Novartis Datathon 2025 - Copilot Agent Master Todo

> **Purpose:** This file provides step-by-step instructions for GitHub Copilot Agent to build the entire project from scratch.
> **Execution:** Work through each step sequentially. Mark `[x]` when complete. Do NOT skip steps. **Local Run:** All Python files are provided in complete `.py` format with demo scripts for testing.

Project Summary

Aspect	Details
Goal	Predict 24-month post-generic sales erosion for pharmaceutical drugs
Target	`volume` (monthly sales units)
Scenarios	Scenario 1: Predict months 0-23 (no post-entry data)
	Scenario 2: Predict months 6-23 (months 0-5 provided)
Critical Focus	Bucket 1 (high erosion, mean 0-0.25) weighted 2×
Metric	Custom Prediction Error (PE), lower is better

STEP 1: Project Setup

1.1 Create Directory Structure

PowerShell Commands to create all directories:

```powershell
# Run from Main_project folder
cd D:\Datathon\novartis_datathon_2025\Main_project

# Create all directories
New-Item -ItemType Directory -Force -Path "data\raw"
New-Item -ItemType Directory -Force -Path "data\processed"
New-Item -ItemType Directory -Force -Path "src"
New-Item -ItemType Directory -Force -Path "notebooks"
New-Item -ItemType Directory -Force -Path "models"
New-Item -ItemType Directory -Force -Path "submissions"
New-Item -ItemType Directory -Force -Path "reports\figures"
```

- [ ] **1.1.1** Create folder: `Main_project/data/raw/`
- [ ] **1.1.2** Create folder: `Main_project/data/processed/`
- [ ] **1.1.3** Create folder: `Main_project/src/`

- [ ] **1.1.4** Create folder: `Main_project/notebooks/`
- [ ] **1.1.5** Create folder: `Main_project/models/`
- [ ] **1.1.6** Create folder: `Main_project/submissions/`
- [ ] **1.1.7** Create folder: `Main_project/reports/`
- [ ] **1.1.8** Create folder: `Main_project/reports/figures/`

1.2 Copy Data Files

PowerShell Commands to copy all data files:

```
# Run from Main_project folder
cd D:\Datathon\novartis_datathon_2025\Main_project

# Copy training data
Copy-Item "..\SUBMISSION\Data files\TRAIN\df_volume_train.csv" -Destination
"data\raw\"
Copy-Item "..\SUBMISSION\Data files\TRAIN\df_generics_train.csv" -Destination
"data\raw\"
Copy-Item "..\SUBMISSION\Data files\TRAIN\df_medicine_info_train.csv" -
Destination "data\raw\"

# Copy test data
Copy-Item "..\SUBMISSION\Data files\TEST\df_volume_test.csv" -Destination
"data\raw\"
Copy-Item "..\SUBMISSION\Data files\TEST\df_generics_test.csv" -Destination
"data\raw\"
Copy-Item "..\SUBMISSION\Data files\TEST\df_medicine_info_test.csv" -
Destination "data\raw\"

# Copy metric and submission files
Copy-Item "..\SUBMISSION\Metric files\metric_calculation.py" -Destination
"src\"
Copy-Item "..\SUBMISSION\Submission example\submission_template.csv" -
Destination "submissions\"
Copy-Item "..\SUBMISSION\Submission example\submission_example.csv" -
Destination "submissions\"
```

- [ ] **1.2.1** Copy `SUBMISSION/Data files/TRAIN/df_volume_train.csv` → `data/raw/`
- [ ] **1.2.2** Copy `SUBMISSION/Data files/TRAIN/df_generics_train.csv` → `data/raw/`
- [ ] **1.2.3** Copy `SUBMISSION/Data files/TRAIN/df_medicine_info_train.csv` → `data/raw/`
- [ ] **1.2.4** Copy `SUBMISSION/Data files/TEST/df_volume_test.csv` → `data/raw/`
- [ ] **1.2.5** Copy `SUBMISSION/Data files/TEST/df_generics_test.csv` → `data/raw/`
- [ ] **1.2.6** Copy `SUBMISSION/Data files/TEST/df_medicine_info_test.csv` → `data/raw/`
- [ ] **1.2.7** Copy `SUBMISSION/Metric files/metric_calculation.py` → `src/`
- [ ] **1.2.8** Copy `SUBMISSION/Submission example/submission_template.csv` → `submissions/`
- [ ] **1.2.9** Copy `SUBMISSION/Submission example/submission_example.csv` → `submissions/`

1.3 Verify Dependencies

Create/Update `requirements.txt`:

```
# Core data processing
pandas>=1.5.0
numpy>=1.23.0
scipy>=1.9.0

# Machine Learning
scikit-learn>=1.1.0
lightgbm>=3.3.0
xgboost>=1.7.0
catboost>=1.1.0

# Time Series (optional)
statsmodels>=0.13.0

# Visualization
matplotlib>=3.6.0
seaborn>=0.12.0
plotly>=5.10.0

# Utilities
tqdm>=4.64.0
joblib>=1.2.0

# Jupyter
jupyter>=1.0.0
notebook>=6.5.0
ipykernel>=6.17.0
```

Install dependencies:

```
# Activate virtual environment first
& D:\Datathon\novartis_datathon_2025\saeed_venv\Scripts\Activate.ps1

# Install requirements
pip install -r requirements.txt
```

- ☐ **1.3.1** Check `requirements.txt` has all needed packages
- ☐ **1.3.2** Install: `pip install -r requirements.txt`

🚀 STEP 2: Create Core Python Modules (`.py` files)

> 🗀 **Architecture:** All core logic in `.py` files for testability, reusability, and clean version control. Notebooks (`.ipynb`) used ONLY for visualization and interactive exploration.

2.1 Create `src/config.py`

- [ ] **2.1.1** Create configuration file:

```
#
================================================================================
# File: src/config.py
# Description: All project paths, constants, and configuration
#
================================================================================

from pathlib import Path
import os

#
================================================================================
# PATHS
#
================================================================================
PROJECT_ROOT = Path(__file__).parent.parent
DATA_RAW = PROJECT_ROOT / "data" / "raw"
DATA_PROCESSED = PROJECT_ROOT / "data" / "processed"
MODELS_DIR = PROJECT_ROOT / "models"
SUBMISSIONS_DIR = PROJECT_ROOT / "submissions"
REPORTS_DIR = PROJECT_ROOT / "reports"
FIGURES_DIR = REPORTS_DIR / "figures"

# Create directories if they don't exist
for dir_path in [DATA_RAW, DATA_PROCESSED, MODELS_DIR, SUBMISSIONS_DIR,
REPORTS_DIR, FIGURES_DIR]:
    dir_path.mkdir(parents=True, exist_ok=True)

#
================================================================================
# DATA FILES
#
================================================================================
VOLUME_TRAIN = DATA_RAW / "df_volume_train.csv"
GENERICS_TRAIN = DATA_RAW / "df_generics_train.csv"
MEDICINE_INFO_TRAIN = DATA_RAW / "df_medicine_info_train.csv"
VOLUME_TEST = DATA_RAW / "df_volume_test.csv"
GENERICS_TEST = DATA_RAW / "df_generics_test.csv"
MEDICINE_INFO_TEST = DATA_RAW / "df_medicine_info_test.csv"

#
================================================================================
# CONSTANTS
#
================================================================================
PRE_ENTRY_MONTHS = 12      # Months before generic entry for Avg_j calculation
POST_ENTRY_MONTHS = 24     # Months to forecast (0-23)
BUCKET_1_THRESHOLD = 0.25  # Mean erosion <= 0.25 = Bucket 1 (high erosion)
```

```
BUCKET_1_WEIGHT = 2        # Bucket 1 weighted 2x in metric
BUCKET_2_WEIGHT = 1        # Bucket 2 weighted 1x in metric

#
================================================================================
# METRIC WEIGHTS - SCENARIO 1 (Predict months 0-23)
#
================================================================================
S1_MONTHLY_WEIGHT = 0.2      # Weight for monthly absolute errors
S1_SUM_0_5_WEIGHT = 0.5      # Weight for accumulated error months 0-5
S1_SUM_6_11_WEIGHT = 0.2     # Weight for accumulated error months 6-11
S1_SUM_12_23_WEIGHT = 0.1    # Weight for accumulated error months 12-23

#
================================================================================
# METRIC WEIGHTS - SCENARIO 2 (Predict months 6-23, given 0-5 actuals)
#
================================================================================
S2_MONTHLY_WEIGHT = 0.2      # Weight for monthly absolute errors
S2_SUM_6_11_WEIGHT = 0.5     # Weight for accumulated error months 6-11
S2_SUM_12_23_WEIGHT = 0.3    # Weight for accumulated error months 12-23

#
================================================================================
# MODEL PARAMETERS
#
================================================================================
RANDOM_STATE = 42
TEST_SIZE = 0.2
N_SPLITS_CV = 5

# LightGBM default parameters
LGBM_PARAMS = {
    'objective': 'regression',
    'metric': 'rmse',
    'boosting_type': 'gbdt',
    'num_leaves': 31,
    'learning_rate': 0.05,
    'feature_fraction': 0.8,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'verbose': -1,
    'n_estimators': 500,
    'random_state': RANDOM_STATE
}

# XGBoost default parameters
XGB_PARAMS = {
    'objective': 'reg:squarederror',
    'eval_metric': 'rmse',
    'max_depth': 6,
    'learning_rate': 0.05,
    'n_estimators': 500,
    'subsample': 0.8,
```

```
    'colsample_bytree': 0.8,
    'random_state': RANDOM_STATE
}

#
==============================================================================
# FEATURE COLUMNS
#
==============================================================================
ID_COLS = ['country', 'brand_name']
TIME_COLS = ['month', 'months_postgx']
TARGET_COL = 'volume'

CATEGORICAL_COLS = ['country', 'ther_area', 'main_package']
BINARY_COLS = ['biological', 'small_molecule']
NUMERIC_COLS = ['hospital_rate', 'n_gxs']


if __name__ == "__main__":
    # Test configuration
    print(f"Project root: {PROJECT_ROOT}")
    print(f"Data raw: {DATA_RAW}")
    print(f"All directories created successfully!")
```

2.2 Create `src/data_loader.py`

- ☐ **2.2.1** Create data loading module:

```
#
==============================================================================
# File: src/data_loader.py
# Description: Functions to load, validate, and merge all datasets
#
==============================================================================

import pandas as pd
import numpy as np
from pathlib import Path
import sys

# Add src to path for imports
sys.path.insert(0, str(Path(__file__).parent))
from config import *


def load_volume_data(train: bool = True) -> pd.DataFrame:
    """
    Load volume dataset (train or test).

    Args:
```

```
        train: If True, load training data; else load test data

    Returns:
        DataFrame with columns: country, brand_name, month, months_postgx,
volume
    """
    path = VOLUME_TRAIN if train else VOLUME_TEST
    df = pd.read_csv(path)

    required_cols = ['country', 'brand_name', 'month', 'months_postgx',
'volume']
    missing = set(required_cols) - set(df.columns)
    if missing:
        raise ValueError(f"Missing columns in volume data: {missing}")

    print(f"☑ Loaded volume {'train' if train else 'test'}: {df.shape}")
    return df


def load_generics_data(train: bool = True) -> pd.DataFrame:
    """
    Load generics dataset (train or test).

    Args:
        train: If True, load training data; else load test data

    Returns:
        DataFrame with columns: country, brand_name, months_postgx, n_gxs
    """
    path = GENERICS_TRAIN if train else GENERICS_TEST
    df = pd.read_csv(path)

    required_cols = ['country', 'brand_name', 'months_postgx', 'n_gxs']
    missing = set(required_cols) - set(df.columns)
    if missing:
        raise ValueError(f"Missing columns in generics data: {missing}")

    print(f"☑ Loaded generics {'train' if train else 'test'}: {df.shape}")
    return df


def load_medicine_info(train: bool = True) -> pd.DataFrame:
    """
    Load medicine info dataset (train or test).

    Args:
        train: If True, load training data; else load test data

    Returns:
        DataFrame with columns: country, brand_name, ther_area, hospital_rate,
                                main_package, biological, small_molecule
    """
    path = MEDICINE_INFO_TRAIN if train else MEDICINE_INFO_TEST
    df = pd.read_csv(path)
```

```
    print(f"☑ Loaded medicine_info {'train' if train else 'test'}:
{df.shape}")
    return df


def load_all_data(train: bool = True) -> tuple:
    """
    Load all three datasets.

    Args:
        train: If True, load training data; else load test data

    Returns:
        Tuple of (volume_df, generics_df, medicine_df)
    """
    volume = load_volume_data(train)
    generics = load_generics_data(train)
    medicine = load_medicine_info(train)
    return volume, generics, medicine


def merge_datasets(volume_df: pd.DataFrame,
                   generics_df: pd.DataFrame,
                   medicine_df: pd.DataFrame) -> pd.DataFrame:
    """
    Merge all datasets into unified modeling table.

    Merge strategy:
    1. volume + generics on (country, brand_name, months_postgx)
    2. result + medicine_info on (country, brand_name)

    Args:
        volume_df: Volume dataset
        generics_df: Generics dataset
        medicine_df: Medicine info dataset

    Returns:
        Merged DataFrame
    """
    # Merge volume + generics
    merged = volume_df.merge(
        generics_df,
        on=['country', 'brand_name', 'months_postgx'],
        how='left'
    )

    # Merge with medicine_info
    merged = merged.merge(
        medicine_df,
        on=['country', 'brand_name'],
        how='left'
    )
```

```
    print(f"☑ Merged dataset shape: {merged.shape}")
    return merged


def validate_data(df: pd.DataFrame, name: str = "data") -> dict:
    """
    Validate dataset and return quality report.

    Args:
        df: DataFrame to validate
        name: Name for reporting

    Returns:
        Dictionary with validation results
    """
    report = {
        'name': name,
        'shape': df.shape,
        'missing_values': df.isnull().sum().to_dict(),
        'duplicates': df.duplicated().sum(),
        'dtypes': df.dtypes.to_dict()
    }

    # Check for negative volumes
    if 'volume' in df.columns:
        report['negative_volumes'] = (df['volume'] < 0).sum()

    # Check hospital_rate range
    if 'hospital_rate' in df.columns:
        report['hospital_rate_out_of_range'] = (
            (df['hospital_rate'] < 0) | (df['hospital_rate'] > 100)
        ).sum()

    return report


def get_unique_brands(df: pd.DataFrame) -> pd.DataFrame:
    """Get unique country-brand combinations."""
    return df[['country', 'brand_name']].drop_duplicates()


def split_train_validation(df: pd.DataFrame,
                           val_brands_ratio: float = 0.2,
                           random_state: int = RANDOM_STATE) -> tuple:
    """
    Split data into train and validation sets by brand (not by row).

    Args:
        df: Full dataset
        val_brands_ratio: Fraction of brands for validation
        random_state: Random seed

    Returns:
        Tuple of (train_df, val_df)
```

```
    """
    np.random.seed(random_state)

    # Get unique brands
    brands = df[['country', 'brand_name']].drop_duplicates()
    n_val = int(len(brands) * val_brands_ratio)

    # Random sample for validation
    val_brands = brands.sample(n=n_val, random_state=random_state)

    # Split
    val_mask = df.set_index(['country', 'brand_name']).index.isin(
        val_brands.set_index(['country', 'brand_name']).index
    )

    train_df = df[~val_mask].copy()
    val_df = df[val_mask].copy()

    print(f"✅ Train: {len(train_df)} rows, {len(train_df[['country',
'brand_name']].drop_duplicates())} brands")
    print(f"✅ Validation: {len(val_df)} rows, {len(val_df[['country',
'brand_name']].drop_duplicates())} brands")

    return train_df, val_df


if __name__ == "__main__":
    # Demo: Load and validate data
    print("=" * 60)
    print("DATA LOADER DEMO")
    print("=" * 60)

    # Load training data
    volume, generics, medicine = load_all_data(train=True)

    # Merge
    merged = merge_datasets(volume, generics, medicine)

    # Validate
    report = validate_data(merged, "merged_train")
    print(f"\nValidation report:")
    print(f"  Shape: {report['shape']}")
    print(f"  Duplicates: {report['duplicates']}")

    # Split
    train_df, val_df = split_train_validation(merged)

    print("\n✅ Data loader demo complete!")
```

