

```
from datetime import datetime
```

```
omr_sql = f"""
SELECT subject_id, chartdate, result_name, result_value
FROM `{PHYS}`.{HOSP}.omr`
WHERE LOWER(result_name) IN ('bmi','height','weight')
"""
```

```
allow_omr_query_failure = os.getenv("COHORT_ALLOW_OMR_QUERY_FAILURE",
↪ "0").strip() == "1"
```

```
try:
```

```
    omr_raw = run_sql_bq(omr_sql)
```

```
except Exception as exc:
```

```
    if allow_omr_query_failure:
```

```
        print("OMR query failed; continuing with empty OMR frame because
↪ COHORT_ALLOW_OMR_QUERY_FAILURE=1:", exc)
```

```
        omr_raw = pd.DataFrame(columns=["subject_id", "chartdate",
↪ "result_name", "result_value"])
```

```
    else:
```

```
        raise
```

```
print("OMR rows:", len(omr_raw))
```

```
omr_prepared = prepare_omr_records(omr_raw)
```

```
ed_df, omr_diagnostics = attach_closest_pre_ed_omr(ed_df, omr_prepared,
↪ window_days=365)
```

```
omr_diagnostics["source_rows_raw"] = int(len(omr_raw))
```

```
omr_diagnostics["prepared_rows"] = int(len(omr_prepared))
```

```
diag_table = (
```

```
    pd.DataFrame({"metric": list(omr_diagnostics.keys()), "value":
```

```
↪ list(omr_diagnostics.values())})
```

```
    .assign(value=lambda d: d["value"].astype(str))
```

```
)
```

```

print(diag_table)

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
out_date = datetime.now().strftime("%Y-%m-%d")
omr_diag_path = prior_runs_dir / f"{out_date} omr_diagnostics.json"
omr_diag_path.write_text(json.dumps(omr_diagnostics, indent=2))
print("Wrote:", omr_diag_path)

# Optional charted anthropometric fallback (nearest-anytime selection).
use_charted_fallback = os.getenv("COHORT_ANTHRO_CHARTED_FALLBACK",
    ↪ "1").strip() == "1"
use_nearest_anytime = os.getenv("COHORT_ANTHRO_NEAREST_ANYTIME", "1").strip()
    ↪ == "1"
charted_anthro_diagnostics = {
    "enabled": bool(use_charted_fallback),
    "nearest_anytime": bool(use_nearest_anytime),
    "charted_rows_input": 0,
    "charted_rows_usable": 0,
    "filled_counts": {},
    "rows_with_any_charted_fill": 0,
}

if use_charted_fallback and len(hadm_list) > 0:
    charted_anthro_sql = f"""
    WITH hadms AS (
        SELECT x AS hadm_id
        FROM UNNEST(@hadms) AS x
    ),
    candidate_items AS (
        SELECT itemid, label, category
        FROM `{PHYS}`.{ICU}.d_items`
        WHERE REGEXP_CONTAINS(LOWER(label),
    ↪ r'(height|weight|admission\\s+weight)')
    ),
    ce AS (
        SELECT
            ce.subject_id,
            ce.hadm_id,
            ce.charttime AS obs_time,
            ci.label,
            ce.valuenum,
            LOWER(REPLACE(COALESCE(ce.valueuom, ''), ' ', '')) AS valueuom
    """

```

```

FROM `{PHYS}`.{ICU}.chartevents` ce
JOIN hadms h
  ON ce.hadm_id = h.hadm_id
JOIN candidate_items ci
  ON ce.itemid = ci.itemid
WHERE ce.valuenum IS NOT NULL
)
SELECT
  subject_id,
  obs_time,
  CASE
    WHEN REGEXP_CONTAINS(LOWER(label), r'height') THEN 'height'
    WHEN REGEXP_CONTAINS(LOWER(label), r'weight') THEN 'weight'
    ELSE NULL
  END AS result_name,
  CASE
    WHEN REGEXP_CONTAINS(LOWER(label), r'height') THEN
      CASE
        WHEN valueuom IN ('cm', 'centimeter', 'centimeters') THEN
↪   valuenum
        WHEN valueuom IN ('m', 'meter', 'meters') THEN valuenum * 100.0
        WHEN valueuom IN ('in', 'inch', 'inches') THEN valuenum * 2.54
        ELSE valuenum
      END
    WHEN REGEXP_CONTAINS(LOWER(label), r'weight') THEN
      CASE
        WHEN valueuom IN ('kg', 'kilogram', 'kilograms') THEN valuenum
        WHEN valueuom IN ('lb', 'lbs', 'pound', 'pounds') THEN valuenum *
↪   0.45359237
        WHEN valueuom IN ('oz', 'ounce', 'ounces') THEN valuenum *
↪   0.0283495231
        ELSE valuenum
      END
    ELSE NULL
  END AS result_value_num,
  'icu_charted' AS source
FROM ce
WHERE REGEXP_CONTAINS(LOWER(label), r'(height|weight)')
"""
charted_raw = run_sql_bq(charted_anthro_sql, {"hadms": hadm_list})
ed_df, charted_anthro_diagnostics = attach_charted_anthro_fallback(
  ed_df,
  charted_raw,

```

```

        nearest_anytime=use_nearest_anytime,
    )

    height_cm = pd.to_numeric(ed_df.get("height_closest_pre_ed"),
↪ errors="coerce")
    weight_kg = pd.to_numeric(ed_df.get("weight_closest_pre_ed"),
↪ errors="coerce")
    bmi_existing = pd.to_numeric(ed_df.get("bmi_closest_pre_ed"),
↪ errors="coerce")
    bmi_calc = weight_kg / (height_cm / 100.0) ** 2
    bmi_fill_mask = bmi_existing.isna() & bmi_calc.notna() & height_cm.gt(0)
    ed_df.loc[bmi_fill_mask, "bmi_closest_pre_ed"] = bmi_calc[bmi_fill_mask]
    charted_anthro_diagnostics["bmi_filled_from_height_weight"] =
↪ int(bmi_fill_mask.sum())
    charted_anthro_diagnostics["enabled"] = True
    charted_anthro_diagnostics["nearest_anytime"] = bool(use_nearest_anytime)

anthro_model_specs = {
    "bmi_closest_pre_ed": ("bmi_closest_pre_ed_model", "bmi_outlier_flag",
↪ 0.0, 100.0),
    "height_closest_pre_ed": ("height_closest_pre_ed_model",
↪ "height_outlier_flag", 100.0, 250.0),
    "weight_closest_pre_ed": ("weight_closest_pre_ed_model",
↪ "weight_outlier_flag", 0.0, 400.0),
}

anthro_audit_rows: list[dict[str, Any]] = []
for raw_col, (model_col, flag_col, lower_bound, upper_bound) in
↪ anthro_model_specs.items():
    raw_numeric = pd.to_numeric(ed_df.get(raw_col), errors="coerce")
    outlier_mask = raw_numeric.notna() & (
        raw_numeric.le(lower_bound) | raw_numeric.gt(upper_bound)
    )
    ed_df[flag_col] = outlier_mask
    ed_df[model_col] = raw_numeric.mask(outlier_mask)
    anthro_audit_rows.append(
        {
            "raw_column": raw_col,
            "model_column": model_col,
            "lower_bound_exclusive": lower_bound,
            "upper_bound_inclusive": upper_bound,
            "raw_non_missing_n": int(raw_numeric.notna().sum()),
            "outlier_n": int(outlier_mask.sum()),
        }
    )

```

```

        "model_non_missing_n": int(pd.to_numeric(ed_df[model_col],
        ↪ errors="coerce").notna().sum()),
    }
)

anthropometric_cleaning_audit = pd.DataFrame(anthro_audit_rows)
anthropometric_cleaning_audit_path = (
    prior_runs_dir / f"{out_date} anthropometric_cleaning_audit.csv"
)
anthropometric_cleaning_audit.to_csv(anthropometric_cleaning_audit_path,
    ↪ index=False)
print("Wrote:", anthropometric_cleaning_audit_path)

anthro_coverage_audit = build_anthro_coverage_audit(ed_df)
anthro_coverage_audit["omr"] = omr_diagnostics
anthro_coverage_audit["charted_fallback"] = charted_anthro_diagnostics
anthro_coverage_audit["cleaning_audit_path"] =
    ↪ str(anthropometric_cleaning_audit_path)
anthro_coverage_audit_path = prior_runs_dir / f"{out_date}"
    ↪ anthropometrics_coverage_audit.json"
anthro_coverage_audit_path.write_text(json.dumps(anthro_coverage_audit,
    ↪ indent=2))
print("Wrote:", anthro_coverage_audit_path)

attached_non_null = omr_diagnostics.get("attached_non_null_counts", {})
attached_total = int(sum(int(v) for v in attached_non_null.values())) if
    ↪ attached_non_null else 0
subject_overlap = int(omr_diagnostics.get("subject_overlap_count", 0))
pre_window_rows = int(omr_diagnostics.get("pre_window_candidate_rows", 0))
post_window_rows = int(omr_diagnostics.get("post_window_candidate_rows", 0))
within_window_rows = int(omr_diagnostics.get("within_window_candidate_rows",
    ↪ 0))
selected_tier_counts = omr_diagnostics.get("selected_tier_counts", {})
selected_tier_rates = omr_diagnostics.get("selected_tier_rates", {})
allow_empty_omr = os.getenv("COHORT_ALLOW_EMPTY_OMR", "0").strip() == "1"
fail_on_omr_attach_inconsistency = (
    os.getenv("COHORT_FAIL_ON_OMR_ATTACH_INCONSISTENCY", "1").strip() == "1"
)

omr_window_diagnostics = {
    "subject_overlap_count": subject_overlap,
    "pre_window_candidate_rows": pre_window_rows,
    "post_window_candidate_rows": post_window_rows,

```

```

    "closest_absolute_candidate_rows":
        ↪ int(omr_diagnostics.get("closest_absolute_candidate_rows", 0)),
    "within_window_candidate_rows": within_window_rows,
    "attached_total_non_null_values": attached_total,
    "selected_tier_counts": selected_tier_counts,
    "selected_tier_rates": selected_tier_rates,
}

if len(omr_raw) > 0 and subject_overlap > 0:
    if pre_window_rows == 0 and post_window_rows > 0:
        print(
            "INFO: OMR subject overlap exists with no pre-ED candidates but
            ↪ post-ED candidates within the 365-day window. "
            "Anthropometric fallback will use post-ED observations and mark
            ↪ them as timing-uncertain."
        )
    elif pre_window_rows == 0 and post_window_rows == 0:
        print(
            "INFO: OMR subject overlap exists but no candidates are within
            ↪ +/-365 days of ED arrival. "
            "Anthropometrics will remain missing for this run."
        )
    if attached_total == 0 and (pre_window_rows > 0 or post_window_rows > 0):
        msg = (
            "OMR attachment produced all-null outputs despite available
            ↪ +/-365-day candidates. "
            "Set COHORT_ALLOW_EMPTY_OMR=1 or
            ↪ COHORT_FAIL_ON_OMR_ATTACH_INCONSISTENCY=0 to bypass this
            ↪ guard."
        )
        if allow_empty_omr or not fail_on_omr_attach_inconsistency:
            print("WARNING:", msg)
        else:
            raise ValueError(msg)

```

OMR rows: 46494

	metric \
0	window_days
1	source_rows_prepared
2	parsed_value_rows
3	ed_rows
4	ed_rows_eligible_for_join
5	subject_overlap_count



```

6  candidate_rows_after_subject_join
7      days_before_min
8      days_before_max
9      nonnegative_candidate_rows
10     pre_window_candidate_rows
11     post_window_candidate_rows
12     closest_absolute_candidate_rows
13     within_window_candidate_rows
14 eligible_ed_stays_with_candidates
15     attached_non_null_counts
16     attached_any_non_null_rows
17     selected_tier_counts
18     selected_tier_rates
19     timing_uncertain_count
20     anthro_source_counts
21     source_rows_raw
22     prepared_rows

```

```

                                value
0                                365
1                               46494
2                               46489
3                               12038
4                               12038
5                                632
6                               1970
7                               -4034
8                               -275
9                                0
10                               0
11                               21
12                               21
13                                0
14                               19
15 {'bmi_closest_pre_ed': 8, 'height_closest_pre_...
16                               19
17 {'pre_ed_365': 0, 'post_ed_365': 19, 'missing'...
18 {'pre_ed_365': 0.0, 'post_ed_365': 0.001578335...
19                               19
20 {'missing': 12019, 'omr': 19}
21                               46494
22                               46494

```

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project

```

CC-NLP/MIMIC tabular data/prior runs/2026-02-20 omr_diagnostics.json
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 anthropometric_cleaning_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 anthropometrics_coverage_audit.json
INFO: OMR subject overlap exists with no pre-ED candidates but post-ED candidates within the
day window. Anthropometric fallback will use post-ED observations and mark them as timing-
uncertain.

