INNOFarms - Comprehensive Code Documentation

Table of Contents

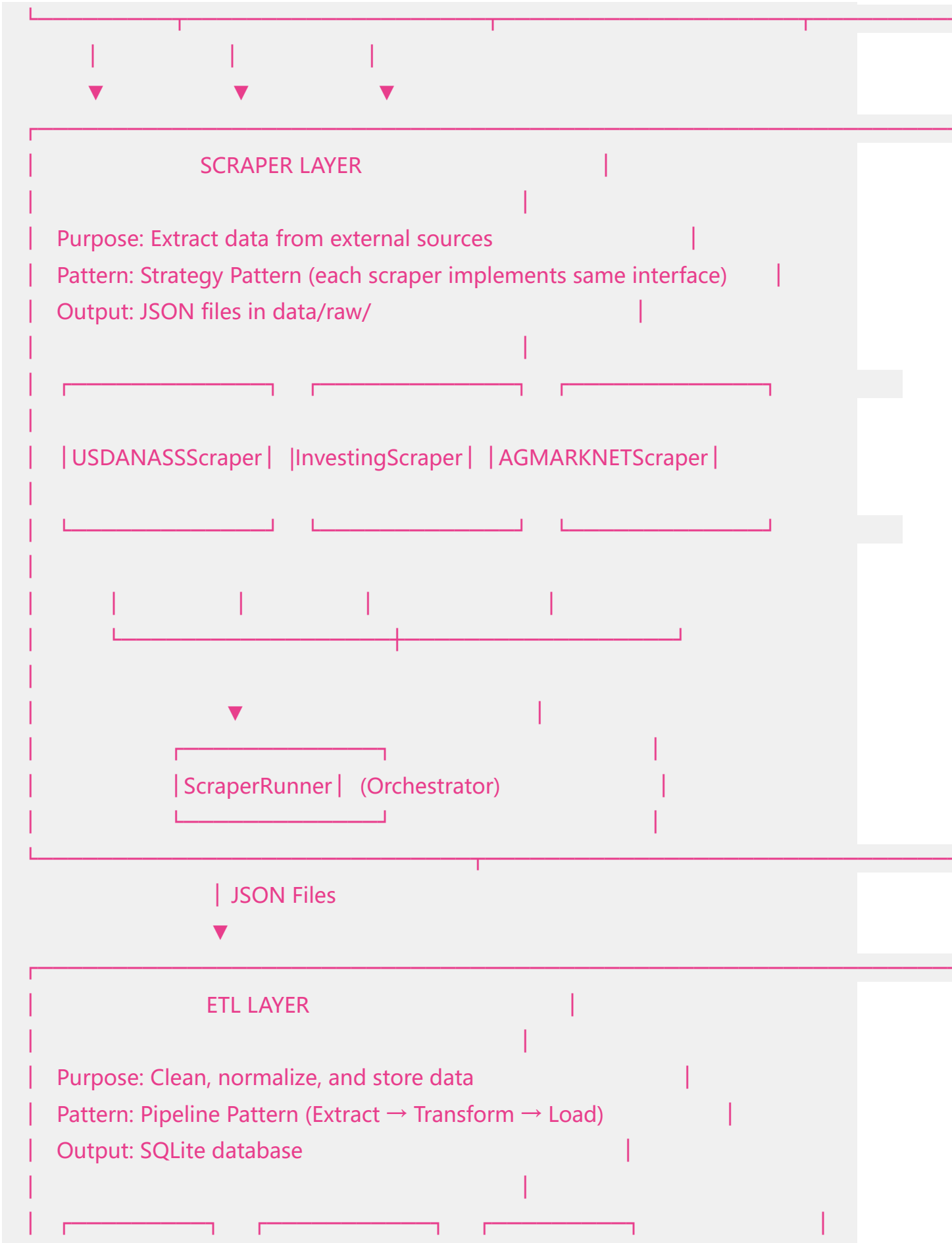
---

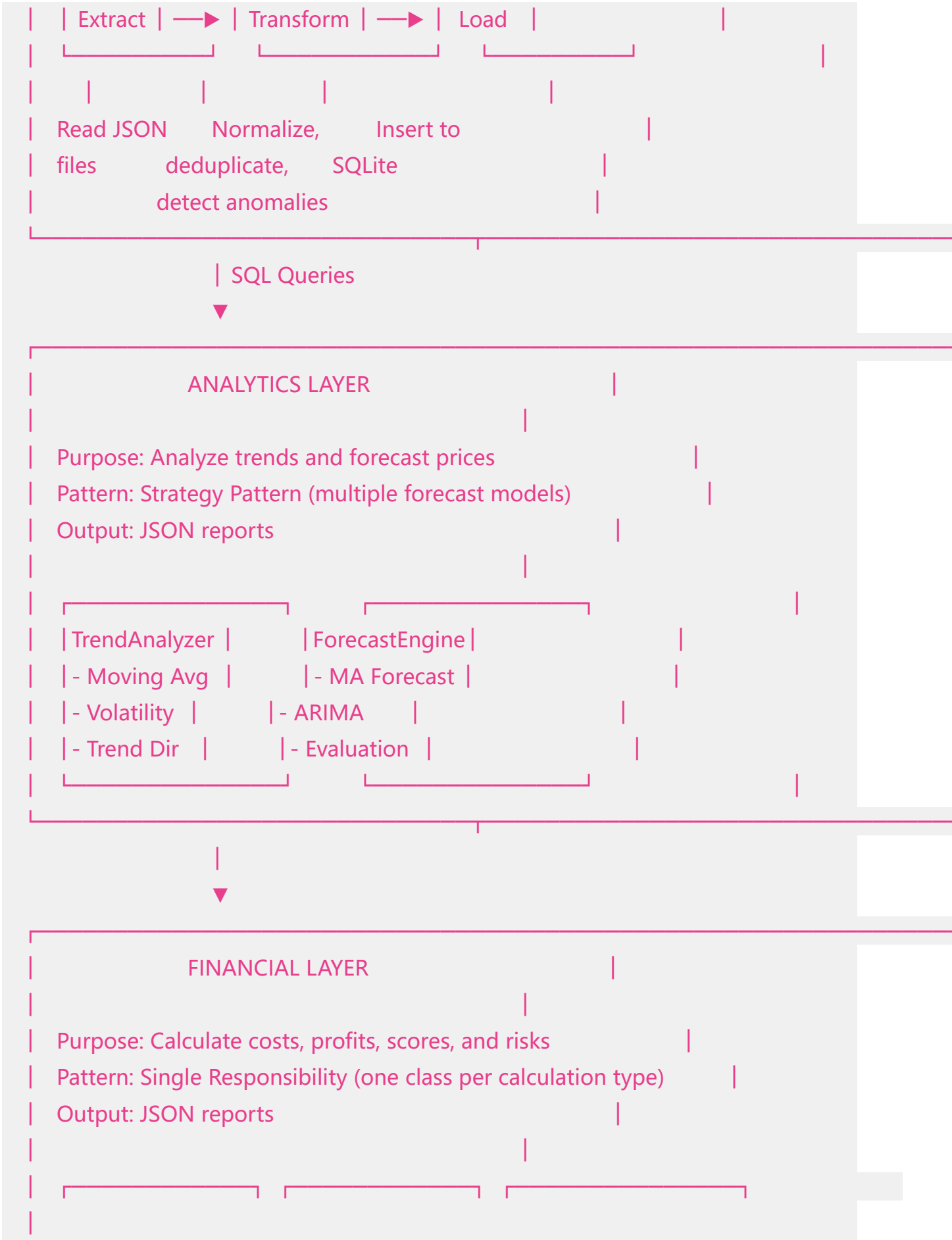
1. High-Level Design (HLD)

