Medical-Evidence-Synthesizer

A Multimodal Diagnostic RAG for Complex Patient Cases

For our final semester long project, we will develop a comprehensive tutorial on building a clinical decision support tool called the "Medical Evidence Synthesizer." This project will guide users through the data science pipeline to create a Retrieval Augmented Generation (RAG) model. The model will ingest complex, de-identified patient data combining structured lab results and unstructured clinical notes to generate a differential diagnosis. A key feature of this tool is its ability to provide verifiable, citable evidence for each diagnostic possibility directly from the source data, showcasing an end-to-end data science solution with a focus on machine learning and explainability.

Structured Data Preprocessing & Cleaning

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, cal
l drive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
%cd /content/drive/MyDrive/MSML_602_FP/Dataset/workspace_data/
%pwd
```

```
/content/drive/MyDrive/MSML_602_FP/Dataset/workspace_data
```

Out[ ]:
```
'/content/drive/MyDrive/MSML_602_FP/Dataset/workspace_data'
```

In [ ]:
```
!ls
```

```
admissions.csv       icustays.csv   omr.csv       processed
d_icd_diagnoses.csv  labevents.csv  patients.csv
```

In [ ]:
```
from __future__ import annotations

import json
from dataclasses import dataclass
from pathlib import Path
from typing import Dict, Iterable, List, Tuple

import numpy as np
import pandas as pd

try:
    from sklearn.feature_selection import mutual_info_classif

    SKLEARN_AVAILABLE = True
except ImportError:
    SKLEARN_AVAILABLE = False
```

```
# Set base directory to the specified folder in Google Drive
BASE_DIR = Path("/content/drive/MyDrive/MSML_602_FP/Dataset/workspace_data")
PROCESSED_DIR = BASE_DIR / "processed"

# Define raw file paths relative to the base directory
RAW_FILES = {
    "admissions": BASE_DIR / "admissions.csv",
    "patients": BASE_DIR / "patients.csv",
    "icustays": BASE_DIR / "icustays.csv",
    "labevents": BASE_DIR / "labevents.csv",
    "omr": BASE_DIR / "omr.csv",
    "diagnoses": BASE_DIR / "d_icd_diagnoses.csv",
}

# Create the processed directory if it doesn't exist
PROCESSED_DIR.mkdir(parents=True, exist_ok=True)
```

In [ ]:
```
from pathlib import Path

# Define input and output paths
base_dir = Path("/content/sample_data")
processed_dir = Path("/content/processed")
processed_dir.mkdir(exist_ok=True)

discharge_input = base_dir / "discharge.csv" # Assuming discharge.csv is in
discharge_output = processed_dir / "discharge_cleaned.csv"

radiology_input = base_dir / "radiology.csv" # Assuming radiology.csv is in
radiology_output = processed_dir / "radiology_cleaned.csv"

# Call the cleaning functions
# Make sure to run the cells defining clean_discharge_notes and clean_radiol
# clean_discharge_notes(discharge_input, discharge_output)
# clean_radiology_notes(radiology_input, radiology_output)

print("Cleaning functions are ready to be called. Uncomment the lines above
```

```
Cleaning functions are ready to be called. Uncomment the lines above and run
this cell to execute them.
```

Utility helpers

In [ ]:
```
def _ensure_int(series: pd.Series) -> pd.Series:
    """Return a nullable integer series; preserves missing values."""

    return pd.to_numeric(series, errors="coerce").astype("Int64")


def _safe_to_numeric(series: pd.Series) -> pd.Series:
    """Convert object series to float when possible, otherwise NaN."""

    if series.dtype.kind in {"i", "u", "f"}:
        return series
    return pd.to_numeric(series.astype(str).str.replace(",", ""), errors="co
```

```python
def _duration_hours(end: pd.Series, start: pd.Series) -> pd.Series:
    """Compute duration in hours between two datetime series."""

    return (end - start).dt.total_seconds() / 3600.0


def _duration_minutes(end: pd.Series, start: pd.Series) -> pd.Series:
    return (end - start).dt.total_seconds() / 60.0


def _recent_records(df: pd.DataFrame, group_cols: List[str], timestamp_col:
    """Return the most recent row per group based on timestamp_col."""

    idx = df.groupby(group_cols)[timestamp_col].transform("idxmax")
    return df.loc[idx].reset_index(drop=True)


def _save_table(df: pd.DataFrame, filename: str) -> str:
    """Persist DataFrame as parquet if available; otherwise CSV.

    Returns the final filename that was written.
    """

    target = PROCESSED_DIR / filename
    suffix = target.suffix.lower()
    if suffix == ".parquet":
        try:
            df.to_parquet(target, index=False)
            return target.name
        except ImportError:
            csv_target = target.with_suffix(".csv")
            df.to_csv(csv_target, index=False)
            return csv_target.name
    elif suffix == ".csv":
        df.to_csv(target, index=False)
        return target.name
    else:
        raise ValueError(f"Unsupported file extension for {target}")
```

Admissions

In [ ]:
```python
def load_and_clean_admissions(path: Path) -> pd.DataFrame:
    df = pd.read_csv(
        path,
        parse_dates=[
            "admittime",
            "dischtime",
            "deathtime",
            "edregtime",
            "edouttime",
        ],
        na_values=["", "NA", "NaN", "?"],
    )

    df = df.drop_duplicates(subset=["hadm_id"], keep="last")
    df["subject_id"] = _ensure_int(df["subject_id"])
    df["hadm_id"] = _ensure_int(df["hadm_id"])

    # Feature engineering
    df["los_hours"] = _duration_hours(df["dischtime"], df["admittime"])
    df.loc[df["los_hours"] < 0, "los_hours"] = np.nan  # guard bad timestamp

    df["ed_wait_minutes"] = _duration_minutes(df["admittime"], df["edregtime
    df["ed_stay_minutes"] = _duration_minutes(df["edouttime"], df["edregtime

    df["died_in_hospital"] = df["hospital_expire_flag"].fillna(0).astype("In

    # Harmonise key categoricals
    categorical_cols = [
        "admission_type",
        "admission_location",
        "discharge_location",
        "insurance",
        "language",
        "marital_status",
        "race",
    ]
    for col in categorical_cols:
        df[col] = (
            df[col]
            .astype(str)
            .str.strip()
            .str.upper()
            .replace({"NAN": np.nan, "?": np.nan, "": np.nan})
        )

    df["marital_status"] = df["marital_status"].fillna("UNKNOWN")
    df["language"] = df["language"].fillna("UNKNOWN")

    return df
```

Patients

```
In [ ]: def load_and_clean_patients(path: Path) -> pd.DataFrame:
    df = pd.read_csv(path, parse_dates=["dod"], na_values=["", "NA", "?"])
    df = df.drop_duplicates(subset=["subject_id"], keep="last")

    df["subject_id"] = _ensure_int(df["subject_id"])
    df["anchor_age"] = pd.to_numeric(df["anchor_age"], errors="coerce")
    df["anchor_year"] = pd.to_numeric(df["anchor_year"], errors="coerce")

    df["dod_available"] = df["dod"].notna().astype("Int64")

    age_bins = [0, 40, 60, 75, 200]
    age_labels = ["<40", "40-59", "60-74", "75+"]
    df["age_group"] = pd.cut(df["anchor_age"], bins=age_bins, labels=age_lab

    df["gender"] = df["gender"].str.upper().str.strip()

    return df
```

ICU stays

In [ ]:
```
def load_and_aggregate_icustays(path: Path) -> pd.DataFrame:
    df = pd.read_csv(
        path,
        parse_dates=["intime", "outtime"],
        na_values=["", "NA", "NaN"],
    )

    df["subject_id"] = _ensure_int(df["subject_id"])
    df["hadm_id"] = _ensure_int(df["hadm_id"])
    df["stay_id"] = _ensure_int(df["stay_id"])

    df["los"] = pd.to_numeric(df["los"], errors="coerce")

    aggregations = {
        "stay_id": "count",
        "los": ["sum", "mean", "max"],
    }
    grouped = df.groupby("hadm_id").agg(aggregations)
    grouped.columns = [
        "icu_stay_count",
        "icu_los_hours_sum",
        "icu_los_hours_mean",
        "icu_los_hours_max",
    ]

    first_last_units = df.sort_values("intime").groupby("hadm_id").agg(
        first_careunit_first=("first_careunit", "first"),
        first_careunit_last=("first_careunit", "last"),
        last_careunit_first=("last_careunit", "first"),
        last_careunit_last=("last_careunit", "last"),
    )

    aggregated = grouped.join(first_last_units, how="left")
    aggregated.reset_index(inplace=True)

    return aggregated
```

Lab events

In [ ]:
```
@dataclass
class LabFeatureSpec:
    itemid: int
    name: str


def _select_lab_items(
    df: pd.DataFrame, eligible_hadm: Iterable[int], top_n: int = 15, min_cov
) -> List[LabFeatureSpec]:
    hadm_set = pd.Index(eligible_hadm)
    coverage = (
        df.dropna(subset=["hadm_id"])
        .groupby("itemid")["hadm_id"]
        .nunique()
        .sort_values(ascending=False)
    )
    coverage_ratio = coverage / hadm_set.nunique()

    selected_itemids = coverage_ratio[coverage_ratio >= min_coverage].head(t
    specs = [LabFeatureSpec(int(item), f"lab_{item}") for item in selected_i
    return specs


def load_and_aggregate_labs(path: Path, eligible_hadm: Iterable[int]) -> Tup
    df = pd.read_csv(
        path,
        parse_dates=["charttime", "storetime"],
        na_values=["", "NA", "NaN", "?", "___"],
    )

    df["subject_id"] = _ensure_int(df["subject_id"])
    df["hadm_id"] = _ensure_int(df["hadm_id"])

    # Prefer valuenum; fall back to parsed value
    df["value_num_clean"] = df["valuenum"]
    missing_mask = df["value_num_clean"].isna()
    df.loc[missing_mask, "value_num_clean"] = _safe_to_numeric(df.loc[missin

    df = df.dropna(subset=["hadm_id", "value_num_clean", "charttime"])

    specs = _select_lab_items(df, eligible_hadm)
    if not specs:
        return pd.DataFrame(columns=["hadm_id"]), []

    frames = []
    for spec in specs:
        subset = df[df["itemid"] == spec.itemid].copy()
        if subset.empty:
            continue
        subset.sort_values("charttime", inplace=True)

        agg = subset.groupby("hadm_id").agg(
            **{
                f"{spec.name}_count": ("value_num_clean", "count"),
                f"{spec.name}_mean": ("value_num_clean", "mean"),
                f"{spec.name}_std": ("value_num_clean", "std"),
                f"{spec.name}_min": ("value_num_clean", "min"),
                f"{spec.name}_max": ("value_num_clean", "max"),
            }
        )

        last_values = (
            subset.sort_values("charttime")
            .groupby("hadm_id", as_index=False)
            .tail(1)
            .set_index("hadm_id")["value_num_clean"]
        )
        agg[f"{spec.name}_last"] = agg.index.map(last_values)
        frames.append(agg)

    if not frames:
        return pd.DataFrame(columns=["hadm_id"]), []

    lab_features = pd.concat(frames, axis=1)
    lab_features.reset_index(inplace=True)
    lab_features = lab_features.loc[:, ~lab_features.columns.duplicated()]

    return lab_features, specs
```

Outpatient (OMR) measurements

In [ ]:
```
def _parse_blood_pressure(value: str) -> Tuple[float | None, float | None]:
    if not isinstance(value, str):
        return (np.nan, np.nan)
    parts = value.replace(" ", "").split("/")
    if len(parts) != 2:
        return (np.nan, np.nan)
    systolic = pd.to_numeric(parts[0], errors="coerce")
    diastolic = pd.to_numeric(parts[1], errors="coerce")
    return (systolic, diastolic)


def load_and_aggregate_omr(path: Path) -> pd.DataFrame:
    df = pd.read_csv(path, parse_dates=["chartdate"], na_values=["", "NA", "
    df["subject_id"] = _ensure_int(df["subject_id"])

    df["result_name"] = df["result_name"].str.strip().str.upper()

    numeric_mask = df["result_name"].isin(
        [
            "HEIGHT (INCHES)",
            "WEIGHT (LBS)",
            "BMI (KG/M2)",
            "WEIGHT (LBS)",
        ]
    )
    df.loc[numeric_mask, "result_value_numeric"] = _safe_to_numeric(df.loc[r

    latest = _recent_records(df, ["subject_id", "result_name"], "chartdate")

    # Convert to feature columns
    features: Dict[str, pd.Series] = {"subject_id": latest["subject_id"]}

    def _assign_feature(name: str, mask: pd.Series, values: pd.Series) -> No
        colname = f"omr_{name}"
        features[colname] = values.where(mask).groupby(latest["subject_id"])

    height_mask = latest["result_name"] == "HEIGHT (INCHES)"
    height_cm = latest.loc[height_mask, "result_value_numeric"] * 2.54
    _assign_feature("height_cm", height_mask, height_cm)

    weight_mask = latest["result_name"] == "WEIGHT (LBS)"
    weight_kg = latest.loc[weight_mask, "result_value_numeric"] * 0.45359237
    _assign_feature("weight_kg", weight_mask, weight_kg)

    bmi_mask = latest["result_name"] == "BMI (KG/M2)"
    bmi_val = latest.loc[bmi_mask, "result_value_numeric"]
    _assign_feature("bmi", bmi_mask, bmi_val)

    bp_mask = latest["result_name"] == "BLOOD PRESSURE"
    bp_vals = latest.loc[bp_mask, "result_value"].apply(_parse_blood_pressur
    if not bp_vals.empty:
        systolic = bp_vals.apply(lambda x: x[0])
        diastolic = bp_vals.apply(lambda x: x[1])
        _assign_feature("blood_pressure_systolic", bp_mask, systolic)
        _assign_feature("blood_pressure_diastolic", bp_mask, diastolic)

    result = pd.DataFrame(features).groupby("subject_id").first().reset_inde
    return result
```

Feature assembly & selection

In [ ]:

```
import pandas as pd
import numpy as np
from typing import List, Tuple, Dict
from dataclasses import dataclass

# Assuming mutual_info_classif is needed and SKLEARN_AVAILABLE is defined el
# from sklearn.feature_selection import mutual_info_classif

def assemble_feature_table(
    admissions: pd.DataFrame,
    patients: pd.DataFrame,
    icu: pd.DataFrame,
    lab: pd.DataFrame,
    omr: pd.DataFrame,
) -> pd.DataFrame:
    df = admissions.merge(patients, on="subject_id", how="left", suffixes=("

    if not icu.empty:
        df = df.merge(icu, on="hadm_id", how="left")

    if not lab.empty:
        df = df.merge(lab, on="hadm_id", how="left")

    if not omr.empty:
        df = df.merge(omr, on="subject_id", how="left")

