Module 2: Data Collection & Processing

Introduction

High-quality data is the foundation of any successful MEV strategy. This module covers comprehensive data collection, cleaning, and preprocessing pipelines necessary for robust backtesting and live trading. You'll learn to work with blockchain data, DEX information, and market microstructure data.

Learning Objectives

By the end of this module, you will be able to:
- Collect real-time and historical blockchain data from multiple sources
- Process DEX pool data and liquidity information
- Build efficient data pipelines for continuous data ingestion
- Implement data quality checks and validation systems
- Create feature engineering pipelines for ML-based strategies

2.1 Blockchain Data Collection

2.1.1 Web3 Data Extraction

```
from web3 import Web3
from typing import Dict, List, Optional, Any, Tuple
from dataclasses import dataclass
from datetime import datetime, timedelta
import requests
import json
import asyncio
from concurrent.futures import ThreadPoolExecutor, as_completed

@dataclass
class TransactionData:
    """Blockchain transaction data structure"""
    tx_hash: str
    block_number: int
    timestamp: datetime
    from_address: str
    to_address: str
    gas_used: int
    gas_price: int
    value: float
    transaction_index: int
    success: bool
    input_data: str

@dataclass
class LogEvent:
    """Smart contract log event data"""
    transaction_hash: str
    log_index: int
    block_number: int
    address: str
    topics: List[str]
    data: str
    timestamp: datetime

class BlockchainDataCollector:
    """Collects and processes blockchain data"""

    def __init__(self, provider_urls: List[str], rate_limit: int =
100):
        self.providers = [Web3(Web3.HTTPProvider(url)) for url in
provider_urls]
```

```
        self.current_provider = 0
        self.rate_limit = rate_limit
        self.request_count = 0

    def get_current_provider(self) -> Web3:
        """Get current Web3 provider with rotation"""
        provider = self.providers[self.current_provider]
        self.request_count += 1

        # Rotate providers every 1000 requests
        if self.request_count % 1000 == 0:
            self.current_provider = (self.current_provider + 1) %
len(self.providers)

        return provider

    def collect_transactions_by_block(self, block_number: int) ->
List[TransactionData]:
        """Collect all transactions from a specific block"""
        w3 = self.get_current_provider()

        try:
            block = w3.eth.get_block(block_number,
full_transactions=True)
            transactions = []

            for tx in block['transactions']:
                transaction_data = TransactionData(
                    tx_hash=tx['hash'].hex(),
                    block_number=block_number,

timestamp=datetime.fromtimestamp(block['timestamp']),
                    from_address=tx['from'],
                    to_address=tx['to'] if tx['to'] else
'0x0000000000000000000000000000000000000000',
                    gas_used=tx['gas'],
                    gas_price=tx['gasPrice'],
                    value=w3.from_wei(tx['value'], 'ether'),
                    transaction_index=tx['transactionIndex'],
                    success=True,  # Would need to check receipt
                    input_data=tx['input']
                )
```

```
                transactions.append(transaction_data)

            return transactions

        except Exception as e:
            print(f"Error collecting block {block_number}: {e}")
            return []

    def collect_transactions_by_time_range(self, start_block: int,
end_block: int) -> List[TransactionData]:
        """Collect transactions over a range of blocks"""
        all_transactions = []

        for block_num in range(start_block, end_block + 1):
            block_transactions =
self.collect_transactions_by_block(block_num)
            all_transactions.extend(block_transactions)

        return all_transactions

    def collect_log_events(self, contract_address: str,
event_signature: str,
                           start_block: int, end_block: int) ->
List[LogEvent]:
        """Collect specific log events from contract"""
        w3 = self.get_current_provider()

        try:
            event_abi = {
                "anonymous": False,
                "inputs": [],
                "name": event_signature,
                "type": "event"
            }

            event_filter = w3.eth.filter({
                'address': contract_address,
                'fromBlock': start_block,
                'toBlock': end_block,
                'topics': [w3.keccak(text=event_signature).hex()]
            })
```

```
            logs = event_filter.get_all_entries()
            events = []

            for log in logs:
                event = LogEvent(
                    transaction_hash=log['transactionHash'].hex(),
                    log_index=log['logIndex'],
                    block_number=log['blockNumber'],
                    address=log['address'],
                    topics=[topic.hex() for topic in log['topics']],
                    data=log['data'],
                    timestamp=datetime.fromtimestamp(
                        w3.eth.get_block(log['blockNumber'])
['timestamp']
                    )
                )
                events.append(event)

            return events

        except Exception as e:
            print(f"Error collecting events: {e}")
            return []

    def collect_dex_pool_data(self, factory_address: str, start_block:
int, end_block: int) -> Dict[str, Any]:
        """Collect Uniswap V3 pool creation events"""
        w3 = self.get_current_provider()

        # Uniswap V3 PoolCreated event signature
        pool_created_signature =
w3.keccak(text="PoolCreated(address,address,uint24,int24,address)").hex()

        pools = {}

        try:
            event_filter = w3.eth.filter({
                'address': factory_address,
                'fromBlock': start_block,
                'toBlock': end_block,
                'topics': [pool_created_signature]
            })
```

```
            for log in event_filter.get_all_entries():
                # Decode pool creation parameters
                pool_address = w3.to_checksum_address(log['topics'][1]
[-40:])
                token0 = w3.to_checksum_address(log['topics'][2][-40:])
                token1 = w3.to_checksum_address(log['topics'][3][-40:])
                fee = int(log['topics'][4], 16)

                pools[pool_address] = {
                    'token0': token0,
                    'token1': token1,
                    'fee': fee,
                    'creation_block': log['blockNumber'],
                    'creation_timestamp': datetime.fromtimestamp(
                        w3.eth.get_block(log['blockNumber'])
['timestamp']
                    )
                }

        except Exception as e:
            print(f"Error collecting pool data: {e}")

        return pools

# Alternative: Use public APIs
class APIDataCollector:
    """Collects blockchain data using public APIs"""

    def __init__(self, api_key: str = None):
        self.api_key = api_key
        self.base_url = "https://api.etherscan.io/api"

    def get_transactions_by_address(self, address: str, start_block:
int,
                                    end_block: int, page: int = 1) ->
List[Dict[str, Any]]:
        """Get transactions for an address using Etherscan API"""
        params = {
            'module': 'account',
            'action': 'txlist',
            'address': address,
```

```
            'startblock': start_block,
            'endblock': end_block,
            'page': page,
            'offset': 10000,
            'sort': 'desc'
        }

        if self.api_key:
            params['apikey'] = self.api_key

        response = requests.get(self.base_url, params=params)
        data = response.json()

        if data['status'] == '1':
            return data['result']
        else:
            print(f"API Error: {data['message']}")
            return []

    def get_token_transfers(self, contract_address: str, start_block:
int,
                            end_block: int) -> List[Dict[str, Any]]:
        """Get ERC-20 token transfers"""
        params = {
            'module': 'account',
            'action': 'tokentx',
            'contractaddress': contract_address,
            'startblock': start_block,
            'endblock': end_block,
            'sort': 'desc',
            'offset': 10000
        }

        if self.api_key:
            params['apikey'] = self.api_key

        response = requests.get(self.base_url, params=params)
        data = response.json()

        if data['status'] == '1':
            return data['result']
```

```
        else:
            return []
```

