Module 3: Real-time Data Streaming

Master WebSocket connections, event processing, and high-frequency data handling for MEV systems

Course Overview

Duration: 150 minutes
Level: Advanced
Prerequisites: Modules 1 & 2, understanding of blockchain data structures and MEV concepts
Certificate: Available upon completion

Learning Objectives

By the end of this module, you will be able to:
- Build high-performance WebSocket clients for real-time blockchain data
- Implement sophisticated event processing and filtering systems
- Design efficient data pipelines for MEV opportunity detection
- Create robust connection management with automatic reconnection
- Build real-time price feeds and market data aggregation systems
- Implement low-latency transaction monitoring and analysis

Table of Contents

---

1. Real-time Data Architecture

Core Data Streaming Concepts

```
import asyncio
import websockets
import json
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Callable, Any, Union, Set
from enum import Enum
import logging
import time
from datetime import datetime, timedelta
from collections import deque, defaultdict
import threading
from abc import ABC, abstractmethod

class DataType(Enum):
    BLOCK = "block"
    TRANSACTION = "transaction"
    LOG = "log"
    MEMPOOL = "mempool"
    PRICE = "price"
    LIQUIDATION = "liquidation"
    ARBITRAGE = "arbitrage"

class EventPriority(Enum):
    HIGH = 1
    NORMAL = 2
    LOW = 3

@dataclass
class DataEvent:
    """Represents a real-time data event"""
    event_type: DataType
    priority: EventPriority
    data: Dict[str, Any]
    timestamp: datetime = field(default_factory=datetime.utcnow)
    processed: bool = False
    source: str = ""
    metadata: Dict[str, Any] = field(default_factory=dict)

    @property
    def age_ms(self) -> float:
        return (datetime.utcnow() - self.timestamp).total_seconds() *
1000
```

```
    def to_dict(self) -> Dict[str, Any]:
        return {
            'event_type': self.event_type.value,
            'priority': self.priority.value,
            'data': self.data,
            'timestamp': self.timestamp.isoformat(),
            'source': self.source,
            'metadata': self.metadata
        }

@dataclass
class StreamConfig:
    """Configuration for data streaming"""
    name: str
    data_types: Set[DataType]
    filters: Dict[str, Any]
    buffer_size: int = 1000
    max_processing_time_ms: float = 100.0
    retry_attempts: int = 3
    heartbeat_interval: float = 30.0
    compression: bool = True

class DataStreamError(Exception):
    """Base exception for data streaming operations"""
    pass

class StreamConnectionError(DataStreamError):
    """Raised when stream connection fails"""
    pass

class ProcessingTimeoutError(DataStreamError):
    """Raised when event processing exceeds timeout"""
    pass

class DataProcessor(ABC):
    """Abstract base class for data processors"""

    def __init__(self, config: StreamConfig):
        self.config = config
        self.logger = logging.getLogger(f"{__name__}.{config.name}")
        self.processing_stats = {
```

```
            'events_processed': 0,
            'events_failed': 0,
            'processing_time_avg': 0.0,
            'last_processed': None
        }

    @abstractmethod
    async def process_event(self, event: DataEvent) ->
Optional[DataEvent]:
        """Process a data event and return result"""
        pass

    def update_stats(self, processing_time: float, success: bool):
        """Update processing statistics"""
        self.processing_stats['events_processed'] += 1
        if not success:
            self.processing_stats['events_failed'] += 1

        # Update average processing time
        total_processed = self.processing_stats['events_processed']
        current_avg = self.processing_stats['processing_time_avg']
        self.processing_stats['processing_time_avg'] = (
            (current_avg * (total_processed - 1) + processing_time) /
total_processed
        )

        self.processing_stats['last_processed'] = datetime.utcnow()

class RealtimeDataStream:
    """High-performance real-time data streaming system"""

    def __init__(self, name: str):
        self.name = name
        self.logger = logging.getLogger(__name__)

        # Event queues
        self.event_queue = asyncio.Queue(maxsize=10000)
        self.priority_queues = {
            EventPriority.HIGH: asyncio.Queue(maxsize=1000),
            EventPriority.NORMAL: asyncio.Queue(maxsize=5000),
            EventPriority.LOW: asyncio.Queue(maxsize=2000)
        }
```

```
        # Processors
        self.processors: List[DataProcessor] = []

        # Event handlers
        self.event_handlers: Dict[DataType, List[Callable]] =
defaultdict(list)

        # State management
        self.is_running = False
        self.connection_manager = None

        # Performance monitoring
        self.metrics = {
            'events_received': 0,
            'events_processed': 0,
            'events_dropped': 0,
            'avg_processing_time': 0.0,
            'current_queue_size': 0,