    # Drop columns with excessive missingness (> 70%)
    missing_ratio = df.isna().mean()
    keep_cols = missing_ratio[missing_ratio <= 0.7].index.tolist()
    df = df[keep_cols]

    return df


def _prepare_numeric_matrix(df: pd.DataFrame, target_col: str) -> Tuple[pd.D
    feature_df = df.drop(columns=[target_col]).copy()
    target = df[target_col].astype("Int64").fillna(0)

    # Exclude identifiers from modeling features
    identifier_cols = [col for col in feature_df.columns if col in {"subject
    feature_df = feature_df.drop(columns=identifier_cols, errors="ignore")

    # Convert datetime and timedelta columns to numeric representations
    datetime_cols = feature_df.select_dtypes(include=["datetime64[ns]", "dat
    for col in datetime_cols:
        feature_df[col] = feature_df[col].apply(lambda x: x.value if pd.notr

    timedelta_cols = feature_df.select_dtypes(include=["timedelta64[ns]"]).c
    for col in timedelta_cols:
        feature_df[col] = feature_df[col].dt.total_seconds()

    # One-hot encode categorical variables using a potentially more memory-e
    categorical_cols = feature_df.select_dtypes(include=["object", "category
    # Use get_dummies with sparse=True if available in your pandas version a
    # feature_df = pd.get_dummies(feature_df, columns=categorical_cols, dumm
    # For broader compatibility, stick to standard get_dummies but be mindfu
    feature_df = pd.get_dummies(feature_df, columns=categorical_cols, dummy_


    # Impute missing values *after* one-hot encoding to handle new dummy col
    # Use median imputation for numeric columns
    for col in feature_df.select_dtypes(include=[np.number]).columns:
        if feature_df[col].isnull().any():
            feature_df[col].fillna(feature_df[col].median(), inplace=True)

    # Drop constant columns
    nunique = feature_df.nunique()
    feature_df = feature_df.loc[:, nunique > 1]

    return feature_df, target


def select_features(df: pd.DataFrame, target_col: str, k: int = 20) -> Tuple
    global SKLEARN_AVAILABLE # Declare SKLEARN_AVAILABLE as global
    X, y = _prepare_numeric_matrix(df, target_col)

    if X.empty:
        return df[["subject_id", "hadm_id", target_col]].copy(), [], {}

    feature_scores: Dict[str, float]

    # Check if SKLEARN_AVAILABLE and mutual_info_classif are available befor
    # Assuming SKLEARN_AVAILABLE is a boolean defined in a previous cell
    # Assuming mutual_info_classif is imported in a previous cell if SKLEARN
    if 'SKLEARN_AVAILABLE' in globals() and SKLEARN_AVAILABLE and y.nunique(
        try:
            from sklearn.feature_selection import mutual_info_classif
            scores = mutual_info_classif(X, y)
            feature_scores = dict(zip(X.columns, scores))
        except ImportError:
            print("Scikit-learn not available or mutual_info_classif not imp
            SKLEARN_AVAILABLE = False # Update flag if import fails
            corrs = {}
            for col in X.columns:
                series = X[col]
                values = series.to_numpy(dtype=float)
                if np.nanstd(values) == 0:
                    corrs[col] = 0.0
                    continue
                # Ensure y has no NaNs for correlation calculation
                valid_indices = ~np.isnan(values)
                if np.sum(valid_indices) > 1: # Need at least two non-NaN va
                    corrs[col] = abs(np.corrcoef(values[valid_indices], y[va
                else:
                    corrs[col] = 0.0 # Cannot compute correlation with less

            feature_scores = corrs

    else:
        # Fallback: absolute Pearson correlation
        corrs = {}
        for col in X.columns:
            series = X[col]
            values = series.to_numpy(dtype=float)
            if np.nanstd(values) == 0:
                corrs[col] = 0.0
                continue
            # Ensure y has no NaNs for correlation calculation
            valid_indices = ~np.isnan(values)
            if np.sum(valid_indices) > 1: # Need at least two non-NaN values
                corrs[col] = abs(np.corrcoef(values[valid_indices], y[valid_
            else:
                corrs[col] = 0.0 # Cannot compute correlation with less thar

        feature_scores = corrs


    # Ensure feature_scores is not empty before sorting
    if not feature_scores:
        return df[["subject_id", "hadm_id", target_col]].copy(), [], {}

    top_features = sorted(feature_scores, key=feature_scores.get, reverse=Tr

    # Ensure selected_df is created correctly even if top_features is empty
    if top_features:
      selected_df = df[["subject_id", "hadm_id", target_col]].join(X[top_fea
    else:
      selected_df = df[["subject_id", "hadm_id", target_col]].copy()


    return selected_df, top_features, feature_scores
```

Main orchestration

In [ ]:
```
from pathlib import Path

def main() -> None:
    # Use the BASE_DIR defined in cell 9cG3wUVrmBFQ
    # BASE_DIR = Path(".").resolve()
    PROCESSED_DIR = BASE_DIR / "processed"
    PROCESSED_DIR.mkdir(exist_ok=True)

    # Check if features_selected.csv already exists
    features_selected_path = PROCESSED_DIR / "features_cleaned_selected.csv"
    if features_selected_path.exists():
        print(f"'{features_selected_path}' found. Skipping data processing a
        # Optionally load the existing features_selected.csv here if needed
        # features_selected_df = pd.read_csv(features_selected_path)
        return # Exit the function if the file exists


    admissions = load_and_clean_admissions(BASE_DIR / RAW_FILES["admissions"
    patients = load_and_clean_patients(BASE_DIR / RAW_FILES["patients"])
    icu = load_and_aggregate_icustays(BASE_DIR / RAW_FILES["icustays"])

    # Commenting out processing of lab events and feature selection to preve
    lab_path = BASE_DIR / RAW_FILES["labevents"]
    labs = pd.DataFrame()
    lab_specs: List[LabFeatureSpec] = []
    if lab_path.exists():
        print(f"Processing lab events from {lab_path} in chunks...")
        # Process labevents in chunks
        chunk_size = 100000 # Adjust chunk size based on available RAM and f
        eligible_hadm = admissions["hadm_id"].dropna().unique()
        all_lab_features = []
        first_chunk = True
        for chunk in pd.read_csv(lab_path, chunksize=chunk_size, on_bad_line
                                 parse_dates=["charttime", "storetime"], na
            chunk["subject_id"] = _ensure_int(chunk["subject_id"])
            chunk["hadm_id"] = _ensure_int(chunk["hadm_id"])

            # Prefer valuenum; fall back to parsed value
            chunk["value_num_clean"] = chunk["valuenum"]
            missing_mask = chunk["value_num_clean"].isna()
            chunk.loc[missing_mask, "value_num_clean"] = _safe_to_numeric(ch

            chunk = chunk.dropna(subset=["hadm_id", "value_num_clean", "char

            if first_chunk:
                # Select lab items based on the first chunk (or a representa
                # For better accuracy, might need a separate pass or larger
                lab_specs = _select_lab_items(chunk, eligible_hadm, top_n=50
                first_chunk = False

            if not lab_specs:
                 print("No eligible lab items found in the initial chunk. Sk
                 break # Exit loop if no specs are found

            frames = []
            for spec in lab_specs:
                subset = chunk[chunk["itemid"] == spec.itemid].copy()
                if subset.empty:
                    continue
                subset.sort_values("charttime", inplace=True)

                agg = subset.groupby("hadm_id").agg(
                    **{
                        f"{spec.name}_count": ("value_num_clean", "count"),
                        f"{spec.name}_mean": ("value_num_clean", "mean"),
                        f"{spec.name}_std": ("value_num_clean", "std"),
                        f"{spec.name}_min": ("value_num_clean", "min"),
                        f"{spec.name}_max": ("value_num_clean", "max"),
                    }
                )

                last_values = (
                    subset.sort_values("charttime")
                    .groupby("hadm_id", as_index=False)
                    .tail(1)
                    .set_index("hadm_id")["value_num_clean"]
                )
                agg[f"{spec.name}_last"] = agg.index.map(last_values)

                frames.append(agg)

            if frames:
                 chunk_lab_features = pd.concat(frames, axis=1)
                 chunk_lab_features.reset_index(inplace=True)
                 chunk_lab_features = chunk_lab_features.loc[:, ~chunk_lab_f
                 all_lab_features.append(chunk_lab_features)
            print(f"Processed a lab chunk.")

        if all_lab_features:

            labs = pd.concat(all_lab_features, ignore_index=True)

            labs = labs.groupby('hadm_id', as_index=False).agg('first') # Si


    omr_path = BASE_DIR / RAW_FILES["omr"]
    omr = pd.DataFrame() # Initialize omr as empty DataFrame
    if omr_path.exists():
        print(f"Processing OMR data from {omr_path}...")
        # Uncommenting OMR processing
        omr = load_and_aggregate_omr(omr_path)
        print("OMR processing complete.")
    else:
         print(f"OMR file not found at {omr_path}. Skipping OMR processing."


    # Persist intermediate datasets (only admissions, patients, icu, and now
    saved_files = {
        "admissions": _save_table(admissions, "clean_admissions.parquet"),
        "patients": _save_table(patients, "clean_patients.parquet"),
        "icu": _save_table(icu, "icu_aggregates.parquet"),
        # "labs": _save_table(labs, "lab_aggregates.parquet") if not labs.em
        "omr": _save_table(omr, "omr_latest.parquet") if not omr.empty else
    }
    # Commenting out feature assembly and selection
    feature_table = assemble_feature_table(admissions, patients, icu, labs,

    # Handle case where feature_table might be empty after filtering
    if not feature_table.empty:
        selected_df, top_features, feature_scores = select_features(feature_
    else:
        selected_df = pd.DataFrame()
        top_features = []
        feature_scores = {}
        print("Feature table is empty after assembly. Skipping feature selec

    full_features_filename = _save_table(feature_table, "features_full.csv")
    selected_features_filename = _save_table(selected_df, "features_selectec

    # Creating dummy metadata to allow the cell to complete
    metadata = {
        "lab_features": [], # Empty as lab processing is skipped
        "selected_feature_names": [], # Empty as feature selection is skippe
        "feature_scores": {}, # Empty as feature selection is skipped
        "sklearn_used": False, # False as sklearn is not used in the skipped
        "rows": {
            "feature_full": 0, # 0 as feature assembly is skipped
            "feature_selected": 0, # 0 as feature selection is skipped
        },
        "saved_files": saved_files # Includes admissions, patients, icu, and
    }


    with open(PROCESSED_DIR / "feature_metadata.json", "w", encoding="utf-8"
        json.dump(metadata, f, indent=2)

    print("Data preprocessing (partial) complete. Outputs saved under 'proce
    print("Note: Processing of lab events, feature assembly, and feature sel


main()
```

```
'/content/drive/MyDrive/MSML_602_FP/Dataset/workspace_data/processed/feature
s_cleaned_selected.csv' found. Skipping data processing and feature selectio
n.
```

Unstructured Data Preprocessing & Cleaning

Discharge Notes

In [ ]:
```
import re
import pandas as pd

def clean_text(text):
    """
    Cleans a single medical note text by removing boilerplate, placeholders,
    """
    if not isinstance(text, str): # Handle potential non-string inputs
```

```python
        return ""
    # Replace masked strings (like [**...**]) and placeholders (like ___)
    text = re.sub(r'\\n', ' ', text)
    text = re.sub(r'\[\*\*.*?\*\*\]', ' ', text)
    text = re.sub(r'___+', ' ', text)

    # Remove header sections by finding the start of the main content.
    start_markers = [
        "History of Present Illness:",
        "Past Medical History:",
        "Social History:",
        "Family History:",
        "Physical Exam:",
        "Brief Hospital Course:",
        "Medications on Admission:",
        "Discharge Diagnosis:"
    ]

    start_index = -1
    for marker in start_markers:
        try:
            index = text.lower().index(marker.lower())
            if start_index == -1 or index < start_index:
                start_index = index
        except ValueError:
            continue

    if start_index != -1:
        text = text[start_index:]

    # Replace all excessive newlines, tabs, and multiple spaces with a singl
    text = text.replace('\t', ' ').replace('\r', ' ')
    text = re.sub(r'\s+', ' ', text)

    return text.strip()

def clean_discharge_notes(input_file: str, output_file: str):
    """
    Reads, cleans, and writes discharge summary notes in chunks.
    """
    print(f"Starting cleaning process for {input_file}...")

    # We will read and process the CSV in chunks to handle large files.
    # Adjust chunksize based on available RAM and file size
    chunk_size = 5000
    chunk_iter = pd.read_csv(input_file, chunksize=chunk_size, on_bad_lines=

    first_chunk = True
    for chunk in chunk_iter:
        # Apply cleaning to the 'text' column
        chunk['text'] = chunk['text'].apply(clean_text)

        if first_chunk:
            chunk.to_csv(output_file, index=False, mode='w')
            first_chunk = False
        else:
            chunk.to_csv(output_file, index=False, mode='a', header=False)

        print(f"Processed a chunk of {chunk_size} rows and appended to {outp

    print(f"Cleaning complete. Cleaned data saved to {output_file}")

# Example usage (replace with your actual file paths)
# clean_discharge_notes('/content/sample_data/discharge.csv', '/content/proc
```

Radiology

In [ ]:

```
import re
import pandas as pd

def clean_text(text):
    """
    Cleans a single medical note text by removing boilerplate text, placehol
    """
    if not isinstance(text, str): # Handle potential non-string inputs
        return ""
    # Replace masked strings (like [**...**]) and placeholders (like ___)
    text = re.sub(r'\\n', ' ', text)
    text = re.sub(r'\[\*\*.*?\*\*\]', ' ', text)
    text = re.sub(r'___+', ' ', text)

    # Remove header sections by finding the start of the main content.
    start_markers = [
        "EXAMINATION:",
        "INDICATION:",
        "TECHNIQUE:",
        "FINDINGS:",
        "IMPRESSION:"
    ]

    start_index = -1
    for marker in start_markers:
        try:
            index = text.lower().index(marker.lower())
            if start_index == -1 or index < start_index:
                start_index = index
        except ValueError:
            continue

    if start_index != -1:
        text = text[start_index:]

    # Replace all excessive newlines, tabs, and multiple spaces with a singl
    text = text.replace('\t', ' ').replace('\r', ' ')
    text = re.sub(r'\s+', ' ', text)

    return text.strip()

def clean_radiology_notes(input_file: str, output_file: str):
    """
    Reads, cleans, and writes radiology notes in chunks.
    """
    print(f"Starting cleaning process for {input_file}...")