2.1.2 Real-time Data Streaming

```
import asyncio
from websockets import connect
import json
from typing import Callable, Dict, Any

class RealTimeDataStreamer:
    """Real-time blockchain data streaming"""

    def __init__(self, ws_url: str, callback: Callable = None):
        self.ws_url = ws_url
        self.callback = callback
        self.connected = False

    async def connect_to_websocket(self):
        """Connect to WebSocket endpoint for real-time data"""
        try:
            async with connect(self.ws_url) as websocket:
                self.connected = True
                print(f"Connected to {self.ws_url}")

                # Subscribe to new blocks
                await websocket.send(json.dumps({
                    "jsonrpc": "2.0",
                    "id": 1,
                    "method": "eth_subscribe",
                    "params": ["newHeads"]
                }))

                # Listen for new blocks
                async for message in websocket:
                    data = json.loads(message)
                    if 'params' in data:
                        block_data = data['params']['result']
                        await self.process_new_block(block_data)

        except Exception as e:
            print(f"WebSocket connection error: {e}")
            self.connected = False

    async def process_new_block(self, block_data: Dict[str, Any]):
        """Process new block data"""
        try:
```

```
            block_number = int(block_data['number'], 16)

            # Extract relevant information
            processed_block = {
                'block_number': block_number,
                'timestamp': int(block_data['timestamp'], 16),
                'difficulty': int(block_data['difficulty'], 16),
                'gas_limit': int(block_data['gasLimit'], 16),
                'gas_used': int(block_data['gasUsed'], 16),
                'hash': block_data['hash']
            }

            # Call user-defined callback
            if self.callback:
                await self.callback(processed_block)

        except Exception as e:
            print(f"Error processing block: {e}")

    def start_streaming(self):
        """Start the streaming loop"""
        asyncio.run(self.connect_to_websocket())

class DEXDataCollector:
    """Collects real-time DEX data"""

    def __init__(self):
        self.dexes = {
            'uniswap_v2': 'https://api.thegraph.com/subgraphs/name/
uniswap/uniswap-v2',
            'uniswap_v3': 'https://api.thegraph.com/subgraphs/name/
uniswap/uniswap-v3',
            'sushiswap': 'https://api.thegraph.com/subgraphs/name/
sushiswap/exchange'
        }

    async def get_uniswap_v2_pairs(self, first: int = 1000) ->
List[Dict[str, Any]]:
        """Get Uniswap V2 pair data from GraphQL"""
        query = f"""
        {{
            pairs(first: {first}, orderBy: reserveUSD, orderDirection:
```

```
desc) {{
                id
                token0 {{
                    id
                    symbol
                    name
                }}
                token1 {{
                    id
                    symbol
                    name
                }}
                reserveUSD
                volumeUSD
                token0Price
                token1Price
                totalSupply
            }}
        }}
        """

        async with aiohttp.ClientSession() as session:
            async with session.post(self.dexes['uniswap_v2'],
                                    json={'query': query}) as response:
                data = await response.json()
                return data['data']['pairs']

    async def get_uniswap_v3_pools(self, first: int = 1000) ->
List[Dict[str, Any]]:
        """Get Uniswap V3 pool data"""
        query = f"""
        {{
            pools(first: {first}, orderBy: totalValueLockedUSD,
orderDirection: desc) {{
                id
                token0 {{
                    id
                    symbol
                    name
                }}
                token1 {{
                    id
```

```
                    symbol
                    name
                }}
                feeTier
                liquidity
                totalValueLockedUSD
                volumeUSD
                token0Price
                token1Price
            }}
        }}
        """

        async with aiohttp.ClientSession() as session:
            async with session.post(self.dexes['uniswap_v3'],
                                    json={'query': query}) as response:
                data = await response.json()
                return data['data']['pools']

    async def monitor_price_changes(self, pairs: List[str], interval:
int = 1) -> Dict[str, float]:
        """Monitor price changes for specific pairs"""
        while True:
            try:
                price_data = {}

                # Get current prices
                v2_pairs = await self.get_uniswap_v2_pairs(100)
                v3_pools = await self.get_uniswap_v3_pools(100)

                # Combine data
                all_pairs = v2_pairs + v3_pools

                for pair in all_pairs:
                    if 'pairAddress' in pair:
                        pair_id = pair['pairAddress']
                    elif 'id' in pair:
                        pair_id = pair['id']
                    else:
                        continue

                    if pair_id in pairs:
```

```
                    # Calculate price
                    if 'token0Price' in pair:
                        price = float(pair['token0Price'])
                    elif 'token1Price' in pair:
                        price = float(pair['token1Price'])
                    else:
                        continue

                    price_data[pair_id] = {
                        'price': price,
                        'timestamp': datetime.now(),
                        'liquidity': pair.get('reserveUSD', 0)
                    }

            yield price_data

            await asyncio.sleep(interval)

        except Exception as e:
            print(f"Error monitoring prices: {e}")
            await asyncio.sleep(interval)
```