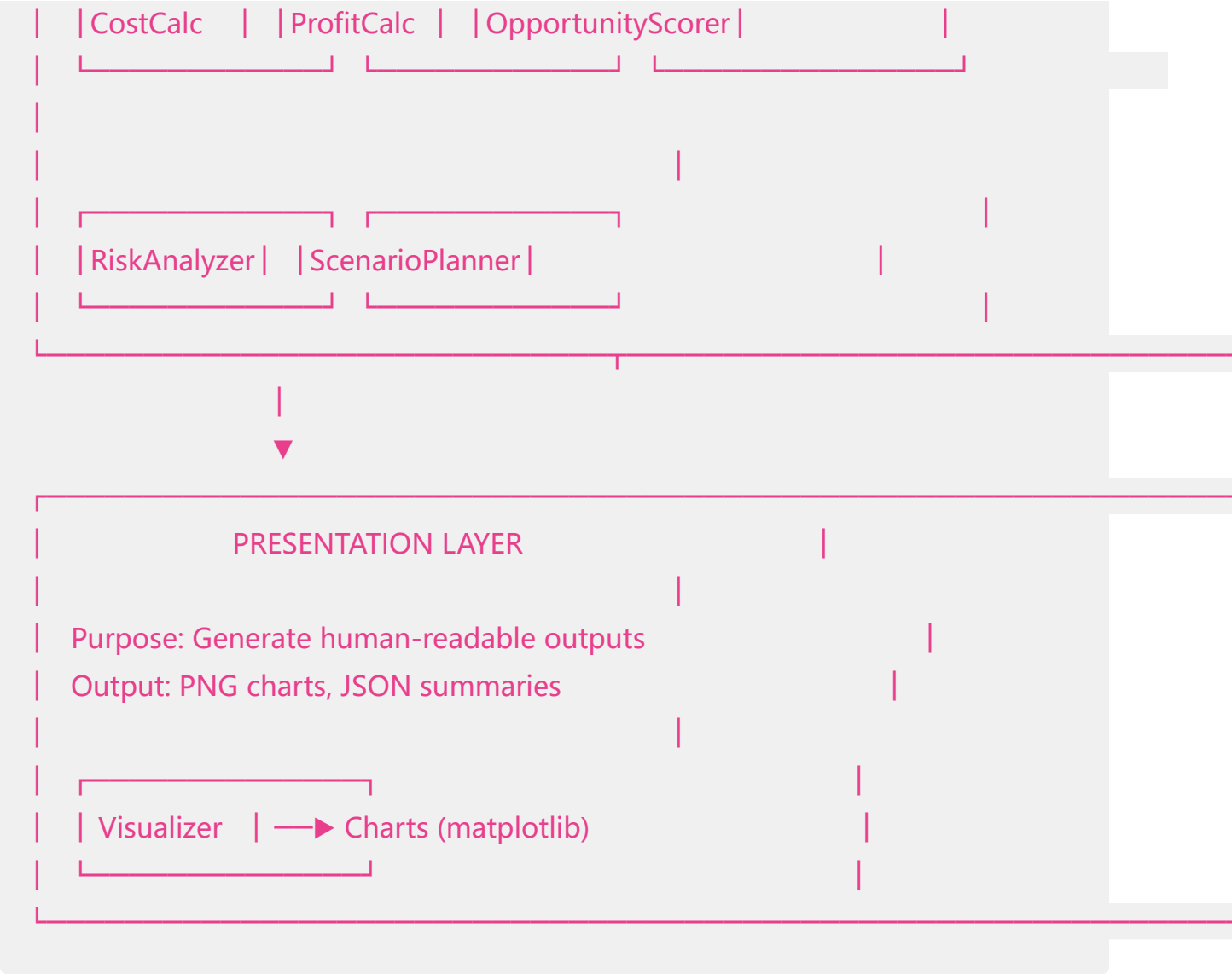
1.1 System Overview

The INNOFarms system is designed as a **data pipeline architecture** that follows the Extract-Transform-Load (ETL) pattern with additional analysis layers.

Data Flow Diagram


SCRAPER LAYER
Purpose: Extract data from external sources
Pattern: Strategy Pattern (each scraper implements same interface)
Output: JSON files in data/raw/
USDANASSScraper
InvestingScraper
AGMARKNETScraper
ScraperRunner
(Orchestrator)
JSON Files
ETL LAYER
Purpose: Clean, normalize, and store data
Pattern: Pipeline Pattern (Extract → Transform → Load)
Output: SQLite database


Extract
Transform
Load
Read JSON files
Normalize, deduplicate, detect anomalies
Insert to SQLite
SQL Queries
ANALYTICS LAYER
Purpose: Analyze trends and forecast prices
Pattern: Strategy Pattern (multiple forecast models)
Output: JSON reports
TrendAnalyzer
- Moving Avg
- Volatility
- Trend Dir
ForecastEngine
- MA Forecast
- ARIMA
- Evaluation
FINANCIAL LAYER
Purpose: Calculate costs, profits, scores, and risks
Pattern: Single Responsibility (one class per calculation type)
Output: JSON reports



1.2 Design Principles

1.2.1 Separation of Concerns

Each module handles one specific responsibility:

- Scrapers: Data collection only
- ETL: Data cleaning only
- Analytics: Analysis only
- Financial: Business logic only

1.2.2 Dependency Injection

- Configuration is injected via `.env` and YAML files

- Database path is configurable
- Logging is centralized

1.2.3 Modularity

- New data sources: Create new scraper class
- New commodities: Edit YAML config
- New analysis: Add new method to analyzer

1.2.4 Error Handling

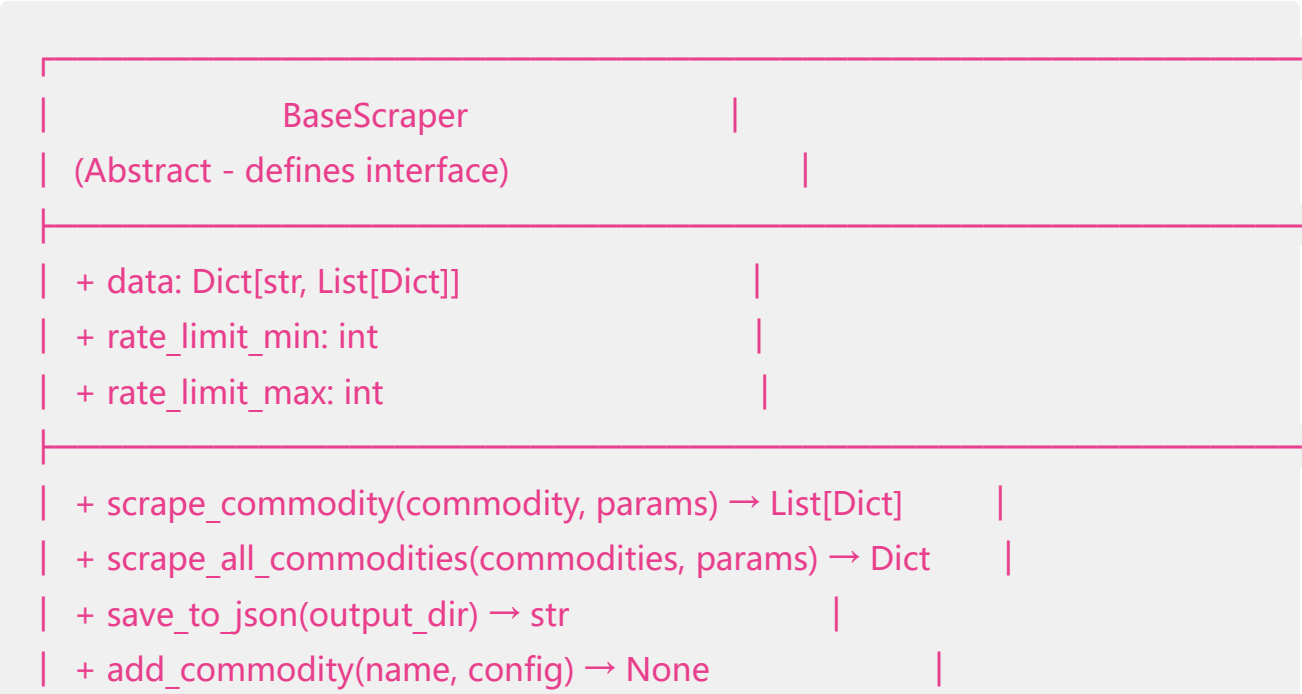
- Each operation logs errors
- Failed records don't crash pipeline
- Graceful degradation with sample data