    # We will read and process the CSV in chunks to handle large files.
    # Adjust chunksize based on available RAM and file size
    chunk_size = 5000
    chunk_iter = pd.read_csv(input_file, chunksize=chunk_size, on_bad_lines=


    first_chunk = True
    for chunk in chunk_iter:
        # Apply cleaning to the 'text' column
        chunk['text'] = chunk['text'].apply(clean_text)

        if first_chunk:
            chunk.to_csv(output_file, index=False, mode='w')
            first_chunk = False
        else:
            chunk.to_csv(output_file, index=False, mode='a', header=False)

        print(f"Processed a chunk of {chunk_size} rows and appended to {outp

    print(f"Cleaning complete. Cleaned data saved to {output_file}")

# Example usage (replace with your actual file paths)
# clean_radiology_notes('/content/sample_data/radiology.csv', '/content/proc
```

Combine Structured & Unstrcutured Data

In [ ]:
```
import pandas as pd
import argparse # Import the argparse module
from pathlib import Path # Import Path

def merge_structured_unstructured_data(discharge_file: Path, radiology_file:
    """
    merge_structured_unstructured_data function to read, merge, and write th
    Accepts file paths as arguments.
    """

    # Remove argparse logic as we are passing paths directly
    # parser = argparse.ArgumentParser(description='Join discharge, radiolog
    # parser.add_argument('discharge_file', help='Path to the cleaned discha
    # parser.add_argument('radiology_file', help='Path to the cleaned radiol
    # parser.add_argument('features_file', help='Path to the features CSV.')
    # parser.add_argument('output_file', help='Path to the output merged CSV
    # args = parser.parse_args()


    print("Reading CSV files...")
    try:
        discharge_df = pd.read_csv(discharge_file)
        radiology_df = pd.read_csv(radiology_file)
        features_df = pd.read_csv(features_file)
    except FileNotFoundError as e:
        print(f"Error: {e.filename} not found.")
        # Create empty dataframes if files are not found to prevent further
        discharge_df = pd.DataFrame(columns=['subject_id', 'text'])
        radiology_df = pd.DataFrame(columns=['subject_id', 'text'])
        features_df = pd.DataFrame(columns=['subject_id'])
        print("Using empty DataFrames due to missing input files.")


    print("Aggregating notes by subject_id...")
    # Aggregate notes by subject_id, joining all notes for a patient into a
    # Ensure subject_id is treated as a common key
    discharge_agg = discharge_df.groupby('subject_id')['text'].apply(' '.joi
    discharge_agg.rename(columns={'text': 'discharge_text'}, inplace=True)

    radiology_agg = radiology_df.groupby('subject_id')['text'].apply(' '.joi
    radiology_agg.rename(columns={'text': 'radiology_text'}, inplace=True)

    print("Merging dataframes on subject_id...")
    # Merge the aggregated notes with the features dataframe.
    # We use a left merge to keep all subjects from the features file.
    merged_df = pd.merge(features_df, discharge_agg, on='subject_id', how='l
    final_df = pd.merge(merged_df, radiology_agg, on='subject_id', how='left

    # Fill NaN values in the new text columns with an empty string
    final_df['discharge_text'] = final_df['discharge_text'].fillna('')
    final_df['radiology_text'] = final_df['radiology_text'].fillna('')

    print(f"Saving merged data to {output_file}...")
    final_df.to_csv(output_file, index=False)
    print(f"Merge complete. Final data saved to {output_file}")



# # Define file paths based on notebook structure
# # Assuming processed files are in the 'processed' directory
# processed_dir = Path("/content/processed") # Use the processed directory p
# discharge_input_path = processed_dir / "discharge_cleaned.csv"
# radiology_input_path = processed_dir / "radiology_cleaned.csv"
# # Note: The features file name might vary based on whether feature selecti
# # Using features_full.csv as a default, but might need adjustment if selec
# features_input_path = processed_dir / "features_full.csv" # Or "features_s
# output_path = processed_dir / "merged_data.csv" # Define an output file na

# # Call merge_structured_unstructured_data with the defined paths
# merge_structured_unstructured_data(discharge_input_path, radiology_input_p
```

Exploratory Data Analysis

In [ ]:
```
import numpy as np
import pandas as pd
```

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import ttest_ind
```

In [ ]:
```
!ls
```

```
admissions.csv       icustays.csv   omr.csv       processed
d_icd_diagnoses.csv  labevents.csv  patients.csv
```

In [ ]:
```
from pathlib import Path
import pandas as pd

# Corrected file path
large_file_path = Path('/content/drive/MyDrive/MSML_602_FP/Dataset/workspace

sample_size = 50000

sampled_chunks = []

chunk_size = 50000

print(f"Reading and sampling from large file in chunks: {large_file_path}")
print(f"Target sample size: {sample_size}")

total_rows_read = 0
for chunk in pd.read_csv(large_file_path, chunksize=chunk_size, low_memory=F
    total_rows_read += len(chunk)

    sample_fraction = min(1.0, (sample_size - sum(len(sc) for sc in sampled_
    if sample_fraction <= 0:
        break

    sampled_chunk = chunk.sample(frac=sample_fraction, replace=False, random
    sampled_chunks.append(sampled_chunk)
    print(f"Read {total_rows_read} rows, sampled {len(sampled_chunk)} from t

    if sum(len(sc) for sc in sampled_chunks) >= sample_size:
        break


project_df = pd.concat(sampled_chunks, ignore_index=True)

print(f"\nFinished sampling. Total rows in sampled DataFrame: {len(project_d
```

```
Reading and sampling from large file in chunks: /content/drive/MyDrive/MSML_
602_FP/Dataset/workspace_data/processed/final_cleaned_data.csv
Target sample size: 50000
Read 50000 rows, sampled 50000 from this chunk. Total sampled so far: 50000

Finished sampling. Total rows in sampled DataFrame: 50000
```

In [ ]:
```
features_selected_df = pd.read_csv('/content/drive/MyDrive/MSML_602_FP/Datas
```

In [ ]:
```
eda_df = project_df.copy()
```

In [ ]:
```
# 1. IMMEDIATE DATA QUALITY CHECKS
# Check text data completeness - CRITICAL for RAG
text_completeness = {
    'discharge_text': eda_df['discharge_text'].notna().sum(),
    'radiology_text': eda_df['radiology_text'].notna().sum(),
    'both_available': ((eda_df['discharge_text'].notna()) &
                       (eda_df['radiology_text'].notna())).sum()
}
print(f"\nText data availability: {text_completeness}")

# 2. Check text length distributions - short texts may be artifacts
eda_df['discharge_text_len'] = eda_df['discharge_text'].str.len()
eda_df['radiology_text_len'] = eda_df['radiology_text'].str.len()

print("\n")

print(eda_df[['discharge_text_len', 'radiology_text_len']].describe())

# Flag suspiciously short texts (< 100 chars)
short_discharge = (eda_df['discharge_text_len'] < 100).sum()
print(f"\nSuspiciously short discharge notes: {short_discharge}")
```

```
Text data availability: {'discharge_text': np.int64(38868), 'radiology_tex
t': np.int64(41239), 'both_available': np.int64(37489)}


       discharge_text_len  radiology_text_len
count        3.886800e+04        41239.000000
mean         6.031893e+04        32774.408497
std          9.948833e+04        43446.177613
min          7.080000e+02           25.000000
25%          1.211600e+04         5793.000000
50%          2.811900e+04        16904.000000
75%          6.826100e+04        42155.000000
max          1.088152e+06       318705.000000

Suspiciously short discharge notes: 0
```

In [ ]:
```
rows, columns = eda_df.shape
print(f"Number of rows: {rows}")
print(f"Number of columns: {columns}")
```

```
Number of rows: 50000
Number of columns: 29
```

In [ ]:
```
eda_df.head()
```

Out[ ]:

	subject_id	hadm_id	admittime	dischtime	admission_type	admit_provider_id	a
0	10636107	20812092	2183-08-19 16:01:00	2183-08-22 18:45:00	OBSERVATION ADMIT	P48CS5	E
1	10182665	22538295	2126-05-29 11:30:00	2126-06-01 13:17:00	SURGICAL SAME DAY ADMISSION	P8323I	
2	10003019	20962108	2176-01-06 15:52:00	2176-01-14 18:09:00	EW EMER.	P84UKK	E
3	10244511	26255794	2191-11-29 03:09:00	2191-12-02 15:50:00	EW EMER.	P42H7G	
4	10741731	21866879	2140-01-26 09:15:00	2140-01-26 15:38:00	SURGICAL SAME DAY ADMISSION	P60IOB	

5 rows × 29 columns

In [ ]:

```
print("\n Column Types")
eda_df.dtypes
```

```
 Column Types
```

```
Out[ ]:
                          0
           subject_id   int64
              hadm_id   int64
            admittime  object
             dischtime  object
       admission_type  object
     admit_provider_id  object
   admission_location  object
   discharge_location  object
            insurance  object
             language  object
       marital_status  object
                 race  object
            edregtime  object
            edouttime  object
  hospital_expire_flag   int64
            los_hours float64
      ed_wait_minutes float64
      ed_stay_minutes float64
     died_in_hospital   int64
               gender  object
           anchor_age   int64
          anchor_year   int64
    anchor_year_group  object
        dod_available   int64
            age_group  object
       discharge_text  object
       radiology_text  object
   discharge_text_len float64
   radiology_text_len float64

dtype: object
```

```
In [ ]: print(f"Unique Patients (subject_id): {eda_df['subject_id'].nunique():,}")
        print(f"Unique Hospitalizations (hadm_id): {eda_df['hadm_id'].nunique():,}")
        print(f"Unique admit_provider_id (admit_provider_id): {eda_df['admit_provide
        print(f"Total Rows: {len(eda_df):,}")
        print(f"Average Rows per Patient: {len(eda_df) / eda_df['subject_id'].nuniqu
```

```
Unique Patients (subject_id): 20,649
Unique Hospitalizations (hadm_id): 50,000
Unique admit_provider_id (admit_provider_id): 1,507
Total Rows: 50,000
Average Rows per Patient: 2.42
```

```
In [ ]: from pathlib import Path
        import pandas as pd

        processed_dir = Path("/content/processed")
        merged_file_path = processed_dir / "merged_data.csv"

        if merged_file_path.exists():
            print(f"'{merged_file_path}'  ")
            try:

                df_check = pd.read_csv(merged_file_path, nrows=5, low_memory=False)
                print("\n  ")
                display(df_check)
                print(f"\n  {df_check.columns.tolist()}")
            except Exception as e:
                print(f"\n error: {e}")
        else:
            print(f"'{merged_file_path}' ")
```

```
'/content/processed/merged_data.csv'
```

```
In [ ]: duplicates = eda_df.duplicated(subset=['subject_id', 'hadm_id']).sum()
        print(f"Duplicate Hospital Stays: {duplicates}")
```

```
Duplicate Hospital Stays: 0
```

```
In [ ]: def categorize_columns(df): # Changed parameter name to df for clarity
            # Detect by name
            temporal_cols = [c for c in df.columns if any(x in c.lower() for x in ['
            id_cols = [c for c in df.columns if 'id' in c.lower()]
            text_cols = [c for c in df.columns if 'text' in c.lower()]

            # Detect by dtype
            object_cols = df.select_dtypes(include=['object']).columns.tolist()
            numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()

            # Detect categorical (object columns not text/id/time)
            categorical_cols = [
                c for c in object_cols
                if c not in text_cols + id_cols + temporal_cols
            ]

            # Detect numerical (numeric columns not id or binary)
            # Use the passed DataFrame 'df' instead of 'project_df'
            binary_cols = [
                c for c in numeric_cols
                if df[c].dropna().nunique() == 2
            ]

            numerical_cols = [
                c for c in numeric_cols
                if c not in id_cols + binary_cols
            ]

            return {
                'temporal_cols': temporal_cols,
                'id_cols': id_cols,
                'text_cols': text_cols,
                'categorical_cols': categorical_cols,
                'numerical_cols': numerical_cols,
                'binary_cols': binary_cols
            }

        # Example usage:
        col_categories = categorize_columns(eda_df) # Pass eda_df to the function

        # Pretty print the results
        for k, v in col_categories.items():
            print(f"{k}: {v}")
```

```
temporal_cols: ['admittime', 'dischtime', 'edregtime', 'edouttime']
id_cols: ['subject_id', 'hadm_id', 'admit_provider_id']
text_cols: ['discharge_text', 'radiology_text', 'discharge_text_len', 'radio
logy_text_len']
categorical_cols: ['admission_type', 'admission_location', 'discharge_locati
on', 'insurance', 'language', 'marital_status', 'race', 'gender', 'anchor_ye
ar_group', 'age_group']
numerical_cols: ['los_hours', 'ed_wait_minutes', 'ed_stay_minutes', 'anchor_
age', 'anchor_year', 'discharge_text_len', 'radiology_text_len']
binary_cols: ['hospital_expire_flag', 'died_in_hospital', 'dod_available']
```

In [ ]:
```
eda_df.describe()
```

Out[ ]:

	subject_id	hadm_id	hospital_expire_flag	los_hours	ed_wait_min
count	5.000000e+04	5.000000e+04	50000.000000	49984.000000	34779.00
mean	1.047456e+07	2.498851e+07	0.022040	114.684835	341.348
std	2.704533e+05	2.884753e+06	0.146815	185.462732	259.66
min	1.000003e+07	2.000002e+07	0.000000	0.033333	-1411.00
25%	1.024563e+07	2.250185e+07	0.000000	26.716667	184.00
50%	1.048028e+07	2.499292e+07	0.000000	67.258333	282.00
75%	1.070485e+07	2.748968e+07	0.000000	134.404167	420.00
max	1.094027e+07	2.999972e+07	1.000000	12373.500000	4218.00

In [ ]:
```
eda_df.describe().T
```

Out[ ]:

	count	mean	std	min	2
subject_id	50000.0	1.047456e+07	2.704533e+05	1.000003e+07	1.024563e
hadm_id	50000.0	2.498851e+07	2.884753e+06	2.000002e+07	2.250185e
hospital_expire_flag	50000.0	2.204000e-02	1.468151e-01	0.000000e+00	0.000000e
los_hours	49984.0	1.146848e+02	1.854627e+02	3.333333e-02	2.671667e
ed_wait_minutes	34779.0	3.413489e+02	2.596678e+02	-1.411000e+03	1.840000e
ed_stay_minutes	34779.0	6.565459e+02	6.188900e+02	0.000000e+00	3.130000e
died_in_hospital	50000.0	2.204000e-02	1.468151e-01	0.000000e+00	0.000000e
anchor_age	50000.0	5.697286e+01	1.909222e+01	1.800000e+01	4.300000e
anchor_year	50000.0	2.152244e+03	2.378516e+01	2.110000e+03	2.132000e
dod_available	50000.0	2.667800e-01	4.422808e-01	0.000000e+00	0.000000e
discharge_text_len	38868.0	6.031893e+04	9.948833e+04	7.080000e+02	1.211600e
radiology_text_len	41239.0	3.277441e+04	4.344618e+04	2.500000e+01	5.793000e