2.3 Create `src/bucket_calculator.py`

- ☐ **2.3.1** Create bucket calculation module:

```
#
================================================================================
# File: src/bucket_calculator.py
# Description: Functions to compute Avg_j, normalized volume, and erosion
buckets
#
================================================================================

import pandas as pd
import numpy as np
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
from config import *


def compute_avg_j(df: pd.DataFrame) -> pd.DataFrame:
    """
    Compute pre-entry average volume (Avg_j) for each country-brand.

    Avg_j = mean(volume) over months_postgx in [-12, -1]
    This is the key normalization factor for the PE metric.

    Args:
        df: DataFrame with columns [country, brand_name, months_postgx,
volume]

    Returns:
        DataFrame with columns [country, brand_name, avg_vol]
    """
    # Filter to pre-entry months (-12 to -1)
    pre_entry = df[
        (df['months_postgx'] >= -PRE_ENTRY_MONTHS) &
        (df['months_postgx'] <= -1)
    ].copy()

    # Compute mean volume per brand
    avg_j = pre_entry.groupby(['country', 'brand_name'])
['volume'].mean().reset_index()
    avg_j.columns = ['country', 'brand_name', 'avg_vol']

    # Handle brands with no pre-entry data
    all_brands = df[['country', 'brand_name']].drop_duplicates()
    avg_j = all_brands.merge(avg_j, on=['country', 'brand_name'], how='left')

    print(f"☑ Computed avg_vol for {len(avg_j)} brands")
    print(f"   Brands with valid avg_vol: {avg_j['avg_vol'].notna().sum()}")

    return avg_j
```

```
def compute_normalized_volume(df: pd.DataFrame, avg_j_df: pd.DataFrame) ->
pd.DataFrame:
    """
    Compute normalized volume: vol_norm = volume / Avg_j

    Args:
        df: DataFrame with volume data
        avg_j_df: DataFrame with avg_vol per brand

    Returns:
        DataFrame with added vol_norm column
    """
    merged = df.merge(avg_j_df, on=['country', 'brand_name'], how='left')

    # Compute normalized volume
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    # Handle division by zero and inf
    merged['vol_norm'] = merged['vol_norm'].replace([np.inf, -np.inf], np.nan)

    return merged


def compute_mean_erosion(df: pd.DataFrame) -> pd.DataFrame:
    """
    Compute mean generic erosion for each country-brand.

    Mean erosion = mean(vol_norm) over months_postgx in [0, 23]

    Args:
        df: DataFrame with vol_norm column

    Returns:
        DataFrame with columns [country, brand_name, mean_erosion]
    """
    # Filter to post-entry months (0-23)
    post_entry = df[
        (df['months_postgx'] >= 0) &
        (df['months_postgx'] <= 23)
    ].copy()

    # Compute mean normalized volume (erosion)
    mean_erosion = post_entry.groupby(['country', 'brand_name'])
['vol_norm'].mean().reset_index()
    mean_erosion.columns = ['country', 'brand_name', 'mean_erosion']

    return mean_erosion


def assign_buckets(mean_erosion_df: pd.DataFrame) -> pd.DataFrame:
    """
    Assign erosion bucket based on mean erosion.

    Bucket 1: mean_erosion in [0, 0.25] → High erosion (weighted 2x in
```

```
    metric!)
        Bucket 2: mean_erosion > 0.25 → Lower erosion (weighted 1x)

        Args:
            mean_erosion_df: DataFrame with mean_erosion column

        Returns:
            DataFrame with added bucket column
        """
        df = mean_erosion_df.copy()

        df['bucket'] = np.where(
            (df['mean_erosion'] >= 0) & (df['mean_erosion'] <=
    BUCKET_1_THRESHOLD),
            1,  # Bucket 1: high erosion
            2   # Bucket 2: lower erosion
        )

        # Count distribution
        bucket_counts = df['bucket'].value_counts().sort_index()
        print(f"☑ Bucket distribution:")
        print(f"   Bucket 1 (high erosion): {bucket_counts.get(1, 0)} brands")
        print(f"   Bucket 2 (lower erosion): {bucket_counts.get(2, 0)} brands")

        return df


    def create_auxiliary_file(df: pd.DataFrame, save: bool = True) ->
    pd.DataFrame:
        """
        Create auxiliary file with avg_vol and bucket for metric calculation.

        This file is used during evaluation to:
        1. Normalize prediction errors by avg_vol
        2. Apply bucket weights (Bucket 1 = 2x)

        Args:
            df: Full merged dataset
            save: If True, save to data/processed/

        Returns:
            DataFrame with columns [country, brand_name, avg_vol, mean_erosion,
    bucket]
        """
        print("\n" + "=" * 60)
        print("CREATING AUXILIARY FILE")
        print("=" * 60)

        # Step 1: Compute avg_vol (pre-entry average)
        avg_j = compute_avg_j(df)

        # Step 2: Add normalized volume to data
        df_with_norm = compute_normalized_volume(df, avg_j)
```

```
    # Step 3: Compute mean erosion
    mean_erosion = compute_mean_erosion(df_with_norm)

    # Step 4: Assign buckets
    buckets = assign_buckets(mean_erosion)

    # Step 5: Merge avg_vol and bucket
    aux = avg_j.merge(
        buckets[['country', 'brand_name', 'mean_erosion', 'bucket']],
        on=['country', 'brand_name'],
        how='left'
    )

    if save:
        output_path = DATA_PROCESSED / "aux_bucket_avgvol.csv"
        aux.to_csv(output_path, index=False)
        print(f"\n☑ Saved auxiliary file to: {output_path}")

    return aux


def get_bucket_for_brand(aux_df: pd.DataFrame, country: str, brand_name: str)
-> int:
    """Get bucket assignment for a specific brand."""
    mask = (aux_df['country'] == country) & (aux_df['brand_name'] ==
brand_name)
    if mask.sum() == 0:
        return None
    return aux_df.loc[mask, 'bucket'].iloc[0]


def get_avg_vol_for_brand(aux_df: pd.DataFrame, country: str, brand_name: str)
-> float:
    """Get avg_vol for a specific brand."""
    mask = (aux_df['country'] == country) & (aux_df['brand_name'] ==
brand_name)
    if mask.sum() == 0:
        return None
    return aux_df.loc[mask, 'avg_vol'].iloc[0]


if __name__ == "__main__":
    # Demo: Create auxiliary file
    print("=" * 60)
    print("BUCKET CALCULATOR DEMO")
    print("=" * 60)

    from data_loader import load_all_data, merge_datasets

    # Load data
    volume, generics, medicine = load_all_data(train=True)
    merged = merge_datasets(volume, generics, medicine)

    # Create auxiliary file
```

```
    aux = create_auxiliary_file(merged, save=True)

    print(f"\nAuxiliary file preview:")
    print(aux.head(10))

    print("\n☑ Bucket calculator demo complete!")
```

2.4 Create `src/feature_engineering.py`

- ☐ **2.4.1** Create feature engineering module:

```
#
================================================================================
# File: src/feature_engineering.py
# Description: Functions to create all features for modeling
#
================================================================================

import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
from config import *


#
================================================================================
# LAG FEATURES
#
================================================================================

def create_lag_features(df: pd.DataFrame,
                        lags: list = [1, 2, 3, 6, 12],
                        target_col: str = 'volume') -> pd.DataFrame:
    """
    Create lag features for volume.

    Args:
        df: Input DataFrame (must be sorted by brand and time)
        lags: List of lag periods
        target_col: Column to create lags for

    Returns:
        DataFrame with added lag features
    """
    df = df.sort_values(['country', 'brand_name', 'months_postgx']).copy()
```

```
    for lag in lags:
        col_name = f'{target_col}_lag_{lag}'
        df[col_name] = df.groupby(['country', 'brand_name'])
[target_col].shift(lag)

    print(f"✅ Created {len(lags)} lag features: {[f'{target_col}_lag_{l}' for
l in lags]}")
    return df


#
================================================================================
# ROLLING FEATURES
#
================================================================================

def create_rolling_features(df: pd.DataFrame,
                            windows: list = [3, 6, 12],
                            target_col: str = 'volume') -> pd.DataFrame:
    """
    Create rolling mean and std features.

    Args:
        df: Input DataFrame
        windows: List of window sizes
        target_col: Column to compute rolling stats for

    Returns:
        DataFrame with added rolling features
    """
    df = df.sort_values(['country', 'brand_name', 'months_postgx']).copy()

    for window in windows:
        # Rolling mean
        df[f'{target_col}_rolling_mean_{window}'] = df.groupby(['country',
'brand_name'])[target_col].transform(
            lambda x: x.rolling(window, min_periods=1).mean()
        )
        # Rolling std
        df[f'{target_col}_rolling_std_{window}'] = df.groupby(['country',
'brand_name'])[target_col].transform(
            lambda x: x.rolling(window, min_periods=1).std()
        )

    print(f"✅ Created {len(windows) * 2} rolling features for windows:
{windows}")
    return df


#
================================================================================
# PRE-ENTRY FEATURES (Critical for Scenario 1)
#
================================================================================
```

```
def create_pre_entry_features(df: pd.DataFrame, avg_j_df: pd.DataFrame) ->
pd.DataFrame:
    """
    Create pre-entry aggregate features per country-brand.
    These are the ONLY features available at prediction time for Scenario 1.

    Features created:
    - avg_vol: Pre-entry average (from avg_j_df)
    - pre_entry_slope: Linear trend before entry
    - pre_entry_volatility: Std dev of pre-entry volumes
    - pre_entry_growth_rate: Growth rate in pre-entry period
    - pre_entry_min/max: Min and max volumes before entry

    Args:
        df: Full dataset
        avg_j_df: DataFrame with avg_vol per brand

    Returns:
        DataFrame with pre-entry features per brand
    """
    pre_entry = df[df['months_postgx'] < 0].copy()

    # 1. Pre-entry slope (linear trend)
    def calc_slope(group):
        if len(group) < 2:
            return 0
        x = group['months_postgx'].values
        y = group['volume'].values
        if np.std(x) == 0 or np.std(y) == 0:
            return 0
        try:
            return np.polyfit(x, y, 1)[0]
        except:
            return 0

    slopes = pre_entry.groupby(['country',
'brand_name']).apply(calc_slope).reset_index()
    slopes.columns = ['country', 'brand_name', 'pre_entry_slope']

    # 2. Pre-entry volatility
    volatility = pre_entry.groupby(['country', 'brand_name'])
['volume'].std().reset_index()
    volatility.columns = ['country', 'brand_name', 'pre_entry_volatility']

    # 3. Pre-entry min/max
    pre_min = pre_entry.groupby(['country', 'brand_name'])
['volume'].min().reset_index()
    pre_min.columns = ['country', 'brand_name', 'pre_entry_min']

    pre_max = pre_entry.groupby(['country', 'brand_name'])
['volume'].max().reset_index()
    pre_max.columns = ['country', 'brand_name', 'pre_entry_max']
```

```
    # 4. Pre-entry growth rate
    def calc_growth_rate(group):
        group = group.sort_values('months_postgx')
        if len(group) < 2:
            return 0
        first_vol = group['volume'].iloc[0]
        last_vol = group['volume'].iloc[-1]
        if first_vol == 0:
            return 0
        return (last_vol - first_vol) / first_vol

    growth = pre_entry.groupby(['country',
'brand_name']).apply(calc_growth_rate).reset_index()
    growth.columns = ['country', 'brand_name', 'pre_entry_growth_rate']

    # 5. Last pre-entry volume (month -1)
    last_pre = pre_entry[pre_entry['months_postgx'] == -1][['country',
'brand_name', 'volume']].copy()
    last_pre.columns = ['country', 'brand_name', 'pre_entry_last_volume']

    # Merge all features
    features = avg_j_df.copy()
    for feat_df in [slopes, volatility, pre_min, pre_max, growth, last_pre]:
        features = features.merge(feat_df, on=['country', 'brand_name'],
how='left')

    print(f"☑ Created 7 pre-entry features for {len(features)} brands")
    return features


#
================================================================================
# COMPETITION FEATURES (n_gxs)
#
================================================================================

def create_competition_features(df: pd.DataFrame) -> pd.DataFrame:
    """
    Create features from generics competition data.

    Features created:
    - n_gxs: Number of generics at each month (from data)
    - n_gxs_cummax: Cumulative max generics seen
    - n_gxs_change: Change in generics from previous month
    - has_generics: Binary indicator (n_gxs > 0)
    - months_with_generics: Cumulative months with competitors

    Args:
        df: DataFrame with n_gxs column

    Returns:
        DataFrame with added competition features
    """
    df = df.sort_values(['country', 'brand_name', 'months_postgx']).copy()
```

```
    # Cumulative max generics
    df['n_gxs_cummax'] = df.groupby(['country', 'brand_name'])
['n_gxs'].cummax()

    # Change in number of generics
    df['n_gxs_change'] = df.groupby(['country', 'brand_name'])
['n_gxs'].diff().fillna(0)

    # Binary: has any generics
    df['has_generics'] = (df['n_gxs'] > 0).astype(int)

    # Cumulative months with generics
    df['months_with_generics'] = df.groupby(['country', 'brand_name'])
['has_generics'].cumsum()

    # Log transform of n_gxs (diminishing returns)
    df['n_gxs_log'] = np.log1p(df['n_gxs'])

    print(f"☑ Created 5 competition features")
    return df


#
================================================================================
# TIME FEATURES
#
================================================================================

def create_time_features(df: pd.DataFrame) -> pd.DataFrame:
    """
    Create time-based features.

    Features created:
    - months_postgx_squared: Squared term for non-linear erosion
    - months_postgx_log: Log transform
    - is_first_6_months: Indicator for months 0-5
    - is_months_6_11: Indicator for months 6-11
    - is_months_12_plus: Indicator for months 12+
    - quarter: Quarter within year

    Args:
        df: DataFrame with months_postgx column

    Returns:
        DataFrame with added time features
    """
    df = df.copy()

    # Non-linear time features
    df['months_postgx_squared'] = df['months_postgx'] ** 2
    df['months_postgx_log'] = np.log1p(df['months_postgx'].clip(lower=0))
    df['months_postgx_sqrt'] = np.sqrt(df['months_postgx'].clip(lower=0))
```

```
    # Period indicators (aligned with metric weights)
    df['is_first_6_months'] = (df['months_postgx'].between(0, 5)).astype(int)
    df['is_months_6_11'] = (df['months_postgx'].between(6, 11)).astype(int)
    df['is_months_12_plus'] = (df['months_postgx'] >= 12).astype(int)

    # Quarter indicator
    df['quarter'] = ((df['months_postgx'] % 12) // 3) + 1

    print(f"☑ Created 7 time features")
    return df


#
================================================================================
# CATEGORICAL ENCODING
#
================================================================================

def encode_categorical_features(df: pd.DataFrame,
                                columns: list = None) -> tuple:
    """
    Label encode categorical columns.

    Args:
        df: Input DataFrame
        columns: List of columns to encode (default: from config)

    Returns:
        Tuple of (encoded DataFrame, dict of encoders)
    """
    df = df.copy()
    columns = columns or CATEGORICAL_COLS

    encoders = {}
    for col in columns:
        if col in df.columns:
            le = LabelEncoder()
            # Handle missing values
            df[col] = df[col].fillna('UNKNOWN')
            df[f'{col}_encoded'] = le.fit_transform(df[col].astype(str))
            encoders[col] = le

    print(f"☑ Encoded {len(encoders)} categorical columns:
{list(encoders.keys())}")
    return df, encoders


#
================================================================================
# INTERACTION FEATURES
#
================================================================================

def create_interaction_features(df: pd.DataFrame) -> pd.DataFrame:
```

```
    """
    Create interaction features based on EDA insights.

    Args:
        df: DataFrame with base features

    Returns:
        DataFrame with added interaction features
    """
    df = df.copy()

    # Biological × time (biosimilars may erode slower)
    if 'biological' in df.columns:
        df['biological_x_months'] = df['biological'] * df['months_postgx']

    # Hospital rate × time (tender dynamics)
    if 'hospital_rate' in df.columns:
        df['hospital_rate_x_months'] = df['hospital_rate'] *
df['months_postgx']

    # n_gxs × time
    if 'n_gxs' in df.columns:
        df['n_gxs_x_months'] = df['n_gxs'] * df['months_postgx']

    print(f"☑ Created 3 interaction features")
    return df


#
==============================================================================
# FULL FEATURE PIPELINE
#
==============================================================================

def create_all_features(df: pd.DataFrame,
                        avg_j_df: pd.DataFrame = None,
                        include_lags: bool = True,
                        include_rolling: bool = True) -> pd.DataFrame:
    """
    Run complete feature engineering pipeline.

    Args:
        df: Merged dataset
        avg_j_df: Pre-computed avg_vol per brand (optional)
        include_lags: Whether to include lag features
        include_rolling: Whether to include rolling features

    Returns:
        DataFrame with all features
    """
    print("\n" + "=" * 60)
    print("FEATURE ENGINEERING PIPELINE")
    print("=" * 60)
```

```
    # 1. Lag features
    if include_lags:
        df = create_lag_features(df)

    # 2. Rolling features
    if include_rolling:
        df = create_rolling_features(df)

    # 3. Competition features
    df = create_competition_features(df)

    # 4. Time features
    df = create_time_features(df)

    # 5. Interaction features
    df = create_interaction_features(df)

    # 6. Categorical encoding
    df, encoders = encode_categorical_features(df)

    # 7. Add pre-entry features if avg_j provided
    if avg_j_df is not None:
        pre_entry_feats = create_pre_entry_features(df, avg_j_df)
        df = df.merge(pre_entry_feats, on=['country', 'brand_name'],
how='left')

    print(f"\n☑ Feature engineering complete!")
    print(f"   Final shape: {df.shape}")
    print(f"   Total features: {len(df.columns)}")

    return df


def get_feature_columns(df: pd.DataFrame,
                        exclude_cols: list = None) -> list:
    """
    Get list of feature columns for modeling.

    Args:
        df: DataFrame with features
        exclude_cols: Columns to exclude

    Returns:
        List of feature column names
    """
    exclude = exclude_cols or [
        'country', 'brand_name', 'month', 'volume', 'vol_norm',
        'mean_erosion', 'bucket', 'avg_vol'
    ]

    feature_cols = [col for col in df.columns if col not in exclude]
    return feature_cols
```

```
if __name__ == "__main__":
    # Demo: Feature engineering
    print("=" * 60)
    print("FEATURE ENGINEERING DEMO")
    print("=" * 60)

    from data_loader import load_all_data, merge_datasets
    from bucket_calculator import compute_avg_j

    # Load data
    volume, generics, medicine = load_all_data(train=True)
    merged = merge_datasets(volume, generics, medicine)

    # Compute avg_j
    avg_j = compute_avg_j(merged)

    # Create all features
    featured = create_all_features(merged, avg_j)

    # Get feature columns
    feature_cols = get_feature_columns(featured)
    print(f"\nFeature columns ({len(feature_cols)}):")
    for col in feature_cols[:10]:
        print(f"  - {col}")
    print(f"  ... and {len(feature_cols) - 10} more")

    print("\n☑ Feature engineering demo complete!")
```