```

### 3.0.9 Phase 7 — ICD comorbidity flags

**Rationale:** Derive comorbidity indicators from index admission ICD codes for stratified analyses.

```

# Purpose: Compute comorbidity flags from hospital + ED ICD codes with
↳ chunked hadm queries.

```

```

# ICD code pulls for comorbidity flags (hospital + ED; combined OR)
# NOTE: Use prefix filters to reduce CPU and avoid regex-heavy scans.

```

```

FLAGS = [
    "flag_copd", "flag_osa_ohs", "flag_chf", "flag_neuromuscular",
    "flag_opioid_substance", "flag_pneumonia",
]

```

```

SQL["icd_flags_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x),
icd AS (
    SELECT
        hadm_id,
        UPPER(REPLACE(icd_code, ".", "")) AS code_norm
    FROM `{PHYS}`.{HOSP}.diagnoses_icd`
    WHERE hadm_id IN (SELECT hadm_id FROM hadms)
    AND (
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J43") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J44") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "G473") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "E662") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "I50") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "G12") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "G70") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "G71") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "F11") OR

```

```

        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "T40") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "F13") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J12") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J13") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J14") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J15") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J16") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J17") OR
        STARTS_WITH(UPPER(REPLACE(icd_code, ".", "")), "J18")
    )
)
SELECT
    hadm_id,
    MAX(IF(STARTS_WITH(code_norm, "J43") OR STARTS_WITH(code_norm, "J44"), 1,
↪ 0)) AS flag_copd,
    MAX(IF(STARTS_WITH(code_norm, "G473") OR STARTS_WITH(code_norm, "E662"), 1,
↪ 0)) AS flag_osa_ohs,
    MAX(IF(STARTS_WITH(code_norm, "I50"), 1, 0)) AS flag_chf,
    MAX(IF(STARTS_WITH(code_norm, "G12") OR STARTS_WITH(code_norm, "G70") OR
↪ STARTS_WITH(code_norm, "G71"), 1, 0)) AS flag_neuromuscular,
    MAX(IF(STARTS_WITH(code_norm, "F11") OR STARTS_WITH(code_norm, "T40") OR
↪ STARTS_WITH(code_norm, "F13"), 1, 0)) AS flag_opioid_substance,
    MAX(IF(STARTS_WITH(code_norm, "J12") OR STARTS_WITH(code_norm, "J13") OR
↪ STARTS_WITH(code_norm, "J14") OR STARTS_WITH(code_norm, "J15") OR
↪ STARTS_WITH(code_norm, "J16") OR STARTS_WITH(code_norm, "J17") OR
↪ STARTS_WITH(code_norm, "J18"), 1, 0)) AS flag_pneumonia
FROM icd
GROUP BY hadm_id
"""

```

```

SQL["ed_icd_flags_sql"] = f"""
WITH hadms AS (SELECT x AS hadm_id FROM UNNEST(@hadms) AS x),
ed_dx AS (
    SELECT
        s.hadm_id,
        UPPER(REPLACE(d.icd_code, ".", "")) AS code_norm
    FROM `{PHYS}`.{ED}.diagnosis` d
    JOIN `{PHYS}`.{ED}.edstays` s
    ON s.stay_id = d.stay_id
    WHERE s.hadm_id IN (SELECT hadm_id FROM hadms)
    AND (
        STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J43") OR
        STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J44") OR

```

```

STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "G473") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "E662") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "I50") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "G12") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "G70") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "G71") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "F11") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "T40") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "F13") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J12") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J13") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J14") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J15") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J16") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J17") OR
STARTS_WITH(UPPER(REPLACE(d.icd_code, ".", "")), "J18")
)
)
SELECT
    hadm_id,
    MAX(IF(STARTS_WITH(code_norm, "J43") OR STARTS_WITH(code_norm, "J44"), 1,
↪ 0)) AS flag_copd,
    MAX(IF(STARTS_WITH(code_norm, "G473") OR STARTS_WITH(code_norm, "E662"), 1,
↪ 0)) AS flag_osa_ohs,
    MAX(IF(STARTS_WITH(code_norm, "I50"), 1, 0)) AS flag_chf,
    MAX(IF(STARTS_WITH(code_norm, "G12") OR STARTS_WITH(code_norm, "G70") OR
↪ STARTS_WITH(code_norm, "G71"), 1, 0)) AS flag_neuromuscular,
    MAX(IF(STARTS_WITH(code_norm, "F11") OR STARTS_WITH(code_norm, "T40") OR
↪ STARTS_WITH(code_norm, "F13"), 1, 0)) AS flag_opioid_substance,
    MAX(IF(STARTS_WITH(code_norm, "J12") OR STARTS_WITH(code_norm, "J13") OR
↪ STARTS_WITH(code_norm, "J14") OR STARTS_WITH(code_norm, "J15") OR
↪ STARTS_WITH(code_norm, "J16") OR STARTS_WITH(code_norm, "J17") OR
↪ STARTS_WITH(code_norm, "J18"), 1, 0)) AS flag_pneumonia
FROM ed_dx
GROUP BY hadm_id
"""

```

```

def _run_flag_query_chunked(sql_name: str, label: str) -> pd.DataFrame:
    parts = []
    hadm_chunk_size = 10000
    for i in range(0, len(hadm_list), hadm_chunk_size):
        hadm_chunk = hadm_list[i:i + hadm_chunk_size]
        part = run_sql_bq(sql(sql_name), {"hadms": hadm_chunk})

```

```

        if len(part) > 0:
            parts.append(part)
        print(f"{label} chunk {i // hadm_chunk_size + 1}: rows={len(part)}")

    if parts:
        out = pd.concat(parts, ignore_index=True)
        out = out.groupby("hadm_id", as_index=False)[FLAGS].max()
        return out

    return pd.DataFrame(columns=["hadm_id"] + FLAGS)

flag_hosp = _run_flag_query_chunked("icd_flags_sql", "Hosp ICD flags")
flag_ed = _run_flag_query_chunked("ed_icd_flags_sql", "ED ICD flags")

flag_hosp = flag_hosp.rename(columns={k: f"{k}_hosp" for k in FLAGS})
flag_ed = flag_ed.rename(columns={k: f"{k}_ed" for k in FLAGS})

flag_df = flag_hosp.merge(flag_ed, on="hadm_id", how="outer")
for k in FLAGS:
    hosp_col = f"{k}_hosp"
    ed_col = f"{k}_ed"
    if hosp_col not in flag_df.columns:
        flag_df[hosp_col] = 0
    if ed_col not in flag_df.columns:
        flag_df[ed_col] = 0
    flag_df[k] = ((flag_df[hosp_col].fillna(0).astype(int) == 1) |
        ↪ (flag_df[ed_col].fillna(0).astype(int) == 1)).astype(int)

# Merge into admission-level and ED-stay-level frames (override if present)
for _df_name in ["df", "ed_df"]:
    if _df_name in globals():
        _df = globals()[_df_name]
        drop_cols = [c for c in flag_df.columns if c != "hadm_id" and c in
            ↪ _df.columns]
        if drop_cols:
            _df = _df.drop(columns=drop_cols)
            _df = _df.merge(flag_df, on="hadm_id", how="left")
            globals()[_df_name] = _df

# Prevalence (combined flags)
for k in FLAGS:
    if k in ed_df.columns:
        print(k, int(ed_df[k].fillna(0).sum()))

```

```

Hosp ICD flags chunk 1: rows=3530
Hosp ICD flags chunk 2: rows=3602
Hosp ICD flags chunk 3: rows=3544
Hosp ICD flags chunk 4: rows=3507
Hosp ICD flags chunk 5: rows=213
ED ICD flags chunk 1: rows=332
ED ICD flags chunk 2: rows=402
ED ICD flags chunk 3: rows=379
ED ICD flags chunk 4: rows=402
ED ICD flags chunk 5: rows=27
flag_copd 2184
flag_osa_ohs 1458
flag_chf 3183
flag_neuromuscular 83
flag_opioid_substance 531
flag_pneumonia 2020

```

### 3.0.10 Phase 8 — Timing phenotypes and derived bands

**Rationale:** Compute time-anchored hypercapnia/acidemia phenotypes for ED presentation vs later course.

```

# Purpose: Create time-anchored phenotypes and severity bands relative to
↳ first ED presentation.

# Timing phenotypes

anchor = "ed_intime_first" if "ed_intime_first" in ed_df.columns else
↳ "ed_intime"

ed_df["dt_first_qualifying_gas_hours"] = (ed_df["first_gas_time"] -
↳ ed_df[anchor]).dt.total_seconds() / 3600.0
ed_df["presenting_hypercapnia"] = (ed_df["dt_first_qualifying_gas_hours"] <=
↳ 6).astype("Int64")
ed_df["late_hypercapnia"] = (ed_df["dt_first_qualifying_gas_hours"] >
↳ 6).astype("Int64")

qualifying_hours = pd.to_numeric(ed_df["dt_first_qualifying_gas_hours"],
↳ errors="coerce")
qualifying_observed = qualifying_hours.notna()
ed_df["qualifying_gas_observed"] = qualifying_observed.astype("Int64")
ed_df["presenting_hypercapnia_tri"] = pd.Series(pd.NA, index=ed_df.index,
↳ dtype="Int64")

```

```

ed_df["late_hypercapnia_tri"] = pd.Series(pd.NA, index=ed_df.index,
    ↪ dtype="Int64")
ed_df.loc[qualifying_observed, "presenting_hypercapnia_tri"] = (
    qualifying_hours.loc[qualifying_observed].le(6).astype("Int64")
)
ed_df.loc[qualifying_observed, "late_hypercapnia_tri"] = (
    qualifying_hours.loc[qualifying_observed].gt(6)
    & qualifying_hours.loc[qualifying_observed].le(24)
).astype("Int64")

threshold_any = (
    pd.to_numeric(ed_df.get("pco2_threshold_any", pd.Series(0,
    ↪ index=ed_df.index)), errors="coerce")
    .fillna(0)
    .astype(int)
)
any_icd = (
    pd.to_numeric(ed_df.get("any_hypercap_icd", pd.Series(0,
    ↪ index=ed_df.index)), errors="coerce")
    .fillna(0)
    .astype(int)
)
ed_df["hypercap_timing_class"] = np.select(
    [
        qualifying_observed & qualifying_hours.le(6),
        qualifying_observed & qualifying_hours.gt(6) &
    ↪ qualifying_hours.le(24),
        (qualifying_observed & qualifying_hours.gt(24))
        | (~qualifying_observed & threshold_any.eq(1)),
    ],
    [
        "presenting_0_6h",
        "late_6_24h",
        "none_within_24h",
    ],
    default="icd_only_or_no_qualifying_gas",
)

# NIV/IMV timing relative to first ED presentation
if "first_imv_time" in ed_df.columns:
    ed_df["dt_first_imv_hours"] = (ed_df["first_imv_time"] -
    ↪ ed_df[anchor]).dt.total_seconds() / 3600.0
if "first_niv_time" in ed_df.columns:

```

```

    ed_df["dt_first_niv_hours"] = (ed_df["first_niv_time"] -
↪   ed_df[anchor]).dt.total_seconds() / 3600.0

ed_df["hospital_los_negative_flag"] = (
    pd.to_numeric(ed_df.get("hospital_los_hours"), errors="coerce")
    .lt(0)
    .fillna(False)
)
admittime_series = pd.to_datetime(ed_df.get("admittime"), errors="coerce")
ed_intime_series = pd.to_datetime(ed_df.get("ed_intime"), errors="coerce")
ed_df["admittime_before_ed_intime_flag"] = (
    admittime_series.notna()
    & ed_intime_series.notna()
    & admittime_series.lt(ed_intime_series)
)
disctime_series = pd.to_datetime(ed_df.get("disctime"), errors="coerce")
ed_df["disctime_before_admittime_flag"] = (
    disctime_series.notna()
    & admittime_series.notna()
    & disctime_series.lt(admittime_series)
)
ed_df["time_integrity_any"] = (
    ed_df["hospital_los_negative_flag"].fillna(False).astype(bool)
    | ed_df["admittime_before_ed_intime_flag"].fillna(False).astype(bool)
    | ed_df["disctime_before_admittime_flag"].fillna(False).astype(bool)
)
hospital_los_numeric = pd.to_numeric(ed_df.get("hospital_los_hours"),
↪   errors="coerce")
ed_df["hospital_los_hours_model"] =
↪   hospital_los_numeric.mask(hospital_los_numeric.lt(0))
anchor_series = pd.to_datetime(ed_df.get(anchor), errors="coerce")
for event_col, delta_col, model_col, outside_col in [
    ("first_imv_time", "dt_first_imv_hours", "dt_first_imv_hours_model",
↪   "imv_time_outside_window_flag"),
    ("first_niv_time", "dt_first_niv_hours", "dt_first_niv_hours_model",
↪   "niv_time_outside_window_flag"),
]:
    if event_col not in ed_df.columns:
        ed_df[model_col] = pd.Series(pd.NA, index=ed_df.index,
↪   dtype="Float64")
        ed_df[outside_col] = False
        continue
    event_time = pd.to_datetime(ed_df[event_col], errors="coerce")

```



```

    delta_hours = pd.to_numeric(ed_df.get(delta_col), errors="coerce")
    outside_lower = anchor_series.notna() & event_time.notna() &
↪ event_time.lt(anchor_series)
    outside_upper = disctime_series.notna() & event_time.notna() &
↪ event_time.gt(disctime_series)
    negative_delta = delta_hours.lt(0).fillna(False)
    outside_window = outside_lower | outside_upper | negative_delta
    ed_df[outside_col] = outside_window.fillna(False)
    ed_df[model_col] = delta_hours.mask(outside_window)

# Bands
bins_ph = [-1, 7.20, 7.30, 7.35, 99]
labels_ph = ["<7.20", "7.20-7.29", "7.30-7.34", "7.35"]
ed_df["ph_band"] = pd.cut(ed_df["first_ph"], bins=bins_ph, labels=labels_ph)

bins_hco3 = [-1, 24, 30, 999]
labels_hco3 = ["<24", "24-29", "30"]
ed_df["hco3_band"] = pd.cut(ed_df["first_hco3"], bins=bins_hco3,
↪ labels=labels_hco3)

bins_lac = [-1, 2, 4, 999]
labels_lac = ["<2", "2-4", ">4"]
ed_df["lactate_band"] = pd.cut(ed_df["first_lactate"], bins=bins_lac,
↪ labels=labels_lac)

# Align ED-stay gas source flags to hadm-level thresholds and ICU POC signal.
# This keeps source-specific flags internally consistent in the final ED-stay
↪ export.
if "df" in globals() and "hadm_id" in ed_df.columns and "hadm_id" in
↪ df.columns:
    hadm_thr_cols = [
        c
        for c in [
            "abg_hypercap_threshold",
            "vbg_hypercap_threshold",
            "unknown_hypercap_threshold",
            "other_hypercap_threshold",
        ]
        if c in df.columns
    ]
    if hadm_thr_cols:

```

```

    hadm_thr = df[["hadm_id"] + hadm_thr_cols].drop_duplicates("hadm_id")
    hadm_thr_map = hadm_thr.set_index("hadm_id")

    for col in hadm_thr_cols:
        mapped = pd.to_numeric(ed_df["hadm_id"].map(hadm_thr_map[col]),
↪ errors="coerce")
        existing = pd.to_numeric(ed_df[col], errors="coerce") if col in
↪ ed_df.columns else pd.Series(np.nan, index=ed_df.index)
        ed_df[col] = existing.fillna(mapped).fillna(0).astype("Int64")

# Source-specific flags must include any matching hadm-level threshold.
flag_specs = [
    ("flag_abg_hypercapnia", "abg_hypercap_threshold"),
    ("flag_vbg_hypercapnia", "vbg_hypercap_threshold"),
    ("flag_unknown_hypercapnia", "unknown_hypercap_threshold"),
]
for flag_col, thr_col in flag_specs:
    base = pd.to_numeric(ed_df.get(flag_col, pd.Series(0,
↪ index=ed_df.index)), errors="coerce").fillna(0).astype(int)
    thr = pd.to_numeric(ed_df.get(thr_col, pd.Series(0, index=ed_df.index)),
↪ errors="coerce").fillna(0).astype(int)
    ed_df[flag_col] = ((base == 1) | (thr == 1)).astype("Int64")

any_from_sources = (
    (pd.to_numeric(ed_df.get("flag_abg_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int) == 1)
    | (pd.to_numeric(ed_df.get("flag_vbg_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int) == 1)
    | (pd.to_numeric(ed_df.get("flag_unknown_hypercapnia", 0),
↪ errors="coerce").fillna(0).astype(int) == 1)
)

poc_any = pd.to_numeric(ed_df.get("flag_any_gas_hypercapnia_poc",
↪ pd.Series(0, index=ed_df.index)), errors="coerce").fillna(0).astype(int)
↪ == 1
ed_df["flag_any_gas_hypercapnia"] = (any_from_sources |
↪ poc_any).astype("Int64")
ed_df["flag_other_hypercapnia"] =
↪ ed_df["flag_unknown_hypercapnia"].astype("Int64")

```

### 3.1 QA & Data Fidelity

**Rationale:** Validate joins, missingness, lab completeness, and produce reproducibility artifacts.

```
# Purpose: Create reusable data-quality and merge guardrail helpers to
↳ prevent silent join errors.

# --- QA checks (lightweight, deterministic)
STRICT_QA = False

# 1) Key uniqueness
if 'ed_df' in globals() and 'ed_stay_id' in ed_df.columns:
    try:
        assert_unique(ed_df, 'ed_stay_id', 'ed_df')
        print('QA: ed_stay_id unique OK')
    except Exception as e:
        print('QA FAIL:', e)
        if STRICT_QA:
            raise

# 2) Inclusion sanity: any_hypercap_icd==0 implies gas criteria met
if 'ed_df' in globals() and 'any_hypercap_icd' in ed_df.columns:
    gas_flag = ed_df.get('flag_any_gas_hypercapnia', pd.Series(0,
↳ index=ed_df.index)).fillna(0).astype(int)
    icd_flag = ed_df['any_hypercap_icd'].fillna(0).astype(int)
    viol = (icd_flag == 0) & (gas_flag == 0)
    n_viol = int(viol.sum())
    print('QA: ICD==0 & Gas==0 count:', n_viol)
    if STRICT_QA and n_viol > 0:
        raise AssertionError('Found rows without ICD and without gas
↳ criteria.')
```

```
# 3) Temporal sanity: first_gas_time >= ed_intime (allow small negative
↳ drift)
if 'ed_df' in globals() and 'first_gas_time' in ed_df.columns and 'ed_intime'
↳ in ed_df.columns:
    dt = (pd.to_datetime(ed_df['first_gas_time']) -
↳ pd.to_datetime(ed_df['ed_intime'])).dt.total_seconds() / 3600
    n_neg = int((dt < -1).sum()) # allow 1h clock drift
    print('QA: first_gas_time < ed_intime by >1h:', n_neg)
    if STRICT_QA and n_neg > 0:
        raise AssertionError('first_gas_time before ed_intime by >1h')
```

```

# 4) Range checks (report only)
if 'ed_df' in globals():
    ranges = {
        'first_ph': (6.8, 7.8),
        'first_pco2': (10, 200),
        'first_lactate': (0, 30),
        'creatinine': (0, 20),
    }
    rc = check_ranges(ed_df, ranges)
    if not rc.empty:
        print('QA range check (n out-of-range):')
        print(rc.to_string(index=False))

QA: ed_stay_id unique OK
QA: first_gas_time < ed_intime by >1h: 0
QA range check (n out-of-range):
      col  n_bad
first_ph      9
first_pco2     2
first_lactate   0

# Purpose: Report missingness for key variables to support transparent
↳ data-quality reporting.

import json
from datetime import datetime

# T1 - uniqueness and join explosion guard
if ed_df["ed_stay_id"].nunique() != len(ed_df):
    raise ValueError("ed_stay_id not unique after merges")

# T1b - create cleaned vitals model fields while preserving raw values.
ed_df, vitals_outlier_audit = add_vitals_model_fields(ed_df)
ed_df, gas_outlier_audit = add_gas_model_fields(ed_df)
outliers_audit = pd.concat([vitals_outlier_audit, gas_outlier_audit],
    ↳ ignore_index=True, sort=False)

# T1c - fail-fast cleaned-vitals bounds checks.
clean_bounds = {
    "ed_triage_o2sat_clean": (0.0, 100.0),
    "ed_first_o2sat_clean": (0.0, 100.0),
    "ed_triage_sbp_clean": (20.0, 300.0),

```

```

    "ed_first_sbp_clean": (20.0, 300.0),
    "ed_triage_dbp_clean": (10.0, 200.0),
    "ed_first_dbp_clean": (10.0, 200.0),
}
for column_name, (lower_bound, upper_bound) in clean_bounds.items():
    if column_name not in ed_df.columns:
        continue
    numeric = pd.to_numeric(ed_df[column_name], errors="coerce")
    invalid_n = int((numeric.notna() & ((numeric < lower_bound) | (numeric >
↪ upper_bound)))).sum())
    if invalid_n:
        raise ValueError(
            f"Cleaned vital bounds failure for {column_name}: {invalid_n}
            ↪ values outside [{lower_bound}, {upper_bound}]"
        )

for temp_clean_column in ("ed_triage_temp_f_clean", "ed_first_temp_f_clean"):
    if temp_clean_column not in ed_df.columns:
        continue
    numeric = pd.to_numeric(ed_df[temp_clean_column], errors="coerce")
    residual_celsius_like_n = int(numeric.between(20.0, 50.0,
↪ inclusive="both").sum())
    if residual_celsius_like_n:
        raise ValueError(
            f"Found {residual_celsius_like_n} cleaned values still in 20-50°F
            ↪ band in {temp_clean_column}."
        )

ed_vitals_distribution_summary, ed_vitals_extreme_examples,
↪ ed_vitals_model_delta = build_ed_vitals_audit_artifacts(ed_df)

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
out_date = datetime.now().strftime("%Y-%m-%d")

vitals_outlier_audit_path = prior_runs_dir / f"{out_date}
↪ vitals_outlier_audit.csv"
vitals_outlier_audit.to_csv(vitals_outlier_audit_path, index=False)
print("Wrote:", vitals_outlier_audit_path)
outliers_audit_path = prior_runs_dir / f"{out_date} outliers_audit.csv"
outliers_audit.to_csv(outliers_audit_path, index=False)
print("Wrote:", outliers_audit_path)