2.2 Data Storage and Management

2.2.1 Database Design

```
import sqlite3
import pandas as pd
from typing import Dict, List, Any, Optional
from datetime import datetime
import pickle
import json

class BlockchainDatabase:
    """SQLite database for blockchain data storage"""

    def __init__(self, db_path: str):
        self.db_path = db_path
        self.init_database()

    def init_database(self):
        """Initialize database schema"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Transactions table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS transactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            tx_hash TEXT UNIQUE NOT NULL,
            block_number INTEGER NOT NULL,
            timestamp DATETIME NOT NULL,
            from_address TEXT NOT NULL,
            to_address TEXT NOT NULL,
            gas_used INTEGER NOT NULL,
            gas_price INTEGER NOT NULL,
            value REAL NOT NULL,
            transaction_index INTEGER NOT NULL,
            success BOOLEAN NOT NULL,
            input_data TEXT,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP
        )
        ''')

        # Log events table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS log_events (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
            transaction_hash TEXT NOT NULL,
            log_index INTEGER NOT NULL,
            block_number INTEGER NOT NULL,
            contract_address TEXT NOT NULL,
            event_name TEXT NOT NULL,
            topics TEXT NOT NULL,
            data TEXT NOT NULL,
            timestamp DATETIME NOT NULL,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (transaction_hash) REFERENCES transactions
(tx_hash)
        )
        ''')

        # DEX pools table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS dex_pools (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            pool_address TEXT UNIQUE NOT NULL,
            dex_name TEXT NOT NULL,
            token0_address TEXT NOT NULL,
            token1_address TEXT NOT NULL,
            token0_symbol TEXT,
            token1_symbol TEXT,
            fee_tier INTEGER,
            liquidity REAL,
            volume_24h REAL,
            total_value_locked REAL,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
            updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
        )
        ''')

        # Price data table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS price_data (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            pair_address TEXT NOT NULL,
            token0_address TEXT NOT NULL,
            token1_address TEXT NOT NULL,
            price REAL NOT NULL,
            liquidity REAL,
```

```
            volume_24h REAL,
            block_number INTEGER NOT NULL,
            timestamp DATETIME NOT NULL,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (pair_address) REFERENCES dex_pools
(pool_address)
        )
        ''')

        # Indexes for better performance
        cursor.execute('CREATE INDEX IF NOT EXISTS
idx_transactions_block ON transactions(block_number)')
        cursor.execute('CREATE INDEX IF NOT EXISTS
idx_transactions_timestamp ON transactions(timestamp)')
        cursor.execute('CREATE INDEX IF NOT EXISTS
idx_transactions_hash ON transactions(tx_hash)')
        cursor.execute('CREATE INDEX IF NOT EXISTS
idx_log_events_contract ON log_events(contract_address)')

cursor.execute('CREATE INDEX IF NOT EXISTS idx_log_events_block ON
log_events(block_number)')
        cursor.execute('CREATE INDEX IF NOT EXISTS idx_price_data_pair
ON price_data(pair_address)')
        cursor.execute('CREATE INDEX IF NOT EXISTS
idx_price_data_timestamp ON price_data(timestamp)')

        conn.commit()
        conn.close()

    def insert_transaction(self, tx: TransactionData):
        """Insert transaction into database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            cursor.execute('''
            INSERT OR REPLACE INTO transactions
            (tx_hash, block_number, timestamp, from_address,
to_address,
             gas_used, gas_price, value, transaction_index, success,
input_data)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

```
            ''', (
                tx.tx_hash, tx.block_number, tx.timestamp,
tx.from_address,
                tx.to_address, tx.gas_used, tx.gas_price, tx.value,
                tx.transaction_index, tx.success, tx.input_data
            ))
            conn.commit()
        except Exception as e:
            print(f"Error inserting transaction: {e}")
        finally:
            conn.close()

    def insert_log_event(self, event: LogEvent, event_name: str):
        """Insert log event into database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            cursor.execute('''
            INSERT OR REPLACE INTO log_events
            (transaction_hash, log_index, block_number,
contract_address,
             event_name, topics, data, timestamp)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
            ''', (
                event.transaction_hash, event.log_index,
event.block_number,
                event.address, event_name, json.dumps(event.topics),
                event.data, event.timestamp
            ))
            conn.commit()
        except Exception as e:
            print(f"Error inserting log event: {e}")
        finally:
            conn.close()

    def get_transactions_by_time_range(self, start_time: datetime,
                                       end_time: datetime) ->
pd.DataFrame:
        """Get transactions in time range"""
        conn = sqlite3.connect(self.db_path)
```

```
        query = '''
        SELECT * FROM transactions
        WHERE timestamp BETWEEN ? AND ?
        ORDER BY timestamp
        '''

        df = pd.read_sql_query(query, conn, params=(start_time,
end_time))
        conn.close()

        return df

    def get_dex_pools(self) -> pd.DataFrame:
        """Get all DEX pools"""
        conn = sqlite3.connect(self.db_path)
        df = pd.read_sql_query('SELECT * FROM dex_pools', conn)
        conn.close()

        return df

    def insert_price_data(self, pair_address: str, token0_address: str,
                          token1_address: str, price: float, liquidity:
float,
                          volume_24h: float, block_number: int,
timestamp: datetime):
        """Insert price data"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            cursor.execute('''
            INSERT INTO price_data
            (pair_address, token0_address, token1_address, price,
liquidity,
             volume_24h, block_number, timestamp)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
            ''', (pair_address, token0_address, token1_address, price,
                  liquidity, volume_24h, block_number, timestamp))
            conn.commit()
        except Exception as e:
            print(f"Error inserting price data: {e}")
        finally:
```

```
            conn.close()

    def get_price_series(self, pair_address: str, start_time: datetime,
                         end_time: datetime) -> pd.DataFrame:
        """Get price time series for a pair"""
        conn = sqlite3.connect(self.db_path)

        query = '''
        SELECT timestamp, price, liquidity, volume_24h
        FROM price_data
        WHERE pair_address = ? AND timestamp BETWEEN ? AND ?
        ORDER BY timestamp
        '''

        df = pd.read_sql_query(query, conn,
                               params=(pair_address, start_time,
end_time))
        conn.close()

        return df

class DataCompressionManager:
    """Manages data compression and archival"""

    def __init__(self, database: BlockchainDatabase,
compression_threshold: int = 1000000):
        self.database = database
        self.compression_threshold = compression_threshold

    def compress_old_data(self, cutoff_date: datetime):
        """Compress and archive old transaction data"""
        conn = sqlite3.connect(self.database.db_path)
        cursor = conn.cursor()

        try:
            # Move old transactions to compressed storage
            old_transactions = cursor.execute('''
            SELECT * FROM transactions
            WHERE timestamp < ? AND compressed = 0
            ''', (cutoff_date,)).fetchall()

            if old_transactions:
```

```
                # Serialize and compress
                compressed_data = pickle.dumps(old_transactions)

                # Store in separate table
                cursor.execute('''
                CREATE TABLE IF NOT EXISTS compressed_transactions (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    cutoff_date DATE NOT NULL,
                    compressed_data BLOB NOT NULL,
                    record_count INTEGER NOT NULL,
                    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
                )
                ''')

                cursor.execute('''
                INSERT INTO compressed_transactions
                (cutoff_date, compressed_data, record_count)
                VALUES (?, ?, ?)
                ''', (cutoff_date, compressed_data,
len(old_transactions)))

