Complete LEI Data Enrichment Code Explanation

Import Statements and Dependencies

```python
import pandas as pd
import requests
import time
import logging
from typing import Dict, Any
import json
from pathlib import Path
```

Line-by-line breakdown:

- `pandas as pd`: Data manipulation library for handling structured data (DataFrames)
- `requests`: HTTP library for making API calls to external services
- `time`: Built-in module for time-related functions (sleep, delays)
- `logging`: Built-in module for structured logging and debugging
- `typing.Dict, Any`: Type hints for better code documentation and IDE support
- `json`: Built-in module for JSON serialization/deserialization
- `pathlib.Path`: Modern, object-oriented approach to file system paths

Custom Exception Definition

```python
class LEIEnrichmentError(Exception):
    """Custom exception for LEI enrichment errors"""
    pass
```

Purpose: Creates a specific exception type for this application. This follows Python best practices by:

- Making error handling more specific and meaningful
- Allowing different exception types to be caught and handled differently
- Improving debugging by clearly identifying the source of errors

Main Class Definition and Constructor

```python
class LEIDataEnricher:
    """
    Production-ready data enrichment service.

    Enriches transaction data by fetching legal entity information
    from the GLEIF API based on LEI codes.
    """
```

Class Purpose: Encapsulates all LEI enrichment functionality in a reusable, maintainable class.

```python
def __init__(self,
             base_url: str = "https://api.gleif.org/api/v1/lei-records",
             rate_limit_delay: float = 0.1,
             max_retries: int = 3,
             timeout: int = 30):
```

Constructor Parameters:

- `base_url`: GLEIF (Global Legal Entity Identifier Foundation) API endpoint
- `rate_limit_delay`: Prevents overwhelming the API (respectful API usage)
- `max_retries`: Resilience against temporary network failures
- `timeout`: Prevents hanging requests that could block the application

```python
self.base_url = base_url
self.rate_limit_delay = rate_limit_delay
self.max_retries = max_retries
self.timeout = timeout
self._lei_cache = {}
```

Instance Variables:

- Stores configuration parameters as instance variables
- `_lei_cache = {}`: Private dictionary (indicated by underscore) to cache API responses and reduce redundant calls

```python
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
self.logger = logging.getLogger(__name__)
```

Logging Setup:

- `basicConfig`: Configures the root logger
- `level=logging.INFO`: Only logs INFO level and above (INFO, WARNING, ERROR, CRITICAL)
- `format`: Timestamp, log level, and message for each log entry
- `getLogger(__name__)`: Creates a logger specific to this module

Core API Fetching Method

```python
def _fetch_lei_data(self, lei_code: str) -> Dict[str, Any]:
```

Method Signature:

- Private method (underscore prefix)
- Takes LEI code string, returns dictionary with lei data
- Type hints improve code documentation and catch type errors

```python
if lei_code in self._lei_cache:
    self.logger.debug(f"Using cached data for LEI: {lei_code}")
    return self._lei_cache[lei_code]
```

Caching Logic:

- Checks cache first to avoid redundant API calls
- Uses debug-level logging (won't appear with INFO level)
- Early return pattern for efficiency

```python
url = f"{self.base_url}?filter[lei]={lei_code}"
```

URL Construction:

- f-string formatting for clean string interpolation
- GLEIF API filter syntax to query specific LEI

```
python

for attempt in range(self.max_retries):
    try:
```

Retry Loop:

- Implements retry logic for resilience
- `range(self.max_retries)` creates attempts 0, 1, 2 (for max_retries=3)

```
python

self.logger.debug(f"Fetching LEI data for: {lei_code} (attempt {attempt + 1})")

response = requests.get(url, timeout=self.timeout)
response.raise_for_status()
```

HTTP Request:

- Debug logging shows which attempt we're on
- `requests.get()` makes HTTP GET request
- `timeout` prevents hanging requests
- `raise_for_status()` automatically raises exception for HTTP error codes (4xx, 5xx)

```
python

data = response.json()

if 'data' in data and len(data['data']) > 0:
    lei_record = data['data'][0]
    attributes = lei_record.get('attributes', {})
```

Response Processing:

- Converts JSON response to Python dictionary
- Defensive programming: checks if 'data' key exists and has content
- Uses `.get()` method with default empty dict to prevent KeyError

```python
legal_name = ''
entity = attributes.get('entity', {})
if entity and 'legalName' in entity:
    legal_name = entity['legalName'].get('name', '')
```

Data Extraction - Legal Name:

- Initializes empty string default
- Safely navigates nested JSON structure
- Multiple safety checks prevent crashes on malformed data

```python
bic = ''
bic_list = attributes.get('bic', [])
if isinstance(bic_list, list) and len(bic_list) > 0:
    bic = bic_list[0]
elif isinstance(bic_list, str):
    bic = bic_list
```

Data Extraction - BIC Code:

- BIC (Bank Identifier Code) extraction
- Handles both list and string formats from API
- Takes first BIC if multiple exist
- `isinstance()` checks object type for safe handling

```python
country = ''
if entity and 'legalAddress' in entity:
    country = entity['legalAddress'].get('country', '')

result = {
    'legalName': legal_name,
    'bic': bic,
    'country': country
}
```