            'peak_queue_size': 0
        }

        # Statistics
        self.stats = {
            'start_time': None,
            'uptime': 0.0,
            'total_events': 0,
            'error_count': 0
        }

    def add_processor(self, processor: DataProcessor):
        """Add data processor to the stream"""
        self.processors.append(processor)
        self.logger.info(f"Added processor: {processor.config.name}")

    def add_event_handler(self, event_type: DataType, handler:
Callable):
        """Add event handler for specific data type"""
        self.event_handlers[event_type].append(handler)
        self.logger.info(f"Added handler for {event_type.value}")

    async def start(self):
```

```
        """Start the data stream"""
        if self.is_running:
            return

        self.is_running = True
        self.stats['start_time'] = datetime.utcnow()

        # Start processing tasks
        tasks = [
            asyncio.create_task(self._event_processor_loop()),
            asyncio.create_task(self._metrics_collector_loop()),
            asyncio.create_task(self._health_checker_loop())
        ]

        self.logger.info(f"Data stream '{self.name}' started")

        try:
            await asyncio.gather(*tasks)
        except Exception as e:
            self.logger.error(f"Data stream error: {e}")
        finally:
            await self.stop()

    async def stop(self):
        """Stop the data stream"""
        self.is_running = False

        # Clear queues
        for queue in self.priority_queues.values():
            while not queue.empty():
                try:
                    queue.get_nowait()
                except asyncio.QueueEmpty:
                    break

        self.logger.info(f"Data stream '{self.name}' stopped")

    async def emit_event(self, event: DataEvent):
        """Emit an event to the stream"""
        try:
            # Route to priority queue
            if event.priority in self.priority_queues:
```

```
                queue = self.priority_queues[event.priority]

                # Try to add to queue without blocking
                try:
                    queue.put_nowait(event)
                except asyncio.QueueFull:
                    # Drop lowest priority event if full
                    self._drop_lowest_priority_event()
                    queue.put_nowait(event)  # Retry
            else:
                # Default to normal priority

self.priority_queues[EventPriority.NORMAL].put_nowait(event)

            self.metrics['events_received'] += 1
            self.stats['total_events'] += 1

        except Exception as e:
            self.logger.error(f"Error emitting event: {e}")
            self.metrics['events_dropped'] += 1

    def _drop_lowest_priority_event(self):
        """Drop lowest priority event when queue is full"""
        low_queue = self.priority_queues[EventPriority.LOW]

        try:
            low_queue.get_nowait()
            self.metrics['events_dropped'] += 1
        except asyncio.QueueEmpty:
            pass  # Nothing to drop

    async def _event_processor_loop(self):
        """Main event processing loop"""
        while self.is_running:
            try:
                # Get next event from highest priority queue
                event = await self._get_next_event()

                if event:
                    start_time = time.time()

                    try:
```

```
                        # Process through all processors
                        processed_event = await
self._process_event_chain(event)

                        # Notify handlers if processing succeeded
                        if processed_event:
                            await
self._notify_handlers(processed_event)

                        processing_time = (time.time() - start_time) *
1000
                        self.metrics['events_processed'] += 1
                        self.metrics['avg_processing_time'] = (
                            (self.metrics['avg_processing_time'] * 0.9)
+ (processing_time * 0.1)
                        )

                    except Exception as e:
                        self.logger.error(f"Event processing error:
{e}")
                        self.stats['error_count'] += 1

                # Update queue metrics
                total_queue_size = sum(q.qsize() for q in
self.priority_queues.values())
                self.metrics['current_queue_size'] = total_queue_size
                self.metrics['peak_queue_size'] = max(
                    self.metrics['peak_queue_size'],
                    total_queue_size
                )

            except Exception as e:
                self.logger.error(f"Event processor loop error: {e}")
                await asyncio.sleep(1)

    async def _get_next_event(self) -> Optional[DataEvent]:
        """Get next event from highest priority non-empty queue"""
        # Check queues in priority order
        for priority in [EventPriority.HIGH, EventPriority.NORMAL,
EventPriority.LOW]:
            queue = self.priority_queues[priority]
```

```
            try:
                event = await asyncio.wait_for(queue.get(),
timeout=0.1)
                return event
            except asyncio.TimeoutError:
                continue

        return None

    async def _process_event_chain(self, event: DataEvent) ->
Optional[DataEvent]:
        """Process event through all configured processors"""
        processed_event = event

        for processor in self.processors:
            try:
                start_time = time.time()

                processed_event = await asyncio.wait_for(
                    processor.process_event(processed_event),
                    timeout=processor.config.max_processing_time_ms /
1000.0
                )

                processing_time = (time.time() - start_time) * 1000
                processor.update_stats(processing_time, processed_event
is not None)

                if processed_event is None:
                    break  # Event was filtered out

            except asyncio.TimeoutError:
                self.logger.warning(f"Processor
{processor.config.name} timed out")

processor.update_stats(processor.config.max_processing_time_ms, False)
                break
            except Exception as e:
                self.logger.error(f"Processor {processor.config.name}
error: {e}")
                processor.update_stats(0, False)
                break
```

```
        return processed_event

    async def _notify_handlers(self, event: DataEvent):
        """Notify all registered handlers for event type"""
        handlers = self.event_handlers.get(event.event_type, [])

        if not handlers:
            return

        # Notify handlers concurrently but don't wait for all
        tasks = []
        for handler in handlers:
            task = asyncio.create_task(self._safe_handler_call(handler,
event))
            tasks.append(task)

        # Wait for a short time, then cancel remaining tasks
        try:
            await asyncio.wait_for(
                asyncio.gather(*tasks, return_exceptions=True),
                timeout=0.1
            )
        except asyncio.TimeoutError:
            for task in tasks:
                if not task.done():
                    task.cancel()

    async def _safe_handler_call(self, handler: Callable, event:
DataEvent):
        """Safely call event handler"""
        try:
            if asyncio.iscoroutinefunction(handler):
                await handler(event)
            else:
                handler(event)
        except Exception as e:
            self.logger.error(f"Event handler error: {e}")

    async def _metrics_collector_loop(self):
        """Collect and log metrics periodically"""
        while self.is_running:
```

```
            try:
                # Calculate uptime
                if self.stats['start_time']:
                    self.stats['uptime'] = (datetime.utcnow() -
self.stats['start_time']).total_seconds()

                # Log metrics every 60 seconds
                if int(self.stats['uptime']) % 60 == 0:
                    self.logger.info(f"Stream metrics: {self.metrics}")

                await asyncio.sleep(10)

            except Exception as e:
                self.logger.error(f"Metrics collection error: {e}")
                await asyncio.sleep(30)

    async def _health_checker_loop(self):
        """Monitor stream health and performance"""
        while self.is_running:
            try:
                # Check queue sizes
                for priority, queue in self.priority_queues.items():
                    if queue.qsize() > queue.maxsize * 0.8:  # 80% full
                        self.logger.warning(f"High queue usage:
{priority.name} queue is {queue.qsize()}/{queue.maxsize}")

                # Check processing delays
                if self.metrics['avg_processing_time'] > 1000:  # 1
second
                    self.logger.warning(f"High processing time:
{self.metrics['avg_processing_time']:.2f}ms")

                # Check error rate
                if self.metrics['events_processed'] > 0:
                    error_rate = self.stats['error_count'] /
self.metrics['events_processed']
                    if error_rate > 0.05:  # 5% error rate
                        self.logger.warning(f"High error rate:
{error_rate:.2%}")

                await asyncio.sleep(30)
```

```
            except Exception as e:
                self.logger.error(f"Health checker error: {e}")
                await asyncio.sleep(60)

    def get_stats(self) -> Dict[str, Any]:
        """Get comprehensive stream statistics"""
        return {
            'name': self.name,
            'is_running': self.is_running,
            'metrics': self.metrics.copy(),
            'stats': self.stats.copy(),
            'processor_count': len(self.processors),
            'handler_counts': {
                event_type.value: len(handlers)
                for event_type, handlers in self.event_handlers.items()
            },
            'queue_sizes': {
                priority.name: queue.qsize()
                for priority, queue in self.priority_queues.items()
            }
        }
```