                # Mark as compressed in main table
                cursor.execute('''
                UPDATE transactions
                SET compressed = 1
                WHERE timestamp < ?
                ''', (cutoff_date,))

                conn.commit()
                print(f"Compressed {len(old_transactions)} old
transactions")

        except Exception as e:
            print(f"Error compressing data: {e}")
        finally:
            conn.close()
```

2.2.2 Data Quality and Validation

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

@dataclass
class DataQualityReport:
    """Data quality assessment results"""
    total_records: int
    missing_values: Dict[str, int]
    duplicate_records: int
    data_types_correct: bool
    value_ranges_acceptable: Dict[str, bool]
    consistency_score: float
    recommendations: List[str]

class DataValidator:
    """Validates data quality and integrity"""

    def __init__(self, database: BlockchainDatabase):
        self.database = database

    def validate_transactions(self, start_time: datetime,
                              end_time: datetime) -> DataQualityReport:
        """Validate transaction data quality"""
        conn = sqlite3.connect(self.database.db_path)

        # Get transaction data
        df = pd.read_sql_query('''
        SELECT * FROM transactions
        WHERE timestamp BETWEEN ? AND ?
        ORDER BY timestamp
        ''', conn, params=(start_time, end_time))

        conn.close()

        if df.empty:
            return DataQualityReport(0, {}, 0, True, {}, 1.0, [])

        # Check for missing values
        missing_values = {}
        for column in ['tx_hash', 'block_number', 'timestamp',
'from_address', 'to_address']:
```

```
            missing_count = df[column].isnull().sum()
            if missing_count > 0:
                missing_values[column] = missing_count

        # Check for duplicates
        duplicate_records = df['tx_hash'].duplicated().sum()

        # Check data types
        data_types_correct = True
        if not pd.api.types.is_numeric_dtype(df['block_number']):
            data_types_correct = False
        if not pd.api.types.is_numeric_dtype(df['gas_used']):
            data_types_correct = False

        # Check value ranges
        value_ranges_acceptable = {}

        # Gas price should be reasonable (between 0.1 gwei and 1000
gwei)
        if df['gas_price'].min() >= 100_000_000 and
df['gas_price'].max() <= 1_000_000_000_000:
            value_ranges_acceptable['gas_price'] = True
        else:
            value_ranges_acceptable['gas_price'] = False

        # Block numbers should be increasing
        if df['block_number'].is_monotonic_increasing:
            value_ranges_acceptable['block_number'] = True
        else:
            value_ranges_acceptable['block_number'] = False

        # Calculate consistency score
        total_checks = len(missing_values) + 1 +
len(value_ranges_acceptable)  # +1 for data types
        passed_checks = 0

        passed_checks += len(missing_values) == 0  # Missing values
check
        passed_checks += data_types_correct  # Data types check
        passed_checks += sum(value_ranges_acceptable.values())
# Value ranges check
```

```
        consistency_score = passed_checks / total_checks if
total_checks > 0 else 1.0

        # Generate recommendations
        recommendations = []
        if missing_values:

recommendations.append(f"Address missing values in columns:
{list(missing_values.keys())}")
        if duplicate_records > 0:
            recommendations.append(f"Remove {duplicate_records}
duplicate records")
        if not value_ranges_acceptable.get('gas_price', True):
            recommendations.append("Review gas price outliers")
        if not value_ranges_acceptable.get('block_number', True):
            recommendations.append("Check block number ordering")

        return DataQualityReport(
            total_records=len(df),
            missing_values=missing_values,
            duplicate_records=duplicate_records,
            data_types_correct=data_types_correct,
            value_ranges_acceptable=value_ranges_acceptable,
            consistency_score=consistency_score,
            recommendations=recommendations
        )

    def validate_price_data(self, pair_address: str, start_time:
datetime,
                            end_time: datetime) -> Dict[str, Any]:
        """Validate price data quality"""
        price_series = self.database.get_price_series(pair_address,
start_time, end_time)

        if price_series.empty:
            return {'error': 'No price data found'}

        validation_results = {
            'total_data_points': len(price_series),
            'time_gaps': self._detect_time_gaps(price_series),
            'price_outliers':
self._detect_price_outliers(price_series),
```

```
            'liquidity_issues':
self._check_liquidity_issues(price_series),
            'data_freshness': self._assess_data_freshness(price_series)
        }

        return validation_results

    def _detect_time_gaps(self, df: pd.DataFrame) -> Dict[str, Any]:
        """Detect gaps in time series"""
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        df = df.sort_values('timestamp')

        # Calculate time differences
        time_diffs = df['timestamp'].diff().dt.total_seconds()

        # Detect gaps (anything longer than expected)
        max_gap = time_diffs.max()
        avg_gap = time_diffs.mean()
        std_gap = time_diffs.std()

        # Flag gaps longer than 3 standard deviations from mean
        threshold = avg_gap + 3 * std_gap
        large_gaps = time_diffs[time_diffs > threshold]

        return {
            'max_gap_seconds': max_gap,
            'avg_gap_seconds': avg_gap,
            'large_gaps_count': len(large_gaps),
            'large_gaps_threshold': threshold
        }

    def _detect_price_outliers(self, df: pd.DataFrame) -> Dict[str,
Any]:
        """Detect price outliers using statistical methods"""
        prices = df['price'].values

        # Z-score method
        z_scores = np.abs((prices - np.mean(prices)) / np.std(prices))
        outliers_zscore = np.sum(z_scores > 3)

        # IQR method
        Q1 = np.percentile(prices, 25)
```

```
        Q3 = np.percentile(prices, 75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        outliers_iqr = np.sum((prices < lower_bound) | (prices >
upper_bound))

        return {
            'outliers_zscore_count': outliers_zscore,
            'outliers_iqr_count': outliers_iqr,
            'zscore_threshold': 3,
            'iqr_bounds': {'lower': lower_bound, 'upper': upper_bound},
            'price_range': {'min': prices.min(), 'max': prices.max()}
        }

    def _check_liquidity_issues(self, df: pd.DataFrame) -> Dict[str,
Any]:
        """Check for liquidity-related data issues"""
        liquidity = df['liquidity'].values

        # Check for zero or negative liquidity
        zero_liquidity = np.sum(liquidity <= 0)

        # Check for sudden liquidity drops
        liquidity_drops = np.diff(liquidity) / liquidity[:-1]
        large_drops = np.sum(liquidity_drops < -0.5)  # Drops > 50%

        return {
            'zero_liquidity_count': zero_liquidity,
            'large_liquidity_drops': large_drops,
            'liquidity_volatility': np.std(liquidity) /
np.mean(liquidity) if np.mean(liquidity) > 0 else 0
        }

    def _assess_data_freshness(self, df: pd.DataFrame) -> Dict[str,
Any]:
        """Assess how fresh the data is"""
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        latest_timestamp = df['timestamp'].max()
        current_time = datetime.now()
```

```
        data_age_seconds = (current_time -
latest_timestamp).total_seconds()

        return {
            'data_age_seconds': data_age_seconds,
            'data_age_minutes': data_age_seconds / 60,
            'is_recent': data_age_seconds < 300  # Less than 5 minutes
old
        }

class DataCleaner:
    """Cleans and preprocesses raw blockchain data"""

    def __init__(self, database: BlockchainDatabase):
        self.database = database

    def clean_transactions(self, start_time: datetime, end_time:
datetime) -> Dict[str, int]:
        """Clean transaction data"""
        conn = sqlite3.connect(self.database.db_path)
        cursor = conn.cursor()

        stats = {
            'removed_duplicates': 0,
            'corrected_gas_prices': 0,
            'removed_invalid_addresses': 0,
            'total_cleaned': 0
        }

        try:
            # Remove duplicate transactions
            cursor.execute('''
            DELETE FROM transactions
            WHERE id NOT IN (
                SELECT MIN(id)
                FROM transactions
                WHERE timestamp BETWEEN ? AND ?
                GROUP BY tx_hash
            ) AND timestamp BETWEEN ? AND ?
            ''', (start_time, end_time, start_time, end_time))

            stats['removed_duplicates'] = cursor.rowcount
```

```
            # Correct unrealistic gas prices (extreme outliers)
            cursor.execute('''
            UPDATE transactions
            SET gas_price = (
                SELECT AVG(gas_price) * 1.1
                FROM transactions
                WHERE timestamp BETWEEN ? AND ?
                AND gas_price > 0
            )
            WHERE gas_price > 1000000000000
            AND timestamp BETWEEN ? AND ?
            ''', (start_time, end_time, start_time, end_time))

            stats['corrected_gas_prices'] = cursor.rowcount

            # Remove transactions with invalid addresses
            cursor.execute('''
            DELETE FROM transactions
            WHERE (length(from_address) != 42 OR length(to_address) !=
42)
            AND timestamp BETWEEN ? AND ?
            ''', (start_time, end_time))

            stats['removed_invalid_addresses'] = cursor.rowcount

            conn.commit()
            stats['total_cleaned'] = sum(v for k, v in stats.items() if
k != 'total_cleaned')

        except Exception as e:
            print(f"Error cleaning transactions: {e}")
        finally:
            conn.close()

        return stats

    def impute_missing_prices(self, pair_address: str, method: str =
'forward_fill'):
        """Impute missing price data"""
        conn = sqlite3.connect(self.database.db_path)
```

```
        try:
            # Get raw price data
            df = pd.read_sql_query('''
            SELECT timestamp, price, liquidity
            FROM price_data
            WHERE pair_address = ?
            ORDER BY timestamp
            ''', conn, params=(pair_address,))

            df['timestamp'] = pd.to_datetime(df['timestamp'])
            df = df.set_index('timestamp')

            if method == 'forward_fill':
                df['price'] = df['price'].fillna(method='ffill')
                df['liquidity'] =
df['liquidity'].fillna(method='ffill')
            elif method == 'linear':
                df['price'] = df['price'].interpolate(method='linear')
                df['liquidity'] =
df['liquidity'].interpolate(method='linear')
            elif method == 'mean':
                df['price'] = df['price'].fillna(df['price'].mean())
                df['liquidity'] =
df['liquidity'].fillna(df['liquidity'].mean())

            # Write back to database
            for idx, row in df.iterrows():
                cursor = conn.cursor()
                cursor.execute('''
                UPDATE price_data
                SET price = ?, liquidity = ?
                WHERE pair_address = ? AND timestamp = ?
                ''', (row['price'], row['liquidity'], pair_address,
idx))
                cursor.close()

            conn.commit()

        except Exception as e:
            print(f"Error imputing prices: {e}")
        finally:
            conn.close()
```