2.5 Create `src/models.py`

- ☐ **2.5.1** Create modeling module:

```
#
==============================================================================
# File: src/models.py
# Description: Model classes for baseline and gradient boosting models
#
==============================================================================

import pandas as pd
import numpy as np
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.metrics import mean_squared_error, mean_absolute_error
import lightgbm as lgb
import xgboost as xgb
import joblib
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
```

```python
from config import *


#
# ============================================================================
# BASELINE MODELS
#
# ============================================================================

class BaselineModels:
    """Simple baseline models for comparison."""

    @staticmethod
    def naive_persistence(avg_j_df: pd.DataFrame,
                          months_to_predict: list) -> pd.DataFrame:
        """
        Predict avg_j (pre-entry average) for all future months.
        This is the "no erosion" baseline.

        Args:
            avg_j_df: DataFrame with avg_vol per brand
            months_to_predict: List of months to predict

        Returns:
            DataFrame with predictions
        """
        predictions = []
        for _, row in avg_j_df.iterrows():
            for month in months_to_predict:
                predictions.append({
                    'country': row['country'],
                    'brand_name': row['brand_name'],
                    'months_postgx': month,
                    'volume': row['avg_vol']
                })
        return pd.DataFrame(predictions)

    @staticmethod
    def linear_decay(avg_j_df: pd.DataFrame,
                     months_to_predict: list,
                     decay_rate: float = 0.03) -> pd.DataFrame:
        """
        Predict linear decay from avg_j.
        volume = avg_vol * (1 - decay_rate * months_postgx)

        Args:
            avg_j_df: DataFrame with avg_vol per brand
            months_to_predict: List of months to predict
            decay_rate: Monthly decay rate

        Returns:
            DataFrame with predictions
        """
        predictions = []
```

```
            for _, row in avg_j_df.iterrows():
                for month in months_to_predict:
                    decayed_volume = row['avg_vol'] * (1 - decay_rate * month)
                    decayed_volume = max(0, decayed_volume)  # No negative volumes
                    predictions.append({
                        'country': row['country'],
                        'brand_name': row['brand_name'],
                        'months_postgx': month,
                        'volume': decayed_volume
                    })
            return pd.DataFrame(predictions)

        @staticmethod
        def exponential_decay(avg_j_df: pd.DataFrame,
                              months_to_predict: list,
                              decay_rate: float = 0.05) -> pd.DataFrame:
            """
            Predict exponential decay from avg_j.
            volume = avg_vol * exp(-decay_rate * months_postgx)

            Args:
                avg_j_df: DataFrame with avg_vol per brand
                months_to_predict: List of months to predict
                decay_rate: Decay rate parameter

            Returns:
                DataFrame with predictions
            """
            predictions = []
            for _, row in avg_j_df.iterrows():
                for month in months_to_predict:
                    decayed_volume = row['avg_vol'] * np.exp(-decay_rate * month)
                    predictions.append({
                        'country': row['country'],
                        'brand_name': row['brand_name'],
                        'months_postgx': month,
                        'volume': decayed_volume
                    })
            return pd.DataFrame(predictions)

        @staticmethod
        def tune_decay_rate(actual_df: pd.DataFrame,
                            avg_j_df: pd.DataFrame,
                            decay_type: str = 'exponential',
                            decay_rates: list = None) -> tuple:
            """
            Tune decay rate on actual data.

            Args:
                actual_df: DataFrame with actual volumes
                avg_j_df: DataFrame with avg_vol per brand
                decay_type: 'linear' or 'exponential'
                decay_rates: List of rates to try
```

```
        Returns:
            Tuple of (best_rate, results_df)
        """
        decay_rates = decay_rates or np.arange(0.01, 0.15, 0.01)
        months = sorted(actual_df['months_postgx'].unique())

        results = []
        for rate in decay_rates:
            if decay_type == 'linear':
                preds = BaselineModels.linear_decay(avg_j_df, months, rate)
            else:
                preds = BaselineModels.exponential_decay(avg_j_df, months,
rate)

            merged = actual_df.merge(
                preds, on=['country', 'brand_name', 'months_postgx'],
                suffixes=('_actual', '_pred')
            )
            mse = mean_squared_error(merged['volume_actual'],
merged['volume_pred'])
            mae = mean_absolute_error(merged['volume_actual'],
merged['volume_pred'])

            results.append({'decay_rate': rate, 'mse': mse, 'mae': mae})

        results_df = pd.DataFrame(results)
        best_rate = results_df.loc[results_df['mae'].idxmin(), 'decay_rate']

        print(f"☑ Best {decay_type} decay rate: {best_rate:.3f}")
        return best_rate, results_df


#
================================================================================
# GRADIENT BOOSTING MODELS
#
================================================================================

class GradientBoostingModel:
    """LightGBM/XGBoost model wrapper for volume prediction."""

    def __init__(self, model_type: str = 'lightgbm', params: dict = None):
        """
        Initialize model.

        Args:
            model_type: 'lightgbm' or 'xgboost'
            params: Model hyperparameters (uses defaults if None)
        """
        self.model_type = model_type
        self.model = None
        self.params = params or self._default_params()
        self.feature_names = None
```

```
    def _default_params(self) -> dict:
        """Get default parameters for model type."""
        if self.model_type == 'lightgbm':
            return LGBM_PARAMS.copy()
        else:
            return XGB_PARAMS.copy()

    def fit(self, X_train: pd.DataFrame, y_train: pd.Series,
            X_val: pd.DataFrame = None, y_val: pd.Series = None,
            early_stopping_rounds: int = 50) -> 'GradientBoostingModel':
        """
        Train the model.

        Args:
            X_train: Training features
            y_train: Training target
            X_val: Validation features (optional)
            y_val: Validation target (optional)
            early_stopping_rounds: Early stopping patience

        Returns:
            Self (fitted model)
        """
        self.feature_names = list(X_train.columns)

        if self.model_type == 'lightgbm':
            self.model = lgb.LGBMRegressor(**self.params)

            if X_val is not None and y_val is not None:
                self.model.fit(
                    X_train, y_train,
                    eval_set=[(X_val, y_val)],
                    callbacks=[lgb.early_stopping(early_stopping_rounds,
verbose=False)]
                )
            else:
                self.model.fit(X_train, y_train)

        else:  # xgboost
            self.model = xgb.XGBRegressor(**self.params)

            if X_val is not None and y_val is not None:
                self.model.fit(
                    X_train, y_train,
                    eval_set=[(X_val, y_val)],
                    verbose=False
                )
            else:
                self.model.fit(X_train, y_train)

        print(f"☑ {self.model_type} model trained successfully")
        return self

    def predict(self, X: pd.DataFrame) -> np.ndarray:
```

```
        """
        Generate predictions.

        Args:
            X: Features to predict on

        Returns:
            Array of predictions
        """
        if self.model is None:
            raise ValueError("Model not fitted. Call fit() first.")

        predictions = self.model.predict(X)
        # Ensure no negative predictions
        predictions = np.clip(predictions, 0, None)
        return predictions

    def get_feature_importance(self, top_n: int = 20) -> pd.DataFrame:
        """
        Get feature importance.

        Args:
            top_n: Number of top features to return

        Returns:
            DataFrame with feature importances
        """
        if self.model is None:
            return None

        importance_df = pd.DataFrame({
            'feature': self.feature_names,
            'importance': self.model.feature_importances_
        }).sort_values('importance', ascending=False)

        return importance_df.head(top_n)

    def cross_validate(self, X: pd.DataFrame, y: pd.Series,
                       n_splits: int = 5) -> dict:
        """
        Perform cross-validation.

        Args:
            X: Features
            y: Target
            n_splits: Number of CV folds

        Returns:
            Dictionary with CV results
        """
        if self.model_type == 'lightgbm':
            model = lgb.LGBMRegressor(**self.params)
        else:
            model = xgb.XGBRegressor(**self.params)
```

```
        tscv = TimeSeriesSplit(n_splits=n_splits)

        scores_neg_mse = cross_val_score(model, X, y, cv=tscv,
scoring='neg_mean_squared_error')
        scores_neg_mae = cross_val_score(model, X, y, cv=tscv,
scoring='neg_mean_absolute_error')

        results = {
            'rmse_mean': np.sqrt(-scores_neg_mse.mean()),
            'rmse_std': np.sqrt(-scores_neg_mse).std(),
            'mae_mean': -scores_neg_mae.mean(),
            'mae_std': (-scores_neg_mae).std()
        }

        print(f"☑ Cross-validation results:")
        print(f"   RMSE: {results['rmse_mean']:.4f} ±
{results['rmse_std']:.4f}")
        print(f"   MAE:  {results['mae_mean']:.4f} ±
{results['mae_std']:.4f}")

        return results

    def save(self, name: str) -> Path:
        """Save model to disk."""
        path = MODELS_DIR / f"{name}.joblib"
        joblib.dump({
            'model': self.model,
            'model_type': self.model_type,
            'params': self.params,
            'feature_names': self.feature_names
        }, path)
        print(f"☑ Model saved to {path}")
        return path

    def load(self, name: str) -> 'GradientBoostingModel':
        """Load model from disk."""
        path = MODELS_DIR / f"{name}.joblib"
        data = joblib.load(path)
        self.model = data['model']
        self.model_type = data['model_type']
        self.params = data['params']
        self.feature_names = data['feature_names']
        print(f"☑ Model loaded from {path}")
        return self


#
==============================================================================
# MODEL TRAINING UTILITIES
#
==============================================================================

def prepare_training_data(df: pd.DataFrame,
```

```
                          feature_cols: list,
                          target_col: str = 'volume',
                          filter_post_entry: bool = True) -> tuple:
    """
    Prepare data for model training.

    Args:
        df: Featured dataset
        feature_cols: List of feature columns
        target_col: Target column name
        filter_post_entry: If True, only use post-entry months

    Returns:
        Tuple of (X, y)
    """
    data = df.copy()

    if filter_post_entry:
        data = data[data['months_postgx'] >= 0]

    X = data[feature_cols].fillna(0)
    y = data[target_col]

    print(f"☑ Prepared training data: X={X.shape}, y={y.shape}")
    return X, y


def train_and_evaluate(X_train: pd.DataFrame, y_train: pd.Series,
                       X_val: pd.DataFrame, y_val: pd.Series,
                       model_type: str = 'lightgbm') -> tuple:
    """
    Train model and evaluate on validation set.

    Args:
        X_train, y_train: Training data
        X_val, y_val: Validation data
        model_type: 'lightgbm' or 'xgboost'

    Returns:
        Tuple of (model, metrics_dict)
    """
    model = GradientBoostingModel(model_type=model_type)
    model.fit(X_train, y_train, X_val, y_val)

    # Evaluate
    train_preds = model.predict(X_train)
    val_preds = model.predict(X_val)

    metrics = {
        'train_rmse': np.sqrt(mean_squared_error(y_train, train_preds)),
        'train_mae': mean_absolute_error(y_train, train_preds),
        'val_rmse': np.sqrt(mean_squared_error(y_val, val_preds)),
        'val_mae': mean_absolute_error(y_val, val_preds)
    }
```

```
        print(f"\n📊 {model_type.upper()} Evaluation:")
        print(f"   Train RMSE: {metrics['train_rmse']:.4f}")
        print(f"   Val RMSE:   {metrics['val_rmse']:.4f}")

        return model, metrics


    if __name__ == "__main__":
        # Demo: Train a simple model
        print("=" * 60)
        print("MODELS DEMO")
        print("=" * 60)

        from data_loader import load_all_data, merge_datasets,
    split_train_validation
        from bucket_calculator import compute_avg_j
        from feature_engineering import create_all_features, get_feature_columns

        # Load and prepare data
        volume, generics, medicine = load_all_data(train=True)
        merged = merge_datasets(volume, generics, medicine)
        avg_j = compute_avg_j(merged)

        # Create features
        featured = create_all_features(merged, avg_j)

        # Split
        train_df, val_df = split_train_validation(featured)

        # Prepare training data
        feature_cols = get_feature_columns(featured)
        X_train, y_train = prepare_training_data(train_df, feature_cols)
        X_val, y_val = prepare_training_data(val_df, feature_cols)

        # Train model
        model, metrics = train_and_evaluate(X_train, y_train, X_val, y_val,
    'lightgbm')

        # Feature importance
        print("\n📊 Top 10 Features:")
        print(model.get_feature_importance(10))

        # Save model
        model.save("demo_lightgbm")

        print("\n✅ Models demo complete!")
```