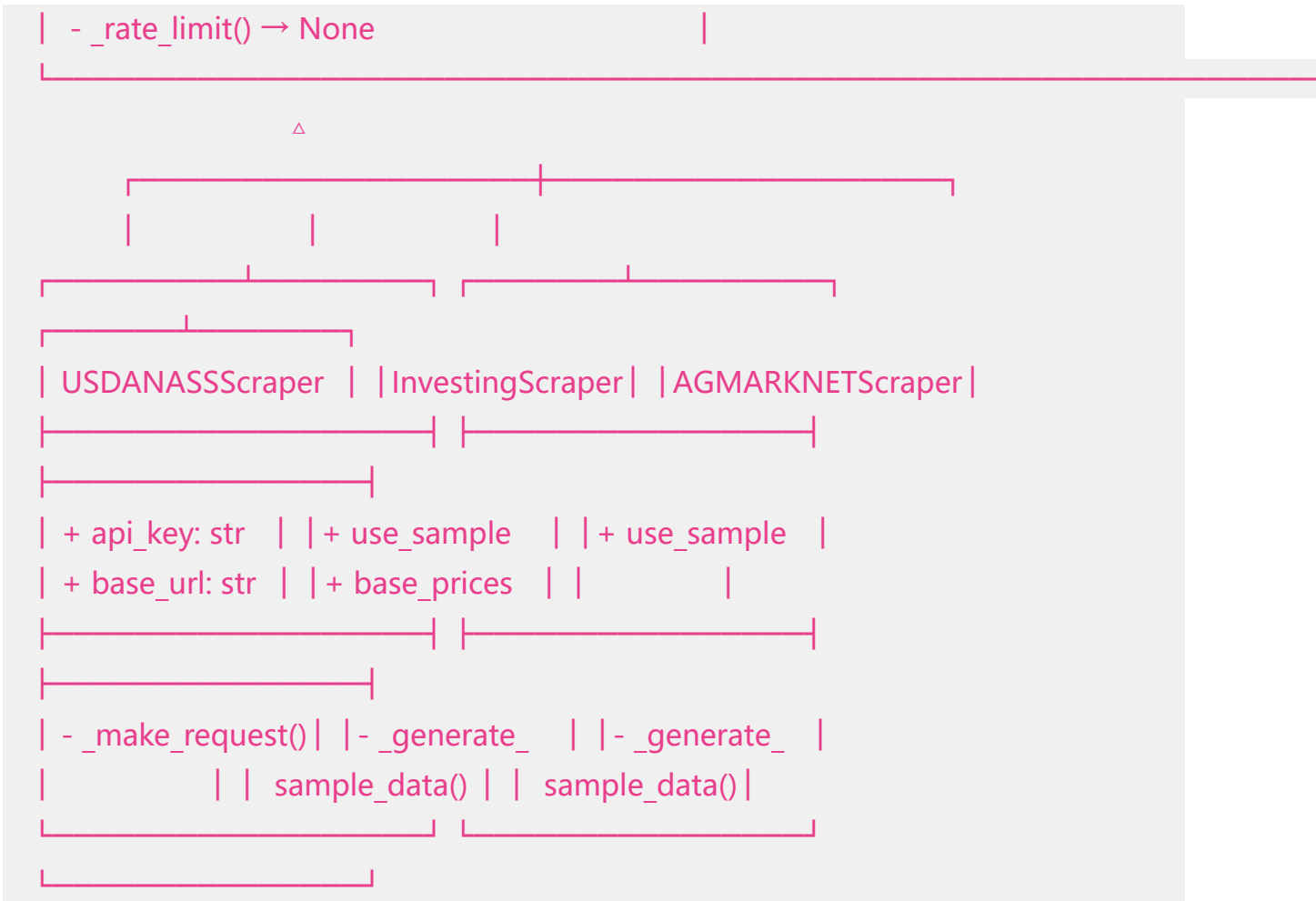
---

2. Low-Level Design (LLD)

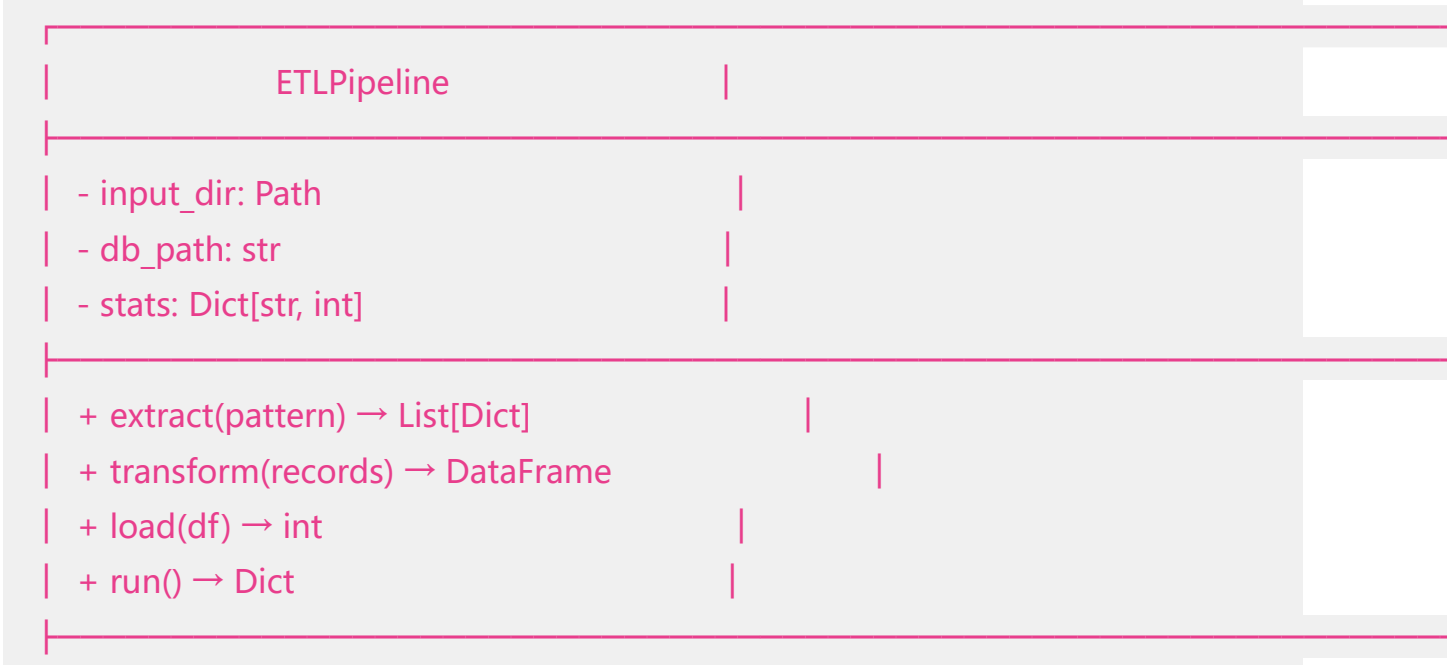
2.1 Class Diagrams

2.1.1 Scraper Module





2.1.2 ETL Pipeline

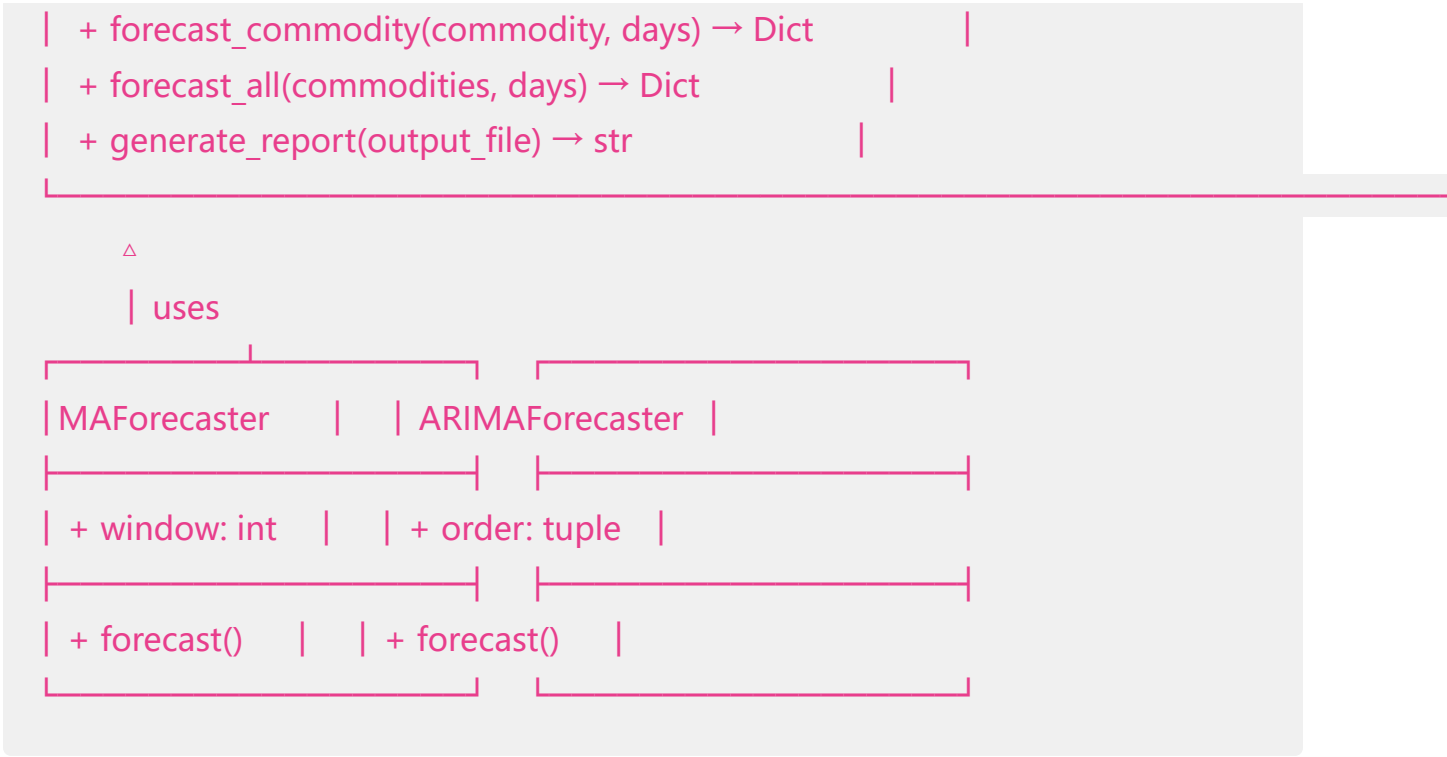


```
| - _create_schema(conn)                                    |
| - _get_commodity_map(conn, commodities) → Dict            |
| - _get_source_map(conn, sources) → Dict                   |
| - _guess_category(commodity) → str                        |
| - _create_aggregates(conn)                                |
└──────────────────────────────────────────────────────────┘
```