---

2. WebSocket Implementation

Advanced WebSocket Client

```
import websockets
import ssl
from urllib.parse import urlparse
import socket

class WebSocketStreamManager:
    """Advanced WebSocket manager for real-time blockchain data"""

    def __init__(self, stream: RealtimeDataStream):
        self.stream = stream
        self.logger = logging.getLogger(__name__)

        # WebSocket connections
        self.connections: Dict[str, websockets.WebSocketServerProtocol]
= {}
        self.connection_configs: Dict[str, Dict[str, Any]] = {}

        # Subscription management
        self.subscriptions: Dict[str, Dict] = {}

        # Connection statistics
        self.connection_stats = {
            'total_connections': 0,
            'successful_connections': 0,
            'failed_connections': 0,
            'reconnection_attempts': 0,
            'last_connected': None
        }

        # Heartbeat management
        self.heartbeat_task = None
        self.last_heartbeat_times: Dict[str, datetime] = {}

    async def add_connection(self,
                             connection_id: str,
                             endpoint: str,
                             subscriptions: List[Dict[str, Any]],
                             auth_token: Optional[str] = None):
        """Add WebSocket connection configuration"""

        config = {
            'endpoint': endpoint,
```

```
            'subscriptions': subscriptions,
            'auth_token': auth_token,
            'max_reconnect_attempts': 10,
            'reconnect_delay': 5.0,
            'heartbeat_interval': 30.0,
            'connection_timeout': 10.0,
            'message_queue_size': 1000
        }

        self.connection_configs[connection_id] = config
        self.logger.info(f"Added WebSocket connection: {connection_id}
-> {endpoint}")

    async def start_all_connections(self):
        """Start all configured WebSocket connections"""

        tasks = []
        for connection_id in self.connection_configs:
            task = asyncio.create_task(
                self._manage_connection(connection_id)
            )
            tasks.append(task)

        # Start heartbeat monitor
        self.heartbeat_task =
asyncio.create_task(self._heartbeat_monitor())

        # Wait for initial connections
        await asyncio.sleep(2)

        self.logger.info(f"Started {len(tasks)} WebSocket connections")

    async def _manage_connection(self, connection_id: str):
        """Manage a single WebSocket connection with automatic
reconnection"""

        config = self.connection_configs[connection_id]
        max_attempts = config['max_reconnect_attempts']
        reconnect_delay = config['reconnect_delay']

        for attempt in range(max_attempts):
            try:
```

```
                self.connection_stats['total_connections'] += 1

                # Connect to WebSocket
                await self._connect_to_endpoint(connection_id, config)

                self.connection_stats['successful_connections'] += 1
                self.connection_stats['last_connected'] =
datetime.utcnow()

                # Connected successfully, keep connection alive
                await self._keep_connection_alive(connection_id)

                break  # Connection successful, exit retry loop

            except Exception as e:
                self.connection_stats['failed_connections'] += 1

                self.logger.error(f"Connection attempt {attempt + 1}
failed for {connection_id}: {e}")

                if attempt < max_attempts - 1:
                    self.connection_stats['reconnection_attempts'] += 1

                    # Exponential backoff
                    delay = reconnect_delay * (1.5 ** attempt)
                    self.logger.info(f"Reconnecting {connection_id} in
{delay:.1f} seconds...")
                    await asyncio.sleep(delay)
                else:
                    self.logger.error(f"Max reconnection attempts
reached for {connection_id}")
                    break

    async def _connect_to_endpoint(self, connection_id: str, config:
Dict[str, Any]):
        """Establish WebSocket connection"""

        endpoint = config['endpoint']
        auth_token = config.get('auth_token')
        timeout = config['connection_timeout']

        # Parse endpoint for SSL configuration
```

```
        parsed = urlparse(endpoint)
        ssl_context = None

        if parsed.scheme == 'wss':
            ssl_context = ssl.create_default_context()

# In production, you might want to disable SSL verification for testing
            # ssl_context.check_hostname = False
            # ssl_context.verify_mode = ssl.CERT_NONE

        # Create WebSocket connection
        extra_headers = {}
        if auth_token:
            extra_headers['Authorization'] = f'Bearer {auth_token}'

        # Connection with timeout
        self.connections[connection_id] = await asyncio.wait_for(