2.6 Create `src/evaluation.py`

- [ ] **2.6.1** Create evaluation module:

```
#
================================================================================
# File: src/evaluation.py
# Description: Functions to evaluate models using official PE metric
#
================================================================================

import pandas as pd
import numpy as np
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
from config import *


def compute_pe_scenario1(actual_df: pd.DataFrame,
                         pred_df: pd.DataFrame,
                         aux_df: pd.DataFrame) -> pd.DataFrame:
    """
    Compute Prediction Error for Scenario 1 (months 0-23).

    PE = 0.2 * (Σ|actual-pred| / 24*avg_vol)
       + 0.5 * (|Σ(0-5) actual - Σ(0-5) pred| / 6*avg_vol)
       + 0.2 * (|Σ(6-11) actual - Σ(6-11) pred| / 6*avg_vol)
       + 0.1 * (|Σ(12-23) actual - Σ(12-23) pred| / 12*avg_vol)

    Args:
        actual_df: DataFrame with actual volumes
        pred_df: DataFrame with predicted volumes
        aux_df: Auxiliary file with avg_vol and bucket

    Returns:
        DataFrame with PE per brand
    """
    # Merge actual, predictions, and auxiliary data
    merged = actual_df.merge(
        pred_df, on=['country', 'brand_name', 'months_postgx'],
        suffixes=('_actual', '_pred')
    ).merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
            on=['country', 'brand_name'])

    results = []
    for (country, brand), group in merged.groupby(['country', 'brand_name']):
        avg_vol = group['avg_vol'].iloc[0]
        bucket = group['bucket'].iloc[0]

        if avg_vol == 0 or np.isnan(avg_vol):
            continue

        # Term 1: Monthly absolute error (0-23)
        monthly_err =
```

```
    group['volume_actual'].sub(group['volume_pred']).abs().sum()
            term1 = S1_MONTHLY_WEIGHT * monthly_err / (24 * avg_vol)

            # Term 2: Accumulated error months 0-5
            m0_5 = group[group['months_postgx'].between(0, 5)]
            sum_err_0_5 = abs(m0_5['volume_actual'].sum() -
    m0_5['volume_pred'].sum())
            term2 = S1_SUM_0_5_WEIGHT * sum_err_0_5 / (6 * avg_vol)

            # Term 3: Accumulated error months 6-11
            m6_11 = group[group['months_postgx'].between(6, 11)]
            sum_err_6_11 = abs(m6_11['volume_actual'].sum() -
    m6_11['volume_pred'].sum())
            term3 = S1_SUM_6_11_WEIGHT * sum_err_6_11 / (6 * avg_vol)

            # Term 4: Accumulated error months 12-23
            m12_23 = group[group['months_postgx'].between(12, 23)]
            sum_err_12_23 = abs(m12_23['volume_actual'].sum() -
    m12_23['volume_pred'].sum())
            term4 = S1_SUM_12_23_WEIGHT * sum_err_12_23 / (12 * avg_vol)

            pe = term1 + term2 + term3 + term4

            results.append({
                'country': country,
                'brand_name': brand,
                'bucket': bucket,
                'avg_vol': avg_vol,
                'PE': pe,
                'term1_monthly': term1,
                'term2_sum_0_5': term2,
                'term3_sum_6_11': term3,
                'term4_sum_12_23': term4
            })

        return pd.DataFrame(results)


    def compute_pe_scenario2(actual_df: pd.DataFrame,
                             pred_df: pd.DataFrame,
                             aux_df: pd.DataFrame) -> pd.DataFrame:
        """
        Compute Prediction Error for Scenario 2 (months 6-23).

        PE = 0.2 * (Σ|actual-pred| / 18*avg_vol)
           + 0.5 * (|Σ(6-11) actual - Σ(6-11) pred| / 6*avg_vol)
           + 0.3 * (|Σ(12-23) actual - Σ(12-23) pred| / 12*avg_vol)

        Args:
            actual_df: DataFrame with actual volumes (months 6-23)
            pred_df: DataFrame with predicted volumes (months 6-23)
            aux_df: Auxiliary file with avg_vol and bucket

        Returns:
```

```
        DataFrame with PE per brand
    """
    # Merge actual, predictions, and auxiliary data
    merged = actual_df.merge(
        pred_df, on=['country', 'brand_name', 'months_postgx'],
        suffixes=('_actual', '_pred')
    ).merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
            on=['country', 'brand_name'])

    results = []
    for (country, brand), group in merged.groupby(['country', 'brand_name']):
        avg_vol = group['avg_vol'].iloc[0]
        bucket = group['bucket'].iloc[0]

        if avg_vol == 0 or np.isnan(avg_vol):
            continue

        # Filter to months 6-23 only
        group = group[group['months_postgx'] >= 6]

        # Term 1: Monthly absolute error (6-23)
        monthly_err =
group['volume_actual'].sub(group['volume_pred']).abs().sum()
        term1 = S2_MONTHLY_WEIGHT * monthly_err / (18 * avg_vol)

        # Term 2: Accumulated error months 6-11
        m6_11 = group[group['months_postgx'].between(6, 11)]
        sum_err_6_11 = abs(m6_11['volume_actual'].sum() -
m6_11['volume_pred'].sum())
        term2 = S2_SUM_6_11_WEIGHT * sum_err_6_11 / (6 * avg_vol)

        # Term 3: Accumulated error months 12-23
        m12_23 = group[group['months_postgx'].between(12, 23)]
        sum_err_12_23 = abs(m12_23['volume_actual'].sum() -
m12_23['volume_pred'].sum())
        term3 = S2_SUM_12_23_WEIGHT * sum_err_12_23 / (12 * avg_vol)

        pe = term1 + term2 + term3

        results.append({
            'country': country,
            'brand_name': brand,
            'bucket': bucket,
            'avg_vol': avg_vol,
            'PE': pe,
            'term1_monthly': term1,
            'term2_sum_6_11': term2,
            'term3_sum_12_23': term3
        })

    return pd.DataFrame(results)


def compute_final_metric(pe_df: pd.DataFrame) -> float:
```

```
    """
    Compute final weighted metric.

    PE_final = (2/n_B1) * Σ(PE_B1) + (1/n_B2) * Σ(PE_B2)

    Bucket 1 (high erosion) is weighted 2x!

    Args:
        pe_df: DataFrame with PE per brand and bucket

    Returns:
        Final weighted PE score
    """
    bucket1 = pe_df[pe_df['bucket'] == 1]
    bucket2 = pe_df[pe_df['bucket'] == 2]

    n_b1 = len(bucket1)
    n_b2 = len(bucket2)

    if n_b1 == 0 and n_b2 == 0:
        return np.nan

    score = 0
    if n_b1 > 0:
        score += (BUCKET_1_WEIGHT / n_b1) * bucket1['PE'].sum()
    if n_b2 > 0:
        score += (BUCKET_2_WEIGHT / n_b2) * bucket2['PE'].sum()

    return score


def evaluate_model(actual_df: pd.DataFrame,
                   pred_df: pd.DataFrame,
                   aux_df: pd.DataFrame,
                   scenario: int = 1) -> dict:
    """
    Full evaluation pipeline.

    Args:
        actual_df: DataFrame with actual volumes
        pred_df: DataFrame with predicted volumes
        aux_df: Auxiliary file with avg_vol and bucket
        scenario: 1 or 2

    Returns:
        Dictionary with evaluation results
    """
    print(f"\n{'=' * 60}")
    print(f"EVALUATING SCENARIO {scenario}")
    print("=" * 60)

    # Compute PE per brand
    if scenario == 1:
        pe_df = compute_pe_scenario1(actual_df, pred_df, aux_df)
```

```
        else:
            pe_df = compute_pe_scenario2(actual_df, pred_df, aux_df)

        # Compute final metric
        final_score = compute_final_metric(pe_df)

        # Bucket breakdown
        bucket1 = pe_df[pe_df['bucket'] == 1]
        bucket2 = pe_df[pe_df['bucket'] == 2]

        bucket1_avg_pe = bucket1['PE'].mean() if len(bucket1) > 0 else np.nan
        bucket2_avg_pe = bucket2['PE'].mean() if len(bucket2) > 0 else np.nan

        results = {
            'scenario': scenario,
            'final_score': final_score,
            'bucket1_avg_pe': bucket1_avg_pe,
            'bucket2_avg_pe': bucket2_avg_pe,
            'n_bucket1': len(bucket1),
            'n_bucket2': len(bucket2),
            'n_total_brands': len(pe_df),
            'pe_details': pe_df
        }

        # Print summary
        print(f"\n📊 EVALUATION RESULTS - Scenario {scenario}")
        print(f"   Final Score: {final_score:.4f}")
        print(f"   Bucket 1 Avg PE: {bucket1_avg_pe:.4f} (n={len(bucket1)},
    weight=2x)")
        print(f"   Bucket 2 Avg PE: {bucket2_avg_pe:.4f} (n={len(bucket2)},
    weight=1x)")

        return results


    def analyze_worst_predictions(pe_df: pd.DataFrame, n: int = 10) ->
    pd.DataFrame:
        """
        Identify worst-predicted brands for error analysis.

        Args:
            pe_df: DataFrame with PE per brand
            n: Number of worst brands to return

        Returns:
            DataFrame with worst predictions
        """
        worst = pe_df.nlargest(n, 'PE')
        print(f"\n⚠️ Top {n} Worst Predictions:")
        print(worst[['country', 'brand_name', 'bucket',
    'PE']].to_string(index=False))
        return worst
```

```
    def compare_models(results_list: list, model_names: list) -> pd.DataFrame:
        """
        Compare multiple models.

        Args:
            results_list: List of evaluation result dicts
            model_names: List of model names

        Returns:
            Comparison DataFrame
        """
        comparison = []
        for name, results in zip(model_names, results_list):
            comparison.append({
                'model': name,
                'final_score': results['final_score'],
                'bucket1_pe': results['bucket1_avg_pe'],
                'bucket2_pe': results['bucket2_avg_pe']
            })

        df = pd.DataFrame(comparison).sort_values('final_score')
        print("\n📊 Model Comparison:")
        print(df.to_string(index=False))
        return df


    if __name__ == "__main__":
        # Demo: Evaluate a baseline model
        print("=" * 60)
        print("EVALUATION DEMO")
        print("=" * 60)

        from data_loader import load_all_data, merge_datasets,
    split_train_validation
        from bucket_calculator import compute_avg_j, create_auxiliary_file
        from models import BaselineModels

        # Load data
        volume, generics, medicine = load_all_data(train=True)
        merged = merge_datasets(volume, generics, medicine)

        # Create auxiliary file
        aux_df = create_auxiliary_file(merged, save=True)
        avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

        # Split for validation
        train_df, val_df = split_train_validation(merged)

        # Get actual post-entry volumes from validation set
        actual_df = val_df[val_df['months_postgx'].between(0, 23)][
            ['country', 'brand_name', 'months_postgx', 'volume']
        ].copy()

        # Generate baseline predictions
```

```
    val_brands = val_df[['country', 'brand_name']].drop_duplicates()
    val_avg_j = avg_j.merge(val_brands, on=['country', 'brand_name'])

    pred_df = BaselineModels.exponential_decay(
        val_avg_j,
        months_to_predict=list(range(0, 24)),
        decay_rate=0.05
    )

    # Evaluate
    results = evaluate_model(actual_df, pred_df, aux_df, scenario=1)

    # Analyze worst predictions
    analyze_worst_predictions(results['pe_details'])

    print("\n☑ Evaluation demo complete!")
```

2.7 Create `src/submission.py`

- ☐ **2.7.1** Create submission generation module:

```
#
================================================================================
# File: src/submission.py
# Description: Functions to generate and validate submission files
#
================================================================================

import pandas as pd
import numpy as np
from datetime import datetime
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
from config import *


def generate_submission(predictions_df: pd.DataFrame,
                        scenario: int,
                        validate: bool = True) -> pd.DataFrame:
    """
    Generate submission file from predictions.

    Args:
        predictions_df: DataFrame with columns [country, brand_name,
months_postgx, volume]
        scenario: 1 or 2
        validate: If True, validate submission before returning
```

```
    Returns:
        Submission DataFrame
    """
    required_cols = ['country', 'brand_name', 'months_postgx', 'volume']
    missing = set(required_cols) - set(predictions_df.columns)
    if missing:
        raise ValueError(f"Missing columns in predictions: {missing}")

    submission = predictions_df[required_cols].copy()

    # Ensure no negative volumes
    submission['volume'] = submission['volume'].clip(lower=0)

    if validate:
        validate_submission(submission, scenario)

    return submission


def validate_submission(submission_df: pd.DataFrame, scenario: int) -> bool:
    """
    Validate submission format and content.

    Args:
        submission_df: Submission DataFrame
        scenario: 1 or 2

    Returns:
        True if valid

    Raises:
        AssertionError if validation fails
    """
    print(f"\n🔍 Validating Scenario {scenario} submission...")

    # Check required columns
    required_cols = ['country', 'brand_name', 'months_postgx', 'volume']
    assert all(col in submission_df.columns for col in required_cols), \
        f"Missing required columns. Expected: {required_cols}"

    # Check no missing values
    assert submission_df['volume'].notna().all(), \
        "Found NaN values in volume predictions"

    # Check no negative volumes
    assert (submission_df['volume'] >= 0).all(), \
        "Found negative volume predictions"

    # Check months_postgx range
    if scenario == 1:
        expected_months = set(range(0, 24))  # 0-23
        expected_rows_per_brand = 24
    else:
        expected_months = set(range(6, 24))  # 6-23
```

```
        expected_rows_per_brand = 18

    # Check each brand has correct months
    errors = []
    for (country, brand), group in submission_df.groupby(['country',
'brand_name']):
        actual_months = set(group['months_postgx'].values)
        if actual_months != expected_months:
            missing = expected_months - actual_months
            extra = actual_months - expected_months
            errors.append(f"{country}/{brand}: missing={missing}, extra=
{extra}")

    if errors:
        print(f"❌ Found {len(errors)} brands with wrong months:")
        for err in errors[:5]:
            print(f"   {err}")
        raise AssertionError(f"Found {len(errors)} brands with incorrect
months_postgx")

    # Check total row count
    n_brands = submission_df[['country',
'brand_name']].drop_duplicates().shape[0]
    expected_total = n_brands * expected_rows_per_brand
    actual_total = len(submission_df)

    assert actual_total == expected_total, \
        f"Wrong total rows. Expected {expected_total} ({n_brands} brands ×
{expected_rows_per_brand} months), got {actual_total}"

    print(f"☑ Submission validation passed!")
    print(f"   Brands: {n_brands}")
    print(f"   Total rows: {actual_total}")
    print(f"   Volume range: [{submission_df['volume'].min():.2f},
{submission_df['volume'].max():.2f}]")

    return True


def save_submission(submission_df: pd.DataFrame,
                    scenario: int,
                    suffix: str = "",
                    include_timestamp: bool = True) -> Path:
    """
    Save submission to file.

    Args:
        submission_df: Validated submission DataFrame
        scenario: 1 or 2
        suffix: Optional suffix for filename
        include_timestamp: If True, add timestamp to filename

    Returns:
        Path to saved file
```

```
    """
    # Build filename
    if include_timestamp:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"scenario{scenario}_{timestamp}{suffix}.csv"
    else:
        filename = f"scenario{scenario}{suffix}.csv"

    filepath = SUBMISSIONS_DIR / filename
    submission_df.to_csv(filepath, index=False)

    print(f"☑ Saved submission to: {filepath}")
    return filepath


def load_submission(filepath: Path) -> pd.DataFrame:
    """Load submission from file."""
    return pd.read_csv(filepath)


def compare_submissions(sub1_path: Path, sub2_path: Path) -> pd.DataFrame:
    """
    Compare two submissions.

    Args:
        sub1_path: Path to first submission
        sub2_path: Path to second submission

    Returns:
        DataFrame with differences
    """
    sub1 = pd.read_csv(sub1_path)
    sub2 = pd.read_csv(sub2_path)

    merged = sub1.merge(
        sub2, on=['country', 'brand_name', 'months_postgx'],
        suffixes=('_sub1', '_sub2')
    )

    merged['diff'] = merged['volume_sub2'] - merged['volume_sub1']
    merged['pct_diff'] = merged['diff'] / merged['volume_sub1'].replace(0,
np.nan) * 100

    print(f"\n📊 Submission Comparison:")
    print(f"   Mean absolute difference: {merged['diff'].abs().mean():.4f}")
    print(f"   Max absolute difference: {merged['diff'].abs().max():.4f}")
    print(f"   Mean % difference: {merged['pct_diff'].abs().mean():.2f}%")

    return merged


def create_scenario1_submission(model,
                                test_df: pd.DataFrame,
                                feature_cols: list,
```

```
                                  save: bool = True) -> pd.DataFrame:
    """
    Create complete Scenario 1 submission.

    Args:
        model: Trained model with predict method
        test_df: Test dataset with features
        feature_cols: List of feature columns
        save: If True, save submission

    Returns:
        Submission DataFrame
    """
    # Filter to months 0-23
    pred_data = test_df[test_df['months_postgx'].between(0, 23)].copy()

    # Generate predictions
    X = pred_data[feature_cols].fillna(0)
    predictions = model.predict(X)

    # Create submission
    submission = pred_data[['country', 'brand_name', 'months_postgx']].copy()
    submission['volume'] = predictions

    submission = generate_submission(submission, scenario=1)

    if save:
        save_submission(submission, scenario=1)

    return submission


def create_scenario2_submission(model,
                                test_df: pd.DataFrame,
                                feature_cols: list,
                                save: bool = True) -> pd.DataFrame:
    """
    Create complete Scenario 2 submission.

    Args:
        model: Trained model with predict method
        test_df: Test dataset with features (including months 0-5 actuals)
        feature_cols: List of feature columns
        save: If True, save submission

    Returns:
        Submission DataFrame
    """
    # Filter to months 6-23
    pred_data = test_df[test_df['months_postgx'].between(6, 23)].copy()

    # Generate predictions
    X = pred_data[feature_cols].fillna(0)
    predictions = model.predict(X)
```

```
    # Create submission
    submission = pred_data[['country', 'brand_name', 'months_postgx']].copy()
    submission['volume'] = predictions

    submission = generate_submission(submission, scenario=2)

    if save:
        save_submission(submission, scenario=2)

    return submission


if __name__ == "__main__":
    # Demo: Create mock submission
    print("=" * 60)
    print("SUBMISSION DEMO")
    print("=" * 60)

    # Create mock data
    brands = [('USA', 'BrandA'), ('USA', 'BrandB'), ('EU', 'BrandC')]

    # Scenario 1 submission (months 0-23)
    rows = []
    for country, brand in brands:
        for month in range(24):
            rows.append({
                'country': country,
                'brand_name': brand,
                'months_postgx': month,
                'volume': 1000 * np.exp(-0.05 * month)  # Mock exponential
decay
            })

    mock_preds = pd.DataFrame(rows)

    # Generate and validate submission
    submission = generate_submission(mock_preds, scenario=1)

    # Save
    filepath = save_submission(submission, scenario=1, suffix="_demo")

    print(f"\nSubmission preview:")
    print(submission.head(10))

    print("\n☑ Submission demo complete!")
```