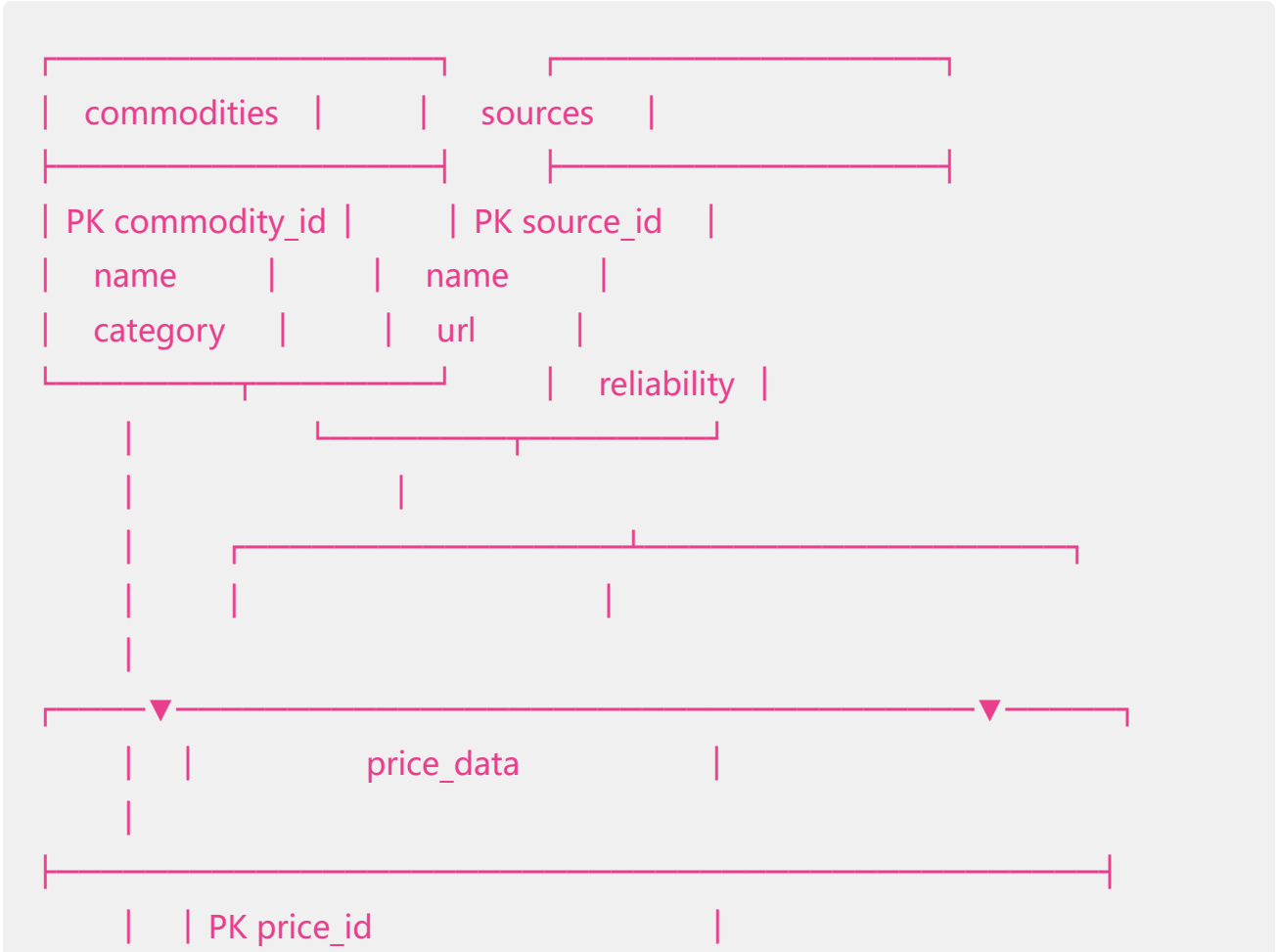
2.1.3 Analytics Module

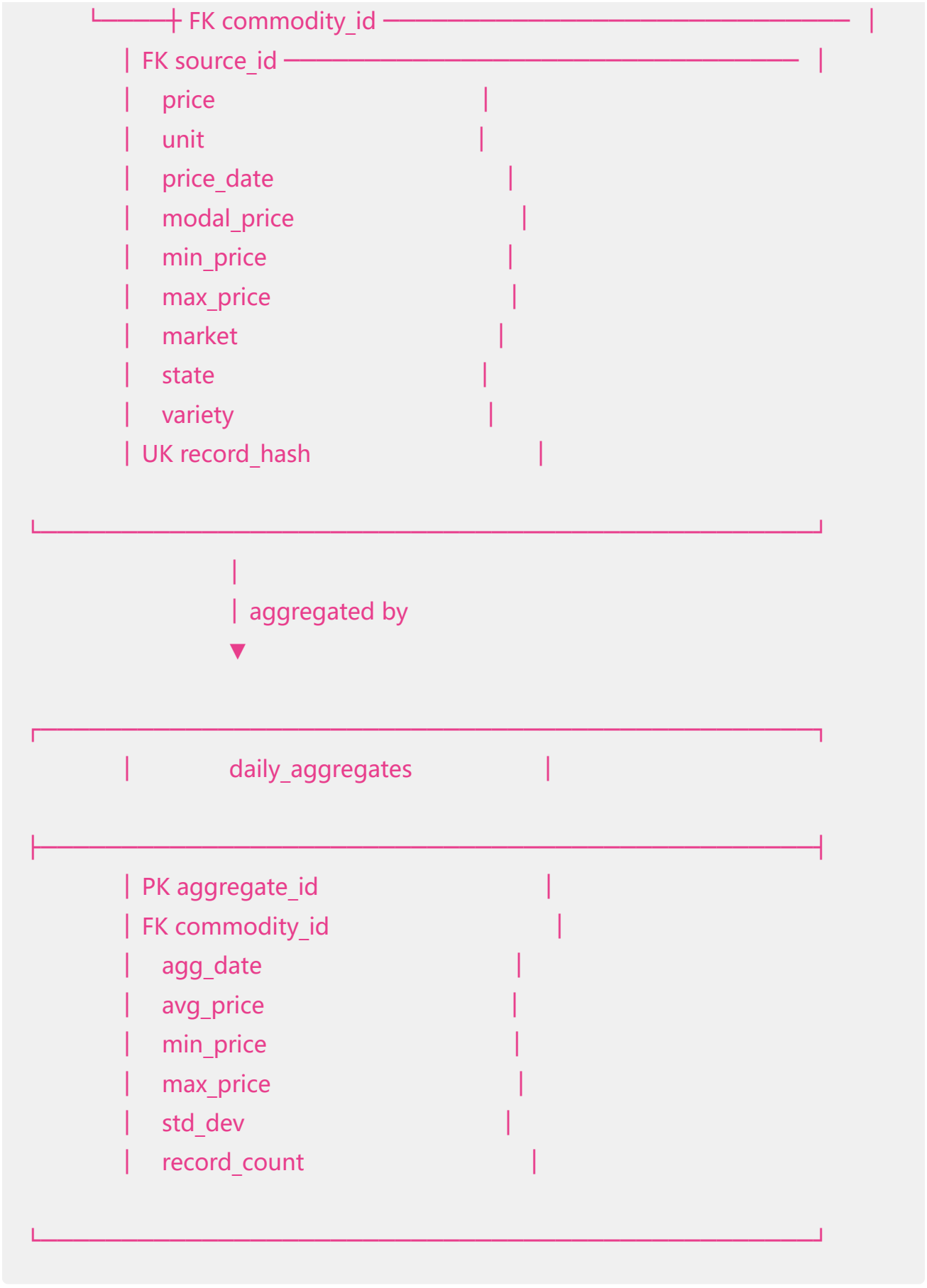
```
┌──────────────────────────────────────────────────────────┐
|                    TrendAnalyzer                         |
├──────────────────────────────────────────────────────────┤
| - db_path: str                                           |
| - results: Dict[str, Any]                                |
├──────────────────────────────────────────────────────────┤
| + get_price_series(commodity, days) → DataFrame          |
| + calculate_moving_averages(df, windows) → Dict          |
| + detect_trend(df, short, long) → Dict                   |
| + calculate_volatility(df) → Dict                        |
| + analyze_commodity(commodity, days) → Dict              |
| + analyze_all(commodities, days) → Dict                  |
| + generate_report(output_file) → str                     |
└──────────────────────────────────────────────────────────┘


┌──────────────────────────────────────────────────────────┐
|                    ForecastEngine                        |
├──────────────────────────────────────────────────────────┤
| - db_path: str                                           |
| - forecasters: Dict[str, Forecaster]                     |
| - results: Dict[str, Any]                                |
├──────────────────────────────────────────────────────────┤
| + get_price_series(commodity, days) → Series             |
| + evaluate_model(series, forecaster) → Dict              |
```