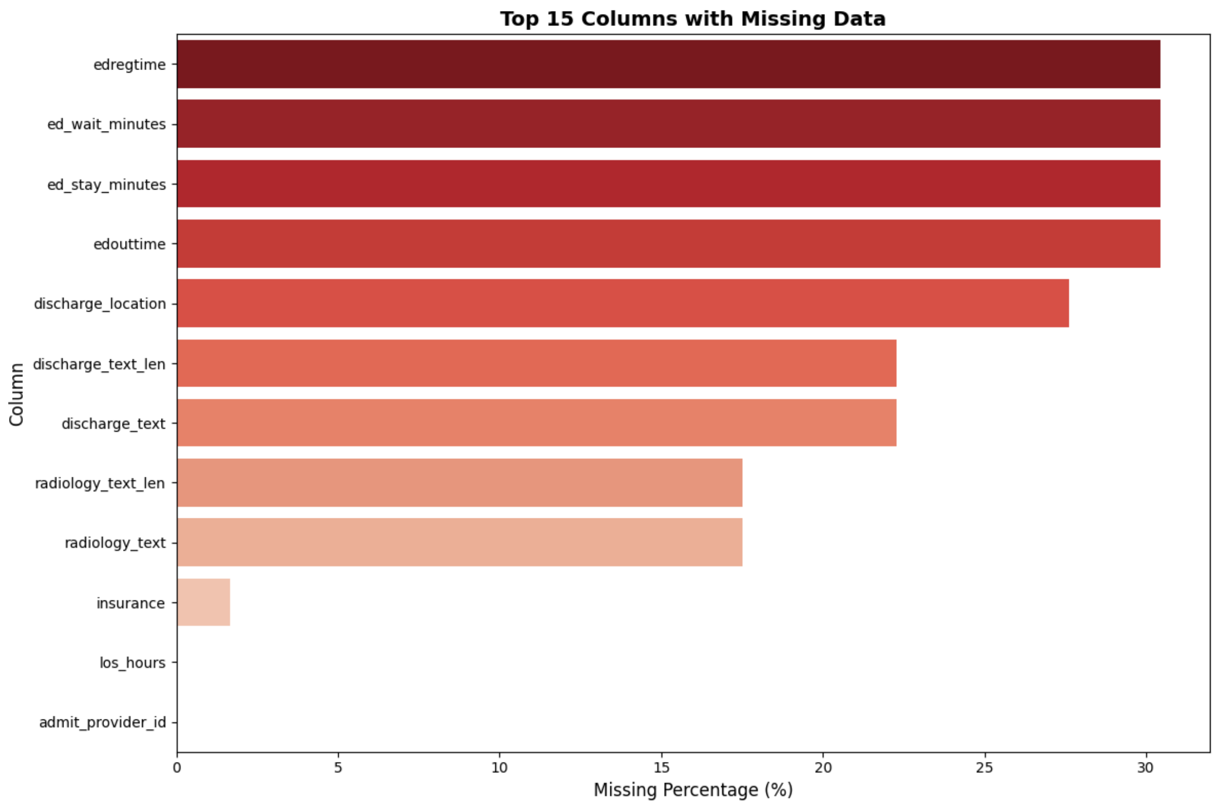
In [ ]:
```python
missing_stats = pd.DataFrame({
    'column': eda_df.columns,
    'missing_count': eda_df.isnull().sum().values,
    'missing_pct': (eda_df.isnull().sum() / len(eda_df) * 100).values,
    'dtype': eda_df.dtypes.values
}).sort_values('missing_pct', ascending=False)
```

In [ ]:
```python
print(missing_stats[missing_stats['missing_pct'] > 0])
```

```
                column  missing_count  missing_pct    dtype
12           edregtime          15221       30.442   object
16     ed_wait_minutes          15221       30.442  float64
17     ed_stay_minutes          15221       30.442  float64
13           edouttime          15221       30.442   object
7   discharge_location          13811       27.622   object
27  discharge_text_len          11132       22.264  float64
25      discharge_text          11132       22.264   object
28  radiology_text_len           8761       17.522  float64
26      radiology_text           8761       17.522   object
8            insurance            839        1.678   object
15           los_hours             16        0.032  float64
5     admit_provider_id              1        0.002   object
```

In [ ]:
```python
fig, ax = plt.subplots(figsize=(12, 8))
missing_plot_data = missing_stats[missing_stats['missing_pct'] > 0].head(15)
sns.barplot(data=missing_plot_data, x='missing_pct', y='column', palette='Re
ax.set_xlabel('Missing Percentage (%)', fontsize=12)
ax.set_ylabel('Column', fontsize=12)
ax.set_title('Top 15 Columns with Missing Data', fontsize=14, fontweight='bc
plt.tight_layout()
plt.savefig('missing_data_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```

```
/tmp/ipython-input-1429206783.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the
same effect.

  sns.barplot(data=missing_plot_data, x='missing_pct', y='column', palette
='Reds_r', ax=ax)
```



In [ ]:
```
print("\nMissing Patterns by Context")

# ED columns missing pattern - expected for non-ED admissions
if 'ed_stay_minutes' in eda_df.columns:
    print("\nED Stay Minutes Missing by Admission Type:")
    ed_missing = eda_df.groupby('admission_type').agg({
        'ed_stay_minutes': lambda x: f"{x.isnull().sum()}/{len(x)} ({x.isnul
    })
    print(ed_missing)

if 'ed_wait_minutes' in eda_df.columns:
    print("\nED Wait Minutes Missing by Admission Type:")
    ed_wait_missing = eda_df.groupby('admission_type').agg({
        'ed_wait_minutes': lambda x: f"{x.isnull().sum()}/{len(x)} ({x.isnul
    })
    print(ed_wait_missing)

# Text data missingness - CRITICAL FOR RAG
print("\nText Data Completeness (CRITICAL FOR RAG)")
print(f"Discharge Text Available: {eda_df['discharge_text'].notna().sum():,}
print(f"Radiology Text Available: {eda_df['radiology_text'].notna().sum():,}
print(f"Both Texts Available: {((eda_df['discharge_text'].notna()) & (eda_df
```

```
Missing Patterns by Context

ED Stay Minutes Missing by Admission Type:
                               ed_stay_minutes
admission_type
AMBULATORY OBSERVATION           600/613 (97.9%)
DIRECT EMER.                   1759/2005 (87.7%)
DIRECT OBSERVATION             1657/2234 (74.2%)
ELECTIVE                       1136/1147 (99.0%)
EU OBSERVATION                  47/11103 (0.4%)
EW EMER.                       689/16124 (4.3%)
OBSERVATION ADMIT              1166/7704 (15.1%)
SURGICAL SAME DAY ADMISSION    3966/3968 (99.9%)
URGENT                         4201/5102 (82.3%)

ED Wait Minutes Missing by Admission Type:
                               ed_wait_minutes
admission_type
AMBULATORY OBSERVATION           600/613 (97.9%)
DIRECT EMER.                   1759/2005 (87.7%)
DIRECT OBSERVATION             1657/2234 (74.2%)
ELECTIVE                       1136/1147 (99.0%)
EU OBSERVATION                  47/11103 (0.4%)
EW EMER.                       689/16124 (4.3%)
OBSERVATION ADMIT              1166/7704 (15.1%)
SURGICAL SAME DAY ADMISSION    3966/3968 (99.9%)
URGENT                         4201/5102 (82.3%)

Text Data Completeness (CRITICAL FOR RAG)
Discharge Text Available: 38,868 (77.7%)
Radiology Text Available: 41,239 (82.5%)
Both Texts Available: 37,489 (75.0%)
```

In [ ]:
```
eda_df['admittime'] = pd.to_datetime(eda_df['admittime'])
eda_df['dischtime'] = pd.to_datetime(eda_df['dischtime'])
if 'edregtime' in eda_df.columns:
    eda_df['edregtime'] = pd.to_datetime(eda_df['edregtime'])
if 'edouttime' in eda_df.columns:
    eda_df['edouttime'] = pd.to_datetime(eda_df['edouttime'])
```

In [ ]:
```
if 'los_hours' not in eda_df.columns:
    eda_df['los_hours'] = (eda_df['dischtime'] - eda_df['admittime']).dt.tot
eda_df['los_days'] = eda_df['los_hours'] / 24

print("\nLength of Stay Statistics")
print("Hours:")
print(eda_df['los_hours'].describe())
print("\nDays:")
print(eda_df['los_days'].describe())
```

```
Length of Stay Statistics
Hours:
count    49984.000000
mean       114.684835
std        185.462732
min          0.033333
25%         26.716667
50%         67.258333
75%        134.404167
max      12373.500000
Name: los_hours, dtype: float64

Days:
count    49984.000000
mean         4.778535
std          7.727614
min          0.001389
25%          1.113194
50%          2.802431
75%          5.600174
max        515.562500
Name: los_days, dtype: float64
```

In [ ]:
```python
print("\nTemporal Data Quality Checks")
print(f"Negative LOS: {(eda_df['los_days'] < 0).sum()}")
print(f"Zero LOS: {(eda_df['los_days'] == 0).sum()}")
print(f"Same-day discharge (< 1 day): {(eda_df['los_days'] < 1).sum()}")
print(f"Extended stay (> 30 days): {(eda_df['los_days'] > 30).sum()}")
print(f"Very long stay (> 365 days): {(eda_df['los_days'] > 365).sum()}")
```

```
Temporal Data Quality Checks
Negative LOS: 0
Zero LOS: 0
Same-day discharge (< 1 day): 11242
Extended stay (> 30 days): 689
Very long stay (> 365 days): 1
```

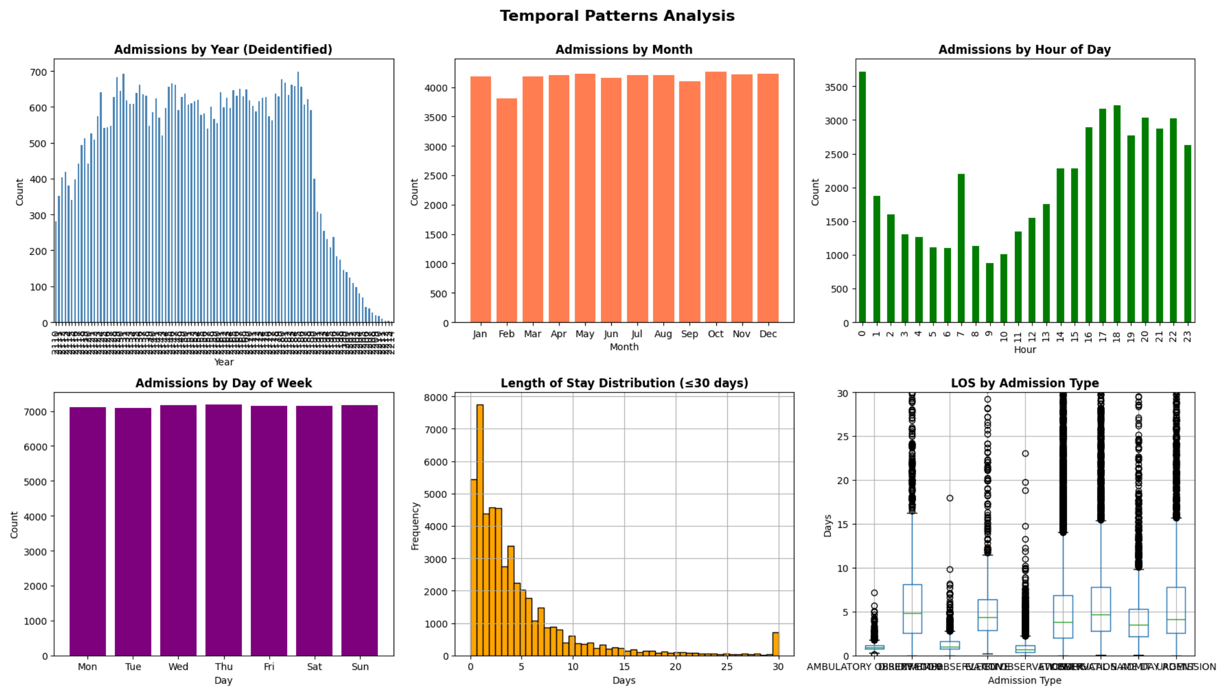
In [ ]:
```python
if 'ed_wait_minutes' in eda_df.columns and eda_df['ed_wait_minutes'].notna()
    print("\nEmergency Department Timing")
    print("ED Wait Time (minutes):")
    print(eda_df['ed_wait_minutes'].describe())
    print("\nED Total Stay Time (minutes):")
    print(eda_df['ed_stay_minutes'].describe())
```

```
Emergency Department Timing
ED Wait Time (minutes):
count    34779.000000
mean       341.348946
std        259.667793
min      -1411.000000
25%        184.000000
50%        282.000000
75%        420.000000
max       4218.000000
Name: ed_wait_minutes, dtype: float64

ED Total Stay Time (minutes):
count    34779.000000
mean       656.545875
std        618.890005
min          0.000000
25%        313.000000
50%        466.000000
75%        776.000000
max      11683.000000
Name: ed_stay_minutes, dtype: float64
```

```
In [ ]: eda_df['admit_year'] = eda_df['admittime'].dt.year
        eda_df['admit_month'] = eda_df['admittime'].dt.month
        eda_df['admit_hour'] = eda_df['admittime'].dt.hour
        eda_df['admit_dow'] = eda_df['admittime'].dt.dayofweek  # 0=Monday
        eda_df['admit_day_name'] = eda_df['admittime'].dt.day_name()
```

```
In [ ]: # Temporal Features
        eda_df['admit_year'] = eda_df['admittime'].dt.year
        eda_df['admit_month'] = eda_df['admittime'].dt.month
        eda_df['admit_hour'] = eda_df['admittime'].dt.hour
        eda_df['admit_dow'] = eda_df['admittime'].dt.dayofweek  # 0=Monday
        eda_df['admit_day_name'] = eda_df['admittime'].dt.day_name()

        # Visualize Temporal Patterns
        fig, axes = plt.subplots(2, 3, figsize=(18, 10))

        # Admissions by year
        eda_df['admit_year'].value_counts().sort_index().plot(kind='bar', ax=axes[0,
        axes[0,0].set_title('Admissions by Year (Deidentified)', fontweight='bold')
        axes[0,0].set_xlabel('Year')
        axes[0,0].set_ylabel('Count')

        # Admissions by month
        month_counts = eda_df['admit_month'].value_counts().sort_index()
        axes[0,1].bar(range(1, 13), month_counts.values, color='coral')
        axes[0,1].set_xticks(range(1, 13))
        axes[0,1].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                                   'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
        axes[0,1].set_title('Admissions by Month', fontweight='bold')
        axes[0,1].set_xlabel('Month')
        axes[0,1].set_ylabel('Count')

        # Admissions by hour
```

```
eda_df['admit_hour'].value_counts().sort_index().plot(kind='bar', ax=axes[0,
axes[0,2].set_title('Admissions by Hour of Day', fontweight='bold')
axes[0,2].set_xlabel('Hour')
axes[0,2].set_ylabel('Count')

# Admissions by day of week
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturc
dow_data = eda_df['admit_day_name'].value_counts().reindex(day_order)
axes[1,0].bar(range(7), dow_data.values, color='purple')
axes[1,0].set_xticks(range(7))
axes[1,0].set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
axes[1,0].set_title('Admissions by Day of Week', fontweight='bold')
axes[1,0].set_xlabel('Day')
axes[1,0].set_ylabel('Count')