2.8 Create `src/pipeline.py`

- ☐ **2.8.1** Create end-to-end pipeline script:

```
#
================================================================================
# File: src/pipeline.py
# Description: End-to-end pipeline from raw data to submission
#
================================================================================

import argparse
import pandas as pd
import numpy as np
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))

from config import *
from data_loader import load_all_data, merge_datasets, split_train_validation
from bucket_calculator import compute_avg_j, create_auxiliary_file
from feature_engineering import create_all_features, get_feature_columns
from models import GradientBoostingModel, BaselineModels,
prepare_training_data
from evaluation import evaluate_model
from submission import generate_submission, save_submission


def run_pipeline(scenario: int = 1,
                 model_type: str = 'lightgbm',
                 validation_mode: bool = True) -> dict:
    """
    Run complete pipeline from raw data to submission.

    Args:
        scenario: 1 or 2
        model_type: 'lightgbm', 'xgboost', or 'baseline'
        validation_mode: If True, evaluate on validation set; else predict on
test

    Returns:
        Dictionary with results
    """
    print("=" * 70)
    print(f"🚀 RUNNING PIPELINE - SCENARIO {scenario}")
    print(f"   Model: {model_type}")
    print(f"   Mode: {'Validation' if validation_mode else 'Test
Prediction'}")
    print("=" * 70)

    #
================================================================================
    # STEP 1: Load Data
    #
================================================================================
```

```
    print("\n📂 STEP 1: Loading data...")
    volume_train, generics_train, medicine_train = load_all_data(train=True)

    if not validation_mode:
        volume_test, generics_test, medicine_test = load_all_data(train=False)

    #
================================================================================
    # STEP 2: Merge Datasets
    #
================================================================================
    print("\n🔗 STEP 2: Merging datasets...")
    train_merged = merge_datasets(volume_train, generics_train,
medicine_train)

    if not validation_mode:
        test_merged = merge_datasets(volume_test, generics_test,
medicine_test)

    #
================================================================================
    # STEP 3: Create Auxiliary File (buckets + avg_vol)
    #
================================================================================
    print("\n📊 STEP 3: Computing buckets and avg_vol...")
    aux_df = create_auxiliary_file(train_merged, save=True)
    avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

    #
================================================================================
    # STEP 4: Feature Engineering
    #
================================================================================
    print("\n🔧 STEP 4: Engineering features...")
    train_featured = create_all_features(train_merged, avg_j)

    if not validation_mode:
        test_featured = create_all_features(test_merged, avg_j)

    #
================================================================================
    # STEP 5: Split Data (if validation mode)
    #
================================================================================
    if validation_mode:
        print("\n✂️ STEP 5: Splitting train/validation...")
        train_df, val_df = split_train_validation(train_featured)
    else:
        train_df = train_featured

    #
================================================================================
    # STEP 6: Prepare Training Data
    #
```

```
# ============================================================================
    print("\n📊 STEP 6: Preparing training data...")
    feature_cols = get_feature_columns(train_df)
    X_train, y_train = prepare_training_data(train_df, feature_cols)

    if validation_mode:
        X_val, y_val = prepare_training_data(val_df, feature_cols)

    #
# ============================================================================
    # STEP 7: Train Model
    #
# ============================================================================
    print(f"\n🤖 STEP 7: Training {model_type} model...")

    if model_type == 'baseline':
        # Use exponential decay baseline
        best_rate, _ = BaselineModels.tune_decay_rate(
            val_df[val_df['months_postgx'] >= 0][['country', 'brand_name',
'months_postgx', 'volume']],
            avg_j,
            decay_type='exponential'
        )
        model = None  # Baseline doesn't need a trained model
    else:
        model = GradientBoostingModel(model_type=model_type)
        if validation_mode:
            model.fit(X_train, y_train, X_val, y_val)
        else:
            model.fit(X_train, y_train)
        model.save(f"scenario{scenario}_{model_type}")

    #
# ============================================================================
    # STEP 8: Generate Predictions
    #
# ============================================================================
    print("\n🔮 STEP 8: Generating predictions...")

    if validation_mode:
        pred_data = val_df
    else:
        pred_data = test_featured

    # Filter to correct months for scenario
    if scenario == 1:
        pred_data = pred_data[pred_data['months_postgx'].between(0,
23)].copy()
    else:
        pred_data = pred_data[pred_data['months_postgx'].between(6,
23)].copy()

    if model_type == 'baseline':
        pred_brands = pred_data[['country', 'brand_name']].drop_duplicates()
```

```
            pred_avg_j = avg_j.merge(pred_brands, on=['country', 'brand_name'])
            months = list(range(0, 24)) if scenario == 1 else list(range(6, 24))
            pred_df = BaselineModels.exponential_decay(pred_avg_j, months,
    best_rate)
        else:
            X_pred = pred_data[feature_cols].fillna(0)
            predictions = model.predict(X_pred)
            pred_df = pred_data[['country', 'brand_name', 'months_postgx']].copy()
            pred_df['volume'] = predictions

        #
    ============================================================================
        # STEP 9: Evaluate (if validation mode)
        #
    ============================================================================
        if validation_mode:
            print("\n📊 STEP 9: Evaluating model...")
            actual_df = val_df[val_df['months_postgx'] >= 0][
                ['country', 'brand_name', 'months_postgx', 'volume']
            ].copy()

            # Get aux_df for validation brands only
            val_brands = val_df[['country', 'brand_name']].drop_duplicates()
            val_aux = aux_df.merge(val_brands, on=['country', 'brand_name'])

            results = evaluate_model(actual_df, pred_df, val_aux,
    scenario=scenario)
        else:
            results = {}

        #
    ============================================================================
        # STEP 10: Generate Submission
        #
    ============================================================================
        print("\n📝 STEP 10: Creating submission...")
        submission = generate_submission(pred_df, scenario=scenario)
        filepath = save_submission(submission, scenario=scenario)

        results['submission_path'] = filepath
        results['submission'] = submission

        print("\n" + "=" * 70)
        print("☑ PIPELINE COMPLETE!")
        if validation_mode and 'final_score' in results:
            print(f"   Final Score: {results['final_score']:.4f}")
        print(f"   Submission: {filepath}")
        print("=" * 70)

        return results


    def main():
        """Main entry point for command-line execution."""
```

```
    parser = argparse.ArgumentParser(description='Novartis Datathon Pipeline')
    parser.add_argument('--scenario', type=int, default=1, choices=[1, 2],
                        help='Scenario to run (1 or 2)')
    parser.add_argument('--model', type=str, default='lightgbm',
                        choices=['lightgbm', 'xgboost', 'baseline'],
                        help='Model type to use')
    parser.add_argument('--test', action='store_true',
                        help='Run on test data (default: validation mode)')

    args = parser.parse_args()

    run_pipeline(
        scenario=args.scenario,
        model_type=args.model,
        validation_mode=not args.test
    )


if __name__ == "__main__":
    main()
```

🚀 STEP 3: Create EDA Analysis Module (`.py`) + Visualization Notebook (`.ipynb`)

> 🗀 **Hybrid Approach:**
>
> - `src/eda_analysis.py` - All EDA logic and computations (reusable, testable)
> - `notebooks/01_eda_visualization.ipynb` - Only visualization and interactive exploration

3.1 Create `src/eda_analysis.py`

- ☐ **3.1.1** Create EDA analysis module:

```
#
================================================================================
# File: src/eda_analysis.py
# Description: EDA functions for data quality, distributions, and analysis
#
================================================================================

import pandas as pd
import numpy as np
from pathlib import Path
import sys

sys.path.insert(0, str(Path(__file__).parent))
from config import *
```

```
#
================================================================================
# DATA QUALITY ANALYSIS
#
================================================================================

def analyze_missing_values(df: pd.DataFrame, name: str = "data") ->
pd.DataFrame:
    """Analyze missing values in dataset."""
    missing = df.isnull().sum()
    missing_pct = (missing / len(df) * 100).round(2)

    report = pd.DataFrame({
        'column': missing.index,
        'missing_count': missing.values,
        'missing_pct': missing_pct.values
    }).sort_values('missing_count', ascending=False)

    print(f"\n📊 Missing Values Analysis - {name}")
    print(report[report['missing_count'] > 0].to_string(index=False))

    return report


def check_duplicates(df: pd.DataFrame, key_cols: list) -> pd.DataFrame:
    """Check for duplicate rows on key columns."""
    duplicates = df.duplicated(subset=key_cols, keep=False)
    n_duplicates = duplicates.sum()

    print(f"   Duplicates on {key_cols}: {n_duplicates}")

    if n_duplicates > 0:
        return df[duplicates]
    return pd.DataFrame()


def validate_ranges(df: pd.DataFrame) -> dict:
    """Validate data ranges are sensible."""
    issues = {}

    # Check negative volumes
    if 'volume' in df.columns:
        neg_vol = (df['volume'] < 0).sum()
        if neg_vol > 0:
            issues['negative_volumes'] = neg_vol

    # Check hospital_rate range
    if 'hospital_rate' in df.columns:
        out_range = ((df['hospital_rate'] < 0) | (df['hospital_rate'] >
100)).sum()
        if out_range > 0:
            issues['hospital_rate_out_of_range'] = out_range

    # Check n_gxs before entry
```

```
    if 'n_gxs' in df.columns and 'months_postgx' in df.columns:
        pre_entry_gxs = ((df['months_postgx'] < 0) & (df['n_gxs'] > 0)).sum()
        if pre_entry_gxs > 0:
            issues['generics_before_entry'] = pre_entry_gxs

    print(f"\n📊 Range Validation Issues: {issues if issues else 'None
found'}")
    return issues


def get_data_summary(df: pd.DataFrame) -> dict:
    """Get comprehensive data summary."""
    summary = {
        'shape': df.shape,
        'n_brands': df[['country', 'brand_name']].drop_duplicates().shape[0],
        'n_countries': df['country'].nunique() if 'country' in df.columns else
0,
        'months_postgx_range': (df['months_postgx'].min(),
df['months_postgx'].max()) if 'months_postgx' in df.columns else None,
        'volume_stats': df['volume'].describe().to_dict() if 'volume' in
df.columns else None
    }
    return summary


#
================================================================================
# BUCKET ANALYSIS
#
================================================================================

def analyze_bucket_distribution(aux_df: pd.DataFrame) -> pd.DataFrame:
    """Analyze distribution of erosion buckets."""
    bucket_counts = aux_df['bucket'].value_counts().sort_index()
    bucket_pct = (bucket_counts / len(aux_df) * 100).round(1)

    report = pd.DataFrame({
        'bucket': bucket_counts.index,
        'count': bucket_counts.values,
        'percentage': bucket_pct.values
    })

    print("\n📊 Bucket Distribution:")
    print(report.to_string(index=False))
    print(f"\n⚠️ Remember: Bucket 1 is weighted 2× in the metric!")

    return report


def analyze_bucket_characteristics(df: pd.DataFrame, aux_df: pd.DataFrame) ->
pd.DataFrame:
    """Analyze characteristics of each bucket."""
    merged = df.merge(aux_df[['country', 'brand_name', 'bucket']], on=
['country', 'brand_name'])
```

```
    # Group by bucket and compute stats
    bucket_stats = []
    for bucket in [1, 2]:
        bucket_data = merged[merged['bucket'] == bucket]

        stats = {
            'bucket': bucket,
            'n_brands': bucket_data[['country',
'brand_name']].drop_duplicates().shape[0],
            'avg_volume': bucket_data['volume'].mean(),
            'avg_n_gxs': bucket_data['n_gxs'].mean() if 'n_gxs' in
bucket_data.columns else None,
        }

        if 'hospital_rate' in bucket_data.columns:
            stats['avg_hospital_rate'] = bucket_data.groupby(['country',
'brand_name'])['hospital_rate'].first().mean()

        if 'biological' in bucket_data.columns:
            stats['pct_biological'] = bucket_data.groupby(['country',
'brand_name'])['biological'].first().mean() * 100

        bucket_stats.append(stats)

    return pd.DataFrame(bucket_stats)


#
================================================================================
# EROSION ANALYSIS
#
================================================================================

def compute_erosion_curves(df: pd.DataFrame, aux_df: pd.DataFrame) ->
pd.DataFrame:
    """
    Compute average erosion curves by bucket.
    Returns normalized volume (vol_norm) over months_postgx.
    """
    # Merge with avg_vol
    merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
                      on=['country', 'brand_name'])

    # Compute normalized volume
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    # Filter to post-entry
    post_entry = merged[merged['months_postgx'] >= 0]

    # Average by bucket and month
    erosion_curves = post_entry.groupby(['bucket', 'months_postgx'])
['vol_norm'].agg(['mean', 'std', 'count']).reset_index()
    erosion_curves.columns = ['bucket', 'months_postgx', 'mean_vol_norm',
```

```
    'std_vol_norm', 'n_brands']

        return erosion_curves


    def analyze_erosion_speed(df: pd.DataFrame, aux_df: pd.DataFrame) ->
    pd.DataFrame:
        """
        Analyze erosion speed metrics per brand.
        - Time to 50% erosion
        - Erosion in first 6 months
        - Final equilibrium level
        """
        merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
                          on=['country', 'brand_name'])
        merged['vol_norm'] = merged['volume'] / merged['avg_vol']

        results = []
        for (country, brand), group in merged.groupby(['country', 'brand_name']):
            group = group.sort_values('months_postgx')
            post_entry = group[group['months_postgx'] >= 0]

            if len(post_entry) == 0:
                continue

            bucket = group['bucket'].iloc[0]

            # Time to 50% erosion
            below_50 = post_entry[post_entry['vol_norm'] <= 0.5]
            time_to_50 = below_50['months_postgx'].min() if len(below_50) > 0 else
    np.nan

            # Erosion in first 6 months
            m0_5 = post_entry[post_entry['months_postgx'] <= 5]
            erosion_0_5 = 1 - m0_5['vol_norm'].mean() if len(m0_5) > 0 else np.nan

            # Final equilibrium (months 18-23)
            m18_23 = post_entry[post_entry['months_postgx'] >= 18]
            final_level = m18_23['vol_norm'].mean() if len(m18_23) > 0 else np.nan

            results.append({
                'country': country,
                'brand_name': brand,
                'bucket': bucket,
                'time_to_50pct': time_to_50,
                'erosion_first_6m': erosion_0_5,
                'final_equilibrium': final_level
            })

        return pd.DataFrame(results)


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

```
# COMPETITION ANALYSIS (n_gxs)
#
================================================================================

def analyze_n_gxs_impact(df: pd.DataFrame, aux_df: pd.DataFrame) ->
pd.DataFrame:
    """Analyze impact of generic competition on erosion."""
    merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol']], on=
['country', 'brand_name'])
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    # Post-entry only
    post_entry = merged[merged['months_postgx'] >= 0]

    # Group by n_gxs level
    n_gxs_impact = post_entry.groupby('n_gxs').agg({
        'vol_norm': ['mean', 'std', 'count']
    }).reset_index()
    n_gxs_impact.columns = ['n_gxs', 'mean_vol_norm', 'std_vol_norm',
'n_observations']

    return n_gxs_impact


def analyze_competition_trajectory(df: pd.DataFrame) -> pd.DataFrame:
    """Analyze how n_gxs evolves over time."""
    post_entry = df[df['months_postgx'] >= 0]

    trajectory = post_entry.groupby('months_postgx')['n_gxs'].agg(['mean',
'std', 'max']).reset_index()
    trajectory.columns = ['months_postgx', 'mean_n_gxs', 'std_n_gxs',
'max_n_gxs']

    return trajectory


#
================================================================================
# DRUG CHARACTERISTICS ANALYSIS
#
================================================================================

def analyze_by_therapeutic_area(df: pd.DataFrame, aux_df: pd.DataFrame) ->
pd.DataFrame:
    """Analyze erosion patterns by therapeutic area."""
    if 'ther_area' not in df.columns:
        return pd.DataFrame()

    merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
                      on=['country', 'brand_name'])
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    post_entry = merged[merged['months_postgx'] >= 0]
```