```

```

ed_vitals_distribution_summary_path = prior_runs_dir / f"{out_date}"
    ↪ ed_vitals_distribution_summary.csv"
ed_vitals_distribution_summary.to_csv(ed_vitals_distribution_summary_path,
    ↪ index=False)
print("Wrote:", ed_vitals_distribution_summary_path)
ed_vitals_extreme_examples_path = prior_runs_dir / f"{out_date}"
    ↪ ed_vitals_extreme_examples.csv"
ed_vitals_extreme_examples.to_csv(ed_vitals_extreme_examples_path,
    ↪ index=False)
print("Wrote:", ed_vitals_extreme_examples_path)
ed_vitals_model_delta_path = prior_runs_dir / f"{out_date}"
    ↪ ed_vitals_model_delta.csv"
ed_vitals_model_delta.to_csv(ed_vitals_model_delta_path, index=False)
print("Wrote:", ed_vitals_model_delta_path)

timing_integrity_audit = pd.DataFrame(
    [
        {
            "rows_total": int(len(ed_df)),
            "time_integrity_any_n": int(ed_df.get("time_integrity_any",
    ↪ pd.Series(False,
    ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "hospital_los_negative_n":
    ↪ int(ed_df.get("hospital_los_negative_flag", pd.Series(False,
    ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "admittime_before_ed_intime_n":
    ↪ int(ed_df.get("admittime_before_ed_intime_flag",
    ↪ pd.Series(False,
    ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "disctime_before_admittime_n":
    ↪ int(ed_df.get("disctime_before_admittime_flag",
    ↪ pd.Series(False,
    ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "qualifying_gas_observed_n":
    ↪ int(pd.to_numeric(ed_df.get("qualifying_gas_observed"),
    ↪ errors="coerce").fillna(0).astype(int).sum()),
            "qualifying_gas_missing_n":
    ↪ int(pd.to_numeric(ed_df.get("qualifying_gas_observed"),
    ↪ errors="coerce").fillna(0).astype(int).eq(0).sum()),
        }
    ]
)

```

```

timing_integrity_audit_path = prior_runs_dir / f"{out_date}"
    ↪ timing_integrity_audit.csv"
timing_integrity_audit.to_csv(timing_integrity_audit_path, index=False)
print("Wrote:", timing_integrity_audit_path)

ventilation_timing_audit = pd.DataFrame(
    [
        {
            "rows_total": int(len(ed_df)),
            "dt_first_imv_hours_negative_n":
                ↪ int(pd.to_numeric(ed_df.get("dt_first_imv_hours"),
                ↪ errors="coerce").lt(0).fillna(False).sum()),
            "dt_first_niv_hours_negative_n":
                ↪ int(pd.to_numeric(ed_df.get("dt_first_niv_hours"),
                ↪ errors="coerce").lt(0).fillna(False).sum()),
            "imv_time_outside_window_n":
                ↪ int(ed_df.get("imv_time_outside_window_flag",
                ↪ pd.Series(False,
                ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "niv_time_outside_window_n":
                ↪ int(ed_df.get("niv_time_outside_window_flag",
                ↪ pd.Series(False,
                ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "dt_first_imv_hours_model_nonnull_n":
                ↪ int(pd.to_numeric(ed_df.get("dt_first_imv_hours_model"),
                ↪ errors="coerce").notna().sum()),
            "dt_first_niv_hours_model_nonnull_n":
                ↪ int(pd.to_numeric(ed_df.get("dt_first_niv_hours_model"),
                ↪ errors="coerce").notna().sum()),
        }
    ]
)
ventilation_timing_audit_path = prior_runs_dir / f"{out_date}"
    ↪ ventilation_timing_audit.csv"
ventilation_timing_audit.to_csv(ventilation_timing_audit_path, index=False)
print("Wrote:", ventilation_timing_audit_path)

hco3_numeric = pd.to_numeric(ed_df.get("first_hco3"), errors="coerce")
hco3_source = ed_df.get("first_hco3_source", pd.Series("missing",
    ↪ index=ed_df.index)).astype("string").fillna("missing")
hco3_source_value_mismatch_n = int(((hco3_source != "missing") &
    ↪ hco3_numeric.isna()).sum())
hco3_source_counts = (

```

```

    hco3_source.value_counts(dropna=False)
    .rename_axis("first_hco3_source")
    .reset_index(name="row_count")
)
hco3_source_counts["null_value_count"] =
    ↪ hco3_source_counts["first_hco3_source"].map(
        lambda source_name: int(
            ((hco3_source == source_name) & hco3_numeric.isna()).sum()
        )
    )
)
hco3_source_counts["nonnull_value_count"] =
    ↪ hco3_source_counts["first_hco3_source"].map(
        lambda source_name: int(
            ((hco3_source == source_name) & hco3_numeric.notna()).sum()
        )
    )
)
hco3_source_counts["null_value_rate"] = np.where(
    hco3_source_counts["row_count"] > 0,
    hco3_source_counts["null_value_count"] / hco3_source_counts["row_count"],
    0.0,
)
hco3_source_counts["source_value_mismatch_n"] = hco3_source_value_mismatch_n
hco3_integrity_audit_path = prior_runs_dir / f"{out_date}"
    ↪ hco3_integrity_audit.csv"
hco3_source_counts.to_csv(hco3_integrity_audit_path, index=False)
print("Wrote:", hco3_integrity_audit_path)

print("ED vitals cleaning summary (raw->clean removals):")
display(
    ed_vitals_model_delta[
        [
            "vital_name",
            "raw_n",
            "clean_n",
            "raw_removed_n",
            "raw_removed_pct",
            "converted_celsius_like_n",
        ]
    ].sort_values("vital_name")
)

# T2 - missingness summary with expected-vs-unexpected classifications.
expected_sparse_fields = set()

```



```

omr_pre_window_rows = int(omr_diagnostics.get("pre_window_candidate_rows",
↪ 0)) if "omr_diagnostics" in globals() else 0
omr_post_window_rows = int(omr_diagnostics.get("post_window_candidate_rows",
↪ 0)) if "omr_diagnostics" in globals() else 0
if omr_pre_window_rows == 0 and omr_post_window_rows == 0:
    expected_sparse_fields.update({"bmi_closest_pre_ed",
↪ "height_closest_pre_ed", "weight_closest_pre_ed"})

miss = classify_missingness_expectations(
    ed_df,
    TARGET_FIELDS,
    expected_sparse_fields=expected_sparse_fields,
)
print(miss.sort_values(["missing_pct", "field"], ascending=[False,
↪ True]).head(30))

unexpected_full_null_fields = miss.loc[
    miss["expectation"].eq("unexpected_full_null"), "field"
].tolist()
expected_structural_null_fields = sorted(
    set(EXPECTED_STRUCTURAL_NULL_FIELDS).intersection(ed_df.columns)
)

# Compute UOM checks against canonical export frame contract (or
↪ deterministic preview).
if "final_cc" in globals():
    uom_scope_df = final_cc
    uom_scope_frame = "final_cc"
elif "df" in globals() and "ed_triage_cc" in df.columns:
    _mask_df_cc = df["ed_triage_cc"].notna() &
↪ (df["ed_triage_cc"].astype(str).str.strip() != "")
    _df_cc = df.loc[_mask_df_cc].copy()
    _mask_cc = ed_df["ed_triage_cc"].notna() &
↪ (ed_df["ed_triage_cc"].astype(str).str.strip() != "")
    _ed_cc_only = ed_df.loc[_mask_cc].copy()
    _ed_tmp = _ed_cc_only.copy()
    if "ed_intime" in _ed_tmp.columns:
        _ed_tmp = _ed_tmp.sort_values(["hadm_id", "ed_intime", "ed_stay_id"],
↪ na_position="last")
    else:
        _ed_tmp = _ed_tmp.sort_values(["hadm_id", "ed_stay_id"],
↪ na_position="last")
    _ed_first = _ed_tmp.drop_duplicates("hadm_id", keep="first")

```

```

    _add_ed_cols = [c for c in _ed_first.columns if c not in _df_cc.columns]
    uom_scope_df = _df_cc.merge(_ed_first[["hadm_id"] + _add_ed_cols],
    ↪ on="hadm_id", how="left")
    uom_scope_frame = "final_cc_preview"
else:
    uom_scope_df = ed_df
    uom_scope_frame = "ed_df"

uom_expectation_checks = evaluate_uom_expectations(uom_scope_df)
uom_expectation_scope = {"frame": uom_scope_frame, "row_count":
    ↪ int(len(uom_scope_df))}

# T2b - route-stratified first_other_pco2 audit using canonical QA scope.
first_other_scope_df = (
    uom_scope_df
    if {"first_other_src", "first_other_pco2"}.issubset(uom_scope_df.columns)
    else ed_df
)
first_other_scope_frame = "uom_scope_df" if first_other_scope_df is
    ↪ uom_scope_df else "ed_df"
first_other_pco2_audit = build_first_other_pco2_audit(first_other_scope_df)
first_other_pco2_audit_path = prior_runs_dir / f"{out_date}"
    ↪ first_other_pco2_audit.csv"
first_other_pco2_audit.to_csv(first_other_pco2_audit_path, index=False)
print("Wrote:", first_other_pco2_audit_path)
print("first_other_pco2 audit scope:", first_other_scope_frame)

print("Expected structural null fields:", expected_structural_null_fields)
print("Unexpected full-null fields (top 20):",
    ↪ unexpected_full_null_fields[:20])
print("UOM expectation scope:", uom_expectation_scope)
print("UOM expectation checks:", json.dumps(uom_expectation_checks,
    ↪ indent=2))

# T3 - lab capture completeness.
pct_any_6h = float(p06["ed_stay_id"].nunique() /
    ↪ max(ed_df["ed_stay_id"].nunique(),1))
pct_any_24h = float(p24["ed_stay_id"].nunique() /
    ↪ max(ed_df["ed_stay_id"].nunique(),1))
print("% any gas panel 0-6h:", round(pct_any_6h*100,1))
print("% any gas panel 0-24h:", round(pct_any_24h*100,1))

if "gas_source_audit" not in globals():

```

```

    gas_source_audit = summarize_gas_source(panel if "panel" in globals()
↪     else pd.DataFrame())

gas_overlap_summary = build_gas_source_overlap_summary(ed_df)
gas_overlap_summary_path = prior_runs_dir / f"{out_date}"
↪     gas_source_overlap_summary.csv"
gas_overlap_summary.to_csv(gas_overlap_summary_path, index=False)
print("Wrote:", gas_overlap_summary_path)

gas_source_unknown_rate = float(gas_source_audit.get("source_rates",
↪     {}).get("unknown", 0.0))
gas_source_other_rate = gas_source_unknown_rate
print("% source unknown (panel-level):", round(gas_source_unknown_rate * 100,
↪     1))

# Blood-gas source distribution audit (LAB + POC definitive pCO2 by source
↪     class).
pco2_source_distribution_rows: list[dict[str, Any]] = []
PCC_DIST_COLUMNS = [
    "source_branch",
    "source",
    "n",
    "median",
    "p95",
    "max",
    "pct_ge_50",
]

def _append_distribution_rows(source_branch: str, source: str, values:
↪     pd.Series) -> None:
    numeric = pd.to_numeric(values, errors="coerce").dropna()
    if numeric.empty:
        return
    pco2_source_distribution_rows.append(
        {
            "source_branch": source_branch,
            "source": source,
            "n": int(numeric.shape[0]),
            "median": float(numeric.median()),
            "p95": float(numeric.quantile(0.95)),
            "max": float(numeric.max()),
            "pct_ge_50": float((numeric >= 50).mean()),
        }
    )