Data Extraction - Country & Result Assembly:

- Extracts country from legal address
- Assembles clean result dictionary
- Consistent structure regardless of API response variations

```python
self._lei_cache[lei_code] = result

time.sleep(self.rate_limit_delay)

return result
```

Caching and Rate Limiting:

- Stores result in cache for future use
- `time.sleep()` implements respectful rate limiting
- Returns the processed result

```python
except requests.exceptions.RequestException as e:
    self.logger.warning(f"Request failed for LEI {lei_code} (attempt {attempt + 1}): {e}")
    if attempt < self.max_retries - 1:
        time.sleep(2 ** attempt)  # Exponential backoff
    else:
        raise LEIEnrichmentError(f"Failed to fetch data for LEI {lei_code} after {self.max_retries} attempts: {e}")
```

Error Handling - Network Errors:

- Catches all requests-related exceptions
- Logs warning with attempt number
- **Exponential backoff**: `2 ** attempt` creates delays of 1s, 2s, 4s...
- Only raises custom exception after all retries exhausted
- Wraps original exception in custom one for better error context

```python
except (KeyError, json.JSONDecodeError) as e:
    self.logger.error(f"Error parsing response for LEI {lei_code}: {e}")
    result = {'legalName': '', 'bic': '', 'country': ''}
    self._lei_cache[lei_code] = result
    return result
```

Error Handling - Data Parsing Errors:

- Catches JSON parsing errors and missing key errors
- Logs as ERROR level (more serious than WARNING)
- Returns empty result instead of crashing
- Still caches empty result to avoid repeated failures

Main Dataset Enrichment Method

```python
def enrich_dataset(self, input_data: pd.DataFrame) -> pd.DataFrame:
```

Public Method:

- Main interface for enriching datasets
- Takes and returns pandas DataFrame
- Clear type hints for API contract

```python
if 'lei' not in input_data.columns:
    raise LEIEnrichmentError("Input data must contain 'lei' column")
```

Input Validation:

- Validates required column exists
- Fails fast with clear error message
- Prevents processing invalid data

```python
self.logger.info(f"Starting enrichment for {len(input_data)} records")

enriched_data = input_data.copy()
```

Processing Setup:

- Logs processing start with record count
- Creates copy to avoid modifying original data (immutability principle)

```python
unique_leis = input_data['lei'].unique()
self.logger.info(f"Found {len(unique_leis)} unique LEI codes")
```

Optimization:

- Gets unique LEI codes to minimize API calls
- 1000 records with 10 unique LEIs = only 10 API calls instead of 1000

```python
lei_info = {}
for i, lei_code in enumerate(unique_leis, 1):
    self.logger.info(f"Processing LEI {i}/{len(unique_leis)}: {lei_code}")
    try:
        lei_info[lei_code] = self._fetch_lei_data(lei_code)
    except LEIEnrichmentError as e:
        self.logger.error(f"Failed to enrich LEI {lei_code}: {e}")
        lei_info[lei_code] = {'legalName': '', 'bic': '', 'country': ''}
```

Batch Processing:

- `enumerate(unique_leis, 1)` provides index starting from 1 for user-friendly logging
- Progress logging shows current position
- Graceful error handling: continues processing even if individual LEIs fail
- Stores empty result for failed LEIs instead of crashing entire process

```python
enriched_data['legalName'] = enriched_data['lei'].map(lambda x: lei_info.get(x, {}).get('legalName', ''))
enriched_data['bic'] = enriched_data['lei'].map(lambda x: lei_info.get(x, {}).get('bic', ''))
enriched_data['country'] = enriched_data['lei'].map(lambda x: lei_info.get(x, {}).get('country', ''))
```

Data Mapping:

- `map()` applies function to each element in the column
- `lambda` creates anonymous function for data lookup
- `.get(x, {}).get('field', '')` provides safe nested lookup with defaults
- Creates new columns in DataFrame with enriched data

```python
self.logger.info("Calculating transaction costs based on country-specific logic")
enriched_data['transaction_costs'] = enriched_data.apply(self._calculate_transaction_costs, axis=1)

enriched_data = enriched_data.drop('country', axis=1)
```

Business Logic Application:

- `apply()` with `axis=1` applies function to each row
- Calculates transaction costs using business rules
- Drops temporary country column as it's not needed in final output

Transaction Cost Calculation Method

```python
def _calculate_transaction_costs(self, row) -> float:
```

Business Logic Method:

- Private method for internal calculation
- Takes DataFrame row, returns calculated cost

```python
try:
    country = row.get('country', '').upper()
    notional = float(row.get('notional', 0))
    rate = float(row.get('rate', 0))
```

Data Preparation:

- `.upper()` normalizes country codes to uppercase
- `float()` converts strings to numbers for calculation
- `.get()` with defaults prevents KeyError exceptions

```python
if country == 'GB':
    transaction_costs = notional * rate - notional
elif country == 'NL':
    if rate != 0:
        transaction_costs = abs(notional * (1/rate) - notional)
    else:
        self.logger.warning(f"Zero rate encountered for NL calculation, setting cost to 0")
        transaction_costs = 0.0
else:
    self.logger.info(f"No specific calculation rule for country '{country}', setting cost to 0")
    transaction_costs = 0.0
```