```
    ther_stats = post_entry.groupby('ther_area').agg({
        'vol_norm': 'mean',
        'country': lambda x: x.nunique(),  # n_brands proxy
        'bucket': lambda x: (x == 1).mean() * 100  # pct bucket 1
    }).reset_index()
    ther_stats.columns = ['ther_area', 'mean_erosion', 'n_brands',
'pct_bucket1']

    return ther_stats.sort_values('mean_erosion')


def analyze_biological_vs_small_molecule(df: pd.DataFrame, aux_df:
pd.DataFrame) -> pd.DataFrame:
    """Compare erosion between biological and small molecule drugs."""
    if 'biological' not in df.columns:
        return pd.DataFrame()

    merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol', 'bucket']],
                      on=['country', 'brand_name'])
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    post_entry = merged[merged['months_postgx'] >= 0]

    # Get drug type per brand
    drug_type = merged.groupby(['country', 'brand_name'])[['biological',
'small_molecule']].first()
    drug_type['drug_type'] = np.where(drug_type['biological'] == 1,
'Biological', 'Small Molecule')

    merged = merged.merge(drug_type[['drug_type']], left_on=['country',
'brand_name'], right_index=True)
    post_entry = merged[merged['months_postgx'] >= 0]

    comparison = post_entry.groupby(['drug_type', 'months_postgx'])
['vol_norm'].mean().unstack(level=0)

    return comparison


def analyze_hospital_rate_impact(df: pd.DataFrame, aux_df: pd.DataFrame) ->
pd.DataFrame:
    """Analyze erosion patterns by hospital rate."""
    if 'hospital_rate' not in df.columns:
        return pd.DataFrame()

    merged = df.merge(aux_df[['country', 'brand_name', 'avg_vol']], on=
['country', 'brand_name'])
    merged['vol_norm'] = merged['volume'] / merged['avg_vol']

    # Bin hospital rate
    merged['hospital_rate_bin'] = pd.cut(
        merged['hospital_rate'],
        bins=[0, 25, 75, 100],
        labels=['Low (0-25%)', 'Medium (25-75%)', 'High (75-100%)']
```

```
    )

    post_entry = merged[merged['months_postgx'] >= 0]

    hospital_stats = post_entry.groupby(['hospital_rate_bin',
'months_postgx'])['vol_norm'].mean().unstack(level=0)

    return hospital_stats


#
================================================================================
# FULL EDA REPORT
#
================================================================================

def run_full_eda(df: pd.DataFrame, aux_df: pd.DataFrame, save_report: bool =
True) -> dict:
    """
    Run comprehensive EDA and return all results.

    Args:
        df: Merged dataset
        aux_df: Auxiliary file with buckets
        save_report: If True, save report to file

    Returns:
        Dictionary with all EDA results
    """
    print("=" * 70)
    print("🔍 COMPREHENSIVE EDA ANALYSIS")
    print("=" * 70)

    results = {}

    # Data quality
    print("\n" + "=" * 50)
    print("SECTION 1: DATA QUALITY")
    print("=" * 50)
    results['missing_values'] = analyze_missing_values(df, "merged_data")
    results['range_issues'] = validate_ranges(df)
    results['data_summary'] = get_data_summary(df)

    # Bucket analysis
    print("\n" + "=" * 50)
    print("SECTION 2: BUCKET ANALYSIS")
    print("=" * 50)
    results['bucket_distribution'] = analyze_bucket_distribution(aux_df)
    results['bucket_characteristics'] = analyze_bucket_characteristics(df,
aux_df)

    # Erosion analysis
    print("\n" + "=" * 50)
    print("SECTION 3: EROSION ANALYSIS")
```

```
    print("=" * 50)
    results['erosion_curves'] = compute_erosion_curves(df, aux_df)
    results['erosion_speed'] = analyze_erosion_speed(df, aux_df)

    # Competition analysis
    print("\n" + "=" * 50)
    print("SECTION 4: COMPETITION ANALYSIS")
    print("=" * 50)
    results['n_gxs_impact'] = analyze_n_gxs_impact(df, aux_df)
    results['competition_trajectory'] = analyze_competition_trajectory(df)

    # Drug characteristics
    print("\n" + "=" * 50)
    print("SECTION 5: DRUG CHARACTERISTICS")
    print("=" * 50)
    results['ther_area_analysis'] = analyze_by_therapeutic_area(df, aux_df)
    results['bio_vs_small'] = analyze_biological_vs_small_molecule(df, aux_df)
    results['hospital_rate_analysis'] = analyze_hospital_rate_impact(df,
aux_df)

    if save_report:
        # Save key dataframes
        results['erosion_curves'].to_csv(DATA_PROCESSED /
"eda_erosion_curves.csv", index=False)
        results['bucket_characteristics'].to_csv(DATA_PROCESSED /
"eda_bucket_chars.csv", index=False)
        print(f"\n☑ EDA results saved to {DATA_PROCESSED}")

    print("\n" + "=" * 70)
    print("☑ EDA ANALYSIS COMPLETE")
    print("=" * 70)

    return results


if __name__ == "__main__":
    # Demo: Run full EDA
    print("=" * 60)
    print("EDA ANALYSIS DEMO")
    print("=" * 60)

    from data_loader import load_all_data, merge_datasets
    from bucket_calculator import create_auxiliary_file

    # Load data
    volume, generics, medicine = load_all_data(train=True)
    merged = merge_datasets(volume, generics, medicine)

    # Create auxiliary file
    aux_df = create_auxiliary_file(merged, save=True)

    # Run EDA
    eda_results = run_full_eda(merged, aux_df)
```

```
    print("\n☑ EDA demo complete!")
```

3.2 Create `notebooks/01_eda_visualization.ipynb` (Visualization Only)

- ☐ **3.2.1** Create visualization notebook that imports from `src/`:

```
<!-- filepath: notebooks/01_eda_visualization.ipynb -->
<VSCode.Cell language="markdown">
# 📊 EDA Visualization Notebook

This notebook is for **visualization only**. All analysis logic is in
`src/eda_analysis.py`.

## Setup
</VSCode.Cell>

<VSCode.Cell language="python">
# Setup and imports
import sys
from pathlib import Path

# Add src to path
sys.path.insert(0, str(Path.cwd().parent / 'src'))

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Import our modules
from config import *
from data_loader import load_all_data, merge_datasets
from bucket_calculator import create_auxiliary_file
from eda_analysis import *

# Set style
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("husl")
%matplotlib inline

print("☑ Setup complete!")
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 1. Load Data and Run EDA Analysis
</VSCode.Cell>

<VSCode.Cell language="python">
# Load data
```

```
volume, generics, medicine = load_all_data(train=True)
merged = merge_datasets(volume, generics, medicine)

# Create auxiliary file
aux_df = create_auxiliary_file(merged, save=True)

# Run comprehensive EDA (logic from src/eda_analysis.py)
eda_results = run_full_eda(merged, aux_df, save_report=True)
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 2. Bucket Distribution Visualization
</VSCode.Cell>

<VSCode.Cell language="python">
# Bucket distribution pie chart
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Pie chart
bucket_data = eda_results['bucket_distribution']
colors = ['#ff6b6b', '#4ecdc4']
axes[0].pie(bucket_data['count'], labels=[f"Bucket {b}" for b in
bucket_data['bucket']],
            autopct='%1.1f%%', colors=colors, explode=[0.05, 0])
axes[0].set_title('Bucket Distribution\n(Bucket 1 = High Erosion, Weight 2×)',
fontsize=12)

# Bar chart with counts
axes[1].bar(bucket_data['bucket'].astype(str), bucket_data['count'],
color=colors)
axes[1].set_xlabel('Bucket')
axes[1].set_ylabel('Number of Brands')
axes[1].set_title('Brands per Bucket')
for i, (b, c) in enumerate(zip(bucket_data['bucket'], bucket_data['count'])):
    axes[1].text(i, c + 5, str(c), ha='center', fontsize=12)

plt.tight_layout()
plt.savefig(FIGURES_DIR / 'bucket_distribution.png', dpi=150,
bbox_inches='tight')
plt.show()
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 3. Erosion Curves by Bucket
</VSCode.Cell>

<VSCode.Cell language="python">
# Plot erosion curves by bucket
erosion_curves = eda_results['erosion_curves']

fig, ax = plt.subplots(figsize=(12, 6))

for bucket in [1, 2]:
    data = erosion_curves[erosion_curves['bucket'] == bucket]
```

```
    label = f"Bucket {bucket}" + (" (High Erosion - 2× weight)" if bucket == 1
else " (Lower Erosion)")
    color = '#ff6b6b' if bucket == 1 else '#4ecdc4'

    ax.plot(data['months_postgx'], data['mean_vol_norm'],
            label=label, linewidth=2, color=color)
    ax.fill_between(data['months_postgx'],
                    data['mean_vol_norm'] - data['std_vol_norm'],
                    data['mean_vol_norm'] + data['std_vol_norm'],
                    alpha=0.2, color=color)

ax.axhline(y=1.0, color='gray', linestyle='--', alpha=0.5, label='Pre-entry
baseline')
ax.axhline(y=0.25, color='red', linestyle=':', alpha=0.5, label='Bucket 1
threshold')
ax.axvline(x=0, color='black', linestyle='-', alpha=0.3, label='Generic
entry')

ax.set_xlabel('Months Since Generic Entry', fontsize=12)
ax.set_ylabel('Normalized Volume (Volume / Avg_j)', fontsize=12)
ax.set_title('Generic Erosion Curves by Bucket', fontsize=14)
ax.legend(loc='upper right')
ax.set_xlim(-1, 24)
ax.set_ylim(0, 1.5)

plt.tight_layout()
plt.savefig(FIGURES_DIR / 'erosion_curves_by_bucket.png', dpi=150,
bbox_inches='tight')
plt.show()
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 4. Sample Brand Trajectories
</VSCode.Cell>

<VSCode.Cell language="python">
# Plot sample brand trajectories
sample_brands = merged[['country', 'brand_name']].drop_duplicates().sample(6,
random_state=42)

fig, axes = plt.subplots(2, 3, figsize=(15, 8))
axes = axes.flatten()

for idx, (_, row) in enumerate(sample_brands.iterrows()):
    brand_data = merged[(merged['country'] == row['country']) &
                        (merged['brand_name'] ==
row['brand_name'])].sort_values('months_postgx')

    ax = axes[idx]
    ax.plot(brand_data['months_postgx'], brand_data['volume'], 'b-',
linewidth=2)
    ax.axvline(x=0, color='red', linestyle='--', alpha=0.7, label='Generic
Entry')
    ax.set_title(f"{row['country']} - {row['brand_name'][:15]}", fontsize=10)
```

```
    ax.set_xlabel('Months Post GX')
    ax.set_ylabel('Volume')

    if idx == 0:
        ax.legend()

plt.suptitle('Sample Brand Volume Trajectories', fontsize=14, y=1.02)
plt.tight_layout()
plt.savefig(FIGURES_DIR / 'sample_brand_trajectories.png', dpi=150,
bbox_inches='tight')
plt.show()
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 5. n_gxs Impact on Erosion
</VSCode.Cell>

<VSCode.Cell language="python">
# Plot n_gxs impact
n_gxs_impact = eda_results['n_gxs_impact']

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Mean erosion by n_gxs
axes[0].bar(n_gxs_impact['n_gxs'], n_gxs_impact['mean_vol_norm'],
color='steelblue', alpha=0.7)
axes[0].set_xlabel('Number of Generics (n_gxs)', fontsize=12)
axes[0].set_ylabel('Mean Normalized Volume', fontsize=12)
axes[0].set_title('Erosion vs Number of Generic Competitors', fontsize=12)
axes[0].axhline(y=1.0, color='gray', linestyle='--', alpha=0.5)

# Competition trajectory over time
comp_traj = eda_results['competition_trajectory']
axes[1].plot(comp_traj['months_postgx'], comp_traj['mean_n_gxs'], 'g-',
linewidth=2, label='Mean')
axes[1].fill_between(comp_traj['months_postgx'],
                     comp_traj['mean_n_gxs'] - comp_traj['std_n_gxs'],
                     comp_traj['mean_n_gxs'] + comp_traj['std_n_gxs'],
                     alpha=0.2, color='green')
axes[1].set_xlabel('Months Since Generic Entry', fontsize=12)
axes[1].set_ylabel('Number of Generics', fontsize=12)
axes[1].set_title('Generic Competition Over Time', fontsize=12)

plt.tight_layout()
plt.savefig(FIGURES_DIR / 'n_gxs_impact.png', dpi=150, bbox_inches='tight')
plt.show()
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 6. Therapeutic Area Analysis
</VSCode.Cell>

<VSCode.Cell language="python">
# Therapeutic area erosion comparison
```

```
ther_analysis = eda_results['ther_area_analysis']

if len(ther_analysis) > 0:
    fig, ax = plt.subplots(figsize=(12, 6))

    # Sort by mean erosion
    ther_analysis = ther_analysis.sort_values('mean_erosion')

    colors = plt.cm.RdYlGn(np.linspace(0.2, 0.8, len(ther_analysis)))
    bars = ax.barh(ther_analysis['ther_area'], ther_analysis['mean_erosion'],
color=colors)

    ax.axvline(x=0.25, color='red', linestyle='--', alpha=0.7, label='Bucket 1
threshold')
    ax.set_xlabel('Mean Normalized Volume (Lower = More Erosion)',
fontsize=12)
    ax.set_title('Erosion by Therapeutic Area', fontsize=14)
    ax.legend()

    plt.tight_layout()
    plt.savefig(FIGURES_DIR / 'ther_area_erosion.png', dpi=150,
bbox_inches='tight')
    plt.show()
else:
    print("No therapeutic area data available")
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 7. Biological vs Small Molecule
</VSCode.Cell>

<VSCode.Cell language="python">
# Biological vs Small Molecule comparison
bio_vs_small = eda_results['bio_vs_small']

if len(bio_vs_small) > 0:
    fig, ax = plt.subplots(figsize=(12, 6))

    bio_vs_small.plot(ax=ax, linewidth=2)

    ax.axhline(y=1.0, color='gray', linestyle='--', alpha=0.5)
    ax.axvline(x=0, color='black', linestyle='-', alpha=0.3)
    ax.set_xlabel('Months Since Generic Entry', fontsize=12)
    ax.set_ylabel('Mean Normalized Volume', fontsize=12)
    ax.set_title('Erosion: Biological vs Small Molecule Drugs', fontsize=14)
    ax.legend(title='Drug Type')

    plt.tight_layout()
    plt.savefig(FIGURES_DIR / 'biological_vs_small_molecule.png', dpi=150,
bbox_inches='tight')
    plt.show()
else:
    print("No biological/small molecule data available")
</VSCode.Cell>
```

```
<VSCode.Cell language="markdown">
## 8. Hospital Rate Impact
</VSCode.Cell>

<VSCode.Cell language="python">
# Hospital rate analysis
hospital_analysis = eda_results['hospital_rate_analysis']

if len(hospital_analysis) > 0:
    fig, ax = plt.subplots(figsize=(12, 6))

    hospital_analysis.plot(ax=ax, linewidth=2)

    ax.axhline(y=1.0, color='gray', linestyle='--', alpha=0.5)
    ax.set_xlabel('Months Since Generic Entry', fontsize=12)
    ax.set_ylabel('Mean Normalized Volume', fontsize=12)
    ax.set_title('Erosion by Hospital Rate', fontsize=14)
    ax.legend(title='Hospital Rate')

    plt.tight_layout()
    plt.savefig(FIGURES_DIR / 'hospital_rate_erosion.png', dpi=150,
bbox_inches='tight')
    plt.show()
else:
    print("No hospital rate data available")
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 9. Erosion Speed Analysis
</VSCode.Cell>

<VSCode.Cell language="python">
# Erosion speed by bucket
erosion_speed = eda_results['erosion_speed']

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Time to 50% erosion
for bucket in [1, 2]:
    data = erosion_speed[erosion_speed['bucket'] == bucket]
['time_to_50pct'].dropna()
    color = '#ff6b6b' if bucket == 1 else '#4ecdc4'
    axes[0].hist(data, bins=20, alpha=0.6, label=f'Bucket {bucket}',
color=color)
axes[0].set_xlabel('Months to 50% Erosion')
axes[0].set_title('Time to 50% Volume Loss')
axes[0].legend()