2.3 Feature Engineering for MEV Strategies

2.3.1 Market Microstructure Features

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

@dataclass
class MarketMicrostructureFeatures:
    """Market microstructure features for MEV analysis"""
    spread: float
    bid_ask_spread: float
    market_depth: float
    price_impact: float
    order_flow_imbalance: float
    volatility: float
    volume_weighted_price: float
    trade_intensity: float
    block_utilization: float

class MicrostructureFeatureEngine:
    """Generates market microstructure features from raw data"""

    def __init__(self, window_size: int = 100):
        self.window_size = window_size

    def calculate_spread_features(self, price_data: pd.DataFrame) ->
pd.DataFrame:
        """Calculate bid-ask spread and related features"""
        df = price_data.copy()

        # Calculate mid price
        df['mid_price'] = (df['price'] + df['price'].shift(1)) / 2

        # Estimate bid-ask spread using price volatility
        price_changes = df['price'].pct_change().abs()
        df['spread_estimate'] = 2 * price_changes * df['mid_price']
        df['spread_ratio'] = df['spread_estimate'] / df['mid_price']

        # Rolling statistics
        df['spread_ma'] =
df['spread_estimate'].rolling(window=20).mean()
        df['spread_std'] =
df['spread_estimate'].rolling(window=20).std()
```

```
        df['spread_zscore'] = (df['spread_estimate'] -
df['spread_ma']) / df['spread_std']

        return df

    def calculate_impact_features(self, trade_data: pd.DataFrame) ->
pd.DataFrame:
        """Calculate price impact features"""
        df = trade_data.copy()

        # Volume-weighted average price
        df['vwap'] = (df['price'] * df['volume']).cumsum() /
df['volume'].cumsum()

        # Price impact per unit volume
        df['impact_per_volume'] = df['price'].pct_change() /
df['volume'].pct_change()

        # Market depth approximation
        df['market_depth'] = df['liquidity'] / df['volume']

        # Order flow imbalance (simplified)
        if 'buy_volume' in df.columns and 'sell_volume' in df.columns:
            df['order_flow_imbalance'] = (df['buy_volume'] -
df['sell_volume']) / (df['buy_volume'] + df['sell_volume'])
        else:
            # Estimate using price momentum
            df['order_flow_imbalance'] = np.where(df['price'] >
df['price'].shift(1), 1, -1)

        return df

    def calculate_volatility_features(self, price_data: pd.DataFrame) -
> pd.DataFrame:
        """Calculate various volatility measures"""
        df = price_data.copy()

        # Realized volatility (annualized)
        df['returns'] = df['price'].pct_change()
        df['realized_vol'] = df['returns'].rolling(window=20).std() *
np.sqrt(365 * 24)
```

```
        # GARCH-like volatility
        df['vol_proxy'] = df['returns'] ** 2
        df['conditional_vol'] =
df['vol_proxy'].rolling(window=20).mean()

        # Volatility clustering indicator
        vol_ma = df['realized_vol'].rolling(window=50).mean()
        df['vol_clustering'] = (df['realized_vol'] >
vol_ma).astype(int)

        # Jump detection
        returns = df['returns'].dropna()
        threshold = 3 * returns.std()
        df['is_jump'] = (abs(returns) > threshold).astype(int)

        return df

    def calculate_block_utilization_features(self, block_data:
pd.DataFrame) -> pd.DataFrame:
        """Calculate Ethereum block utilization features"""
        df = block_data.copy()

        # Block gas utilization ratio
        df['gas_utilization'] = df['gas_used'] / df['gas_limit']

        # Rolling utilization statistics
        df['utilization_ma'] =
df['gas_utilization'].rolling(window=50).mean()
        df['utilization_std'] =
df['gas_utilization'].rolling(window=50).std()