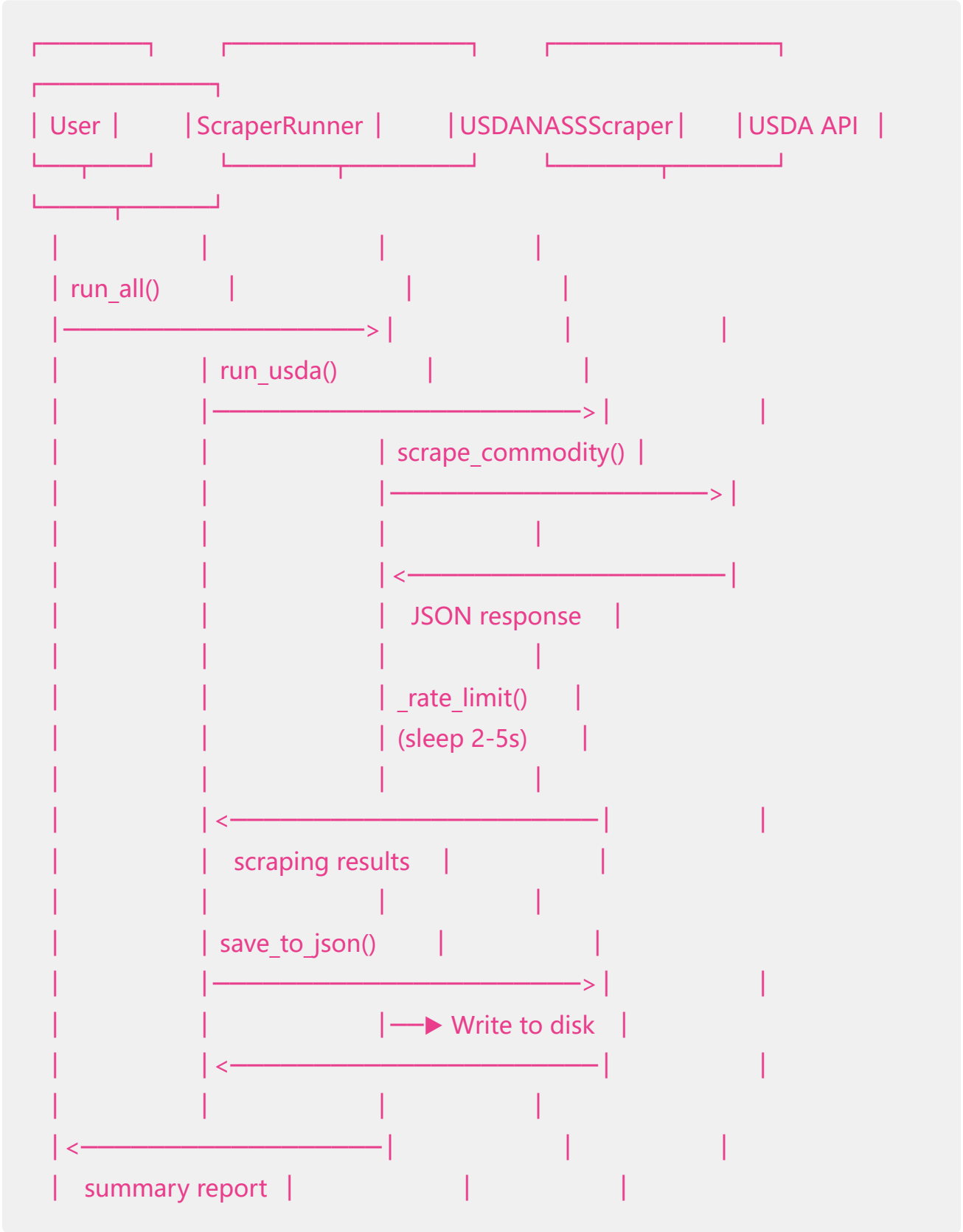
2.2 Database Schema (ER Diagram)



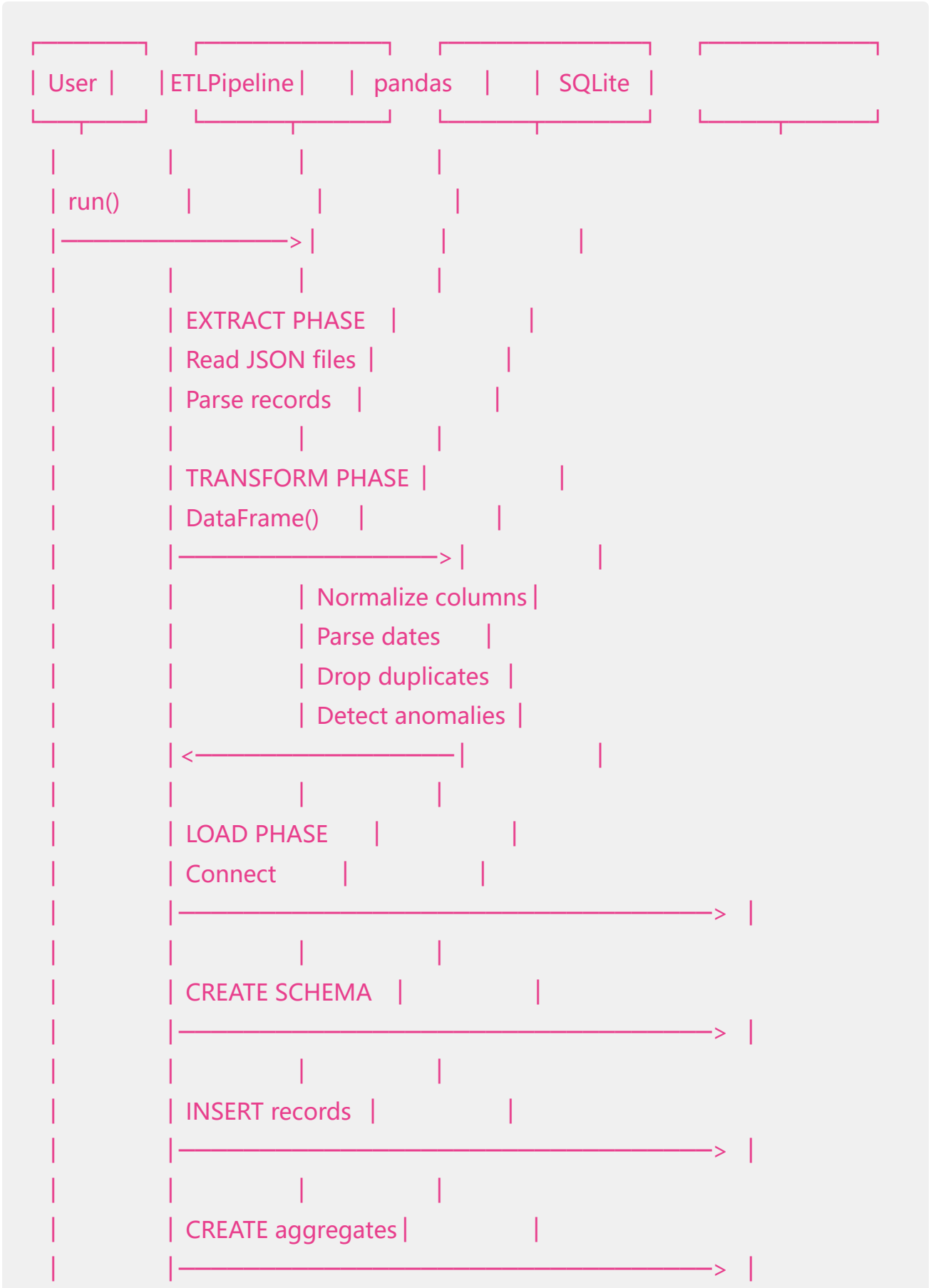

FK commodity_id
FK source_id
price
unit
price_date
modal_price
min_price
max_price
market
state
variety
UK record_hash
aggregated by
daily_aggregates
PK aggregate_id
FK commodity_id
agg_date
avg_price
min_price
max_price
std_dev
record_count

2.3 Sequence Diagrams

2.3.1 Data Scraping Sequence



2.3.2 ETL Pipeline Sequence



---

3. Module Documentation

3.1 Utils Module

3.1.1 logger.py - Centralized Logging

```
"""
Centralized Logger Configuration
================================

Provides a unified logging configuration for all modules.
Import this module to get a configured logger.

Usage:
    from src.utils.logger import get_logger
    logger = get_logger(__name__)
    logger.info("Message here")
"""

import os
import logging
from datetime import datetime
from pathlib import Path
from typing import Optional
```

Line-by-line explanation:

1. `"""..."""` - Module docstring explaining purpose and usage
2. `import os` - Operating system interface for file paths
3. `import logging` - Python's built-in logging module
4. `from datetime import datetime` - For timestamp generation
5. `from pathlib import Path` - Modern path handling
6. `from typing import Optional` - Type hints for optional parameters

```
# ============================================================
# CONFIGURATION
# ============================================================
LOG_DIR = Path(__file__).parent.parent.parent / "logs"
LOG_FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
DATE_FORMAT = "%Y-%m-%d %H:%M:%S"
DEFAULT_LEVEL = logging.INFO
```

Why these constants?