# LOS distribution (capped)
eda_df['los_days'].clip(upper=30).hist(bins=50, ax=axes[1,1], color='orange'
axes[1,1].set_title('Length of Stay Distribution (≤30 days)', fontweight='bc
axes[1,1].set_xlabel('Days')
axes[1,1].set_ylabel('Frequency')

# LOS by admission type
eda_df.boxplot(column='los_days', by='admission_type', ax=axes[1,2])
axes[1,2].set_title('LOS by Admission Type', fontweight='bold')
axes[1,2].set_xlabel('Admission Type')
axes[1,2].set_ylabel('Days')
axes[1,2].set_ylim(0, 30)
plt.suptitle('')  # Remove default title

plt.suptitle('Temporal Patterns Analysis', fontsize=16, fontweight='bold', y
plt.tight_layout()
plt.savefig('temporal_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```



In [ ]:
```
# Anchor Year Analysis
print("\nAnchor Year Groups")
print(eda_df['anchor_year_group'].value_counts().sort_index())
```

```
Anchor Year Groups
anchor_year_group
2008 - 2010    20956
2011 - 2013    10144
2014 - 2016     8484
2017 - 2019     6537
2020 - 2022     3879
Name: count, dtype: int64
```

In [ ]:
```
print("\nAge Distribution")
eda_df['anchor_age'].describe()
```

```
Age Distribution
```

Out[ ]:

	anchor_age
count	50000.000000
mean	56.972860
std	19.092218
min	18.000000
25%	43.000000
50%	58.000000
75%	72.000000
max	91.000000

dtype: float64

In [ ]:
```
print(f"\nAge-censored patients (age=91): {(eda_df['anchor_age'] == 91).sum(
```

```
Age-censored patients (age=91): 1,428 (2.9%)
```

In [ ]:
```
print("\nAge Groups")
print(eda_df['age_group'].value_counts().sort_index())
```

```
Age Groups
age_group
40-59    15450
60-74    13765
75+      10205
<40      10580
Name: count, dtype: int64
```

In [ ]:
```
print("\nGender Distribution")
gender_counts = eda_df['gender'].value_counts()
print(gender_counts)
print("\nPercentages:")
print(eda_df['gender'].value_counts(normalize=True) * 100)
```

```
Gender Distribution
gender
F    25596
M    24404
Name: count, dtype: int64

Percentages:
gender
F    51.192
M    48.808
Name: proportion, dtype: float64
```

In [ ]:
```
print("\nRace/Ethnicity Distribution")
print(eda_df['race'].value_counts().head(10))
```

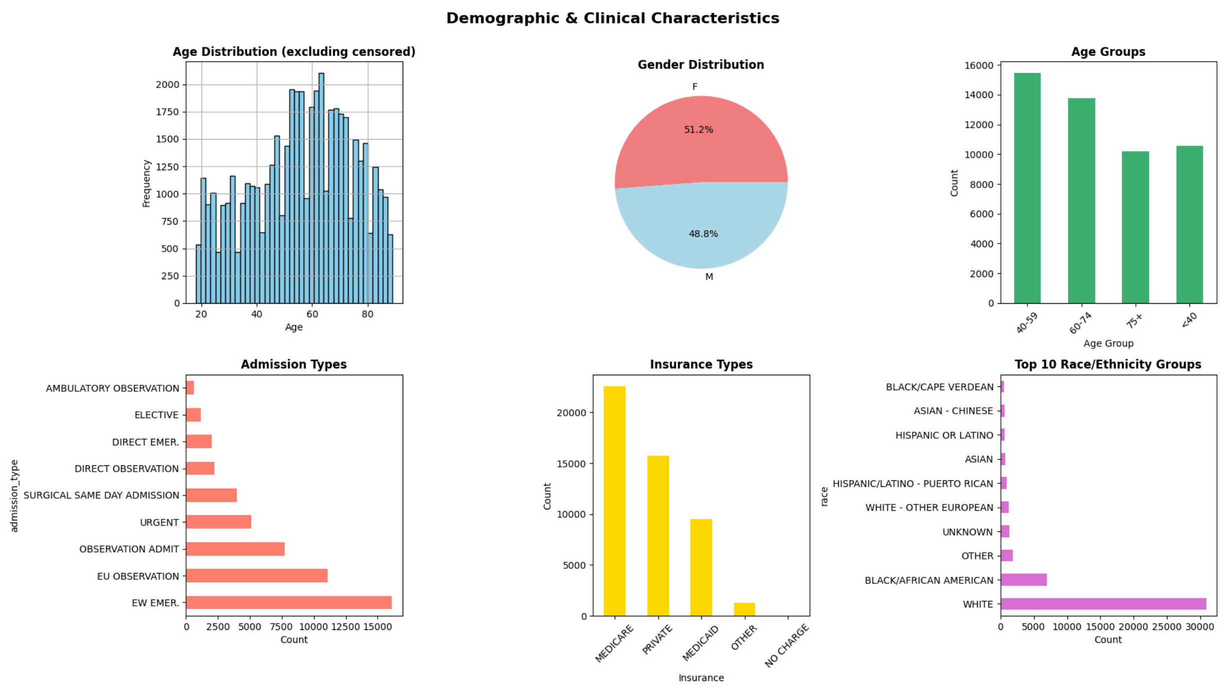
```
Race/Ethnicity Distribution
race
WHITE                             30889
BLACK/AFRICAN AMERICAN             6948
OTHER                              1864
UNKNOWN                            1362
WHITE - OTHER EUROPEAN             1234
HISPANIC/LATINO - PUERTO RICAN      945
ASIAN                               737
HISPANIC OR LATINO                  684
ASIAN - CHINESE                     679
BLACK/CAPE VERDEAN                  592
Name: count, dtype: int64
```

In [ ]:
```
print("\nMarital Status Distribution")
print(eda_df['marital_status'].value_counts())
```

```
Marital Status Distribution
marital_status
MARRIED     20909
SINGLE      18648
WIDOWED      5278
DIVORCED     3915
UNKNOWN      1250
Name: count, dtype: int64
```

In [ ]:
```
fig, axes = plt.subplots(2, 3, figsize=(18, 10))

# Age histogram (excluding censored)
eda_df[eda_df['anchor_age'] < 91]['anchor_age'].hist(bins=40, ax=axes[0,0],
                                                      color='skyblue', edgecolor='b
axes[0,0].set_title('Age Distribution (excluding censored)', fontweight='bol
axes[0,0].set_xlabel('Age')
axes[0,0].set_ylabel('Frequency')

# Gender distribution
gender_counts.plot(kind='pie', ax=axes[0,1], autopct='%1.1f%%', colors=['lig
axes[0,1].set_title('Gender Distribution', fontweight='bold')
axes[0,1].set_ylabel('')
```

```
# Age groups
eda_df['age_group'].value_counts().sort_index().plot(kind='bar', ax=axes[0,2
axes[0,2].set_title('Age Groups', fontweight='bold')
axes[0,2].set_xlabel('Age Group')
axes[0,2].set_ylabel('Count')
axes[0,2].tick_params(axis='x', rotation=45)

# Admission type
eda_df['admission_type'].value_counts().plot(kind='barh', ax=axes[1,0], colo
axes[1,0].set_title('Admission Types', fontweight='bold')
axes[1,0].set_xlabel('Count')

# Insurance type
eda_df['insurance'].value_counts().plot(kind='bar', ax=axes[1,1], color='gol
axes[1,1].set_title('Insurance Types', fontweight='bold')
axes[1,1].set_xlabel('Insurance')
axes[1,1].set_ylabel('Count')
axes[1,1].tick_params(axis='x', rotation=45)

# Top race/ethnicity
eda_df['race'].value_counts().head(10).plot(kind='barh', ax=axes[1,2], color
axes[1,2].set_title('Top 10 Race/Ethnicity Groups', fontweight='bold')
axes[1,2].set_xlabel('Count')

plt.suptitle('Demographic & Clinical Characteristics', fontsize=16, fontweig
plt.tight_layout()
plt.savefig('demographics_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```



In [ ]:
```
print("\nAdmission Characteristics")
print("\nAdmission Types:")
print(eda_df['admission_type'].value_counts())
```

```
Admission Characteristics

Admission Types:
admission_type
EW EMER.                        16124
EU OBSERVATION                  11103
OBSERVATION ADMIT                7704
URGENT                           5102
SURGICAL SAME DAY ADMISSION      3968
DIRECT OBSERVATION               2234
DIRECT EMER.                     2005
ELECTIVE                         1147
AMBULATORY OBSERVATION            613
Name: count, dtype: int64
```

```
In [ ]: print("\nTop 10 Admission Locations:")
        print(eda_df['admission_location'].value_counts().head(10))
```

```
Top 10 Admission Locations:
admission_location
EMERGENCY ROOM                            22509
PHYSICIAN REFERRAL                        14824
TRANSFER FROM HOSPITAL                     5247
WALK-IN/SELF REFERRAL                      3813
CLINIC REFERRAL                            1160
PROCEDURE SITE                              758
INTERNAL TRANSFER TO OR FROM PSYCH          576
TRANSFER FROM SKILLED NURSING FACILITY      553
PACU                                        497
INFORMATION NOT AVAILABLE                    40
Name: count, dtype: int64
```

```
In [ ]: print("\nTop 10 Discharge Locations:")
        print(eda_df['discharge_location'].value_counts().head(10))
```

```
Top 10 Discharge Locations:
discharge_location
HOME                             17830
HOME HEALTH CARE                  8926
SKILLED NURSING FACILITY          4769
REHAB                             1326
DIED                              1099
CHRONIC/LONG TERM ACUTE CARE       730
HOSPICE                            481
AGAINST ADVICE                     314
PSYCH FACILITY                     274
ACUTE HOSPITAL                     223
Name: count, dtype: int64
```

```
In [ ]: print("\nLanguage Distribution:")
        print(eda_df['language'].value_counts().head(10))
```

```
Language Distribution:
language
ENGLISH                  44767
SPANISH                   1833
CHINESE                    755
RUSSIAN                    657
KABUVERDIANU               480
PORTUGUESE                 306
HAITIAN                    205
OTHER                      162
VIETNAMESE                 133
MODERN GREEK (1453-)       116
Name: count, dtype: int64
```

In [ ]:
```python
if 'hospital_expire_flag' in eda_df.columns:
    mortality_col = 'hospital_expire_flag'
else:
    mortality_col = 'died_in_hospital'

mortality_rate = eda_df[mortality_col].mean() * 100
n_deaths = eda_df[mortality_col].sum()

print(f"\nOverall Mortality")
print(f"In-hospital Deaths: {n_deaths:,}")
print(f"In-hospital Mortality Rate: {mortality_rate:.2f}%")
```

```
Overall Mortality
In-hospital Deaths: 1,102
In-hospital Mortality Rate: 2.20%
```

In [ ]:
```python
if 'hospital_expire_flag' in eda_df.columns and 'died_in_hospital' in eda_df
    print("\nMortality Flag Comparison")
    print(pd.crosstab(eda_df['hospital_expire_flag'], eda_df['died_in_hospit
                      rownames=['hospital_expire_flag'], colnames=['died_in_

# DOD availability
if 'dod_available' in eda_df.columns:
    dod_available = eda_df['dod_available'].sum()
    print(f"\nDate of Death Available: {dod_available:,} ({dod_available/ler
```

```
Mortality Flag Comparison
died_in_hospital          0     1
hospital_expire_flag
0                     48898     0
1                         0  1102

Date of Death Available: 13,339 (26.68%)
```

In [ ]:
```python
print("\nMortality by Age Group")
mortality_by_age = eda_df.groupby('age_group')[mortality_col].agg(['sum', 'c
mortality_by_age['rate_%'] = mortality_by_age['mean'] * 100
print(mortality_by_age)
```

```
Mortality by Age Group
            sum  count      mean    rate_%
age_group
40-59       200  15450  0.012945  1.294498
60-74       376  13765  0.027316  2.731566
75+         487  10205  0.047722  4.772171
<40          39  10580  0.003686  0.368620
```

In [ ]:
```
print("\nMortality by Gender")
mortality_by_gender = eda_df.groupby('gender')[mortality_col].agg(['sum', 'c
mortality_by_gender['rate_%'] = mortality_by_gender['mean'] * 100
print(mortality_by_gender)
```

```
Mortality by Gender
         sum  count      mean    rate_%
gender
F        487  25596  0.019026  1.902641
M        615  24404  0.025201  2.520079
```

In [ ]:
```
print("\nMortality by Race/Ethnicity (Top 10)")
mortality_by_race = eda_df.groupby('race')[mortality_col].agg(['sum', 'count
mortality_by_race['rate_%'] = mortality_by_race['mean'] * 100
print(mortality_by_race.sort_values('count', ascending=False).head(10))
```

```
Mortality by Race/Ethnicity (Top 10)
                                sum  count      mean     rate_%
race
WHITE                           613  30889  0.019845   1.984525
BLACK/AFRICAN AMERICAN           96   6948  0.013817   1.381693
OTHER                            34   1864  0.018240   1.824034
UNKNOWN                         162   1362  0.118943  11.894273
WHITE - OTHER EUROPEAN           27   1234  0.021880   2.188006
HISPANIC/LATINO - PUERTO RICAN    7    945  0.007407   0.740741
ASIAN                            19    737  0.025780   2.578019
HISPANIC OR LATINO                6    684  0.008772   0.877193
ASIAN - CHINESE                  16    679  0.023564   2.356406
BLACK/CAPE VERDEAN                7    592  0.011824   1.182432
```

In [ ]:
```
print("\nMortality by Admission Type")
mortality_by_admission = eda_df.groupby('admission_type')[mortality_col].agg
mortality_by_admission['rate_%'] = mortality_by_admission['mean'] * 100
print(mortality_by_admission.sort_values('rate_%', ascending=False))
```

```
Mortality by Admission Type
                             sum  count      mean    rate_%
admission_type
URGENT                       236   5102  0.046256  4.625637
EW EMER.                     627  16124  0.038886  3.888613
DIRECT EMER.                  55   2005  0.027431  2.743142
OBSERVATION ADMIT            159   7704  0.020639  2.063863
ELECTIVE                       6   1147  0.005231  0.523104
SURGICAL SAME DAY ADMISSION    8   3968  0.002016  0.201613
DIRECT OBSERVATION             2   2234  0.000895  0.089526
EU OBSERVATION                 9  11103  0.000811  0.081059
AMBULATORY OBSERVATION         0    613  0.000000  0.000000
```