# Erosion in first 6 months
for bucket in [1, 2]:
    data = erosion_speed[erosion_speed['bucket'] == bucket]
['erosion_first_6m'].dropna()
    color = '#ff6b6b' if bucket == 1 else '#4ecdc4'
```

```
    axes[1].hist(data, bins=20, alpha=0.6, label=f'Bucket {bucket}',
color=color)
axes[1].set_xlabel('Erosion Rate (1 - mean vol_norm)')
axes[1].set_title('Erosion in First 6 Months')
axes[1].legend()

# Final equilibrium
for bucket in [1, 2]:
    data = erosion_speed[erosion_speed['bucket'] == bucket]
['final_equilibrium'].dropna()
    color = '#ff6b6b' if bucket == 1 else '#4ecdc4'
    axes[2].hist(data, bins=20, alpha=0.6, label=f'Bucket {bucket}',
color=color)
axes[2].set_xlabel('Final Normalized Volume (months 18-23)')
axes[2].set_title('Final Equilibrium Level')
axes[2].legend()

plt.tight_layout()
plt.savefig(FIGURES_DIR / 'erosion_speed_analysis.png', dpi=150,
bbox_inches='tight')
plt.show()
</VSCode.Cell>

<VSCode.Cell language="markdown">
## 10. Summary Statistics
</VSCode.Cell>

<VSCode.Cell language="python">
# Print summary
print("=" * 60)
print("📊 EDA SUMMARY")
print("=" * 60)

summary = eda_results['data_summary']
print(f"\nDataset Shape: {summary['shape']}")
print(f"Total Brands: {summary['n_brands']}")
print(f"Countries: {summary['n_countries']}")
print(f"Months Range: {summary['months_postgx_range']}")

print("\n" + "-" * 40)
bucket_dist = eda_results['bucket_distribution']
print("\nBucket Distribution:")
for _, row in bucket_dist.iterrows():
    print(f"  Bucket {row['bucket']}: {row['count']} brands
({row['percentage']}%)")

print("\n⚠️ KEY INSIGHT: Bucket 1 (high erosion) is weighted 2× in the
metric!")
print("   Focus optimization on Bucket 1 predictions!")

print("\n☑️ All figures saved to:", FIGURES_DIR)
</VSCode.Cell>
```

🚀 STEP 4: Feature Engineering (Use `src/feature_engineering.py`)

> **Already implemented in STEP 2.4!** Run the module directly.

4.1 Run Feature Engineering

- [ ] **4.1.1** Run feature engineering from command line:

```
# Navigate to project directory
cd D:\Datathon\novartis_datathon_2025\Main_project

# Activate environment
& .\saeed_venv\Scripts\Activate.ps1

# Run feature engineering demo
python src/feature_engineering.py
```

4.2 Create `notebooks/02_feature_exploration.ipynb` (Visualization Only)

- [ ] **4.2.1** Create notebook for feature visualization:

```
<!-- filepath: notebooks/02_feature_exploration.ipynb -->
<VSCode.Cell language="markdown">
# 🔧 Feature Exploration Notebook

Visualize features created by `src/feature_engineering.py`.
</VSCode.Cell>

<VSCode.Cell language="python">
# Setup
import sys
from pathlib import Path
sys.path.insert(0, str(Path.cwd().parent / 'src'))

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from config import *
from data_loader import load_all_data, merge_datasets
from bucket_calculator import compute_avg_j, create_auxiliary_file
from feature_engineering import create_all_features, get_feature_columns

plt.style.use('seaborn-v0_8-whitegrid')
%matplotlib inline
</VSCode.Cell>
```

```
<VSCode.Cell language="python">
# Load and create features
volume, generics, medicine = load_all_data(train=True)
merged = merge_datasets(volume, generics, medicine)
aux_df = create_auxiliary_file(merged, save=False)
avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

# Create all features
featured = create_all_features(merged, avg_j)
feature_cols = get_feature_columns(featured)

print(f"Total features: {len(feature_cols)}")
print(f"Feature columns:\n{feature_cols[:20]}")
</VSCode.Cell>

<VSCode.Cell language="python">
# Feature correlation heatmap
numeric_features = featured[feature_cols].select_dtypes(include=
[np.number]).columns[:15]

fig, ax = plt.subplots(figsize=(12, 10))
corr = featured[numeric_features].corr()
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm', center=0, ax=ax)
ax.set_title('Feature Correlation Matrix (Top 15)', fontsize=14)
plt.tight_layout()
plt.savefig(FIGURES_DIR / 'feature_correlation.png', dpi=150)
plt.show()
</VSCode.Cell>

<VSCode.Cell language="python">
# Feature distributions
fig, axes = plt.subplots(3, 4, figsize=(16, 12))
axes = axes.flatten()

for idx, col in enumerate(numeric_features[:12]):
    featured[col].hist(bins=50, ax=axes[idx], alpha=0.7)
    axes[idx].set_title(col, fontsize=10)

plt.suptitle('Feature Distributions', fontsize=14, y=1.02)
plt.tight_layout()
plt.savefig(FIGURES_DIR / 'feature_distributions.png', dpi=150)
plt.show()
</VSCode.Cell>
```

🚀 STEP 5: Model Training (Use `src/models.py`)

> **Already implemented in STEP 2.5!** Core logic in `.py` files.

5.1 Run Baseline Models

- [ ] **5.1.1** Run baseline models from command line:

```
# Run models demo
python src/models.py
```

5.2 Create `scripts/train_models.py`

- [ ] **5.2.1** Create training script:

```
#
==============================================================================
# File: scripts/train_models.py
# Description: Script to train and compare all models
#
==============================================================================

import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))

import pandas as pd
import numpy as np

from config import *
from data_loader import load_all_data, merge_datasets, split_train_validation
from bucket_calculator import compute_avg_j, create_auxiliary_file
from feature_engineering import create_all_features, get_feature_columns
from models import GradientBoostingModel, BaselineModels,
prepare_training_data, train_and_evaluate
from evaluation import evaluate_model, compare_models


def train_all_models(scenario: int = 1):
    """Train and compare all models."""
    print("=" * 70)
    print(f"🚀 TRAINING ALL MODELS - SCENARIO {scenario}")
    print("=" * 70)

    # Load and prepare data
    print("\n📂 Loading data...")
    volume, generics, medicine = load_all_data(train=True)
    merged = merge_datasets(volume, generics, medicine)

    # Create auxiliary file
    aux_df = create_auxiliary_file(merged, save=True)
    avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

    # Create features
```

```
    print("\n🔧 Creating features...")
    featured = create_all_features(merged, avg_j)

    # Split train/validation
    train_df, val_df = split_train_validation(featured)

    # Prepare data
    feature_cols = get_feature_columns(featured)
    X_train, y_train = prepare_training_data(train_df, feature_cols)
    X_val, y_val = prepare_training_data(val_df, feature_cols)

    # Get validation actual data for evaluation
    val_actual = val_df[val_df['months_postgx'] >= 0][
        ['country', 'brand_name', 'months_postgx', 'volume']
    ].copy()

    val_brands = val_df[['country', 'brand_name']].drop_duplicates()
    val_aux = aux_df.merge(val_brands, on=['country', 'brand_name'])
    val_avg_j = avg_j.merge(val_brands, on=['country', 'brand_name'])

    #
================================================================================
    # BASELINE MODELS
    #
================================================================================
    print("\n" + "=" * 50)
    print("BASELINE MODELS")
    print("=" * 50)

    months = list(range(0, 24)) if scenario == 1 else list(range(6, 24))

    # Tune exponential decay
    best_rate, _ = BaselineModels.tune_decay_rate(
        val_actual, val_avg_j, decay_type='exponential'
    )

    baseline_preds = BaselineModels.exponential_decay(val_avg_j, months,
best_rate)
    baseline_results = evaluate_model(val_actual, baseline_preds, val_aux,
scenario)

    #
================================================================================
    # LIGHTGBM
    #
================================================================================
    print("\n" + "=" * 50)
    print("LIGHTGBM MODEL")
    print("=" * 50)

    lgbm_model, lgbm_metrics = train_and_evaluate(X_train, y_train, X_val,
y_val, 'lightgbm')

    # Generate predictions for evaluation
```

```
    val_pred_data = val_df[val_df['months_postgx'].between(0, 23) if scenario
== 1 else val_df['months_postgx'].between(6, 23)].copy()
    X_val_pred = val_pred_data[feature_cols].fillna(0)
    lgbm_predictions = lgbm_model.predict(X_val_pred)

    lgbm_pred_df = val_pred_data[['country', 'brand_name',
'months_postgx']].copy()
    lgbm_pred_df['volume'] = lgbm_predictions

    lgbm_results = evaluate_model(val_actual, lgbm_pred_df, val_aux, scenario)

    # Feature importance
    print("\n📊 LightGBM Feature Importance (Top 15):")
    print(lgbm_model.get_feature_importance(15))

    # Save model
    lgbm_model.save(f"scenario{scenario}_lightgbm")

    #
================================================================================
    # XGBOOST
    #
================================================================================
    print("\n" + "=" * 50)
    print("XGBOOST MODEL")
    print("=" * 50)

    xgb_model, xgb_metrics = train_and_evaluate(X_train, y_train, X_val,
y_val, 'xgboost')

    xgb_predictions = xgb_model.predict(X_val_pred)
    xgb_pred_df = val_pred_data[['country', 'brand_name',
'months_postgx']].copy()
    xgb_pred_df['volume'] = xgb_predictions

    xgb_results = evaluate_model(val_actual, xgb_pred_df, val_aux, scenario)

    # Save model
    xgb_model.save(f"scenario{scenario}_xgboost")

    #
================================================================================
    # MODEL COMPARISON
    #
================================================================================
    print("\n" + "=" * 50)
    print("MODEL COMPARISON")
    print("=" * 50)

    comparison = compare_models(
        [baseline_results, lgbm_results, xgb_results],
        ['Exponential Decay', 'LightGBM', 'XGBoost']
    )
```

```
    # Save comparison
    comparison.to_csv(REPORTS_DIR /
f"model_comparison_scenario{scenario}.csv", index=False)

    print("\n" + "=" * 70)
    print("☑ MODEL TRAINING COMPLETE")
    print("=" * 70)

    return {
        'baseline': baseline_results,
        'lightgbm': lgbm_results,
        'xgboost': xgb_results,
        'comparison': comparison
    }


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--scenario', type=int, default=1, choices=[1, 2])
    args = parser.parse_args()

    train_all_models(scenario=args.scenario)
```

5.3 Create `notebooks/03_model_results.ipynb` (Visualization Only)

- ☐ **5.3.1** Create notebook for model results visualization:

```
<!-- filepath: notebooks/03_model_results.ipynb -->
<VSCode.Cell language="markdown">
# 📈 Model Results Visualization

Visualize training results from `scripts/train_models.py`.
</VSCode.Cell>

<VSCode.Cell language="python">
# Setup
import sys
from pathlib import Path
sys.path.insert(0, str(Path.cwd().parent / 'src'))

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from config import *

plt.style.use('seaborn-v0_8-whitegrid')
%matplotlib inline
```

```
</VSCode.Cell>

<VSCode.Cell language="python">
# Load model comparison results
scenario = 1  # Change to 2 for Scenario 2

comparison = pd.read_csv(REPORTS_DIR /
f"model_comparison_scenario{scenario}.csv")
print(comparison)
</VSCode.Cell>

<VSCode.Cell language="python">
# Plot model comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Final Score
axes[0].bar(comparison['model'], comparison['final_score'], color=['gray',
'steelblue', 'orange'])
axes[0].set_ylabel('Final PE Score (Lower is Better)')
axes[0].set_title('Overall Model Comparison')
axes[0].tick_params(axis='x', rotation=45)

# Bucket 1 PE
axes[1].bar(comparison['model'], comparison['bucket1_pe'], color=
['#ff6b6b']*3)
axes[1].set_ylabel('Bucket 1 Avg PE')
axes[1].set_title('Bucket 1 Performance (2× Weight)')
axes[1].tick_params(axis='x', rotation=45)

# Bucket 2 PE
axes[2].bar(comparison['model'], comparison['bucket2_pe'], color=
['#4ecdc4']*3)
axes[2].set_ylabel('Bucket 2 Avg PE')
axes[2].set_title('Bucket 2 Performance (1× Weight)')
axes[2].tick_params(axis='x', rotation=45)

plt.suptitle(f'Scenario {scenario} Model Comparison', fontsize=14, y=1.02)
plt.tight_layout()
plt.savefig(FIGURES_DIR / f'model_comparison_scenario{scenario}.png', dpi=150)
plt.show()
</VSCode.Cell>

<VSCode.Cell language="python">
# Load and visualize feature importance
import joblib

model_path = MODELS_DIR / f"scenario{scenario}_lightgbm.joblib"
if model_path.exists():
    model_data = joblib.load(model_path)
    model = model_data['model']
    feature_names = model_data['feature_names']

    importance_df = pd.DataFrame({
        'feature': feature_names,
```

```
            'importance': model.feature_importances_
        }).sort_values('importance', ascending=False).head(20)

        fig, ax = plt.subplots(figsize=(10, 8))
        ax.barh(importance_df['feature'], importance_df['importance'],
color='steelblue')
        ax.set_xlabel('Importance')
        ax.set_title(f'LightGBM Feature Importance - Scenario {scenario}')
        ax.invert_yaxis()
        plt.tight_layout()
        plt.savefig(FIGURES_DIR / f'feature_importance_scenario{scenario}.png',
dpi=150)
        plt.show()
    else:
        print(f"Model not found at {model_path}. Run train_models.py first.")
    </VSCode.Cell>
```

🚀 STEP 6: Generate Predictions and Submissions

> **Use `src/pipeline.py` and `src/submission.py`** - Already implemented!

6.1 Run Full Pipeline

- [ ] **6.1.1** Run pipeline from command line:

```
# Scenario 1 - Validation mode (evaluate on held-out data)
python src/pipeline.py --scenario 1 --model lightgbm

# Scenario 1 - Test mode (generate submission)
python src/pipeline.py --scenario 1 --model lightgbm --test

# Scenario 2 - Validation mode
python src/pipeline.py --scenario 2 --model lightgbm

# Scenario 2 - Test mode (generate submission)
python src/pipeline.py --scenario 2 --model lightgbm --test
```

6.2 Create `scripts/generate_final_submissions.py`

- [ ] **6.2.1** Create final submission script:

```
#
================================================================================
# File: scripts/generate_final_submissions.py
# Description: Generate final submissions for both scenarios
```

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

import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))

from config import *
from data_loader import load_all_data, merge_datasets
from bucket_calculator import compute_avg_j, create_auxiliary_file
from feature_engineering import create_all_features, get_feature_columns
from models import GradientBoostingModel
from submission import generate_submission, save_submission,
validate_submission


def generate_final_submissions():
    """Generate final submissions for both scenarios."""
    print("=" * 70)
    print("🎯 GENERATING FINAL SUBMISSIONS")
    print("=" * 70)

    # Load training data (for auxiliary file)
    print("\n📂 Loading training data...")
    vol_train, gen_train, med_train = load_all_data(train=True)
    merged_train = merge_datasets(vol_train, gen_train, med_train)
    aux_df = create_auxiliary_file(merged_train, save=True)
    avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

    # Load test data
    print("\n📂 Loading test data...")
    vol_test, gen_test, med_test = load_all_data(train=False)
    merged_test = merge_datasets(vol_test, gen_test, med_test)

    # Create features for test
    print("\n🔧 Creating features for test data...")
    test_featured = create_all_features(merged_test, avg_j)
    feature_cols = get_feature_columns(test_featured)

    #
================================================================================
    # SCENARIO 1
    #
================================================================================
    print("\n" + "=" * 50)
    print("SCENARIO 1: Predict months 0-23")
    print("=" * 50)

    # Load model
    model_s1 = GradientBoostingModel()
    model_s1.load("scenario1_lightgbm")

    # Prepare test data for scenario 1
    test_s1 = test_featured[test_featured['months_postgx'].between(0,
```

```
23)].copy()
    X_test_s1 = test_s1[feature_cols].fillna(0)

    # Generate predictions
    predictions_s1 = model_s1.predict(X_test_s1)

    # Create submission
    submission_s1 = test_s1[['country', 'brand_name', 'months_postgx']].copy()
    submission_s1['volume'] = predictions_s1

    submission_s1 = generate_submission(submission_s1, scenario=1)
    filepath_s1 = save_submission(submission_s1, scenario=1, suffix="_final")

    # Expected: 228 brands × 24 months = 5,472 rows
    print(f"   Expected rows: 5,472")
    print(f"   Actual rows: {len(submission_s1)}")

    #
================================================================================
    # SCENARIO 2
    #
================================================================================
    print("\n" + "=" * 50)
    print("SCENARIO 2: Predict months 6-23 (given 0-5 actuals)")
    print("=" * 50)

    # Load model
    model_s2 = GradientBoostingModel()
    model_s2.load("scenario2_lightgbm")

    # Prepare test data for scenario 2
    test_s2 = test_featured[test_featured['months_postgx'].between(6,
23)].copy()
    X_test_s2 = test_s2[feature_cols].fillna(0)

    # Generate predictions
    predictions_s2 = model_s2.predict(X_test_s2)

    # Create submission
    submission_s2 = test_s2[['country', 'brand_name', 'months_postgx']].copy()
    submission_s2['volume'] = predictions_s2

    submission_s2 = generate_submission(submission_s2, scenario=2)
    filepath_s2 = save_submission(submission_s2, scenario=2, suffix="_final")

    # Expected: 112 brands × 18 months = 2,016 rows
    print(f"   Expected rows: 2,016")
    print(f"   Actual rows: {len(submission_s2)}")

    #
================================================================================
    # SUMMARY
    #
================================================================================
```

```
    print("\n" + "=" * 70)
    print("☑ FINAL SUBMISSIONS GENERATED")
    print("=" * 70)
    print(f"\n   Scenario 1: {filepath_s1}")
    print(f"   Scenario 2: {filepath_s2}")

    return submission_s1, submission_s2


if __name__ == "__main__":
    generate_final_submissions()
```