- `LOG_DIR`: Places logs in project root's `logs/` folder
    - `Path(__file__)` = current file path
    - `.parent.parent.parent` = go up 3 directories (utils → src → project root)
- `LOG_FORMAT`: Standard logging format showing:
    - `%(asctime)s` - Timestamp
    - `%(name)s` - Logger name (usually module name)
    - `%(levelname)s` - DEBUG/INFO/WARNING/ERROR
    - `%(message)s` - Actual log message
- `DATE_FORMAT`: ISO-like format for readability
- `DEFAULT_LEVEL`: INFO level shows important events, not debug noise

```
def get_logger(
    name: str,
    level: int = DEFAULT_LEVEL,
    log_to_file: bool = True,
```

```
    log_file: Optional[str] = None
) -> logging.Logger:
    """
    Get a configured logger instance.

    Args:
        name: Logger name (typically __name__)
        level: Logging level (default: INFO)
        log_to_file: Whether to also log to file
        log_file: Custom log file path (optional)

    Returns:
        Configured logger instance
    """
```

Function signature explanation:

- `name: str` - Logger identifier, use `__name__` to get module name automatically
- `level: int` - Control verbosity (DEBUG=10, INFO=20, WARNING=30, ERROR=40)
- `log_to_file: bool` - Enable file logging for debugging
- `log_file: Optional[str]` - Custom path, or auto-generate from name
- `-> logging.Logger` - Returns configured logger object

```
    logger = logging.getLogger(name)

    # Only configure if not already configured
    if logger.handlers:
        return logger
```

Why check handlers?

- `getLogger(name)` returns same object for same name (singleton pattern)
- If already configured (has handlers), don't add duplicate handlers
- Prevents duplicate log messages when importing module multiple times

```
    logger.setLevel(level)

    # Console handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(level)
    console_handler.setFormatter(logging.Formatter(LOG_FORMAT, DATE_FORMA
        T))
    logger.addHandler(console_handler)
```

Console handler setup:

1. `StreamHandler()` - Outputs to stdout (terminal)
2. `setLevel()` - Filter messages below this level
3. `setFormatter()` - Apply our format string
4. `addHandler()` - Attach to logger

```
    # File handler (optional)
    if log_to_file:
        LOG_DIR.mkdir(parents=True, exist_ok=True)

        if log_file is None:
            # Use module name as log file
            safe_name = name.replace(".", "_").replace("/", "_")
            log_file = LOG_DIR / f"{safe_name}_{datetime.now().strftim
                e('%Y%m%d')}.log"
        else:
            log_file = Path(log_file)

        file_handler = logging.FileHandler(log_file)
        file_handler.setLevel(level)
        file_handler.setFormatter(logging.Formatter(LOG_FORMAT, DATE_FORMAT))
        logger.addHandler(file_handler)

    return logger
```

File handler setup:

1. `mkdir(parents=True, exist_ok=True)` - Create logs directory if missing
2. `safe_name` - Replace dots and slashes with underscores for valid filename
3. `datetime.now().strftime('%Y%m%d')` - Daily log rotation by date
4. `FileHandler` - Writes to disk for future debugging

3.1.2 config.py - Environment Configuration

```
"""
Configuration loader with environment variable support.
"""

import os
from pathlib import Path
from typing import Dict, Any, Optional

# Try to load python-dotenv
try:
    from dotenv import load_dotenv
    load_dotenv()
except ImportError:
    pass
```

Why try/except for dotenv?

- `python-dotenv` loads `.env` file into environment variables
- If not installed, fail silently (doesn't crash the app)
- `load_dotenv()` reads `.env` file and sets `os.environ` values

```
def get_env(key: str, default: Optional[str] = None) -> Optional[str]:
    """Get environment variable with optional default."""
    return os.environ.get(key, default)
```

```python
def get_env_int(key: str, default: int = 0) -> int:
    """Get environment variable as integer."""
    value = os.environ.get(key)
    return int(value) if value else default


def get_env_float(key: str, default: float = 0.0) -> float:
    """Get environment variable as float."""
    value = os.environ.get(key)
    return float(value) if value else default


def get_env_bool(key: str, default: bool = False) -> bool:
    """Get environment variable as boolean."""
    value = os.environ.get(key, "").lower()
    if value in ("true", "1", "yes"):
        return True
    if value in ("false", "0", "no"):
        return False
    return default
```

Helper functions explained:

- `get_env()` - Basic string retrieval with default
- `get_env_int()` - Parse string to int (e.g., “5” → 5)
- `get_env_float()` - Parse string to float (e.g., “3.14” → 3.14)
- `get_env_bool()` - Handle various boolean representations

Why these helpers?

- Environment variables are always strings
- Type conversion with error handling
- Consistent default value pattern

```python
class Config:
    """Application configuration from environment variables."""

    # API Keys
    USDA_API_KEY: str = get_env("USDA_API_KEY", "")

    # Database
    DATABASE_PATH: str = get_env("DATABASE_PATH", "database/market_data.d
            b")

    # Logging
    LOG_LEVEL: str = get_env("LOG_LEVEL", "INFO")

    # Scraper settings
    RATE_LIMIT_MIN: int = get_env_int("SCRAPER_RATE_LIMIT_MIN", 2)
    RATE_LIMIT_MAX: int = get_env_int("SCRAPER_RATE_LIMIT_MAX", 5)
    MAX_RETRIES: int = get_env_int("SCRAPER_MAX_RETRIES", 3)

    # Paths
    PROJECT_ROOT: Path = Path(__file__).parent.parent.parent
    DATA_DIR: Path = PROJECT_ROOT / "data" / "raw"
    REPORTS_DIR: Path = PROJECT_ROOT / "reports"
    CONFIG_DIR: Path = PROJECT_ROOT / "config"


config = Config()
```

Why a Config class?

- **Single source of truth** - All settings in one place
- **Type hints** - IDE autocomplete and error checking
- **Default values** - Works without .env file
- **Singleton pattern** - `config = Config()` creates single instance

3.2 Scrapers Module

3.2.1 usda_scraper.py - USDA NASS API Scraper

```
# ============================================================
# CONSTANTS
# ============================================================
USDA_API_URL = "https://quickstats.nass.usda.gov/api/api_GET/"

# Commodity configurations - easily extendable
COMMODITIES = {
    "WHEAT": {
        "commodity_desc": "WHEAT",
        "statisticcat_desc": "PRICE RECEIVED",
        "unit_desc": "$ / BU"
    },
    "CORN": {
        "commodity_desc": "CORN",
        "statisticcat_desc": "PRICE RECEIVED",
        "unit_desc": "$ / BU"
    },
    # ... more commodities
}
```

Constants design:

- `USDA_API_URL` - API endpoint (could move to config)
- `COMMODITIES` - Dictionary for each commodity's API parameters
    - `commodity_desc` - USDA commodity name
    - `statisticcat_desc` - What statistic to fetch (price, production, etc.)
    - `unit_desc` - Expected unit for price

Why dictionary format?

- Easy to add new commodities

- Self-documenting (field names explain purpose)
- Can be overridden per-call

```python
class USDANASSScraper:
    """
    Scraper for USDA NASS QuickStats API.
    """

    def __init__(self, api_key: str = None):
        """
        Initialize the USDA scraper.

        Args:
            api_key: USDA API key (defaults to env variable)
        """
        self.api_key = api_key or config.USDA_API_KEY
        self.base_url = USDA_API_URL
        self.data: Dict[str, List[Dict]] = {}
        self.rate_limit_min = config.RATE_LIMIT_MIN
        self.rate_limit_max = config.RATE_LIMIT_MAX

        if not self.api_key:
            logger.warning("USDA API key not set")
```

Constructor explained:

- `api_key or config.USDA_API_KEY` - Use provided key OR fall back to env
- `self.data` - Stores scraped data (key: commodity name, value: records list)
- Rate limits from config - respect API terms of service
- Warning if no API key - helps debug issues

```python
    def _make_request(self, params: Dict) -> Optional[Dict]:
        """
        Make API request with error handling.
```

```
        """
        params['key'] = self.api_key
        params['format'] = 'JSON'

        try:
            response = requests.get(
                self.base_url,
                params=params,
                timeout=30
            )
            response.raise_for_status()
            return response.json()

        except requests.exceptions.RequestException as e:
            logger.error(f"API request failed: {e}")
            return None
```

HTTP request pattern:

1. Add authentication (`key`) and format to params
2. `requests.get()` - Make HTTP GET request
3. `timeout=30` - Don't hang forever
4. `raise_for_status()` - Raise exception for 4xx/5xx errors
5. Return `None` on error (caller handles gracefully)

```
    def _rate_limit(self):
        """Apply rate limiting delay."""
        delay = random.uniform(self.rate_limit_min, self.rate_limit_max)
        time.sleep(delay)
```

Why rate limiting?