```
In [ ]: print("\nMortality by Insurance Type")
        mortality_by_insurance = eda_df.groupby('insurance')[mortality_col].agg(['su
        mortality_by_insurance['rate_%'] = mortality_by_insurance['mean'] * 100
        print(mortality_by_insurance.sort_values('rate_%', ascending=False))
```

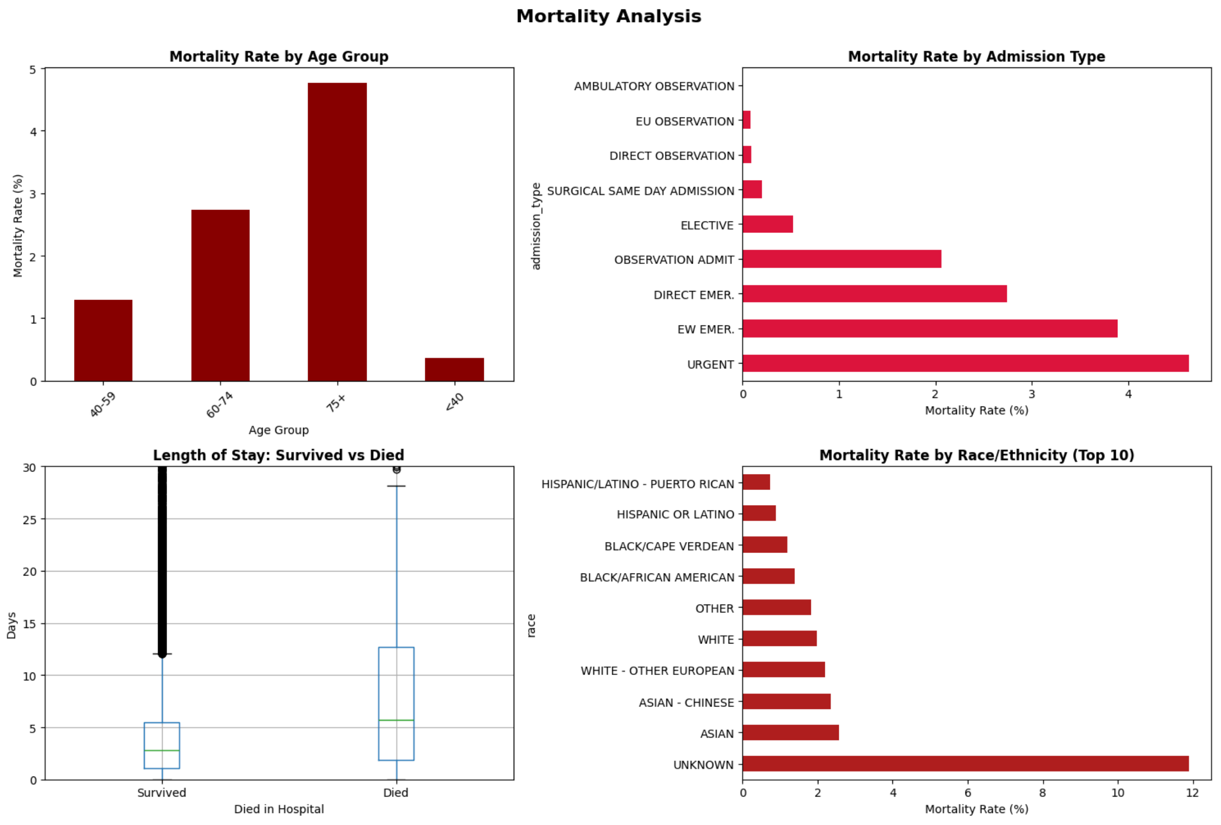
```
Mortality by Insurance Type
           sum  count      mean    rate_%
insurance
OTHER       52   1271  0.040913  4.091267
MEDICARE   703  22595  0.031113  3.111308
PRIVATE    204  15754  0.012949  1.294909
MEDICAID    99   9499  0.010422  1.042215
NO CHARGE    0     42  0.000000  0.000000
```

```
In [ ]: print("\nData Validation: Mortality vs Discharge Location")
        mortality_discharge_check = pd.crosstab(
            eda_df[mortality_col],
            eda_df['discharge_location'],
            margins=True,
            margins_name='Total'
        )
        print(mortality_discharge_check)
```

```
Data Validation: Mortality vs Discharge Location
discharge_location    ACUTE HOSPITAL  AGAINST ADVICE  ASSISTED LIVING  \
hospital_expire_flag
0                                223             314               63
1                                  0               0                0
Total                            223             314               63

discharge_location    CHRONIC/LONG TERM ACUTE CARE  DIED  HEALTHCARE FACILIT
Y  \
hospital_expire_flag
0                                              730    24
7
1                                                0  1075
0
Total                                          730  1099
7

discharge_location     HOME  HOME HEALTH CARE  HOSPICE  OTHER FACILITY  \
hospital_expire_flag
0                     17819              8926      481             147
1                        11                 0        0               0
Total                 17830              8926      481             147

discharge_location    PSYCH FACILITY  REHAB  SKILLED NURSING FACILITY  Total
hospital_expire_flag
0                                274   1325                      4766  35099
1                                  0      1                         3   1090
Total                            274   1326                      4769  36189
```

```
In [ ]: fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Mortality by age group
        mortality_by_age['rate_%'].plot(kind='bar', ax=axes[0,0], color='darkred')
```

```
axes[0,0].set_title('Mortality Rate by Age Group', fontweight='bold')
axes[0,0].set_xlabel('Age Group')
axes[0,0].set_ylabel('Mortality Rate (%)')
axes[0,0].tick_params(axis='x', rotation=45)

# Mortality by admission type
mortality_by_admission['rate_%'].sort_values(ascending=False).plot(kind='bar
axes[0,1].set_title('Mortality Rate by Admission Type', fontweight='bold')
axes[0,1].set_xlabel('Mortality Rate (%)')

# LOS comparison: survived vs died
eda_df.boxplot(column='los_days', by=mortality_col, ax=axes[1,0])
axes[1,0].set_title('Length of Stay: Survived vs Died', fontweight='bold')
axes[1,0].set_xlabel('Died in Hospital')
axes[1,0].set_ylabel('Days')
axes[1,0].set_xticklabels(['Survived', 'Died'])
axes[1,0].set_ylim(0, 30)
plt.suptitle('')  # Remove default title

# Mortality by race (top 10)
top_races = eda_df['race'].value_counts().head(10).index
mortality_by_race_top = eda_df[eda_df['race'].isin(top_races)].groupby('race
mortality_by_race_top.sort_values(ascending=False).plot(kind='barh', ax=axes
axes[1,1].set_title('Mortality Rate by Race/Ethnicity (Top 10)', fontweight=
axes[1,1].set_xlabel('Mortality Rate (%)')

plt.suptitle('Mortality Analysis', fontsize=16, fontweight='bold', y=1.00)
plt.tight_layout()
plt.savefig('mortality_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```



```
In [ ]: def analyze_clinical_text(text_series, text_len_series, name):
            """Comprehensive text analysis for RAG preparation"""

            print(f"\n{'='*70}")
            print(f"{name.upper()} ANALYSIS")
            print(f"{'='*70}")

            # Remove nulls
            texts = text_series.dropna()
            print(f"\nTotal Available: {len(texts):,} ({len(texts)/len(eda_df)*100:.

            if len(texts) == 0:
                print("No text data available!")
                return

            # Character length analysis (use pre-calculated if available)
            if text_len_series is not None:
                char_lengths = text_len_series.dropna()
            else:
                char_lengths = texts.str.len()

            print(f"\nCharacter Length Statistics")
            print(char_lengths.describe())

            # Word count analysis
            word_counts = texts.str.split().str.len()
            print(f"\nWord Count Statistics")
            print(word_counts.describe())

            # Identify suspiciously short texts
            short_threshold = 100
            very_short = (char_lengths < short_threshold).sum()
            print(f"\nSuspiciously Short Texts (< {short_threshold} chars): {very_sh

            # Medical terminology presence
            medical_terms = {
                'diagnosis': r'\bdiagnos(is|es|ed|ing)\b',
                'history': r'\bhistory\b',
                'examination': r'\bexamin(ation|ed|ing)\b',
                'treatment': r'\btreat(ment|ed|ing)\b',
                'symptoms': r'\bsymptom(s)?\b',
                'patient': r'\bpatient(s)?\b',
                'admitted': r'\badmit(ted|ting)?\b',
                'procedure': r'\bprocedure(s)?\b',
                'medication': r'\bmedication(s)?\b',
                'discharge': r'\bdischarge(d)?\b'
            }

            print(f"\nMedical Terminology Presence")
            for term, pattern in medical_terms.items():
                count = texts.str.contains(pattern, case=False, regex=True, na=False
                print(f"{term.capitalize():15}: {count:6,} ({count/len(texts)*100:5.

            # Visualize distributions
            fig, axes = plt.subplots(1, 2, figsize=(14, 5))
```

```
    # Character length distribution
    char_lengths.clip(upper=char_lengths.quantile(0.95)).hist(bins=50, ax=ax
                                                            color='teal'
    axes[0].set_title(f'{name} - Character Length Distribution (95th percent
    axes[0].set_xlabel('Characters')
    axes[0].set_ylabel('Frequency')
    axes[0].axvline(char_lengths.median(), color='red', linestyle='--', labe
    axes[0].legend()

    # Word count distribution
    word_counts.clip(upper=word_counts.quantile(0.95)).hist(bins=50, ax=axes
                                                         color='darkorar
    axes[1].set_title(f'{name} - Word Count Distribution (95th percentile)',
    axes[1].set_xlabel('Words')
    axes[1].set_ylabel('Frequency')
    axes[1].axvline(word_counts.median(), color='red', linestyle='--', label
    axes[1].legend()

    plt.tight_layout()
    plt.savefig(f'{name.lower().replace(" ", "_")}_analysis.png', dpi=300, b
    plt.show()

    # Sample text
    print(f"\nSample Text (First 500 characters)")
    sample_text = texts.iloc[0][:500]
    print(sample_text)
    print("...")
```

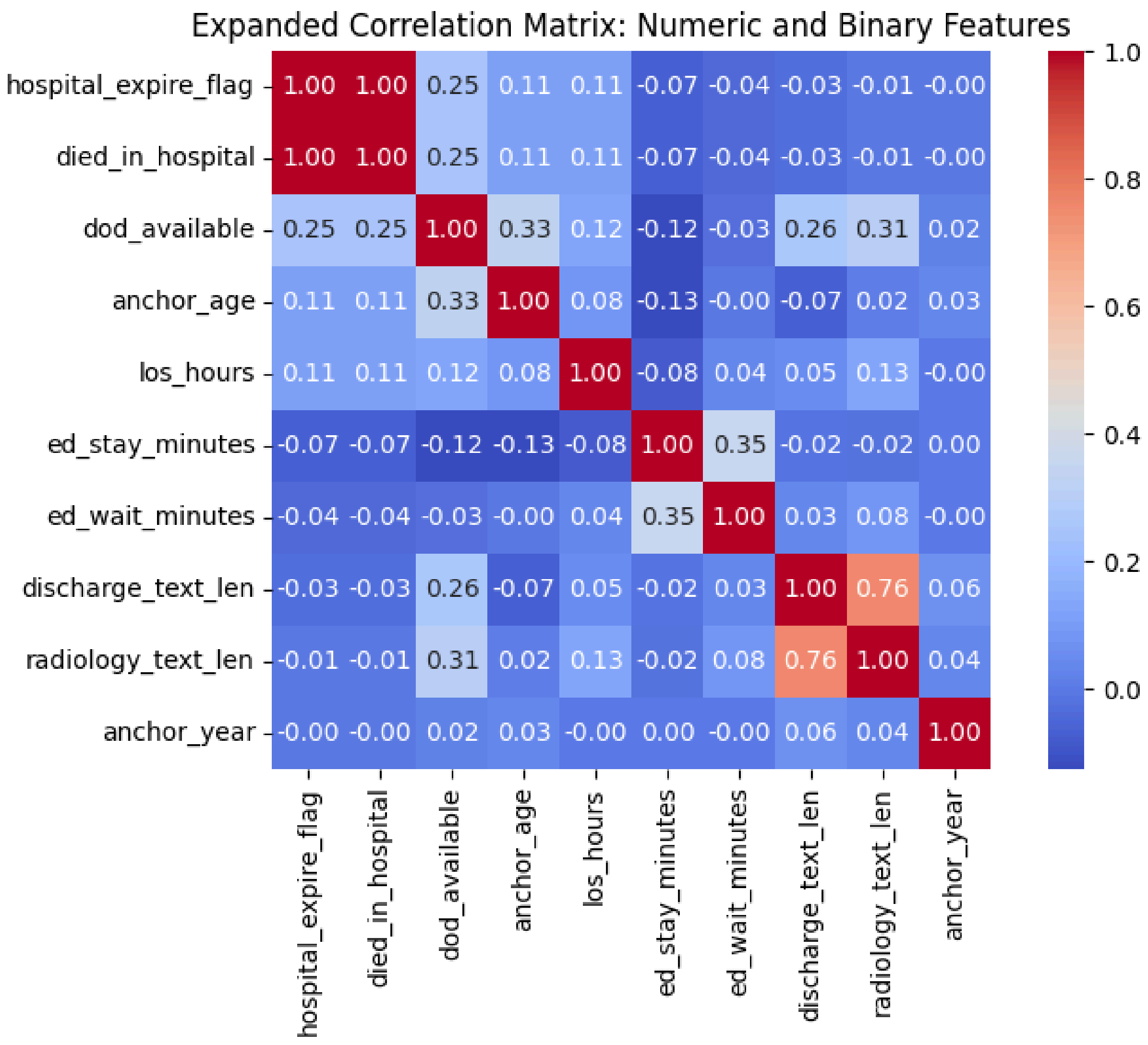
In [ ]:
```
"""
analyze_clinical_text(eda_df['discharge_text'],
                      eda_df['discharge_text_len'] if 'discharge_text_len' in
                      "DISCHARGE NOTES")
analyze_clinical_text(eda_df['radiology_text'],
                      eda_df['radiology_text_len'] if 'radiology_text_len' in
                      "RADIOLOGY REPORTS")
"""
```

Out[ ]:
```
'\nanalyze_clinical_text(eda_df[\'discharge_text\'],\n
eda_df[\'discharge_text_len\'] if \'discharge_text_len\' in eda_df.columns
else None,\n                     "DISCHARGE NOTES")\nanalyze_clinical_text
(eda_df[\'radiology_text\'],\n                     eda_df[\'radiology_text_
len\'] if \'radiology_text_len\' in eda_df.columns else None,\n
"RADIOLOGY REPORTS")\n'
```

Computing and visualizing correlations among key numerical and binary variables to identify relationships with in-hospital mortality. This helps highlight which features are most strongly associated with patient outcomes.