Business Rules Implementation:

- **GB (Great Britain)**: `notional * rate - notional` (interest calculation)
- **NL (Netherlands)**: `abs(notional * (1/rate) - notional)` (inverse rate with absolute value)
- **Division by zero protection** for NL calculation
- **Default case** for unknown countries
- Detailed logging for debugging and audit trail

```python
except (ValueError, TypeError) as e:
    self.logger.error(f"Error calculating transaction costs: {e}")
    return 0.0
```

Error Handling:

- Catches conversion errors (invalid numbers)
- Logs specific error for debugging
- Returns safe default value instead of crashing

Caching Methods

```python
def save_cache(self, cache_file: str = "lei_cache.json"):
    """Save the LEI cache to a file for future use."""
    try:
        with open(cache_file, 'w') as f:
            json.dump(self._lei_cache, f, indent=2)
        self.logger.info(f"Cache saved to {cache_file}")
    except Exception as e:
        self.logger.error(f"Failed to save cache: {e}")
```

Cache Persistence:

- Saves in-memory cache to JSON file
- `indent=2` makes JSON human-readable
- `with open()` ensures file is properly closed
- Generic exception handling for any file I/O errors

```python
def load_cache(self, cache_file: str = "lei_cache.json"):
    """Load LEI cache from a file."""
    try:
        if Path(cache_file).exists():
            with open(cache_file, 'r') as f:
                self._lei_cache = json.load(f)
            self.logger.info(f"Cache loaded from {cache_file}")
    except Exception as e:
        self.logger.error(f"Failed to load cache: {e}")
```

Cache Loading:

- Checks if file exists before attempting to load
- Uses modern `Path.exists()` instead of `os.path.exists()`
- Graceful failure: continues working even if cache loading fails

Main Function

```python
def main():
    """
    Main function to demonstrate the enrichment process.
    """
    try:
        enricher = LEIDataEnricher()
        enricher.load_cache()
```

Initialization:

- Creates enricher instance with default parameters
- Loads existing cache if available

```python
df = pd.read_csv("sample_input.csv")

print("Original data shape:", df.shape)
print("\nFirst few rows of original data:")
print(df.head())
```

Data Loading and Inspection:

- Loads CSV file into pandas DataFrame
- `df.shape` shows (rows, columns) for data overview
- `df.head()` shows first 5 rows for manual inspection

```python
enriched_df = enricher.enrich_dataset(df)

print("\nEnriched data shape:", enriched_df.shape)
print("\nFirst few rows of enriched data:")
print(enriched_df.head())
```

Data Processing:

- Calls main enrichment method
- Shows before/after comparison of data shape and content

```python
output_file = 'output.csv'
enriched_df.to_csv(output_file, index=False)
print(f"\nEnriched data saved to: {output_file}")


enricher.save_cache()
```

Output and Persistence:

- `index=False` prevents pandas from writing row numbers to CSV
- Saves processed data for future use
- Persists cache for faster future runs

```python
print(f"\nEnrichment Summary:")
print(f"- Total records processed: {len(enriched_df)}")
print(f"- Unique LEIs processed: {len(df['lei'].unique())}")
print(f"- Records with legal names: {len(enriched_df[enriched_df['legalName'] != ''])}")
print(f"- Records with BIC codes: {len(enriched_df[enriched_df['bic'] != ''])}")
print(f"- Records with transaction costs calculated: {len(enriched_df[enriched_df['transaction_costs'] != 0])}")
```

Success Metrics:

- Provides comprehensive processing statistics
- Boolean indexing `enriched_df[enriched_df['legalName'] != '']` counts non-empty values
- Helps validate enrichment success rate

```python
print(f"\nSample Transaction Costs:")
cost_sample = enriched_df[['lei', 'notional', 'rate', 'transaction_costs']].head()
print(cost_sample.to_string(index=False))
```

Sample Output:

- Shows subset of columns for focused analysis
- `.to_string(index=False)` formats output nicely without row indices

```python
except Exception as e:
    logging.error(f"Error in enrichment process: {e}")
```

Top-Level Error Handling:

- Catches any unhandled exceptions in main process
- Uses module-level logging for critical errors

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

Script Execution:

- Python idiom for script vs. module usage
- Only runs main() when file is executed directly, not when imported

Key Design Patterns and Best Practices

1. **Error Handling**: Multiple layers with specific exception types
2. **Caching**: Reduces API calls and improves performance
3. **Rate Limiting**: Respectful API usage
4. **Retry Logic**: Resilience against network issues
5. **Logging**: Comprehensive tracking and debugging
6. **Type Hints**: Better code documentation and IDE support
7. **Defensive Programming**: Safe data access with defaults
8. **Single Responsibility**: Each method has one clear purpose
9. **Immutability**: Creates copies instead of modifying original data
10. **Configuration**: Parameterized for different environments