        # Congestion indicator
        df['is_congested'] = (df['gas_utilization'] > 0.9).astype(int)

        # Block timing features
        if 'block_time' in df.columns:
            df['block_time_volatility'] =
df['block_time'].rolling(window=20).std()

        return df

class MEVOpportunityFeatures:
```

```
    """Generate features specifically for MEV opportunity detection"""

    def __init__(self):
        self.price_impact_threshold = 0.01  # 1%
        self.gas_price_threshold = 20_000_000_000  # 20 gwei

    def calculate_arbitrage_features(self, dex_data: Dict[str,
pd.DataFrame]) -> pd.DataFrame:
        """Calculate arbitrage opportunity features across DEXs"""
        features = []

        # Get all token pairs
        all_tokens = set()
        for dex_name, data in dex_data.items():
            if not data.empty:
                all_tokens.update([data['token0'].iloc[0],
data['token1'].iloc[0]])

        for token in all_tokens:
            # Get prices from different DEXs
            prices = {}
            for dex_name, data in dex_data.items():
                if not data.empty and token in data['token'].values:
                    price_data = data[data['token'] == token]['price']
                    if not price_data.empty:
                        prices[dex_name] = price_data.iloc[-1]

            if len(prices) >= 2:
                max_price = max(prices.values())
                min_price = min(prices.values())

                arbitrage_features = {
                    'token': token,
                    'max_price': max_price,
                    'min_price': min_price,
                    'price_spread': max_price - min_price,
                    'price_spread_pct': (max_price - min_price) /
min_price,
                    'num_dexes': len(prices),
                    'arbitrage_opportunity': (max_price - min_price) /
min_price > 0.001,  # 0.1% threshold
                    'timestamp': datetime.now()
```

```
                }

                features.append(arbitrage_features)

        return pd.DataFrame(features)

    def calculate_liquidation_features(self, lending_data:
pd.DataFrame) -> pd.DataFrame:
        """Calculate liquidation opportunity features"""
        if lending_data.empty:
            return pd.DataFrame()

        df = lending_data.copy()

        # Calculate health factor
        df['health_factor'] = df['collateral_value'] / df['debt_value']

        # Identify near-liquidation positions
        df['near_liquidation'] = (df['health_factor'] <
1.1).astype(int)

        # Calculate liquidation potential
        df['liquidation_bonus'] = 0.08  # Typical liquidation bonus
        df['liquidation_value'] = df['debt_value'] * (1 +
df['liquidation_bonus'])

        # Market impact estimation
        df['estimated_impact'] = df['liquidation_value'] /
df['total_liquidity']

        # Prioritization score
        df['priority_score'] = (df['liquidation_value'] * (2 -
df['health_factor'])) / (1 + df['estimated_impact'])

        return df

    def calculate_sandwich_features(self, mempool_data: pd.DataFrame) -
> pd.DataFrame:
        """Calculate sandwich attack opportunity features"""
        if mempool_data.empty:
            return pd.DataFrame()
```

```
        df = mempool_data.copy()

        # Identify large transactions
        df['is_large_tx'] = (df['value_usd'] > 100000).astype(int)  #
>$100k

        # Gas price premium analysis
        df['gas_premium'] = df['gas_price'] / df['market_gas_price']

        # Profit estimation (simplified)
        if 'slippage' in df.columns:
            df['estimated_profit'] = df['value_usd'] * df['slippage'] *
0.1  # 10% of slippage

        # MEV bot competition analysis
        df['mev_competition'] = df['gas_price'] >
(df['market_gas_price'] * 1.5)

        # Execution timing analysis
        if 'block_number' in df.columns:
            df['time_to_block'] = df['block_number'] -
df['current_block']

        return df

class FeaturePipeline:
    """Orchestrates feature generation and processing"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.microstructure_engine = MicrostructureFeatureEngine()
        self.mev_features = MEVOpportunityFeatures()
        self.feature_cache = {}

    def generate_all_features(self, data_dict: Dict[str, Any]) ->
Dict[str, pd.DataFrame]:
        """Generate all features from raw data"""
        features = {}

        # Market microstructure features
        if 'price_data' in data_dict:
            microstructure_features =
```

```
self.microstructure_engine.calculate_spread_features(
                data_dict['price_data']
            )
            features['microstructure'] = microstructure_features

        # Volatility features
        if 'price_data' in data_dict:
            volatility_features =
self.microstructure_engine.calculate_volatility_features(
                data_dict['price_data']
            )
            features['volatility'] = volatility_features

        # Block utilization features
        if 'block_data' in data_dict:
            block_features =
self.microstructure_engine.calculate_block_utilization_features(
                data_dict['block_data']
            )
            features['block_utilization'] = block_features

        # MEV opportunity features
        if 'dex_data' in data_dict:
            arbitrage_features =
self.mev_features.calculate_arbitrage_features(
                data_dict['dex_data']
            )
            features['arbitrage'] = arbitrage_features

        if 'lending_data' in data_dict:
            liquidation_features =
self.mev_features.calculate_liquidation_features(
                data_dict['lending_data']
            )
            features['liquidation'] = liquidation_features

        if 'mempool_data' in data_dict:
            sandwich_features =
self.mev_features.calculate_sandwich_features(
                data_dict['mempool_data']
            )
            features['sandwich'] = sandwich_features
```

```
        return features

    def select_features_for_model(self, features: Dict[str,
pd.DataFrame],
                                  model_type: str = 'arbitrage') ->
List[str]:
        """Select relevant features for specific model types"""

        feature_sets = {
            'arbitrage': [
                'price_spread_pct', 'volatility', 'gas_utilization',
                'market_depth', 'liquidity'
            ],
            'liquidation': [
                'health_factor', 'liquidation_value',
'estimated_impact',
                'priority_score', 'gas_premium'
            ],
            'sandwich': [
                'estimated_profit', 'mev_competition', 'gas_premium',
                'slippage', 'time_to_block'
            ]
        }

        available_features = []
        if model_type in feature_sets:
            for feature_name in feature_sets[model_type]:
                for df_name, df in features.items():
                    if feature_name in df.columns:
                        available_features.append(f"{df_name}.
{feature_name}")
                        break

        return available_features
```

2.4 Data Pipeline Automation

2.4.1 Continuous Data Ingestion

```
import schedule
import time
from threading import Thread
from queue import Queue
from typing import Callable, Any
import logging

class DataPipelineManager:
    """Manages automated data collection and processing pipelines"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.database = BlockchainDatabase(config['database_path'])
        self.validators = DataValidator(self.database)
        self.feature_pipeline = FeaturePipeline(config.get('features',
{}))

        self.running = False
        self.data_queues = {
            'transactions': Queue(),
            'price_data': Queue(),
            'block_data': Queue()
        }

        self.setup_logging()

    def setup_logging(self):
        """Setup logging for the pipeline"""
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s',
            handlers=[
                logging.FileHandler('data_pipeline.log'),
                logging.StreamHandler()
            ]
        )
        self.logger = logging.getLogger('DataPipeline')

    def start_pipeline(self):
        """Start all data collection and processing pipelines"""
        self.running = True
```