            websockets.connect(
                endpoint,
                ssl=ssl_context,
                extra_headers=extra_headers,
                max_size=2**20  # 1MB max message size
            ),
            timeout=timeout
        )

        # Initialize heartbeat tracking
        self.last_heartbeat_times[connection_id] = datetime.utcnow()

        # Subscribe to configured topics
        await self._send_subscriptions(connection_id,
config['subscriptions'])

        self.logger.info(f"Connected to WebSocket: {connection_id}")

    async def _send_subscriptions(self, connection_id: str,
subscriptions: List[Dict[str, Any]]):
        """Send subscription requests"""

        websocket = self.connections[connection_id]

        for subscription in subscriptions:
```

```
            try:
                message = {
                    'jsonrpc': '2.0',
                    'method': subscription['method'],
                    'params': subscription['params'],
                    'id': len(self.subscriptions) + 1
                }

                await websocket.send(json.dumps(message))

                # Track subscription
                self.subscriptions[len(self.subscriptions) + 1] = {
                    'connection_id': connection_id,
                    'subscription': subscription,
                    'timestamp': datetime.utcnow()
                }

                self.logger.debug(f"Sent subscription: {subscription}")

                # Small delay between subscriptions
                await asyncio.sleep(0.1)

            except Exception as e:
                self.logger.error(f"Failed to send subscription: {e}")

    async def _keep_connection_alive(self, connection_id: str):
        """Keep connection alive and handle incoming messages"""

        websocket = self.connections[connection_id]

        try:
            async for message in websocket:
                try:
                    # Parse incoming message
                    data = json.loads(message)

                    # Handle subscription responses
                    if 'method' in data and data['method'] ==
'eth_subscription':
                        await
self._handle_subscription_message(connection_id, data)
```

```
                    # Handle other message types
                    elif 'result' in data:
                        await
self._handle_response_message(connection_id, data)

                    else:
                        await self._handle_other_message(connection_id,
data)

                except json.JSONDecodeError as e:
                    self.logger.error(f"Invalid JSON from
{connection_id}: {e}")

                except Exception as e:
                    self.logger.error(f"Error processing message from
{connection_id}: {e}")

        except websockets.exceptions.ConnectionClosed:
            self.logger.warning(f"WebSocket connection closed:
{connection_id}")
            raise

        except Exception as e:
            self.logger.error(f"WebSocket error for {connection_id}:
{e}")
            raise

    async def _handle_subscription_message(self, connection_id: str,
data: Dict[str, Any]):
        """Handle subscription notification message"""

        subscription_id = data['params']['subscription']
        result = data['params']['result']

        # Find corresponding subscription
        subscription_info = None
        for sub_id, info in self.subscriptions.items():
            if info.get('subscription_id') == subscription_id:
                subscription_info = info
                break

        if not subscription_info:
```

```
            return

        # Update heartbeat time
        self.last_heartbeat_times[connection_id] = datetime.utcnow()

        # Convert to DataEvent and emit
        event_type =
self._determine_event_type(subscription_info['subscription'])

        event = DataEvent(
            event_type=event_type,
            priority=EventPriority.HIGH if event_type in
[DataType.BLOCK, DataType.TRANSACTION] else EventPriority.NORMAL,
            data=result,
            source=connection_id,
            metadata={
                'subscription_id': subscription_id,
                'connection_id': connection_id
            }
        )

        await self.stream.emit_event(event)

    def _determine_event_type(self, subscription: Dict[str, Any]) ->
DataType:
        """Determine event type from subscription"""

        method = subscription['method']
        params = subscription.get('params', [])

        if 'newHeads' in method:
            return DataType.BLOCK
        elif 'newPendingTransactions' in method:
            return DataType.TRANSACTION
        elif 'logs' in method:
            return DataType.LOG
        else:
            return DataType.MEMPOOL

    async def _handle_response_message(self, connection_id: str, data:
Dict[str, Any]):
        """Handle response to subscription request"""