- Prevents overwhelming the API server
- Avoids getting IP blocked
- `random.uniform()` - Random delay looks more human-like
- Typically 2-5 seconds between requests

```
def scrape_commodity(
    self,
    commodity: str,
    years: int = 10,
    commodity_config: Dict = None
) -> List[Dict]:
    """
    Scrape data for a single commodity.
    """
    if commodity_config is None:
        commodity_config = COMMODITIES.get(commodity.upper(), {})

    if not commodity_config:
        logger.warning(f"No configuration for commodity: {commodity}")
        return []

    current_year = datetime.now().year
    start_year = current_year - years

    params = {
        "source_desc": "SURVEY",
        "sector_desc": "CROPS",
        "commodity_desc": commodity_config.get("commodity_desc", commodit
            y),
        "statisticcat_desc": commodity_config.get("statisticcat_desc", "PRICE REC
            EIVED"),
        "year__GE": str(start_year),
        "year__LE": str(current_year),
        "freq_desc": "ANNUAL"
    }

    logger.info(f"Fetching {commodity} data ({start_year}-{current_year})...")

    response = self._make_request(params)
```

Query building:

- USDA API uses filter parameters:
  - `source_desc="SURVEY"` - Survey data (not census)
  - `sector_desc="CROPS"` - Crop sector (not livestock)
  - `year__GE` - Year greater-than-or-equal (start)
  - `year__LE` - Year less-than-or-equal (end)
- Uses commodity config for commodity-specific parameters

```
    if not response or 'data' not in response:
        logger.warning(f"No data returned for {commodity}")
        return []

    records = []
    for item in response['data']:
        try:
            value = item.get('Value', '').replace(',', '')
            if value and value != '(D)' and value != '(NA)':
                record = {
                    'commodity': commodity.title(),
                    'date': f"{item.get('year', datetime.now().year)}-01-01",
                    'price': float(value),
                    'unit': item.get('unit_desc', 'USD'),
                    'source': 'USDA NASS',
                    'state': item.get('state_name', 'US TOTAL'),
                    'market': 'National Average',
                    'frequency': item.get('freq_desc', 'ANNUAL')
                }
                records.append(record)
        except (ValueError, KeyError) as e:
            logger.debug(f"Skipping invalid record: {e}")
            continue

    logger.info(f"Scraped {len(records)} records for {commodity}")
    self._rate_limit()
```

```
        return records
```

Record parsing:

- `value.replace(',', '')` - Remove thousands separator (1,234 → 1234)
- `value != '(D)'` - Skip suppressed data
- `value != '(NA)'` - Skip not available
- `try/except` - Skip malformed records without crashing
- Normalize field names (commodity, date, price, etc.)

```
    def add_commodity(self, name: str, config_dict: Dict) -> None:
        """
        Add a new commodity configuration for scraping.
        """
        COMMODITIES[name.upper()] = config_dict
        logger.info(f"Added commodity: {name}")
```

Why add_commodity method?

- Allows runtime addition without editing source
- Extensibility for users
- Logs the addition for debugging

3.3 ETL Module

3.3.1 etl_pipeline.py - Extract Transform Load

This is the core data processing module. Let me explain the key methods:

```
    def extract(self, file_pattern: str = "*.json") -> List[Dict]:
        """
        Extract data from raw JSON files.
        """
```

```
        logger.info(f"Extracting data from {self.input_dir}")

        all_records = []
        json_files = list(self.input_dir.glob(file_pattern))

        logger.info(f"Found {len(json_files)} JSON files")

        for json_file in json_files:
            try:
                with open(json_file) as f:
                    data = json.load(f)

                # Extract source
                source = data.get('source', 'Unknown')

                # Extract records from commodities dict
                commodities_data = data.get('commodities', {})
                for commodity_name, records in commodities_data.items():
                    if isinstance(records, list):
                        for record in records:
                            record['source'] = source
                            record['commodity'] = commodity_name
                            all_records.append(record)

                self.stats['files_processed'] += 1

            except Exception as e:
                logger.error(f"Error extracting {json_file}: {e}")

        self.stats['records_extracted'] = len(all_records)
        return all_records
```

Extract phase explained:

1. `glob(file_pattern)` - Find all matching files

2. Loop through each JSON file
3. Parse nested structure (commodities → array of records)
4. Flatten into single list of records
5. Add source/commodity to each record
6. Count statistics

```
def transform(self, records: List[Dict]) -> pd.DataFrame:
    """
    Transform extracted records.
    """
    if not records:
        return pd.DataFrame()

    df = pd.DataFrame(records)
    initial_count = len(df)

    # Normalize column names
    df.columns = df.columns.str.lower().str.strip()
```

Why normalize columns?

- Different sources use different cases (Price vs price vs PRICE)
- `str.lower()` - Standardize to lowercase
- `str.strip()` - Remove whitespace

```
    # Parse dates
    df['date'] = pd.to_datetime(df['date'], errors='coerce')
    df = df.dropna(subset=['date'])
```

Date handling:

- `pd.to_datetime()` - Parse various date formats
- `errors='coerce'` - Invalid dates become NaT (null)
- `dropna(subset=['date'])` - Remove rows with invalid dates

```
        # Remove duplicates
        before_dedupe = len(df)
        df = df.drop_duplicates(
            subset=['commodity', 'date', 'source', 'market'],
            keep='first'
        )
        self.stats['duplicates_removed'] = before_dedupe - len(df)
```

Deduplication:

- Same commodity+date+source+market = duplicate
- `keep='first'` - Keep first occurrence
- Track duplicates removed for statistics

```
        # Detect anomalies (prices > 3 std from mean)
        for commodity in df['commodity'].unique():
            mask = df['commodity'] == commodity
            prices = df.loc[mask, 'price']
            if len(prices) > 10:
                mean, std = prices.mean(), prices.std()
                if std > 0:
                    z_scores = abs((prices - mean) / std)
                    anomaly_mask = z_scores > 3
                    anomaly_count = anomaly_mask.sum()
                    if anomaly_count > 0:
                        self.stats['anomalies_detected'] += anomaly_count
```

Anomaly detection:

- Z-score = (value - mean) / standard deviation
- |Z| > 3 means value is >3 standard deviations from mean
- 99.7% of normal data falls within 3 std
- Flag (but don't remove) anomalies

```
    # Create record hash for deduplication
    df['record_hash'] = df.apply(
        lambda r: hashlib.md5(
            f"{r['commodity']}{r['date']}{r['source']}{r.get('market', '')}".encode()
        ).hexdigest()[:16],
        axis=1
    )
```

Why record hash?

- Unique identifier for each record
- Used as database primary key
- MD5 produces consistent hash for same input
- `[:16]` - First 16 chars is enough for uniqueness

3.4 Analytics Module

3.4.1 trend_analyzer.py

```
def calculate_moving_averages(
    self,
    df: pd.DataFrame,
    windows: List[int] = [7, 15, 30]
) -> Dict[str, float]:
    """
    Calculate moving averages for different windows.
    """
    result = {}

    for window in windows:
        if len(df) >= window:
            ma = df['price'].rolling(window=window).mean().iloc[-1]
```

```
            result[f'ma_{window}_day'] = round(ma, 2)
        else:
            result[f'ma_{window}_day'] = None

    return result
```

Moving averages explained:

- `rolling(window=7)` - Create 7-day sliding window
- `.mean()` - Average of each window
- `.iloc[-1]` - Get latest (most recent) value
- Different windows show different trends:
    - 7-day: Short-term, responsive to recent changes
    - 15-day: Medium-term
    - 30-day: Long-term, smooth out noise

Mathematical formula:

```
MA_n = (P_1 + P_2 + ... + P_n) / n
```

```
def detect_trend(
    self,
    df: pd.DataFrame,
    short_window: int = 7,
    long_window: int = 30
) -> Dict[str, Any]:
    """
    Detect price trend direction.
    """
    if len(df) < long_window:
        return {'trend': 'Unknown', 'strength': 0}

    short_ma = df['price'].rolling(window=short_window).mean().iloc[-1]
    long_ma = df['price'].rolling(window=long_window).mean().iloc[-1]
```

```
    if long_ma == 0:
        return {'trend': 'Unknown', 'strength': 0}

    diff_pct = ((short_ma - long_ma) / long_ma) * 100

    if diff_pct > 2:
        trend = 'Upward'
    elif diff_pct < -2:
        trend = 'Downward'
    else:
        trend = 'Stable'

    return {
        'trend': trend,
        'strength': round(abs(diff_pct), 2)
    }
```

Trend detection logic:

1. Calculate short-term MA (responsive to recent changes)
2. Calculate long-term MA (baseline)
3. Compare: `(short - long) / long * 100`
4. If short > long by >2%: Upward trend
5. If short < long by >2%: Downward trend
6. Otherwise: Stable

Why 2% threshold?

- Small differences could be noise
- 2% is significant enough to indicate real trend
- Adjustable based on commodity volatility

```
def calculate_volatility(self, df: pd.DataFrame) -> Dict[str, Any]:
    """
    Calculate price volatility using coefficient of variation.
```

```
    """
    if df.empty or len(df) < 10:
        return {'volatility_pct': 0, 'level': 'Unknown'}

    mean_price = df['price'].mean()
    std_price = df['price'].std()

    if mean_price == 0:
        return {'volatility_pct': 0, 'level': 'Unknown'}

    cv = (std_price / mean_price) * 100

    if cv < 10:
        level = 'Low'
    elif cv < 20:
        level = 'Medium'
    else:
        level = 'High'

    return {
        'volatility_pct': round(cv, 2),
        'level': level
    }
```

Coefficient of Variation (CV):

```
CV = (Standard Deviation / Mean) × 100%
```

Why CV instead of just standard deviation?

- CV is normalized by mean
- Allows comparison between different price scales
- $100 commodity with $10 std vs $1000 commodity with $10 std
- Same std, but very different volatility!