```

```

    )

pco2_distribution_scope = ed_df
if "uom_scope_df" in globals() and isinstance(uom_scope_df, pd.DataFrame):
    poc_scope_columns = ["poc_abg_paco2", "poc_vbg_paco2", "poc_other_paco2"]
    has_poc_values = any(
        col in uom_scope_df.columns
        and pd.to_numeric(uom_scope_df[col], errors="coerce").notna().any()
        for col in poc_scope_columns
    )
    if has_poc_values:
        pco2_distribution_scope = uom_scope_df

if "panel" in globals() and len(panel) > 0 and "pco2" in panel.columns:
    panel_pco2 = panel.copy()
    panel_pco2["pco2"] = pd.to_numeric(panel_pco2["pco2"], errors="coerce")
    panel_pco2 = panel_pco2.loc[panel_pco2["pco2"].notna()].copy()
    if len(panel_pco2) > 0:
        for source_name, subset in panel_pco2.groupby("source",
            ↪ dropna=False):
            _append_distribution_rows("LAB", str(source_name),
            ↪ subset["pco2"])

if "poc_abg_paco2" in pco2_distribution_scope.columns:
    _append_distribution_rows("POC", "arterial",
    ↪ pco2_distribution_scope["poc_abg_paco2"])
if "poc_vbg_paco2" in pco2_distribution_scope.columns:
    _append_distribution_rows("POC", "venous",
    ↪ pco2_distribution_scope["poc_vbg_paco2"])
if "poc_other_paco2" in pco2_distribution_scope.columns:
    _append_distribution_rows("POC", "unknown",
    ↪ pco2_distribution_scope["poc_other_paco2"])

pco2_source_distribution_audit = pd.DataFrame(pco2_source_distribution_rows)
for column_name in PCC_DIST_COLUMNS:
    if column_name not in pco2_source_distribution_audit.columns:
        pco2_source_distribution_audit[column_name] = pd.NA
pco2_source_distribution_audit =
    ↪ pco2_source_distribution_audit[PCC_DIST_COLUMNS]
pco2_source_distribution_audit_path = prior_runs_dir / f"{out_date}"
    ↪ pco2_source_distribution_audit.csv"
pco2_source_distribution_audit.to_csv(pco2_source_distribution_audit_path,
    ↪ index=False)

```

```

print("Wrote:", pco2_source_distribution_audit_path)

poc_other_values_all = (
    pd.to_numeric(ed_df["poc_other_paco2"], errors="coerce")
    if "poc_other_paco2" in ed_df.columns
    else pd.Series(dtype="float64")
)
unknown_threshold_series = (
    pd.to_numeric(ed_df["unknown_hypercap_threshold"], errors="coerce")
    if "unknown_hypercap_threshold" in ed_df.columns
    else pd.Series(0, index=ed_df.index, dtype="float64")
)
poc_unknown_threshold_series = (
    pd.to_numeric(ed_df["poc_unknown_hypercap_threshold"], errors="coerce")
    if "poc_unknown_hypercap_threshold" in ed_df.columns
    else pd.Series(0, index=ed_df.index, dtype="float64")
)
first_other_detail_series = (
    ed_df["first_other_src_detail"].astype("string")
    if "first_other_src_detail" in ed_df.columns
    else pd.Series("missing", index=ed_df.index, dtype="string")
)
poc_other_candidate_n = int(poc_other_values_all.notna().sum())
poc_other_hypercap_n = int((poc_other_values_all >= 50).sum())
poc_unknown_hadm_n = int(
    poc_unknown_threshold_series.fillna(0).astype(int).sum()
)
unknown_from_poc_detail_n = int(
    (
        unknown_threshold_series.fillna(0).astype(int).eq(1)
        & first_other_detail_series.str.lower().isin({"poc_bg_unknown",
        ↪ "poc_bg_unknown_available"})
    ).sum()
)
other_route_quarantine_audit = pd.DataFrame(
    [
        {
            "poc_other_candidate_n": poc_other_candidate_n,
            "poc_other_ge50_n": poc_other_hypercap_n,
            "poc_other_ge50_rate": float(poc_other_hypercap_n /
            ↪ poc_other_candidate_n)
            if poc_other_candidate_n
            else 0.0,

```

```

        "poc_unknown_hadm_n": poc_unknown_hadm_n,
        "unknown_hypercap_threshold_hadm_n":
            ↪ int(unknown_threshold_series.fillna(0).astype(int).sum()),
        "unknown_from_poc_detail_hadm_n": unknown_from_poc_detail_n,
    }
]
)
other_route_quarantine_audit_path = prior_runs_dir / f"{out_date}"
    ↪ other_route_quarantine_audit.csv"
other_route_quarantine_audit.to_csv(other_route_quarantine_audit_path,
    ↪ index=False)
print("Wrote:", other_route_quarantine_audit_path)

first_gas_anchor_audit = pd.DataFrame(
    [
        {
            "row_count": int(len(ed_df)),
            "first_gas_time_nonnull_n":
                ↪ int(pd.to_datetime(ed_df.get("first_gas_time"),
                ↪ errors="coerce").notna().sum())
            if "first_gas_time" in ed_df.columns
            else 0,
            "first_gas_anchor_has_pco2_n":
                ↪ int(ed_df.get("first_gas_anchor_has_pco2", pd.Series(False,
                ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
            "first_gas_anchor_source_validated_n":
                ↪ int(ed_df.get("first_gas_anchor_source_validated",
                ↪ pd.Series(False,
                ↪ index=ed_df.index)).fillna(False).astype(bool).sum()),
        }
    ]
)
first_gas_anchor_audit["first_gas_without_pco2_anchor_n"] = (
    first_gas_anchor_audit["first_gas_time_nonnull_n"] -
    ↪ first_gas_anchor_audit["first_gas_anchor_has_pco2_n"]
)
first_gas_anchor_audit_path = prior_runs_dir / f"{out_date}"
    ↪ first_gas_anchor_audit.csv"
first_gas_anchor_audit.to_csv(first_gas_anchor_audit_path, index=False)
print("Wrote:", first_gas_anchor_audit_path)

hadm_rows = int(len(df)) if "df" in globals() else None
hadm_cc_rows = None

```

```

if "df" in globals() and "ed_triage_cc" in df.columns:
    _hadm_cc_mask = df["ed_triage_cc"].notna() &
    ↪ (df["ed_triage_cc"].astype(str).str.strip() != "")
    hadm_cc_rows = int(_hadm_cc_mask.sum())

uom_failed_checks = []
for uom_column, details in uom_expectation_checks.get("paco2_uom_checks",
    ↪ {}).items():
    if details.get("present") and not details.get("passes", True):
        uom_failed_checks.append(uom_column)

qa_warnings = []
qa_infos = []
if unexpected_full_null_fields:
    qa_warnings.append("unexpected_full_null_fields_detected")
if gas_source_audit.get("all_other_or_unknown", False):
    qa_warnings.append("gas_source_all_other_or_unknown")
if omr_pre_window_rows == 0 and omr_post_window_rows > 0:
    qa_infos.append("omr_pre_window_empty_post_window_fallback_used")
if omr_pre_window_rows == 0 and omr_post_window_rows == 0:
    qa_infos.append("omr_pre_post_window_empty_expected_sparse")
if uom_failed_checks:
    qa_warnings.append("uom_checks_failed")
if "status" in first_other_pco2_audit.columns and
    ↪ (first_other_pco2_audit["status"] == "missing_columns").any():
    qa_warnings.append("first_other_pco2_audit_missing_columns")
if "unknown_from_poc_detail_hadm_n" in other_route_quarantine_audit.columns:
    if int(other_route_quarantine_audit["unknown_from_poc_detail_hadm_n"]
    ↪ ).iloc[0]) > 0:
        qa_infos.append("unknown_route_includes_poc_detail")
if "first_gas_without_pco2_anchor_n" in first_gas_anchor_audit.columns:
    if int(first_gas_anchor_audit["first_gas_without_pco2_anchor_n"].iloc[0])
    ↪ > 0:
        qa_warnings.append("first_gas_anchor_without_pco2_detected")
if "source_value_mismatch_n" in hco3_source_counts.columns:
    if int(hco3_source_counts["source_value_mismatch_n"].max()) > 0:
        qa_warnings.append("first_hco3_source_value_mismatch_detected")
qa_other_warn_threshold = float(os.getenv("COHORT_WARN_OTHER_RATE", "0.50"))
qa_other_fail_raw = os.getenv("COHORT_FAIL_OTHER_RATE", "").strip()
qa_other_fail_threshold = float(qa_other_fail_raw) if qa_other_fail_raw else
    ↪ None
if qa_other_fail_threshold is not None and gas_source_other_rate >=
    ↪ qa_other_fail_threshold:

```

```

        qa_warnings.append("gas_source_other_rate_fail_threshold_exceeded")
    elif gas_source_other_rate >= qa_other_warn_threshold:
        qa_infos.append("gas_source_other_rate_high")

first_other_pco2_audit_records =
    ↪ json.loads(first_other_pco2_audit.to_json(orient="records"))
vitals_outlier_audit_records =
    ↪ json.loads(vitals_outlier_audit.to_json(orient="records"))
outliers_audit_records = json.loads(outliers_audit.to_json(orient="records"))
ed_vitals_distribution_summary_records =
    ↪ json.loads(ed_vitals_distribution_summary.to_json(orient="records"))
ed_vitals_model_delta_records =
    ↪ json.loads(ed_vitals_model_delta.to_json(orient="records"))
gas_overlap_summary_records =
    ↪ json.loads(gas_overlap_summary.to_json(orient="records")) if
    ↪ "gas_overlap_summary" in globals() else []
pco2_source_distribution_audit_records = json.loads(
    pco2_source_distribution_audit.to_json(orient="records")
)
other_route_quarantine_audit_records = json.loads(
    other_route_quarantine_audit.to_json(orient="records")
)
first_gas_anchor_audit_records =
    ↪ json.loads(first_gas_anchor_audit.to_json(orient="records"))
pco2_itemid_qc_audit_records = (
    json.loads(pco2_itemid_qc_audit.to_json(orient="records"))
    if "pco2_itemid_qc_audit" in globals() and not pco2_itemid_qc_audit.empty
    else []
)
timing_integrity_audit_records = (
    json.loads(timing_integrity_audit.to_json(orient="records"))
    if "timing_integrity_audit" in globals()
    else []
)
ventilation_timing_audit_records = (
    json.loads(ventilation_timing_audit.to_json(orient="records"))
    if "ventilation_timing_audit" in globals()
    else []
)
anthropometric_cleaning_audit_records = (
    json.loads(anthropometric_cleaning_audit.to_json(orient="records"))
    if "anthropometric_cleaning_audit" in globals()
    else []
)

```



```

)
hco3_integrity_audit_records = (
    json.loads(hco3_source_counts.to_json(orient="records"))
    if "hco3_source_counts" in globals()
    else []
)
blood_gas_itemid_manifest_audit_records = (
    json.loads(blood_gas_itemid_manifest_audit.to_json(orient="records"))
    if "blood_gas_itemid_manifest_audit" in globals() and not
        ↪ blood_gas_itemid_manifest_audit.empty
    else []
)

qa_status = "warning" if qa_warnings else "pass"

# T4 - QA summary artifact.
qa_summary = {
    "ed_spine_rows": int(len(ed_df)),
    "ed_rows": int(len(ed_df)),
    "ed_unique": int(ed_df["ed_stay_id"].nunique()),
    "hadm_rows": hadm_rows,
    "hadm_cc_rows": hadm_cc_rows,
    "icu_link_rate": float(ed_df["icu_intime_first"].notna().mean()) if
        ↪ "icu_intime_first" in ed_df.columns else None,
    "pct_any_gas_0_6h": pct_any_6h,
    "pct_any_gas_0_24h": pct_any_24h,
    "gas_source_other_rate": gas_source_other_rate,
    "gas_source_other_rate_warn_threshold": qa_other_warn_threshold,
    "gas_source_other_rate_fail_threshold": qa_other_fail_threshold,
    "source_unknown_rate": gas_source_unknown_rate,
    "gas_source_audit": gas_source_audit,
    "gas_source_overlap_summary": gas_overlap_summary_records,
    "blood_gas_manifest": {
        "version": BLOOD_GAS_MANIFEST.get("version"),
        "path": BLOOD_GAS_MANIFEST.get("path"),
    },
    "blood_gas_itemid_manifest_audit":
        ↪ blood_gas_itemid_manifest_audit_records,
    "pco2_source_distribution_audit": pco2_source_distribution_audit_records,
    "other_route_quarantine_audit": other_route_quarantine_audit_records,
    "first_gas_anchor_audit": first_gas_anchor_audit_records,
    "pco2_itemid_qc_audit": pco2_itemid_qc_audit_records,
    "poc_inclusion_enabled": bool(poc_inclusion_enabled),