In [ ]:
```
temporal_cols = ['admittime', 'dischtime', 'edregtime', 'edouttime']
id_cols = ['subject_id', 'hadm_id', 'admit_provider_id']
text_cols = ['discharge_text', 'radiology_text', 'discharge_text_len', 'radi
categorical_cols = ['admission_type', 'admission_location', 'discharge_locat
                    'language', 'marital_status', 'race', 'gender', 'anchor_
numerical_cols = ['los_hours', 'ed_wait_minutes', 'ed_stay_minutes', 'anchor
```

```
                   'discharge_text_len', 'radiology_text_len']
binary_cols = ['hospital_expire_flag', 'died_in_hospital', 'dod_available']

# Subset and prepare correlation-ready DataFrame
corr_cols = numerical_cols + binary_cols
corr_df = eda_df[corr_cols].copy()

# Ensure binary columns are numeric (just in case)
for c in binary_cols:
    if corr_df[c].dtype == 'object':
        corr_df[c] = corr_df[c].map({'Yes': 1, 'No': 0}).astype(float)

# Compute correlation matrix
corr_matrix = corr_df.corr(numeric_only=True)

# Sort by strength of correlation with mortality
if 'died_in_hospital' in corr_matrix.columns:
    sorted_cols = corr_matrix['died_in_hospital'].abs().sort_values(ascendir
    corr_matrix = corr_matrix.loc[sorted_cols, sorted_cols]

# Visualization
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", square=True
plt.title("Expanded Correlation Matrix: Numeric and Binary Features")
plt.tight_layout()
plt.show()
```



	hospital_expire_flag	died_in_hospital	dod_available	anchor_age	los_hours	ed_stay_minutes	ed_wait_minutes	discharge_text_len	radiology_text_len	anchor_year
hospital_expire_flag	1.00	1.00	0.25	0.11	0.11	-0.07	-0.04	-0.03	-0.01	-0.00
died_in_hospital	1.00	1.00	0.25	0.11	0.11	-0.07	-0.04	-0.03	-0.01	-0.00
dod_available	0.25	0.25	1.00	0.33	0.12	-0.12	-0.03	0.26	0.31	0.02
anchor_age	0.11	0.11	0.33	1.00	0.08	-0.13	-0.00	-0.07	0.02	0.03
los_hours	0.11	0.11	0.12	0.08	1.00	-0.08	0.04	0.05	0.13	-0.00
ed_stay_minutes	-0.07	-0.07	-0.12	-0.13	-0.08	1.00	0.35	-0.02	-0.02	0.00
ed_wait_minutes	-0.04	-0.04	-0.03	-0.00	0.04	0.35	1.00	0.03	0.08	-0.00
discharge_text_len	-0.03	-0.03	0.26	-0.07	0.05	-0.02	0.03	1.00	0.76	0.06
radiology_text_len	-0.01	-0.01	0.31	0.02	0.13	-0.02	0.08	0.76	1.00	0.04
anchor_year	-0.00	-0.00	0.02	0.03	-0.00	0.00	-0.00	0.06	0.04	1.00

Identifying key factors potentially linked to patient outcomes.

In [ ]:
```
# Top correlated features with mortality
top_corr = corr_matrix['died_in_hospital'].drop('died_in_hospital').sort_val
print("\nTop correlations with mortality (died_in_hospital):")
print(top_corr)
```

```
Top correlations with mortality (died_in_hospital):
hospital_expire_flag    1.000000
dod_available           0.248878
anchor_age              0.106551
los_hours               0.106355
ed_stay_minutes        -0.068048
ed_wait_minutes        -0.044221
discharge_text_len     -0.031312
radiology_text_len     -0.008705
anchor_year            -0.003191
Name: died_in_hospital, dtype: float64
```

The strongest correlation with in-hospital mortality is the hospital_expire_flag as expected, this indicates perfect alignment between the two indicators of death. Other moderate correlations include date of death availability (dod_available) and length of stay (los_hours), suggesting that longer hospital stays and older age may be associated with higher mortality.

Most other features show weak correlations, indicating limited linear relationships with mortality.

Statistical Analysis: Length of Stay and Mortality (Welch's t-test)

Performing a Welch’s t-test to compare the average length of stay between patients who died in the hospital and those who survived.

This is to test whether the difference in means (averages) is statistically significant.

Hypotheses:

- **Null Hypothesis (H0):** The average length of hospital stay for patients who died is equal to the average length of hospital stay for patients who survived.
- **Alternative Hypothesis (H1):** The average length of hospital stay for patients who died is not equal to the average length of hospital stay for patients who survived.

```
In [ ]: # Split groups
died = eda_df.loc[eda_df['died_in_hospital'] == 1, 'los_hours'].dropna()
survived = eda_df.loc[eda_df['died_in_hospital'] == 0, 'los_hours'].dropna()

# Performing Welch's t-test
t_stat, p_val = ttest_ind(died, survived, equal_var=False)

print(f"T-statistic: {t_stat:.3f}")
print(f"P-value: {p_val:.4f}")

if p_val < 0.05:
    print("Statistically significant difference: hospital stay is longer for
else:
    print("No statistically significant difference detected.")
```

```
T-statistic: 11.435
P-value: 0.0000
Statistically significant difference: hospital stay is longer for patients w
ho died.
```

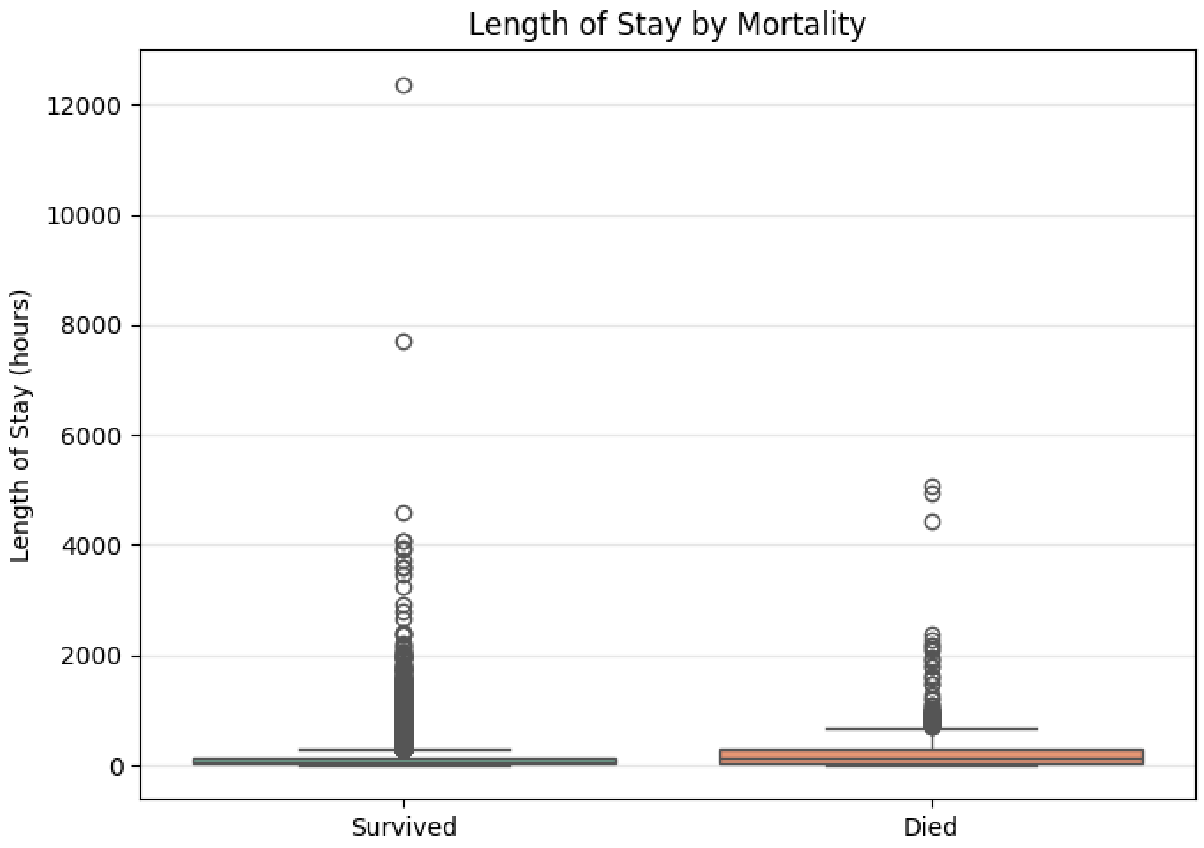
The results show a highly significant difference in hospital stay length between patients who died and those who survived. Patients who died tended to have **substantially longer hospital stays**, as indicated by the **large t-statistic and zero p-value**.

This visualization compares the distribution of hospital stay lengths between patients who survived and those who died.

It highlights that patients who died generally had longer hospital stays, consistent with the statistical test results.

In [ ]:
```python
# Visualization
plt.figure(figsize=(7, 5))
sns.boxplot(x='died_in_hospital', y='los_hours', data=eda_df, palette='Set2'
plt.xticks([0, 1], ['Survived', 'Died'])
plt.title('Length of Stay by Mortality')
plt.xlabel('')
plt.ylabel('Length of Stay (hours)')
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-1185102687.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sns.boxplot(x='died_in_hospital', y='los_hours', data=eda_df, palette='Set
2')
```



This boxplot shows that patients who **died in the hospital generally had longer lengths of stay** compared to those who survived.

The wide range and presence of extreme outliers indicate that while most patients had relatively short stays, a subset—especially among those who died—remained hospitalized much longer.

Statistical Analysis: Gender and Mortality (Chi-Squared Test)

Performing a Chi-Squared test to determine if there is a statistically significant association between patient gender and in-hospital mortality.

The Chi-Squared test was used for the analysis of Admission Type and Mortality because it's the appropriate statistical test for examining the association between two categorical variables.

Admission Type is a categorical variable (e.g., 'EW EMER.', 'ELECTIVE', 'URGENT'). In-hospital mortality is also a categorical variable (binary: Died/Survived, or 0/1). The Chi-Squared test helps us determine if the observed frequencies in each category combination (e.g., how many patients admitted via 'EW EMER.' died vs. survived) are significantly different from what would be expected if there were no association between the two variables.

Hypotheses:

- **Null Hypothesis (H0):** There is no statistically significant association between patient gender and in-hospital mortality.
- **Alternative Hypothesis (H1):** There is a statistically significant association between patient gender and in-hospital mortality.

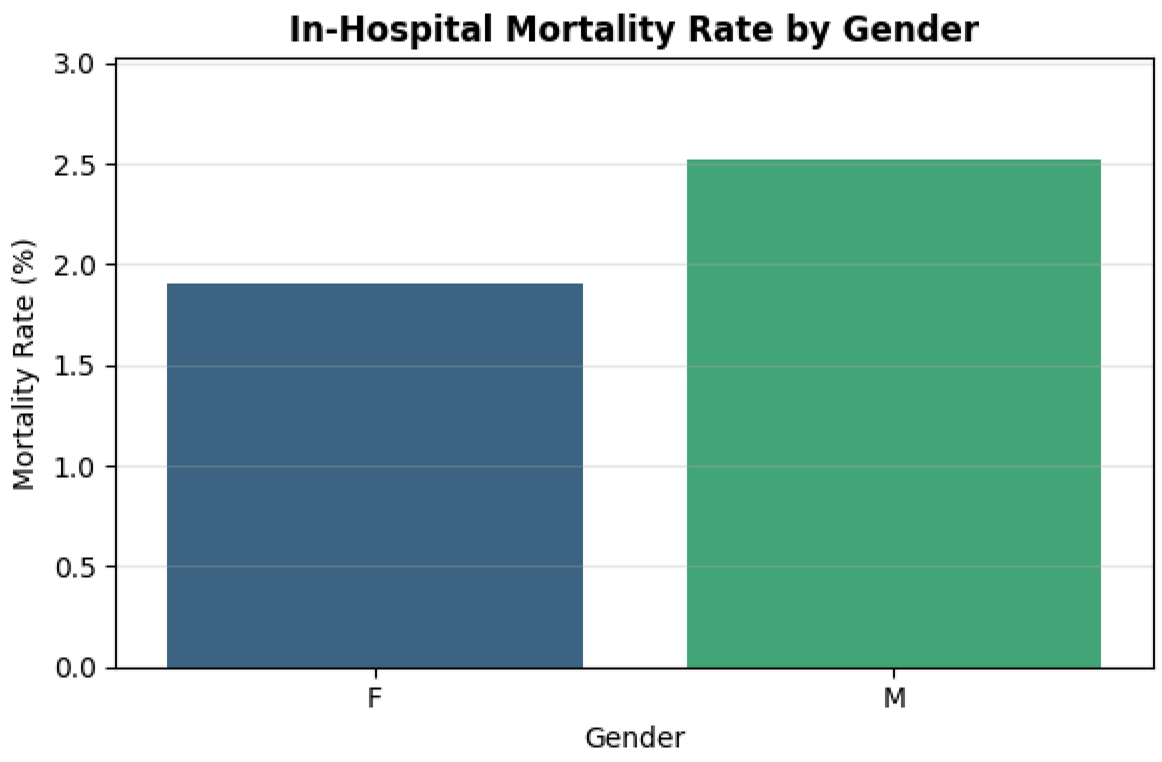
Performing a Chi-Squared test to determine if there is a statistically significant association between patient gender and in-hospital mortality.

In [ ]:
```
from scipy.stats import chi2_contingency

# Ensure mortality column is correctly identified
if 'hospital_expire_flag' in eda_df.columns:
    mortality_col = 'hospital_expire_flag'
else:
    mortality_col = 'died_in_hospital'

# Create a contingency table of Gender and Mortality
contingency_table = pd.crosstab(eda_df['gender'], eda_df[mortality_col])

print("Contingency Table (Gender vs. Mortality):")
print(contingency_table)

# Perform the Chi-Squared test
chi2, p, dof, expected = chi2_contingency(contingency_table)

print(f"\nChi-Squared Statistic: {chi2:.3f}")
print(f"P-value: {p:.4f}")
print(f"Degrees of Freedom: {dof}")

if p < 0.05:
    print("\nResult: There is a statistically significant association betwee
else:
    print("\nResult: There is no statistically significant association betwe
Contingency Table (Gender vs. Mortality):
hospital_expire_flag      0    1
gender
F                     25109  487
M                     23789  615

Chi-Squared Statistic: 21.811
P-value: 0.0000
Degrees of Freedom: 1

Result: There is a statistically significant association between gender and
in-hospital mortality.
```

This visualization shows the mortality rate for each gender, visually supporting the findings of the Chi-Squared test.