```

```
        self.logger.debug(f"Received response from {connection_id}:
{data}")

    async def _handle_other_message(self, connection_id: str, data:
Dict[str, Any]):
        """Handle other types of messages"""

        self.logger.debug(f"Received other message from
{connection_id}: {data}")

    async def _heartbeat_monitor(self):
        """Monitor heartbeat times and detect stale connections"""

        while self.stream.is_running:
            try:
                current_time = datetime.utcnow()
                stale_threshold = timedelta(seconds=90)  # 90 seconds

                for connection_id, last_heartbeat in
self.last_heartbeat_times.items():
                    if current_time - last_heartbeat > stale_threshold:
                        self.logger.warning(f"Stale connection
detected: {connection_id}")

                        # Close stale connection
                        if connection_id in self.connections:
                            await
self.connections[connection_id].close()

                await asyncio.sleep(30)

            except Exception as e:
                self.logger.error(f"Heartbeat monitor error: {e}")
                await asyncio.sleep(60)

class EthereumWebSocketClient:
    """Specialized Ethereum WebSocket client for MEV data"""

    def __init__(self, stream: RealtimeDataStream):
        self.stream = stream
        self.ws_manager = WebSocketStreamManager(stream)
```

```
        self.logger = logging.getLogger(__name__)

        # Ethereum-specific subscriptions
        self.ethereum_subscriptions = []

    async def add_ethereum_endpoint(self,
                                    endpoint: str,
                                    api_key: Optional[str] = None):
        """Add Ethereum WebSocket endpoint"""

        # Define comprehensive subscriptions for MEV
        subscriptions = [
            {
                'method': 'eth_subscribe',
                'params': ['newHeads']
            },
            {
                'method': 'eth_subscribe',
                'params': ['newPendingTransactions']
            }
        ]

        # Add specific contract event subscriptions for MEV
        mev_contracts = {
            'uniswap_v2_factory':
'0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f',
            'uniswap_v3_factory':
'0x1F98431c8aD98523631AE4a59f267346ea31F984',
            'aave_pool': '0x87870Bca3F3fD6335C3F4ce8392D69350B4fA4E2'
        }

        for contract_name, contract_address in mev_contracts.items():
            subscriptions.append({
                'method': 'eth_subscribe',
                'params': [
                    'logs',
                    {
                        'address': contract_address,
                        'topics': []  # All events
                    }
                ]
            })
```

```
        await self.ws_manager.add_connection(
            connection_id='ethereum_main',
            endpoint=endpoint,
            subscriptions=subscriptions,
            auth_token=api_key
        )

    async def start_monitoring(self):
        """Start Ethereum real-time monitoring"""

        await self.ws_manager.start_all_connections()

        # Add specialized processors for Ethereum data
        ethereum_processors = [
            MempoolAnalyzer(),
            BlockAnalyzer(),
            LiquidationDetector(),
            ArbitrageDetector()
        ]

        for processor in ethereum_processors:
            self.stream.add_processor(processor)

        self.logger.info("Started Ethereum real-time monitoring")

class BinanceWebSocketClient:
    """Binance WebSocket client for price data"""

    def __init__(self, stream: RealtimeDataStream):
        self.stream = stream
        self.ws_manager = WebSocketStreamManager(stream)
        self.logger = logging.getLogger(__name__)

        # Price data subscriptions
        self.price_subscriptions = {}

    async def add_price_stream(self, symbol: str, data_type: str =
'ticker'):
        """Add price stream for symbol"""

        endpoint = f"wss://stream.binance.com:9443/ws/{symbol.lower()}
```

```
@{data_type}"

        subscription = {
            'method': 'SUBSCRIBE',
            'params': [f"{symbol.lower()}@{data_type}"],
            'id': len(self.price_subscriptions) + 1
        }

        await self.ws_manager.add_connection(
            connection_id=f'binance_{symbol.lower()}',
            endpoint=endpoint,
            subscriptions=[subscription]
        )

        self.price_subscriptions[symbol] = subscription

        self.logger.info(f"Added Binance price stream: {symbol}")

    async def start_price_monitoring(self):
        """Start Binance price monitoring"""

        await self.ws_manager.start_all_connections()

        # Add price processor
        self.stream.add_processor(PriceDataProcessor())

        self.logger.info("Started Binance price monitoring")
```