3.4.2 forecasters.py

```python
class MovingAverageForecaster:
    """Simple Moving Average forecaster."""

    def __init__(self, window: int = 7):
        self.window = window
        self.name = "Moving Average"

    def forecast(self, series: pd.Series, periods: int = 7) -> List[float]:
        """Generate forecast using moving average trend."""
        if len(series) < self.window:
            return [series.mean()] * periods

        ma = series.tail(self.window).mean()
        trend = (series.iloc[-1] - series.iloc[-self.window]) / self.window

        forecasts = []
        for i in range(1, periods + 1):
            forecasts.append(round(ma + trend * i, 2))

        return forecasts
```

MA Forecasting logic:

1. Calculate recent MA (last 7 days)
2. Calculate trend: `(latest price - price 7 days ago) / 7`
3. Project forward: `MA + trend * days_ahead`

Limitations:

- Assumes linear trend continues
- Doesn't capture seasonality
- Simple but fast baseline

```python
class ARIMAForecaster:
    """ARIMA time series forecaster."""

    def __init__(self, order: Tuple[int, int, int] = (5, 1, 0)):
        self.order = order
        self.name = "ARIMA"

    def forecast(self, series: pd.Series, periods: int = 7) -> List[float]:
        """Generate ARIMA forecast."""
        try:
            from statsmodels.tsa.arima.model import ARIMA

            if len(series) < 30:
                return [series.mean()] * periods

            model = ARIMA(series, order=self.order)
            fitted = model.fit()
            forecasts = fitted.forecast(steps=periods)

            return [round(f, 2) for f in forecasts.tolist()]

        except Exception as e:
            logger.warning(f"ARIMA failed: {e}, using fallback")
            return [round(series.mean(), 2)] * periods
```

ARIMA explained:

- **AR(p)**: AutoRegressive - uses past values
- **I(d)**: Integrated - differencing for stationarity
- **MA(q)**: Moving Average - uses past errors

Order (5, 1, 0) means:

- p=5: Use last 5 values for autoregression
- d=1: First difference (makes non-stationary data stationary)
- q=0: No moving average of errors

Why (5, 1, 0)?

- Simple model that often works
- 1 differencing handles linear trends
- 5 lags capture weekly patterns

```
def calculate_mape(actual: np.ndarray, predicted: np.ndarray) -> float:
    """Calculate Mean Absolute Percentage Error."""
    actual = np.array(actual)
    predicted = np.array(predicted)

    mask = actual != 0
    if not mask.any():
        return 0.0

    mape = np.mean(np.abs((actual[mask] - predicted[mask]) / actual[mask])) * 10
           0
    return round(mape, 2)
```

MAPE formula:

```
MAPE = (1/n) × Σ |actual - predicted| / actual × 100%
```

Why MAPE?

- Percentage-based (scale-independent)
- Easy to interpret (5% error means 5% off on average)
- Mask zeros to avoid division by zero

3.5 Financial Module

3.5.1 cost_calculator.py

```
class CostCalculator:
    """
    Calculates production costs per hectare for commodities.
    """

    def __init__(self, config_path: str = "config/production_costs.yaml"):
        self.config = self._load_config(config_path)

    def _load_config(self, path: str) -> Dict:
        """Load cost configuration from YAML."""
        try:
            with open(path) as f:
                return yaml.safe_load(f)
        except:
            return self._get_default_costs()

    def get_total_cost(self, commodity: str) -> float:
        """
        Get total production cost per hectare.
        """
        costs = self.config.get('costs', {}).get(commodity, {})

        return sum([
            costs.get('seeds', 0),
            costs.get('fertilizers', 0),
            costs.get('pesticides', 0),
            costs.get('labor', 0),
            costs.get('irrigation', 0),
            costs.get('machinery', 0),
```

```
            costs.get('other', 0)
        ])
```

Cost categories:

1. **Seeds**: Initial planting material
2. **Fertilizers**: NPK, micronutrients
3. **Pesticides**: Herbicides, insecticides, fungicides
4. **Labor**: Planting, weeding, harvesting
5. **Irrigation**: Water, pumping costs
6. **Machinery**: Tractors, equipment rental
7. **Other**: Transport, storage, misc

3.5.2 profit_calculator.py

```
    def calculate_profit(
        self,
        commodity: str,
        price_per_unit: float = None,
        yield_per_hectare: float = None
    ) -> Dict[str, float]:
        """
        Calculate profit for a commodity.

        Returns:
            Dict with revenue, cost, profit, roi, margin
        """
        # Get current price if not provided
        if price_per_unit is None:
            price_per_unit = self._get_current_price(commodity)

        # Get expected yield if not provided
        if yield_per_hectare is None:
            yield_per_hectare = self._get_expected_yield(commodity)
```

```
    # Calculate
    total_cost = self.cost_calc.get_total_cost(commodity)
    revenue = price_per_unit * yield_per_hectare
    profit = revenue - total_cost

    roi = (profit / total_cost) * 100 if total_cost > 0 else 0
    margin = (profit / revenue) * 100 if revenue > 0 else 0

    return {
        'commodity': commodity,
        'price_per_unit': price_per_unit,
        'yield_per_hectare': yield_per_hectare,
        'revenue': round(revenue, 2),
        'total_cost': round(total_cost, 2),
        'profit': round(profit, 2),
        'roi_percent': round(roi, 2),
        'profit_margin_percent': round(margin, 2)
    }
```

Financial formulas:

```
Revenue = Price × Yield
Profit = Revenue - Cost
ROI = (Profit / Cost) × 100%
Margin = (Profit / Revenue) × 100%
```

Why both ROI and Margin?

- **ROI**: Return relative to investment (cost)
    - Answers: “How much do I get back per rupee invested?”
    - Higher is better for capital efficiency
- **Margin**: Profit relative to sales (revenue)

- Answers: “What % of revenue is profit?”
- Important for pricing decisions

3.5.3 opportunity_scorer.py

```python
def calculate_score(self, commodity: str) -> Dict[str, Any]:
    """
    Calculate opportunity score for a commodity.

    Score = weighted sum of:
    - ROI score (35%)
    - Trend score (25%)
    - Stability score (20%)
    - Demand score (20%)
    """
    # Get component scores
    roi_score = self._get_roi_score(commodity)
    trend_score = self._get_trend_score(commodity)
    stability_score = self._get_stability_score(commodity)
    demand_score = self._get_demand_score(commodity)

    # Weighted sum
    total_score = (
        roi_score * 0.35 +
        trend_score * 0.25 +
        stability_score * 0.20 +
        demand_score * 0.20
    )

    # Determine recommendation
    if total_score >= 70:
        recommendation = "Strong Buy"
    elif total_score >= 50:
```

```
        recommendation = "Buy"
    elif total_score >= 30:
        recommendation = "Hold"
    else:
        recommendation = "Avoid"

    return {
        'commodity': commodity,
        'score': round(total_score, 1),
        'recommendation': recommendation,
        'components': {
            'roi': roi_score,
            'trend': trend_score,
            'stability': stability_score,
            'demand': demand_score
        }
    }
```

Scoring methodology:

1. **ROI Score (35%)** - Higher ROI = higher score
2. **Trend Score (25%)** - Upward trend = higher score
3. **Stability Score (20%)** - Lower volatility = higher score
4. **Demand Score (20%)** - Market demand indicators

Why these weights?

- ROI is most important (direct profit indicator)
- Trend shows future potential
- Stability reduces risk
- Demand ensures market exists

3.6 Visualization Module

3.6.1 visualizer.py

```
def plot_price_trends(
    self,
    commodity: str,
    days: int = 180
) -> Optional[str]:
    """
    Plot price trends with moving averages.
    """
    # ... query data ...

    df['ma_7'] = df['price'].rolling(7).mean()
    df['ma_30'] = df['price'].rolling(30).mean()

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

    ax.plot(df['price_date'], df['price'],
            label='Daily Price', alpha=0.7, linewidth=1)
    ax.plot(df['price_date'], df['ma_7'],
            label='7-Day MA', linewidth=2)
    ax.plot(df['price_date'], df['ma_30'],
            label='30-Day MA', linewidth=2)

    ax.fill_between(df['price_date'], df['price'], alpha=0.1)

    ax.set_title(f'{commodity} Price Trends ({days} Days)', fontweight='bold')
    ax.set_xlabel('Date')
    ax.set_ylabel('Price')
    ax.legend(loc='upper left')
    ax.grid(True, alpha=0.3)
```

Matplotlib components:

- `plt.subplots()` - Create figure and axes
- `ax.plot()` - Line plot
- `ax.fill_between()` - Shaded area under curve
- `ax.set_title()` - Chart title
- `ax.legend()` - Show label legend
- `ax.grid()` - Add gridlines

Summary

This documentation covered:

1. **High-Level Design** - System architecture, data flow, design principles
2. **Low-Level Design** - Class diagrams, database schema, sequence diagrams
3. **Module Documentation** - Line-by-line code explanation for each module

Key design patterns used:

- **Strategy Pattern** - Multiple scrapers with same interface
- **Pipeline Pattern** - ETL with distinct phases
- **Singleton Pattern** - Config and logger instances
- **Factory Pattern** - Logger creation

Key algorithms:

- **Moving Average** - Trend smoothing
- **ARIMA** - Time series forecasting
- **Coefficient of Variation** - Volatility measurement
- **Weighted Scoring** - Multi-criteria decision making

Last updated: February 1, 2026