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Calculate mortality rate by gender
mortality_by_gender_plot = eda_df.groupby('gender')[mortality_col].mean().re
mortality_by_gender_plot['rate_%'] = mortality_by_gender_plot[mortality_col]

plt.figure(figsize=(6, 4))
sns.barplot(x='gender', y='rate_%', data=mortality_by_gender_plot, palette='
plt.title('In-Hospital Mortality Rate by Gender', fontweight='bold')
plt.xlabel('Gender')
plt.ylabel('Mortality Rate (%)')
plt.ylim(0, mortality_by_gender_plot['rate_%'].max() * 1.2) # Add some paddi
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-2788157173.py:9: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sns.barplot(x='gender', y='rate_%', data=mortality_by_gender_plot, palette
='viridis')
```



Statistical Analysis: Admission Type and Mortality (Chi-Squared Test)

Performing a Chi-Squared test to determine if there is a statistically significant association between patient admission type and in-hospital mortality.

The Chi-Squared test was used for the analysis of Admission Type and Mortality because it's the appropriate statistical test for examining the association between two categorical variables.

Admission Type is a categorical variable (e.g., 'EW EMER.', 'ELECTIVE', 'URGENT'). In-hospital mortality is also a categorical variable (binary: Died/Survived, or 0/1). The Chi-Squared test helps us determine if the observed frequencies in each category combination (e.g., how many patients admitted via 'EW EMER.' died vs. survived) are significantly different from what would be expected if there were no association between the two variables.

Hypotheses:

- **Null Hypothesis (H0):** There is no statistically significant association between patient admission type and in-hospital mortality.
- **Alternative Hypothesis (H1):** There is a statistically significant association between patient admission type and in-hospital mortality.

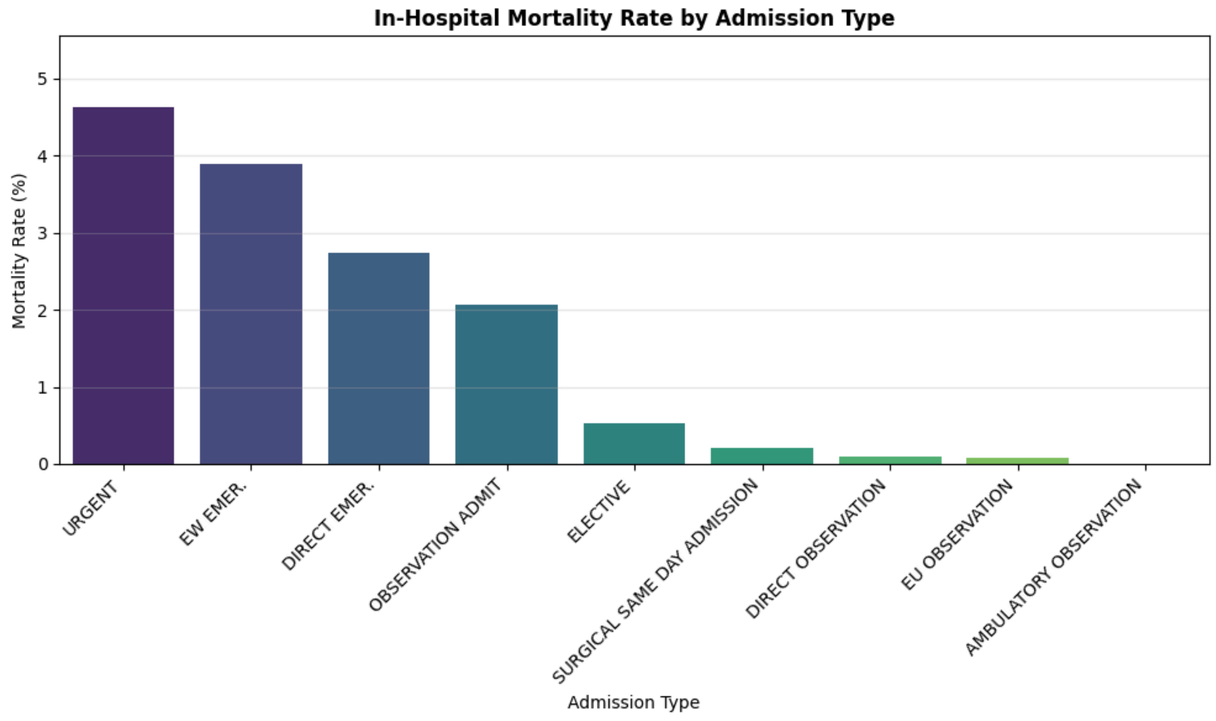
Performing a Chi-Squared test to determine if there is a statistically significant association between patient admission type and in-hospital mortality.

```
In [ ]: from scipy.stats import chi2_contingency
        import pandas as pd

        # Ensure mortality column is correctly identified
        if 'hospital_expire_flag' in eda_df.columns:
            mortality_col = 'hospital_expire_flag'
        else:
            mortality_col = 'died_in_hospital'

        # Create a contingency table of Admission Type and Mortality
        contingency_table_admission = pd.crosstab(eda_df['admission_type'], eda_df[m

        print("Contingency Table (Admission Type vs. Mortality):")
        print(contingency_table_admission)

        # Perform the Chi-Squared test
        chi2_admission, p_admission, dof_admission, expected_admission = chi2_contir

        print(f"\nChi-Squared Statistic: {chi2_admission:.3f}")
        print(f"P-value: {p_admission:.4f}")
        print(f"Degrees of Freedom: {dof_admission}")

        if p_admission < 0.05:
            print("\nResult: There is a statistically significant association betwee
        else:
            print("\nResult: There is no statistically significant association betwe
```

```
Contingency Table (Admission Type vs. Mortality):
hospital_expire_flag             0    1
admission_type
AMBULATORY OBSERVATION         613    0
DIRECT EMER.                  1950   55
DIRECT OBSERVATION            2232    2
ELECTIVE                      1141    6
EU OBSERVATION               11094    9
EW EMER.                     15497  627
OBSERVATION ADMIT             7545  159
SURGICAL SAME DAY ADMISSION   3960    8
URGENT                        4866  236

Chi-Squared Statistic: 735.674
P-value: 0.0000
Degrees of Freedom: 8

Result: There is a statistically significant association between admission t
ype and in-hospital mortality.
```

```
In [ ]: import matplotlib.pyplot as plt
        import seaborn as sns

        # Calculate mortality rate by admission type
        mortality_by_admission_plot = eda_df.groupby('admission_type')[mortality_col
        mortality_by_admission_plot['rate_%'] = mortality_by_admission_plot[mortalit

        plt.figure(figsize=(10, 6))
        sns.barplot(x='admission_type', y='rate_%', data=mortality_by_admission_plot
        plt.title('In-Hospital Mortality Rate by Admission Type', fontweight='bold')
        plt.xlabel('Admission Type')
        plt.ylabel('Mortality Rate (%)')
        plt.ylim(0, mortality_by_admission_plot['rate_%'].max() * 1.2) # Add some pa
        plt.xticks(rotation=45, ha='right') # Rotate labels for better readability
        plt.grid(axis='y', alpha=0.3)
        plt.tight_layout()
        plt.show()
```

```
/tmp/ipython-input-495625549.py:9: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sns.barplot(x='admission_type', y='rate_%', data=mortality_by_admission_pl
ot.sort_values('rate_%', ascending=False), palette='viridis')
```



Statistical Analysis: Age and Mortality (Independent t-test)

Performing an independent t-test to compare the average age of patients who died in the hospital versus those who survived.

The independent t-test is used here because we are comparing the means of a continuous variable (patient age) between two independent groups (patients who died in the hospital and patients who survived). This test helps determine if the observed difference in average age between these two groups is statistically significant or likely due to random chance.

Hypotheses:

- **Null Hypothesis (H0):** The average age of patients who died in the hospital is equal to the average age of patients who survived.
- **Alternative Hypothesis (H1):** The average age of patients who died in the hospital is not equal to the average age of patients who survived.

Performing an independent t-test to compare the average age of patients who died in the hospital versus those who survived.

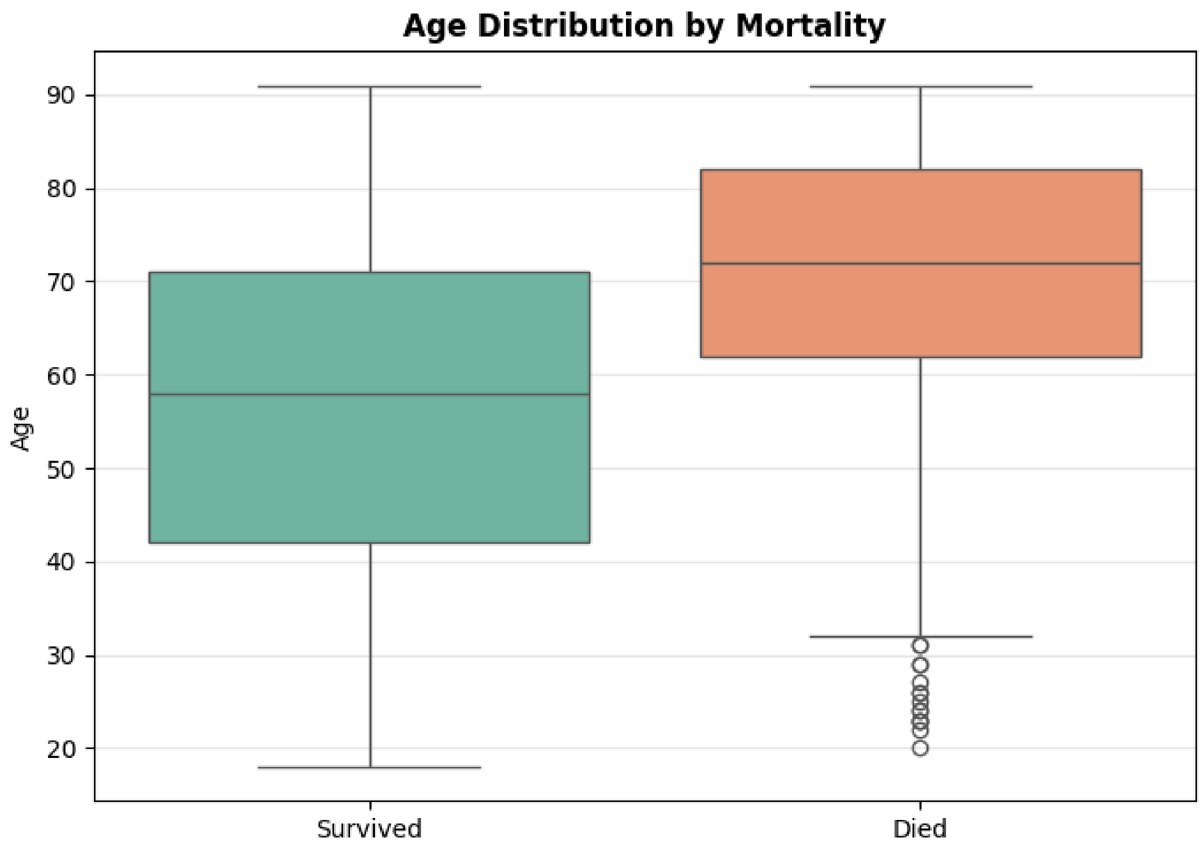
```
In [ ]: from scipy.stats import ttest_ind

        # Ensure mortality column is correctly identified
        if 'hospital_expire_flag' in eda_df.columns:
            mortality_col = 'hospital_expire_flag'
        else:
            mortality_col = 'died_in_hospital'

        # Separate age data for the two groups (died vs. survived)
        age_died = eda_df.loc[eda_df[mortality_col] == 1, 'anchor_age'].dropna()
        age_survived = eda_df.loc[eda_df[mortality_col] == 0, 'anchor_age'].dropna()

        # Perform the independent t-test (assuming unequal variances - Welch's t-tes
        t_stat, p_val = ttest_ind(age_died, age_survived, equal_var=False)

        print(f"T-statistic: {t_stat:.3f}")
        print(f"P-value: {p_val:.4f}")

        if p_val < 0.05:
            print("\nResult: There is a statistically significant difference in aver
            print(f"Average age (Died): {age_died.mean():.2f}")
            print(f"Average age (Survived): {age_survived.mean():.2f}")
        else:
            print("\nResult: There is no statistically significant difference in ave
```

```
T-statistic: 30.940
P-value: 0.0000

Result: There is a statistically significant difference in average age betwe
en patients who died and those who survived.
Average age (Died): 70.52
Average age (Survived): 56.67
```

```
In [ ]: import matplotlib.pyplot as plt
        import seaborn as sns

        # Ensure mortality column is correctly identified
        if 'hospital_expire_flag' in eda_df.columns:
            mortality_col = 'hospital_expire_flag'
        else:
            mortality_col = 'died_in_hospital'

        # Visualization
        plt.figure(figsize=(7, 5))
        sns.boxplot(x=mortality_col, y='anchor_age', data=eda_df, palette='Set2')
        plt.xticks([0, 1], ['Survived', 'Died'])
```

```python
plt.title('Age Distribution by Mortality', fontweight='bold')
plt.xlabel('')
plt.ylabel('Age')
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-2486593892.py:12: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sns.boxplot(x=mortality_col, y='anchor_age', data=eda_df, palette='Set2')
```



Summary of Findings and Conclusion

Based on the exploratory data analysis and the statistical tests performed, here is a summary of the key findings regarding the factors associated with in-hospital mortality:

Overall Mortality: The dataset shows an overall in-hospital mortality rate of approximately 2.20%.

Length of Stay: Patients who died in the hospital had significantly longer lengths of stay compared to those who survived (as demonstrated by the Welch's t-test with a highly significant p-value). The correlation analysis also showed a **positive correlation** between `los_hours` and `died_in_hospital`, supporting this finding.

Age: There is a statistically significant difference in average age between patients who died and those who survived. Patients who died were, on average, older (mean age ~70.5) than those who survived (mean age ~56.7) (as shown by the independent t-test with a highly significant p-value). The mortality rate increases with age, with the highest rate observed in the 75+ age group. The correlation analysis showed a **positive correlation** between `anchor_age` and `died_in_hospital`.

Gender: The Chi-Squared test revealed a statistically significant association between patient gender and in-hospital mortality. While the difference in mortality rates between genders is not as pronounced as with age or length of stay, the analysis suggests that gender is a statistically relevant factor. (Visualizations show a slightly higher mortality rate for males compared to females).

Admission Type: The Chi-Squared test demonstrated a highly statistically significant association between admission type and in-hospital mortality. Certain admission types, particularly `URGENT` and `EW EMER.`, are associated with substantially higher mortality rates compared to elective or observation-based admissions. This highlights the acuity and severity of illness upon arrival as a critical factor.

Race/Ethnicity and Insurance: While the Chi-Squared tests for these variables were not explicitly performed in the provided cells, the mortality rate analysis by group showed variations, notably a higher mortality rate in the `UNKNOWN` race category and among patients with `OTHER` and `MEDICARE` insurance types. Further statistical testing would be needed to confirm the significance of these associations.

Text Data Completeness: A significant portion of records have missing discharge and radiology text data (22.3% and 17.5% respectively). While this doesn't directly impact the current mortality analysis based on structured data, it is a crucial data quality issue for the downstream RAG model, which relies heavily on this unstructured text.

Conclusion:

The exploratory data analysis and statistical tests strongly indicate that patient age, length of hospital stay, gender, and admission type are significant factors associated with in-hospital mortality in this dataset. Older patients, those with longer hospitalizations, and those admitted through emergency pathways (EW EMER., URGENT) have a higher risk of mortality. While gender shows a statistically significant association, the practical difference in mortality rate is less pronounced compared to age and admission type. The availability of unstructured text data is a notable limitation for future RAG model development. These findings are crucial for building a predictive model and for interpreting the outputs of the Medical Evidence Synthesizer.