```
        # Start data collection threads
        Thread(target=self._collect_blockchain_data,
daemon=True).start()
        Thread(target=self._collect_dex_data, daemon=True).start()
        Thread(target=self._process_data, daemon=True).start()
        Thread(target=self._validate_and_clean, daemon=True).start()
        Thread(target=self._generate_features, daemon=True).start()

        self.logger.info("Data pipeline started successfully")

    def stop_pipeline(self):
        """Stop all pipelines"""
        self.running = False
        self.logger.info("Data pipeline stopped")

    def _collect_blockchain_data(self):
        """Collect blockchain data continuously"""
        collector =
BlockchainDataCollector(self.config['provider_urls'])

        while self.running:
            try:
                # Get latest block number
                w3 = collector.get_current_provider()
                latest_block = w3.eth.block_number

                # Collect recent blocks
                start_block = latest_block - 100  # Collect last 100
blocks
                for block_num in range(start_block, latest_block):
                    transactions =
collector.collect_transactions_by_block(block_num)

                    for tx in transactions:
                        self.data_queues['transactions'].put(tx)

                    time.sleep(0.1)  # Rate limiting

                time.sleep(12)  # Wait for next block

            except Exception as e:
```

```
                self.logger.error(f"Error in blockchain data
collection: {e}")
                time.sleep(30)

    def _collect_dex_data(self):
        """Collect DEX data continuously"""
        dex_collector = DEXDataCollector()

        while self.running:
            try:
                # Collect Uniswap V2 pairs
                v2_pairs =
asyncio.run(dex_collector.get_uniswap_v2_pairs(100))
                for pair in v2_pairs:
                    # Process pair data
                    price_data = {
                        'pair_address': pair['id'],
                        'price': float(pair.get('token0Price', 0)),
                        'liquidity': float(pair.get('reserveUSD', 0)),
                        'volume_24h': float(pair.get('volumeUSD', 0)),
                        'timestamp': datetime.now()
                    }
                    self.data_queues['price_data'].put(price_data)

                # Collect Uniswap V3 pools
                v3_pools =
asyncio.run(dex_collector.get_uniswap_v3_pools(100))
                for pool in v3_pools:
                    price_data = {
                        'pair_address': pool['id'],
                        'price': float(pool.get('token0Price', 0)),
                        'liquidity':
float(pool.get('totalValueLockedUSD', 0)),
                        'volume_24h': float(pool.get('volumeUSD', 0)),
                        'timestamp': datetime.now()
                    }
                    self.data_queues['price_data'].put(price_data)

                time.sleep(60)  # Update every minute

            except Exception as e:
                self.logger.error(f"Error in DEX data collection: {e}")
```

```
                time.sleep(60)

    def _process_data(self):
        """Process data from queues and store in database"""
        while self.running:
            try:
                # Process transactions
                while not self.data_queues['transactions'].empty():
                    tx = self.data_queues['transactions'].get()
                    self.database.insert_transaction(tx)

                # Process price data
                while not self.data_queues['price_data'].empty():
                    price_data = self.data_queues['price_data'].get()
                    self.database.insert_price_data(
                        price_data['pair_address'],
                        'token0_address',  # Would need proper token
mapping
                        'token1_address',
                        price_data['price'],
                        price_data['liquidity'],
                        price_data['volume_24h'],
                        18000000,  # Current block number
                        price_data['timestamp']
                    )

                time.sleep(1)

            except Exception as e:
                self.logger.error(f"Error in data processing: {e}")
                time.sleep(5)

    def _validate_and_clean(self):
        """Continuously validate and clean data"""
        while self.running:
            try:
                # Validate recent transaction data
                end_time = datetime.now()
                start_time = end_time - timedelta(hours=1)

                report =
self.validators.validate_transactions(start_time, end_time)
```

```
                if report.consistency_score < 0.8:

self.logger.warning(f"Data quality issues detected:
{report.recommendations}")

                    # Clean problematic data
                    cleaner = DataCleaner(self.database)
                    stats = cleaner.clean_transactions(start_time,
end_time)
                    self.logger.info(f"Cleaned data: {stats}")

                time.sleep(300)  # Validate every 5 minutes

            except Exception as e:
                self.logger.error(f"Error in data validation: {e}")
                time.sleep(300)

    def _generate_features(self):
        """Generate and store features continuously"""
        while self.running:
            try:
                # Get recent data for feature generation
                end_time = datetime.now()
                start_time = end_time - timedelta(hours=1)

                # Prepare data dictionary
                data_dict = {}

                # Get price data
                price_df =
self.database.get_price_series('pair_address', start_time, end_time)
                if not price_df.empty:
                    data_dict['price_data'] = price_df

                # Generate features
                features =
self.feature_pipeline.generate_all_features(data_dict)

                # Store features (implementation depends on your
feature storage needs)
                # This could be saved to database, files, or passed to
```

```
ML models

                self.logger.info(f"Generated features for
{len(features)} feature groups")

                time.sleep(300)  # Generate features every 5 minutes

            except Exception as e:
                self.logger.error(f"Error in feature generation: {e}")
                time.sleep(300)

# Scheduled pipeline management
class ScheduledPipelineManager(DataPipelineManager):
    """Pipeline manager with scheduled tasks"""

    def __init__(self, config: Dict[str, Any]):
        super().__init__(config)
        self.scheduled_tasks = []

    def setup_scheduled_tasks(self):
        """Setup scheduled data collection tasks"""

        # Daily comprehensive data collection
        schedule.every().day.at("00:00").do(self._daily_data_backup)

        # Hourly data validation
        schedule.every().hour.do(self._hourly_data_validation)

        # Weekly feature generation

schedule.every().monday.at("02:00").do(self._weekly_feature_generation)

        # Monthly data archival
        schedule.every().month.do(self._monthly_data_archival)

    def _daily_data_backup(self):
        """Daily data backup routine"""
        self.logger.info("Starting daily data backup")

        try:
            # Export recent data
            end_time = datetime.now()
```

```
            start_time = end_time - timedelta(days=1)

            transactions_df =
self.database.get_transactions_by_time_range(start_time, end_time)

            if not transactions_df.empty:
                # Save to CSV
                filename =
f"backup_transactions_{end_time.strftime('%Y%m%d')}.csv"
                transactions_df.to_csv(filename, index=False)
                self.logger.info(f"Backed up {len(transactions_df)}
transactions to {filename}")

        except Exception as e:
            self.logger.error(f"Error in daily backup: {e}")

    def _hourly_data_validation(self):
        """Hourly data validation"""
        self.logger.info("Starting hourly data validation")

        end_time = datetime.now()
        start_time = end_time - timedelta(hours=1)

        report = self.validators.validate_transactions(start_time,
end_time)

        if report.consistency_score < 0.9:
            self.logger.warning(f"Data quality alert:
{report.recommendations}")

    def _weekly_feature_generation(self):
        """Weekly comprehensive feature generation"""
        self.logger.info("Starting weekly feature generation")