🚀 STEP 7: Validation and Testing

7.1 Create `scripts/validate_submissions.py`

- ☐ **7.1.1** Create validation script:

```
#
================================================================================
# File: scripts/validate_submissions.py
# Description: Validate submission files before upload
#
================================================================================

import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))

import pandas as pd
from config import *
from submission import validate_submission


def validate_all_submissions():
    """Validate all submission files in submissions directory."""
    print("=" * 70)
    print("🔍 VALIDATING SUBMISSIONS")
    print("=" * 70)

    submission_files = list(SUBMISSIONS_DIR.glob("*.csv"))

    if not submission_files:
        print("❌ No submission files found!")
        return

    for filepath in submission_files:
        print(f"\n📄 Validating: {filepath.name}")
```

```
        # Load submission
        submission = pd.read_csv(filepath)

        # Determine scenario from filename
        if "scenario1" in filepath.name.lower():
            scenario = 1
        elif "scenario2" in filepath.name.lower():
            scenario = 2
        else:
            print(f"    ⚠ Cannot determine scenario from filename")
            continue

        try:
            validate_submission(submission, scenario)
        except AssertionError as e:
            print(f"    ❌ Validation failed: {e}")


if __name__ == "__main__":
    validate_all_submissions()
```

7.2 Create `scripts/run_demo.py`

- ☐ **7.2.1** Create demo script for quick testing:

```
#
================================================================================
# File: scripts/run_demo.py
# Description: Quick demo to test the entire pipeline
#
================================================================================

import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent / 'src'))

import pandas as pd
import numpy as np

from config import *
from data_loader import load_all_data, merge_datasets, split_train_validation
from bucket_calculator import compute_avg_j, create_auxiliary_file
from feature_engineering import create_all_features, get_feature_columns
from models import GradientBoostingModel, BaselineModels,
prepare_training_data
from evaluation import evaluate_model
from submission import generate_submission


def run_quick_demo():
```

```
    """Run a quick demo of the entire pipeline."""
    print("=" * 70)
    print("🎯 QUICK DEMO - NOVARTIS DATATHON PIPELINE")
    print("=" * 70)

    #
================================================================================
    # 1. LOAD DATA
    #
================================================================================
    print("\n📂 Step 1: Loading data...")
    try:
        volume, generics, medicine = load_all_data(train=True)
        print(f"   Volume shape: {volume.shape}")
        print(f"   Generics shape: {generics.shape}")
        print(f"   Medicine shape: {medicine.shape}")
    except Exception as e:
        print(f"   ❌ Error loading data: {e}")
        print(f"   Make sure data files are in: {DATA_RAW}")
        return

    #
================================================================================
    # 2. MERGE DATA
    #
================================================================================
    print("\n🔗 Step 2: Merging datasets...")
    merged = merge_datasets(volume, generics, medicine)
    print(f"   Merged shape: {merged.shape}")

    #
================================================================================
    # 3. CREATE AUXILIARY FILE
    #
================================================================================
    print("\n📊 Step 3: Creating auxiliary file (buckets)...")
    aux_df = create_auxiliary_file(merged, save=True)
    avg_j = aux_df[['country', 'brand_name', 'avg_vol']].copy()

    #
================================================================================
    # 4. FEATURE ENGINEERING
    #
================================================================================
    print("\n🔧 Step 4: Creating features...")
    featured = create_all_features(merged, avg_j)
    feature_cols = get_feature_columns(featured)
    print(f"   Features created: {len(feature_cols)}")

    #
================================================================================
    # 5. SPLIT DATA
    #
================================================================================
```

```
    print("\n✂️ Step 5: Splitting train/validation...")
    train_df, val_df = split_train_validation(featured, val_brands_ratio=0.2)

    #
================================================================================
    # 6. TRAIN MODEL
    #
================================================================================
    print("\n🤖 Step 6: Training LightGBM model...")
    X_train, y_train = prepare_training_data(train_df, feature_cols)
    X_val, y_val = prepare_training_data(val_df, feature_cols)

    model = GradientBoostingModel(model_type='lightgbm')
    model.fit(X_train, y_train, X_val, y_val)

    #
================================================================================
    # 7. EVALUATE
    #
================================================================================
    print("\n📊 Step 7: Evaluating model...")
    val_actual = val_df[val_df['months_postgx'].between(0, 23)][
        ['country', 'brand_name', 'months_postgx', 'volume']
    ].copy()

    val_pred_data = val_df[val_df['months_postgx'].between(0, 23)].copy()
    X_pred = val_pred_data[feature_cols].fillna(0)
    predictions = model.predict(X_pred)

    val_pred = val_pred_data[['country', 'brand_name',
'months_postgx']].copy()
    val_pred['volume'] = predictions

    val_brands = val_df[['country', 'brand_name']].drop_duplicates()
    val_aux = aux_df.merge(val_brands, on=['country', 'brand_name'])

    results = evaluate_model(val_actual, val_pred, val_aux, scenario=1)

    #
================================================================================
    # 8. FEATURE IMPORTANCE
    #
================================================================================
    print("\n📈 Step 8: Top 10 features:")
    importance = model.get_feature_importance(10)
    print(importance.to_string(index=False))

    #
================================================================================
    # SUMMARY
    #
================================================================================
    print("\n" + "=" * 70)
    print("☑️ DEMO COMPLETE!")
```

```
    print("=" * 70)
    print(f"\n   Final PE Score: {results['final_score']:.4f}")
    print(f"   Bucket 1 Avg PE: {results['bucket1_avg_pe']:.4f} (n=
{results['n_bucket1']})")
    print(f"   Bucket 2 Avg PE: {results['bucket2_avg_pe']:.4f} (n=
{results['n_bucket2']})")
    print(f"\n   ⚠ Remember: Lower PE is better!")
    print(f"   ⚠ Bucket 1 is weighted 2× in final metric!")

    return results


if __name__ == "__main__":
    run_quick_demo()
```

🚀 STEP 8: Run the Complete Project

8.1 Command Line Execution Guide

```
#
================================================================================
# COMPLETE EXECUTION GUIDE
#
================================================================================

# 1. Navigate to project directory
cd D:\Datathon\novartis_datathon_2025\Main_project

# 2. Activate virtual environment
& .\saeed_venv\Scripts\Activate.ps1

# 3. Install requirements (first time only)
pip install -r requirements.txt

#
================================================================================
# STEP-BY-STEP EXECUTION
#
================================================================================

# STEP 1: Test configuration
python src/config.py

# STEP 2: Test data loading
python src/data_loader.py

# STEP 3: Create auxiliary file (buckets)
python src/bucket_calculator.py
```

```
# STEP 4: Run EDA analysis
python src/eda_analysis.py

# STEP 5: Test feature engineering
python src/feature_engineering.py

# STEP 6: Test models
python src/models.py

# STEP 7: Test evaluation
python src/evaluation.py

# STEP 8: Test submission generation
python src/submission.py

#
================================================================================
# RUN COMPLETE PIPELINE
#
================================================================================

# Quick demo (validation mode)
python scripts/run_demo.py

# Train all models
python scripts/train_models.py --scenario 1
python scripts/train_models.py --scenario 2

# Run full pipeline (validation mode)
python src/pipeline.py --scenario 1 --model lightgbm
python src/pipeline.py --scenario 2 --model lightgbm

# Generate final submissions (test mode)
python src/pipeline.py --scenario 1 --model lightgbm --test
python src/pipeline.py --scenario 2 --model lightgbm --test

# Or use the dedicated script
python scripts/generate_final_submissions.py

# Validate submissions
python scripts/validate_submissions.py
```

8.2 Expected Output Files

After running the complete pipeline, you should have:

```
Main_project/
├── data/
│   ├── raw/                        # Input data (copy here)
│   │   ├── df_volume_train.csv
```

```
    │   │   ├── df_volume_test.csv
    │   │   ├── df_generics_train.csv
    │   │   ├── df_generics_test.csv
    │   │   ├── df_medicine_info_train.csv
    │   │   └── df_medicine_info_test.csv
    │   └── processed/
    │       ├── aux_bucket_avgvol.csv        # ☑ Created by bucket_calculator.py
    │       ├── eda_erosion_curves.csv       # ☑ Created by eda_analysis.py
    │       └── eda_bucket_chars.csv         # ☑ Created by eda_analysis.py
    ├── models/
    │   ├── scenario1_lightgbm.joblib        # ☑ Created by train_models.py
    │   ├── scenario2_lightgbm.joblib        # ☑ Created by train_models.py
    │   ├── scenario1_xgboost.joblib         # ☑ Created by train_models.py
    │   └── scenario2_xgboost.joblib         # ☑ Created by train_models.py
    ├── submissions/
    │   ├── scenario1_YYYYMMDD_HHMMSS_final.csv   # ☑ Final submission
    │   └── scenario2_YYYYMMDD_HHMMSS_final.csv   # ☑ Final submission
    ├── reports/
    │   ├── figures/
    │   │   ├── bucket_distribution.png      # ☑ Created by notebook
    │   │   ├── erosion_curves_by_bucket.png # ☑ Created by notebook
    │   │   ├── sample_brand_trajectories.png
    │   │   ├── n_gxs_impact.png
    │   │   ├── feature_correlation.png
    │   │   └── model_comparison_scenario1.png
    │   └── model_comparison_scenario1.csv   # ☑ Created by train_models.py
```

🚀 STEP 9: Notebooks for Visualization (Optional)

> **All notebooks are for visualization only!** Core logic is in `src/` modules.

9.1 Notebook Summary

Notebook	Purpose	Imports From
`01_eda_visualization.ipynb`	Visualize EDA results	`src/eda_analysis.py`
`02_feature_exploration.ipynb`	Visualize features	`src/feature_engineering.py`
`03_model_results.ipynb`	Visualize model comparison	Loads saved results

9.2 Running Notebooks

```
# Start Jupyter from project directory
cd D:\Datathon\novartis_datathon_2025\Main_project
jupyter notebook notebooks/
```

Or open in VS Code with the Jupyter extension.

📊 Progress Tracking

Step	Description	Files	Status
1	Project Setup	`requirements.txt`	☐ Not Started
2	Core Modules	`src/*.py` (8 files)	☐ Not Started
3	EDA Analysis	`src/eda_analysis.py`, `notebooks/01_eda_visualization.ipynb`	☐ Not Started
4	Feature Engineering	Use `src/feature_engineering.py`	☐ Not Started
5	Model Training	`scripts/train_models.py`	☐ Not Started
6	Submissions	`scripts/generate_final_submissions.py`	☐ Not Started
7	Validation	`scripts/validate_submissions.py`	☐ Not Started
8	Run Pipeline	Command line execution	☐ Not Started
9	Visualization	`notebooks/*.ipynb` (optional)	☐ Not Started

🧠 Critical Reminders

Priority 1: Bucket 1 (HIGH EROSION)

- Bucket 1 is weighted **2×** in the final metric
- Always check Bucket 1 performance separately
- Consider training separate models for Bucket 1

Priority 2: Early Months

- **Scenario 1:** Months 0-5 carry **50%** weight
- **Scenario 2:** Months 6-11 carry **50%** weight
- Optimize for early month accuracy

Priority 3: Normalization

- All errors normalized by `Avg_j` (12-month pre-entry average)
- Handle cases where `Avg_j = 0` or `NaN`

Priority 4: No Negative Predictions

- Volume cannot be negative
- Always clip predictions to `>= 0`

Priority 5: Exact Submission Format

- Columns: `country`, `brand_name`, `months_postgx`, `volume`
- Scenario 1: months 0-23 for each country-brand (5,472 rows total)
- Scenario 2: months 6-23 for each country-brand (2,016 rows total)

Final Project Structure (Hybrid Approach)

```
Main_project/
├── src/                          # 🟢 CORE LOGIC (.py files)
│   ├── config.py                 # Configuration and constants
│   ├── data_loader.py            # Data loading and validation
│   ├── bucket_calculator.py      # Avg_j, normalized volume, buckets
│   ├── feature_engineering.py    # All feature creation
│   ├── eda_analysis.py           # EDA computations (not visualization)
│   ├── models.py                 # Baseline and gradient boosting
│   ├── evaluation.py             # PE metric computation
│   ├── submission.py             # Submission generation/validation
│   └── pipeline.py               # End-to-end pipeline
│
├── scripts/                      # 🟢 EXECUTION SCRIPTS (.py files)
│   ├── run_demo.py               # Quick demo of entire pipeline
│   ├── train_models.py           # Train and compare all models
│   ├── generate_final_submissions.py  # Create final submissions
│   └── validate_submissions.py   # Validate before upload
│
├── notebooks/                    # 🟢 VISUALIZATION ONLY (.ipynb)
│   ├── 01_eda_visualization.ipynb     # Visualize EDA results
│   ├── 02_feature_exploration.ipynb   # Visualize features
│   └── 03_model_results.ipynb         # Visualize model comparison
│
├── data/
│   ├── raw/                      # Input data files
│   │   ├── df_volume_train.csv
│   │   ├── df_volume_test.csv
│   │   ├── df_generics_train.csv
│   │   ├── df_generics_test.csv
│   │   ├── df_medicine_info_train.csv
│   │   └── df_medicine_info_test.csv
│   └── processed/                # Generated intermediate files
│       ├── aux_bucket_avgvol.csv
│       ├── eda_erosion_curves.csv
│       └── eda_bucket_chars.csv
│
├── models/                       # Saved trained models
```

```
│   ├── scenario1_lightgbm.joblib
│   ├── scenario2_lightgbm.joblib
│   ├── scenario1_xgboost.joblib
│   └── scenario2_xgboost.joblib
│
├── submissions/                    # Generated submission files
│   ├── scenario1_*_final.csv
│   └── scenario2_*_final.csv
│
├── reports/
│   ├── figures/                    # Saved visualizations
│   └── model_comparison_*.csv      # Model comparison results
│
├── Docs/                           # Documentation
├── SUBMISSION/                     # Official submission templates
├── requirements.txt                # Python dependencies
├── Todo.md                         # Quick task list
└── main_todo.md                    # This file (master guide)
```

Quick Start Commands

```powershell
# 1. Setup
cd D:\Datathon\novartis_datathon_2025\Main_project
& .\saeed_venv\Scripts\Activate.ps1
pip install -r requirements.txt

# 2. Quick Demo (test everything works)
python scripts/run_demo.py

# 3. Train Models
python scripts/train_models.py --scenario 1
python scripts/train_models.py --scenario 2

# 4. Generate Final Submissions
python scripts/generate_final_submissions.py

# 5. Validate
python scripts/validate_submissions.py
```

SUCCESS CRITERIA

Metric	Target
Scenario 1 PE	< 1.0 (good), < 0.5 (excellent)
Scenario 2 PE	< 0.8 (good), < 0.4 (excellent)

Metric	Target
Bucket 1 Accuracy	Better than Bucket 2
Submission Valid	☑ All checks pass

☑ **when complete.**