```

```

"poc_inclusion_reason": str(poc_inclusion_reason),
"omr_diagnostics": omr_diagnostics if "omr_diagnostics" in globals() else
↪ None,
"omr_window_diagnostics": omr_window_diagnostics if
↪ "omr_window_diagnostics" in globals() else None,
"anthro_timing_tier_counts": omr_diagnostics.get("selected_tier_counts",
↪ {}) if "omr_diagnostics" in globals() else {},
"anthro_timing_tier_rates": omr_diagnostics.get("selected_tier_rates",
↪ {}) if "omr_diagnostics" in globals() else {},
"anthro_coverage_audit": anthro_coverage_audit if "anthro_coverage_audit"
↪ in globals() else None,
"anthropometric_cleaning_audit": anthropometric_cleaning_audit_records,
"hco3_integrity_audit": hco3_integrity_audit_records,
"charted_anthro_diagnostics": charted_anthro_diagnostics if
↪ "charted_anthro_diagnostics" in globals() else None,
"timing_integrity_audit": timing_integrity_audit_records,
"ventilation_timing_audit": ventilation_timing_audit_records,
"unexpected_full_null_fields": unexpected_full_null_fields,
"expected_structural_null_fields": expected_structural_null_fields,
"uom_expectation_scope": uom_expectation_scope,
"uom_expectation_checks": uom_expectation_checks,
"first_other_pco2_audit_scope": first_other_scope_frame,
"first_other_pco2_audit": first_other_pco2_audit_records,
"vitals_outlier_audit": vitals_outlier_audit_records,
"ed_vitals_distribution_summary": ed_vitals_distribution_summary_records,
"ed_vitals_model_delta": ed_vitals_model_delta_records,
"outliers_audit": outliers_audit_records,
"ed_vitals_audit_paths": {
    "distribution_summary": str(ed_vitals_distribution_summary_path),
    "extreme_examples": str(ed_vitals_extreme_examples_path),
    "model_delta": str(ed_vitals_model_delta_path),
},
"blood_gas_audit_paths": {
    "manifest_itemids": str(blood_gas_itemid_manifest_audit_path)
    if "blood_gas_itemid_manifest_audit_path" in globals()
    else None,
    "pco2_itemid_qc": str(pco2_itemid_qc_audit_path)
    if "pco2_itemid_qc_audit_path" in globals()
    else None,
    "pco2_source_distribution": str(pco2_source_distribution_audit_path),
    "other_route_quarantine": str(other_route_quarantine_audit_path),
    "first_gas_anchor": str(first_gas_anchor_audit_path),
    "hco3_integrity": str(hco3_integrity_audit_path)
}

```

```

        if "hco3_integrity_audit_path" in globals()
        else None,
    },
    "timing_integrity_audit_path": str(timing_integrity_audit_path)
    if "timing_integrity_audit_path" in globals()
    else None,
    "ventilation_timing_audit_path": str(ventilation_timing_audit_path)
    if "ventilation_timing_audit_path" in globals()
    else None,
    "anthropometric_cleaning_audit_path":
        ↪ str(anthropometric_cleaning_audit_path)
    if "anthropometric_cleaning_audit_path" in globals()
    else None,
    "qa_status": qa_status,
    "qa_warnings": qa_warnings,
    "qa_infos": qa_infos,
    "run_metadata": RUN_METADATA if "RUN_METADATA" in globals() else None,
}
qa_path = WORK_DIR / "qa_summary.json"
qa_path.write_text(json.dumps(qa_summary, indent=2))
print("Wrote:", qa_path)

```

```

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 vitals_outlier_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 outliers_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 ed_vitals_distribution_summary.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 ed_vitals_extreme_examples.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 ed_vitals_model_delta.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 timing_integrity_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 ventilation_timing_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 hco3_integrity_audit.csv
ED vitals cleaning summary (raw->clean removals):

```

	vital_name	raw_n	clean_n	raw_removed_n	raw_removed_pct	converted_celsius_like_n
7	first_dbp	9296	9286	10	0.001076	0
9	first_o2sat	9053	9052	1	0.000110	0
3	first_pain	7163	7163	0	0.000000	0
6	first_sbp	9296	9289	7	0.000753	0
1	first_temp	6993	6993	0	0.000000	56
5	triage_dbp	8842	8829	13	0.001470	0
8	triage_o2sat	8786	8785	1	0.000114	0
2	triage_pain	8244	7674	570	0.069141	0
4	triage_sbp	8883	8876	7	0.000788	0
0	triage_temp	8435	8434	1	0.000119	20

	field	missing_n	missing_pct	expectation
0	abg_before_imv	12038	1.000000	missing_column
62	first_other_time	12038	1.000000	missing_column
92	icu_stay_id	12038	1.000000	missing_column
116	vbg_before_imv	12038	1.000000	missing_column
5	anthro_chartdate	12019	0.998422	conditional_sparse
6	anthro_days_offset	12019	0.998422	conditional_sparse
60	first_hco3	11761	0.976990	conditional_sparse
79	hco3_band	11761	0.976990	conditional_sparse
32	ed_first_rhythm	11367	0.944260	conditional_sparse
18	deathtime	9867	0.819654	conditional_sparse
103	max_pco2_0_6h	8909	0.740073	conditional_sparse
105	min_ph_0_6h	8866	0.736501	conditional_sparse
26	dt_first_niv_hours_model	6893	0.572603	conditional_sparse
25	dt_first_niv_hours	6877	0.571274	conditional_sparse
67	flag_chf	6710	0.557402	conditional_sparse
68	flag_copd	6710	0.557402	conditional_sparse
69	flag_neuromuscular	6710	0.557402	conditional_sparse
70	flag_opioid_substance	6710	0.557402	conditional_sparse
71	flag_osa_ohs	6710	0.557402	conditional_sparse
73	flag_pneumonia	6710	0.557402	conditional_sparse
61	first_lactate	5704	0.473833	conditional_sparse
96	lactate_band	5704	0.473833	conditional_sparse
24	dt_first_imv_hours_model	5315	0.441519	conditional_sparse
23	dt_first_imv_hours	5287	0.439193	conditional_sparse
35	ed_first_temp	5045	0.419090	conditional_sparse
15	bmi_closest_pre_ed_model	4754	0.394916	conditional_sparse
81	height_closest_pre_ed_model	4671	0.388021	conditional_sparse
14	bmi_closest_pre_ed	4638	0.385280	conditional_sparse

```

80          height_closest_pre_ed          4633          0.384865  conditional_sparse
58          first_gas_specimen_type          4379          0.363765  conditional_sparse
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project.
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 first_other_pco2_audit.csv
first_other_pco2 audit scope: uom_scope_df
Expected structural null fields: []
Unexpected full-null fields (top 20): []
UOM expectation scope: {'frame': 'final_cc_preview', 'row_count': 12020}
UOM expectation checks: {
  "expected_structural_null_fields": [
    "poc_abg_ph_uom",
    "poc_vbg_ph_uom",
    "poc_other_ph_uom"
  ],
  "structural_null_checks": {
    "poc_abg_ph_uom": {
      "present": true,
      "missing_n": 12020,
      "missing_pct": 1.0,
      "all_null": true
    },
    "poc_vbg_ph_uom": {
      "present": true,
      "missing_n": 12020,
      "missing_pct": 1.0,
      "all_null": true
    },
    "poc_other_ph_uom": {
      "present": true,
      "missing_n": 12020,
      "missing_pct": 1.0,
      "all_null": true
    }
  },
  "paco2_uom_checks": {
    "poc_abg_paco2_uom": {
      "present": true,
      "paired_value_column": "poc_abg_paco2",
      "value_rows": 6934,
      "missing_uom_when_value_present": 0,
      "non_mmhg_uom_when_value_present": 0,
      "passes": true
    }
  },

```

```

    "poc_vbg_paco2_uom": {
      "present": true,
      "paired_value_column": "poc_vbg_paco2",
      "value_rows": 5375,
      "missing_uom_when_value_present": 0,
      "non_mmhg_uom_when_value_present": 0,
      "passes": true
    },
    "poc_other_paco2_uom": {
      "present": true,
      "paired_value_column": "poc_other_paco2",
      "value_rows": 0,
      "missing_uom_when_value_present": 0,
      "non_mmhg_uom_when_value_present": 0,
      "passes": true
    }
  }
}
% any gas panel 0-6h: 41.8
% any gas panel 0-24h: 95.8
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 gas_source_overlap_summary.csv
% source unknown (panel-level): 66.3
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 pco2_source_distribution_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 other_route_quarantine_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 first_gas_anchor_audit.csv
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project/
CC-NLP/qa_summary.json

```

## 3.2 Outputs (ED-stay cohort + long tables)

**Rationale:** Persist ED-stay analytic datasets and supporting long tables.

```
# Purpose: Write finalized cohort outputs and derivative tables for
↳ downstream analysis workflows.
```

```
# Save outputs (dated parquet artifacts archived under prior runs)
from datetime import datetime
```

```

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

cohort_path = prior_runs_dir / f"cohort_ed_stay_{timestamp}.parquet"
ed_vitals_path = prior_runs_dir / f"ed_vitals_long_{timestamp}.parquet"
labs_path = prior_runs_dir / f"labs_long_{timestamp}.parquet"
gas_panels_path = prior_runs_dir / f"gas_panels_{timestamp}.parquet"

ed_df.to_parquet(cohort_path, index=False)
ed_vitals_long.to_parquet(ed_vitals_path, index=False)
labs_long.to_parquet(labs_path, index=False)
panel.to_parquet(gas_panels_path, index=False)

print("Wrote:", cohort_path)
print("Wrote:", ed_vitals_path)
print("Wrote:", labs_path)
print("Wrote:", gas_panels_path)

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/cohort_ed_stay_20260220_123028.parquet
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/ed_vitals_long_20260220_123028.parquet
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/labs_long_20260220_123028.parquet
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects/
CC-NLP/MIMIC tabular data/prior runs/gas_panels_20260220_123028.parquet

# Purpose: Report missingness for key variables to support transparent
↳ data-quality reporting.

# Also export to Excel (tabular dataset) - optional archive output
from datetime import datetime

WRITE_ARCHIVE_XLSX_EXPORTS = os.getenv("WRITE_ARCHIVE_XLSX_EXPORTS", "0") ==
↳ "1"
if WRITE_ARCHIVE_XLSX_EXPORTS:
    prior_runs_dir = DATA_DIR / "prior runs"
    prior_runs_dir.mkdir(parents=True, exist_ok=True)
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')

    xlsx_path = prior_runs_dir /
↳ f"mimic_hypercapp_EXT_EDstay_bq_gas_{timestamp}.xlsx"