---

3. Event Processing Systems

Event Filters and Processors

```
from typing import Callable, Any
import re
from decimal import Decimal

class EventFilter:
    """Advanced event filtering system"""

    def __init__(self, name: str):
        self.name = name
        self.logger = logging.getLogger(f"{__name__}.{name}")

        # Filter criteria
        self.filters = []
        self.field_validators = {}

        # Statistics
        self.stats = {
            'events_received': 0,
            'events_passed': 0,
            'events_filtered': 0
        }

    def add_field_filter(self, field_path: str, operator: str, value:
Any):
        """Add field-based filter"""

        def create_filter_func(field_path: str, operator: str, value:
Any):
            def filter_func(event_data: Dict[str, Any]) -> bool:
                # Navigate to field (supports nested paths like
'data.gasPrice')
                current = event_data
                for part in field_path.split('.'):
                    if isinstance(current, dict) and part in current:
                        current = current[part]
                    else:
                        return False

                # Apply operator
                if operator == 'equals':
                    return current == value
                elif operator == 'not_equals':
```

```
                    return current != value
                elif operator == 'greater_than':
                    return current > value
                elif operator == 'less_than':
                    return current < value
                elif operator == 'contains':
                    return str(value) in str(current)
                elif operator == 'regex':
                    return bool(re.search(str(value), str(current)))
                else:
                    raise ValueError(f"Unknown operator: {operator}")

            return filter_func

        self.filters.append(create_filter_func(field_path, operator,
value))
        self.logger.debug(f"Added filter: {field_path} {operator}
{value}")

    def add_range_filter(self, field_path: str, min_value: Any,
max_value: Any):
        """Add range-based filter"""

        def range_filter(event_data: Dict[str, Any]) -> bool:
            current = event_data
            for part in field_path.split('.'):
                if isinstance(current, dict) and part in current:
                    current = current[part]
                else:
                    return False

            return min_value <= current <= max_value

        self.filters.append(range_filter)
        self.logger.debug(f"Added range filter: {field_path}
[{min_value}, {max_value}]")

    def add_custom_filter(self, filter_func: Callable[[Dict[str, Any]],
bool]):
        """Add custom filter function"""
        self.filters.append(filter_func)
        self.logger.debug("Added custom filter")
```

```
    def filter_event(self, event: DataEvent) -> bool:
        """Apply all filters to event"""

        self.stats['events_received'] += 1

        # Apply all filters
        for filter_func in self.filters:
            try:
                if not filter_func(event.data):
                    self.stats['events_filtered'] += 1
                    return False
            except Exception as e:
                self.logger.error(f"Filter error: {e}")
                self.stats['events_filtered'] += 1
                return False

        self.stats['events_passed'] += 1
        return True

    def get_stats(self) -> Dict[str, Any]:
        """Get filter statistics"""