        # Generate features for the past week
        end_time = datetime.now()
        start_time = end_time - timedelta(weeks=1)

        # Get all pairs for feature generation
        pools_df = self.database.get_dex_pools()

        for _, pool in pools_df.iterrows():
```

```
            price_series = self.database.get_price_series(
                pool['pool_address'], start_time, end_time
            )

            if len(price_series) > 100:  # Minimum data points required
                features =
self.feature_pipeline.generate_all_features({'price_data':
price_series})
                # Store features for ML training

    def _monthly_data_archival(self):
        """Monthly data archival"""
        self.logger.info("Starting monthly data archival")

        # Archive data older than 3 months
        cutoff_date = datetime.now() - timedelta(days=90)

        compressor = DataCompressionManager(self.database)
        compressor.compress_old_data(cutoff_date)

    def run_scheduler(self):
        """Run the scheduled tasks"""
        self.setup_scheduled_tasks()

        while self.running:
            schedule.run_pending()
            time.sleep(60)  # Check every minute

    def start_complete_pipeline(self):
        """Start both continuous and scheduled pipelines"""
        self.start_pipeline()

        # Start scheduler in separate thread
        scheduler_thread = Thread(target=self.run_scheduler,
daemon=True)
        scheduler_thread.start()

        self.logger.info("Complete pipeline started with scheduled
tasks")
```

2.5 Practical Exercise: Build Complete Data Pipeline

```
def build_data_collection_pipeline():
    """Complete exercise to build a data collection pipeline"""

    # Configuration
    config = {
        'database_path': 'blockchain_data.db',
        'provider_urls': [
            'https://eth-mainnet.g.alchemy.com/v2/YOUR_KEY',
            'https://mainnet.infura.io/v3/YOUR_KEY'
        ],
        'features': {
            'window_size': 100,
            'arbitrage_threshold': 0.001
        }
    }

    print("Building data collection pipeline...")

    # Initialize database
    database = BlockchainDatabase(config['database_path'])
    print("✅ Database initialized")

    # Initialize pipeline manager
    pipeline = ScheduledPipelineManager(config)
    print("✅ Pipeline manager initialized")

    # Start data collection (in test mode for 30 seconds)
    print("Starting data collection test...")

    # Test blockchain data collection
    collector = BlockchainDataCollector(config['provider_urls'])

    # Collect a few recent blocks for testing
    w3 = collector.get_current_provider()
    current_block = w3.eth.block_number

    print(f"Current block: {current_block}")

    # Test data collection
    test_block = current_block - 1
    transactions = collector.collect_transactions_by_block(test_block)
```

```
    print(f"✅ Collected {len(transactions)} transactions from block
{test_block}")

    # Test database operations
    for tx in transactions[:5]:  # Insert first 5 transactions
        database.insert_transaction(tx)

    print("✅ Database insertion successful")

    # Test data validation
    validator = DataValidator(database)

    end_time = datetime.now()
    start_time = end_time - timedelta(hours=1)

    quality_report = validator.validate_transactions(start_time,
end_time)

    print(f"✅ Data quality check completed")
    print(f"   Total records: {quality_report.total_records}")
    print(f"   Consistency score: {quality_report.consistency_score:.
2f}")

    # Test feature generation
    print("\nTesting feature generation...")

    # Generate sample price data for testing
    sample_price_data = pd.DataFrame({
        'timestamp': pd.date_range(start='2024-01-01', periods=100,
freq='H'),
        'price': np.random.normal(2000, 50, 100),
        'liquidity': np.random.uniform(100000, 1000000, 100),
        'volume': np.random.uniform(1000, 10000, 100)
    })

    feature_engine = MicrostructureFeatureEngine()
    features =
feature_engine.calculate_spread_features(sample_price_data)

    print(f"✅ Generated {len(features.columns)} features from sample
data")
```

```
    # Test DEX data collection
    print("\nTesting DEX data collection...")

    dex_collector = DEXDataCollector()

    try:
        # This would normally require internet connection

print("✅ DEX collector initialized (would fetch real data with
internet)")
    except Exception as e:
        print(f"⚠️  DEX collection test skipped: {e}")

    print("\n🎉 Data collection pipeline test completed successfully!")

    print("\nPipeline components:")
    print("- ✅ Blockchain data collection")
    print("- ✅ Database storage and management")
    print("- ✅ Data validation and quality checks")
    print("- ✅ Feature engineering")
    print("- ✅ Automated pipeline management")
    print("- ✅ Scheduled tasks for maintenance")

    return {
        'database': database,
        'pipeline': pipeline,
        'test_results': {
            'transactions_collected': len(transactions),
            'database_inserted': 5,
            'quality_score': quality_report.consistency_score,
            'features_generated': len(features.columns)
        }
    }

# Run the exercise
if __name__ == "__main__":
    pipeline_results = build_data_collection_pipeline()
    print("\nData collection pipeline is ready for production use!")
```

Module Summary

In this module, you have learned to build comprehensive data collection and processing systems for MEV strategies. The pipeline includes:

- **Blockchain Data Collection**: Web3 and API-based data extraction from Ethereum
- **Real-time Data Streaming**: WebSocket connections for live blockchain data
- **Database Management**: SQLite-based storage with proper indexing and compression
- **Data Quality Validation**: Automated checks and cleaning procedures
- **Feature Engineering**: Market microstructure and MEV-specific features
- **Automated Pipelines**: Continuous data collection with scheduled maintenance tasks

Key Takeaways

1. **Data Quality First**: Always validate and clean raw blockchain data before use
2. **Realistic Sampling**: Account for network delays, gas prices, and block times
3. **Feature Engineering**: Create meaningful features specific to MEV opportunities
4. **Pipeline Automation**: Build robust, automated systems for continuous operation
5. **Monitoring and Alerting**: Implement quality checks and failure detection
6. **Scalable Architecture**: Design for high-throughput data processing

Next Steps

1. **Production Deployment**: Deploy the pipeline with proper monitoring and alerting
2. **Data Lake Integration**: Extend to use cloud storage solutions
3. **ML Pipeline Integration**: Connect features directly to machine learning models
4. **Performance Optimization**: Implement caching and batch processing
5. **Multi-chain Support**: Extend to other blockchain networks

This data pipeline provides the foundation for high-quality, real-time data needed for sophisticated MEV strategy development and execution.