```

```

with pd.ExcelWriter(xlsx_path, engine="openpyxl") as xw:
    ed_df.to_excel(xw, sheet_name="cohort_ed_stay", index=False)
    # Optional: include QA tables if present
    try:
        miss.to_excel(xw, sheet_name="missingness", index=False)
    except Exception:
        pass

    print("Saved:", xlsx_path)
else:
    print("Skipping ED-stay archive workbook export (set
    ↪ WRITE_ARCHIVE_XLSX_EXPORTS=1 to enable).")

Skipping ED-stay archive workbook export (set WRITE_ARCHIVE_XLSX_EXPORTS=1 to enable).

# Purpose: Capture ED presentation context (chief complaint, acuity, and
    ↪ early vitals) at the ED-stay level.

# Final Excel outputs requested: all encounters + ED chief-complaint only
from datetime import datetime

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')

# 1) All encounters meeting inclusion criteria (admission-level), with ED
    ↪ linkage flags
ed_link = (
    ed_df.groupby("hadm_id", as_index=False)
        .agg(ed_stay_id_first=("ed_stay_id", "first"),
    ↪ n_ed_stays=("ed_stay_id", "nunique"))
)
all_encounters = df.merge(ed_link, on="hadm_id", how="left")
all_encounters["has_ed_encounter"] =
    ↪ all_encounters["n_ed_stays"].fillna(0).astype(int).gt(0).astype(int)
all_encounters["encounter_source"] =
    ↪ all_encounters["has_ed_encounter"].map({1: "ED-linked", 0:
    ↪ "Inpatient-only"})

if WRITE_ARCHIVE_XLSX_EXPORTS:
    all_path = prior_runs_dir /
    ↪ f"mimic_hypercap_EXT_all_encounters_bq_{timestamp}.xlsx"

```



```

    with pd.ExcelWriter(all_path, engine="openpyxl") as xw:
        all_encounters.to_excel(xw, sheet_name="all_encounters", index=False)
    print("Saved:", all_path)

# 2) ED chief-complaint-only (ED-stay level)
if "ed_triage_cc" not in ed_df.columns:
    raise KeyError("Column 'ed_triage_cc' not found in ed_df. Ensure ED
        ↳ triage merge ran.")

mask_cc = ed_df["ed_triage_cc"].notna() &
    ↳ (ed_df["ed_triage_cc"].astype(str).str.strip() != "")
ed_cc_only = ed_df.loc[mask_cc].copy()

if WRITE_ARCHIVE_XLSX_EXPORTS:
    cc_path = prior_runs_dir /
    ↳ f"mimic_hypercap_EXT_EDcc_only_edstay_bq_{timestamp}.xlsx"
    with pd.ExcelWriter(cc_path, engine="openpyxl") as xw:
        ed_cc_only.to_excel(xw, sheet_name="ed_cc_only", index=False)
    print("Saved:", cc_path)
else:
    print("Skipping all-encounters/ED-CC archive exports (set
        ↳ WRITE_ARCHIVE_XLSX_EXPORTS=1 to enable).")

Skipping all-encounters/ED-CC archive exports (set WRITE_ARCHIVE_XLSX_EXPORTS=1 to enable).

# Purpose: Capture ED presentation context (chief complaint, acuity, and
    ↳ early vitals) at the ED-stay level.

# Final CC output (cohort-only; excludes NLP-derived columns)
from datetime import datetime

# Ensure cc-only frames exist (build from df if upstream optional export
    ↳ cells were skipped)
if "df_cc" not in globals():
    if "df" not in globals():
        raise NameError("df not found. Run cohort assembly cells first.")
    if "ed_triage_cc" not in df.columns:
        raise KeyError("ed_triage_cc missing in df; cannot build df_cc.")
    _mask_df_cc = df["ed_triage_cc"].notna() &
    ↳ (df["ed_triage_cc"].astype(str).str.strip() != "")
    df_cc = df.loc[_mask_df_cc].copy()

```

```

if "ed_cc_only" not in globals():
    if "ed_df" not in globals():
        raise NameError("ed_df not found. Run the ED-stay build cells
↪ first.")
    if "ed_triage_cc" not in ed_df.columns:
        raise KeyError("ed_triage_cc missing in ed_df; cannot build
↪ ed_cc_only.")
    _mask_cc = ed_df["ed_triage_cc"].notna() &
↪ (ed_df["ed_triage_cc"].astype(str).str.strip() != "")
    ed_cc_only = ed_df.loc[_mask_cc].copy()

# Add ED-stay columns (dedup to earliest ED stay per hadm_id)
ed_tmp = ed_cc_only.copy()
if "ed_intime" in ed_tmp.columns:
    ed_tmp = ed_tmp.sort_values(["hadm_id", "ed_intime", "ed_stay_id"],
↪ na_position="last")
else:
    ed_tmp = ed_tmp.sort_values(["hadm_id", "ed_stay_id"],
↪ na_position="last")
ed_first = ed_tmp.drop_duplicates("hadm_id", keep="first")

add_ed_cols = [c for c in ed_first.columns if c not in df_cc.columns]

final_cc = df_cc.merge(ed_first[["hadm_id"] + add_ed_cols], on="hadm_id",
↪ how="left")

prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)
out_date = datetime.now().strftime("%Y-%m-%d")

cohort_contract = validate_cohort_contract(final_cc)
cohort_contract_report = {
    "generated_utc": datetime.utcnow().isoformat() + "Z",
    "status": cohort_contract["status"],
    "contracts": {"cohort": cohort_contract},
    "findings": cohort_contract["findings"],
}
cohort_contract_paths = write_contract_report(cohort_contract_report,
↪ work_dir=WORK_DIR)
print("Wrote:", cohort_contract_paths["contract_report_path"])
if cohort_contract_paths["failed_contract_path"].exists():
    print("Wrote:", cohort_contract_paths["failed_contract_path"])

```

```

contract_mode = os.getenv("PIPELINE_CONTRACT_MODE", "fail").strip().lower()
if contract_mode not in {"fail", "warn"}:
    raise ValueError("PIPELINE_CONTRACT_MODE must be one of {'fail',
↪ 'warn'}")
if cohort_contract["status"] == "fail" and contract_mode == "fail":
    raise ValueError(
        "Cohort contract checks failed. "
        f"See {cohort_contract_paths['failed_contract_path']}"
    )

dated_out_path = prior_runs_dir / f"{out_date} MIMICIV all with CC.xlsx"
canonical_out_path = DATA_DIR / "MIMICIV all with CC.xlsx"

# Write via temporary files then atomically replace targets to avoid
↪ zero-byte outputs on interruption.
def _atomic_write_excel(df_obj, target_path: Path) -> None:
    tmp_path = target_path.with_suffix(target_path.suffix + ".tmp")
    df_obj.to_excel(tmp_path, index=False)
    os.replace(tmp_path, target_path)

def _sha256_file(path: Path) -> str:
    hasher = hashlib.sha256()
    with path.open("rb") as file_handle:
        for chunk in iter(lambda: file_handle.read(1024 * 1024), b''):
            hasher.update(chunk)
    return hasher.hexdigest()

_atomic_write_excel(final_cc, dated_out_path)
_atomic_write_excel(final_cc, canonical_out_path)
canonical_sha256 = _sha256_file(canonical_out_path)
dated_sha256 = _sha256_file(dated_out_path)
print(
    "EXPORT_SHAPE cohort",
    f"rows={len(final_cc):,}",
    f"cols={final_cc.shape[1]:,}",
    f"canonical_sha256={canonical_sha256}",
)

cohort_manifest = collect_run_manifest(
    WORK_DIR,
    run_id=f"cohort_{datetime.now().strftime('%Y%m%d_%H%M%S')}",
)

```

```

cohort_manifest["stage"] = "cohort"
cohort_manifest["outputs"] = {
    "canonical_output_path": str(canonical_out_path),
    "canonical_output_sha256": canonical_sha256,
    "dated_output_path": str(dated_out_path),
    "dated_output_sha256": dated_sha256,
    "gas_source_audit_path": str(gas_source_audit_path) if
        ↪ "gas_source_audit_path" in globals() else None,
    "gas_source_overlap_summary_path": str(gas_overlap_summary_path) if
        ↪ "gas_overlap_summary_path" in globals() else None,
    "anthropometrics_coverage_audit_path": str(anthro_coverage_audit_path) if
        ↪ "anthro_coverage_audit_path" in globals() else None,
    "ed_vitals_distribution_summary_path":
        ↪ str(ed_vitals_distribution_summary_path) if
        ↪ "ed_vitals_distribution_summary_path" in globals() else None,
    "ed_vitals_extreme_examples_path": str(ed_vitals_extreme_examples_path)
        ↪ if "ed_vitals_extreme_examples_path" in globals() else None,
    "ed_vitals_model_delta_path": str(ed_vitals_model_delta_path) if
        ↪ "ed_vitals_model_delta_path" in globals() else None,
    "outliers_audit_path": str(outliers_audit_path) if "outliers_audit_path"
        ↪ in globals() else None,
}
cohort_manifest["gas_source"] = {
    "mode": gas_source_mode if "gas_source_mode" in globals() else None,
    "source_rates": gas_source_audit.get("source_rates", {}) if
        ↪ "gas_source_audit" in globals() else {},
    "tier_rates": gas_source_audit.get("tier_rates", {}) if
        ↪ "gas_source_audit" in globals() else {},
}
cohort_manifest["anthropometrics"] = (
    anthro_coverage_audit if "anthro_coverage_audit" in globals() else None
)
cohort_manifest_path = prior_runs_dir / f"{out_date}"
    ↪ cohort_run_manifest.json"
cohort_manifest_path.write_text(json.dumps(cohort_manifest, indent=2))

print("Saved:", dated_out_path)
print("Saved:", canonical_out_path)
print("Wrote:", cohort_manifest_path)
print("Base rows:", len(df_cc), "| Added from ED-stay:", len(add_ed_cols))
print("Total columns:", len(final_cc.columns))

```

Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Projects

```

CC-NLP/debug/contracts/20260220_193029/contract_report.json
EXPORT_SHAPE cohort rows=12,020 cols=293 canonical_sha256=05d9dcd5be171a20fed4807173492aa7ab
Saved: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 MIMICIV all with CC.xlsx
Saved: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project
CC-NLP/MIMIC tabular data/MIMICIV all with CC.xlsx
Wrote: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Project
CC-NLP/MIMIC tabular data/prior runs/2026-02-20 cohort_run_manifest.json
Base rows: 12020 | Added from ED-stay: 182
Total columns: 293

```

```

# Purpose: Extract and standardize first ABG/VBG physiology fields for
↳ baseline characterization.

# Data dictionary (OSF-style) for final cohort output
from datetime import datetime
import numpy as np

# Choose target dataset for dictionary
if "final_cc" in globals():
    dd_target = final_cc
elif "df_cc" in globals():
    dd_target = df_cc
elif "df" in globals():
    dd_target = df
else:
    raise NameError("No cohort dataframe found for data dictionary (expected
↳ final_cc/df_cc/df).")

# Helper functions

def _readable(name: str) -> str:
    return name.replace("_", " ").strip().title()

def _infer_units(col: str) -> str:
    if col.endswith("_hours"):
        return "hours"
    if col.endswith("_time") or col.endswith("_intime") or
↳ col.endswith("_outtime") or col.endswith("_date"):
        return "datetime"
    if col.startswith("flag_") or col.endswith("_flag") or
↳ col.endswith("_before_imv"):

```

```

        return "binary (0/1)"
    if col.endswith("_paco2") or col.endswith("_pco2"):
        return "mmHg"
    if col.endswith("_ph"):
        return "pH"
    if col.endswith("_lactate"):
        return "mmol/L"
    return ""

def _allowed_values(series):
    s = series.dropna()
    if s.empty:
        return ""
    if s.dtype.kind in "biu":
        vals = sorted(map(int, s.unique()))
        if len(vals) <= 15:
            return ", ".join(map(str, vals))
        return f"{min(vals)}--{max(vals)}"
    if s.dtype.kind in "f":
        return f"{np.nanmin(s):.3g}-{np.nanmax(s):.3g}"
    # object/categorical: use bounded sampling to keep dictionary generation
    ↪ fast.
    sample_n = 5000
    s_str = s.astype(str)
    if len(s_str) > sample_n:
        s_sample = s_str.head(sample_n)
        sample_note = f"sampled {sample_n}/{len(s_str)} rows"
    else:
        s_sample = s_str
        sample_note = None
    unique_n = int(s_sample.nunique(dropna=True))
    top_vals = s_sample.value_counts(dropna=False).head(15).index.tolist()
    if unique_n > len(top_vals):
        tail = f" ... ({unique_n} unique"
        if sample_note:
            tail += f"; {sample_note}"
        tail += ")"
        return ", ".join(top_vals) + tail
    if sample_note:
        return ", ".join(top_vals) + f" ({sample_note})"
    return ", ".join(top_vals)

```

```

META = {
  # IDs
  "subject_id": {
    "label": "Subject identifier",
    "definition": "Identifier for a unique patient in MIMIC-IV.",
    "source": "mimiciv_hosp.patients",
    "derivation": "as recorded",
  },
  "hadm_id": {
    "label": "Hospital admission identifier",
    "definition": "Identifier for a hospital admission.",
    "source": "mimiciv_hosp.admissions",
    "derivation": "as recorded",
  },
  "ed_stay_id": {
    "label": "ED stay identifier",
    "definition": "Identifier for an ED stay.",
    "source": "mimiciv_ed.edstays",
    "derivation": "as recorded",
  },
  "ed_intime": {
    "label": "ED arrival time",
    "definition": "ED arrival time for this ED stay.",
    "source": "mimiciv_ed.edstays",
    "derivation": "as recorded",
    "units": "datetime",
  },
  "ed_outtime": {
    "label": "ED departure time",
    "definition": "ED departure time for this ED stay.",
    "source": "mimiciv_ed.edstays",
    "derivation": "as recorded",
    "units": "datetime",
  },
  "ed_intime_first": {
    "label": "First ED arrival time (admission)",
    "definition": "Earliest ED arrival time among ED stays linked to the
↪ admission.",
    "source": "mimiciv_ed.edstays",
    "derivation": "min(edstays.intime) per hadm_id",
    "units": "datetime",
  },
},

```

```

# Demographics / triage
"ed_gender": {
  "label": "ED gender",
  "definition": "Gender as recorded in ED stay table.",
  "source": "mimiciv_ed.edstays",
  "derivation": "as recorded",
},
"ed_race": {
  "label": "ED race",
  "definition": "Race as recorded in ED stay table.",
  "source": "mimiciv_ed.edstays",
  "derivation": "as recorded",
},
"hosp_race": {
  "label": "Hospital race",
  "definition": "Race as recorded in admissions table.",
  "source": "mimiciv_hosp.admissions",
  "derivation": "as recorded",
},
"ed_triage_cc": {
  "label": "ED chief complaint",
  "definition": "Chief complaint recorded at ED triage.",
  "source": "mimiciv_ed.triage",
  "derivation": "as recorded",
},
"ed_triage_acuity": {
  "label": "ED triage acuity",
  "definition": "Triage acuity level recorded at ED presentation.",
  "source": "mimiciv_ed.triage",
  "derivation": "as recorded",
},
# Outcomes
"hospital_expire_flag": {
  "label": "Hospital expire flag",
  "definition": "Hospital mortality indicator from admissions table.",
  "source": "mimiciv_hosp.admissions",
  "derivation": "as recorded",
  "units": "binary (0/1)",
},
"in_hospital_death": {
  "label": "In-hospital death",
  "definition": "Indicator for in-hospital death during admission.",
  "source": "mimiciv_hosp.admissions",

```



```

        "derivation": "(hospital_expire_flag == 1) OR (deathtime not null)",
        "units": "binary (0/1)",
    },
    # Ventilation
    "imv_flag": {
        "label": "IMV flag",
        "definition": "Indicator of invasive mechanical ventilation during
        ↪ admission.",
        "source": "ICD procedures + ICU chartevents",
        "derivation": "max(IMV ICD flag, IMV chart flag)",
        "units": "binary (0/1)",
    },
    "niv_flag": {
        "label": "NIV flag",
        "definition": "Indicator of noninvasive ventilation during
        ↪ admission.",
        "source": "ICD procedures + ICU chartevents",
        "derivation": "max(NIV ICD flag, NIV chart flag)",
        "units": "binary (0/1)",
    },
    "first_imv_time": {
        "label": "First IMV time",
        "definition": "Earliest charted time of IMV in ICU chartevents (if
        ↪ present).",
        "source": "mimiciv_icu.chartevents",
        "derivation": "min charttime among IMV charted events",
        "units": "datetime",
    },
    "first_niv_time": {
        "label": "First NIV time",
        "definition": "Earliest charted time of NIV in ICU chartevents (if
        ↪ present).",
        "source": "mimiciv_icu.chartevents",
        "derivation": "min charttime among NIV charted events",
        "units": "datetime",
    },
    "dt_first_imv_hours": {
        "label": "Hours to first IMV",
        "definition": "Hours from first ED arrival to first IMV time.",
        "source": "Derived",
        "derivation": "(first_imv_time - ed_intime_first) in hours",
        "units": "hours",
    },
},

```

```

"dt_first_niv_hours": {
  "label": "Hours to first NIV",
  "definition": "Hours from first ED arrival to first NIV time.",
  "source": "Derived",
  "derivation": "(first_niv_time - ed_intime_first) in hours",
  "units": "hours",
},
"abg_before_imv": {
  "label": "ABG before IMV",
  "definition": "Indicator that first ABG time precedes first IMV
↪ time.",
  "source": "Derived",
  "derivation": "first_abg_time < first_imv_time",
  "units": "binary (0/1)",
},
"vbg_before_imv": {
  "label": "VBG before IMV",
  "definition": "Indicator that first VBG time precedes first IMV
↪ time.",
  "source": "Derived",
  "derivation": "first_vbg_time < first_imv_time",
  "units": "binary (0/1)",
},
# Gas/ABG/VBG
"abg_hypercap_threshold": {
  "label": "ABG hypercapnia threshold met",
  "definition": "Indicator that any arterial pCO2 45 mmHg in hosp/ICU
↪ pCO2 extraction.",
  "source": "mimiciv_hosp.labevents + mimiciv_icu.charthevents",
  "derivation": "max(arterial pCO2 45) per hadm_id",
  "units": "binary (0/1)",
},
"vbg_hypercap_threshold": {
  "label": "VBG hypercapnia threshold met",
  "definition": "Indicator that any venous pCO2 50 mmHg in hosp/ICU
↪ pCO2 extraction.",
  "source": "mimiciv_hosp.labevents + mimiciv_icu.charthevents",
  "derivation": "max(venous pCO2 50) per hadm_id",
  "units": "binary (0/1)",
},
"other_hypercap_threshold": {
  "label": "Other-source hypercapnia threshold met",

```

```

    "definition": "Indicator that any other/unspecified-source pCO2 50
    ↪ mmHg in hosp/ICU pCO2 extraction.",
    "source": "mimiciv_hosp.labevents + mimiciv_icu.charthevents",
    "derivation": "max(other-source pCO2 50) per hadm_id",
    "units": "binary (0/1)",
  },
  "first_gas_time": {
    "label": "First gas panel time",
    "definition": "Earliest gas panel time in the ED 0-24h window.",
    "source": "mimiciv_hosp.labevents (gas panels)",
    "derivation": "min_panel_time within ED 0-24h window",
    "units": "datetime",
  },
  "dt_first_qualifying_gas_hours": {
    "label": "Hours to first gas panel",
    "definition": "Hours from first ED arrival to first gas panel time.",
    "source": "Derived",
    "derivation": "(first_gas_time - ed_intime_first) in hours",
    "units": "hours",
  },
  "first_other_time": {
    "label": "First other-source gas time",
    "definition": "Earliest time of a pCO2 measurement with
    ↪ other/unspecified source classification.",
    "source": "mimiciv_hosp.labevents + mimiciv_icu.charthevents",
    "derivation": "min(other-source pCO2 charttime) across LAB and POC",
    "units": "datetime",
  },
  "presenting_hypercapnia": {
    "label": "Presenting (6h) gas timing",
    "definition": "Indicator that first gas panel occurred within 6h of
    ↪ first ED arrival.",
    "source": "Derived",
    "derivation": "dt_first_qualifying_gas_hours 6",
    "units": "binary (0/1)",
  },
  "late_hypercapnia": {
    "label": "Late (>6h) gas timing",
    "definition": "Indicator that first gas panel occurred >6h after
    ↪ first ED arrival.",
    "source": "Derived",
    "derivation": "dt_first_qualifying_gas_hours > 6",
    "units": "binary (0/1)",
  }

```

```

},
"gas_source_other_rate": {
  "label": "Gas source other/unknown rate",
  "definition": "Proportion of gas panels classified as
    ↪ other/unspecified source per ED stay.",
  "source": "Derived from gas panel source inference",
  "derivation": "mean(source == 'other') per ed_stay_id",
  "units": "proportion",
},
# Comorbidities
"flag_copd": {
  "label": "COPD/emphysema flag",
  "definition": "Indicator for COPD/emphysema ICD codes in ED or
    ↪ hospital diagnoses.",
  "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
  "derivation": "ICD prefixes J43/J44 (ED OR hospital)",
  "units": "binary (0/1)",
},
"flag_osa_ohs": {
  "label": "OSA/OHS flag",
  "definition": "Indicator for OSA/OHS ICD codes in ED or hospital
    ↪ diagnoses.",
  "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
  "derivation": "ICD prefixes G473/E662 (ED OR hospital)",
  "units": "binary (0/1)",
},
"flag_chf": {
  "label": "CHF flag",
  "definition": "Indicator for CHF ICD codes in ED or hospital
    ↪ diagnoses.",
  "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
  "derivation": "ICD prefix I50 (ED OR hospital)",
  "units": "binary (0/1)",
},
"flag_neuromuscular": {
  "label": "Neuromuscular flag",
  "definition": "Indicator for neuromuscular ICD codes in ED or
    ↪ hospital diagnoses.",
  "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
  "derivation": "ICD prefixes G12/G70/G71 (ED OR hospital)",
  "units": "binary (0/1)",
},
"flag_opioid_substance": {

```

```

        "label": "Opioid/substance flag",
        "definition": "Indicator for opioid/substance ICD codes in ED or
↪ hospital diagnoses.",
        "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
        "derivation": "ICD prefixes F11/T40/F13 (ED OR hospital)",
        "units": "binary (0/1)",
    },
    "flag_pneumonia": {
        "label": "Pneumonia flag",
        "definition": "Indicator for pneumonia ICD codes in ED or hospital
↪ diagnoses.",
        "source": "mimiciv_hosp.diagnoses_icd + mimiciv_ed.diagnosis",
        "derivation": "ICD prefixes J12-J18 (ED OR hospital)",
        "units": "binary (0/1)",
    },
}

rows = []
for col in dd_target.columns:
    info = META.get(col, {})
    series = dd_target[col]
    rows.append({
        "variable": col,
        "label": info.get("label", _readable(col)),
        "units": info.get("units", _infer_units(col)),
        "allowed_values": info.get("allowed_values",
↪ _allowed_values(series)),
        "definition": info.get("definition", "UNCONFIRMED"),
        "synonyms": info.get("synonyms", ""),
        "description": info.get("description", ""),
        "source": info.get("source", "UNCONFIRMED"),
        "derivation": info.get("derivation", "UNCONFIRMED"),
        "dtype": str(series.dtype),
        "example_value": series.dropna().iloc[0] if series.dropna().shape[0]
↪ else None,
    })

data_dict = pd.DataFrame(rows)

out_date = datetime.now().strftime("%Y-%m-%d")
prior_runs_dir = DATA_DIR / "prior runs"
prior_runs_dir.mkdir(parents=True, exist_ok=True)

```

```
out_xlsx = prior_runs_dir / f"{out_date} MIMICIV all with  
↳ CC_data_dictionary.xlsx"  
out_csv = prior_runs_dir / f"{out_date} MIMICIV all with  
↳ CC_data_dictionary.csv"  
  
data_dict.to_excel(out_xlsx, index=False)  
data_dict.to_csv(out_csv, index=False)  
  
print("Saved data dictionary:", out_xlsx)  
print("Saved data dictionary:", out_csv)  
print("Rows:", len(data_dict))
```