        total_events = self.stats['events_received']

        return {
            'name': self.name,
            'total_events': total_events,
            'events_passed': self.stats['events_passed'],
            'events_filtered': self.stats['events_filtered'],
            'pass_rate': self.stats['events_passed'] / max(1,
total_events),
            'filter_count': len(self.filters)
        }

class TransactionAnalyzer(DataProcessor):
    """Analyze transactions for MEV opportunities"""

    def __init__(self, config: StreamConfig):
        super().__init__(config)

        # MEV opportunity detection patterns
        self.arbitrage_patterns = [
```

```
            'swapExactTokensForTokens',
            'swapExactETHForTokens',
            'swapExactTokensForETH'
        ]

        self.liquidation_patterns = [
            'liquidationCall',
            'liquidatePosition'
        ]

        # Gas price analysis
        self.gas_analysis_window = 100
        self.gas_price_history = deque(maxlen=self.gas_analysis_window)

        # Transaction value analysis
        self.value_thresholds = {
            'large': 1000,  # $1000+
            'very_large': 10000,  # $10,000+
            'mega': 100000  # $100,000+
        }

    async def process_event(self, event: DataEvent) ->
Optional[DataEvent]:
        """Analyze transaction event"""

        start_time = time.time()

        try:
            if event.event_type != DataType.TRANSACTION:
                return event

            tx_data = event.data

            # Analyze transaction
            analysis = self._analyze_transaction(tx_data)

            if analysis['is_mev_relevant']:
                # Enhance event with analysis
                event.data['mev_analysis'] = analysis
                event.metadata.update({
                    'is_mev_opportunity':
analysis['is_mev_opportunity'],
```

```
                    'opportunity_type':
analysis.get('opportunity_type'),
                    'urgency': analysis['urgency'],
                    'estimated_profit':
analysis.get('estimated_profit', 0)
                })

                # Adjust priority based on opportunity
                if analysis['is_mev_opportunity']:
                    event.priority = EventPriority.HIGH
                elif analysis['urgency'] == 'high':
                    event.priority = EventPriority.HIGH

            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, True)

            return event

        except Exception as e:
            self.logger.error(f"Transaction analysis error: {e}")
            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, False)
            return event

    def _analyze_transaction(self, tx_data: Dict[str, Any]) ->
Dict[str, Any]:
        """Analyze single transaction"""

        analysis = {
            'is_mev_relevant': False,
            'is_mev_opportunity': False,
            'urgency': 'normal',
            'gas_price': 0,
            'value_usd': 0,
            'to_address': '',
            'method_name': '',
            'tokens_involved': [],
            'dex_interactions': []
        }

        try:
            # Extract basic transaction data
```

```
            analysis['to_address'] = tx_data.get('to', '').lower()
            analysis['gas_price'] = int(tx_data.get('gasPrice', 0), 16)
if tx_data.get('gasPrice') else 0
            analysis['value_wei'] = int(tx_data.get('value', '0'), 16)
if tx_data.get('value') else 0

            # Estimate USD value (simplified)
            analysis['value_usd'] =
self._estimate_usd_value(analysis['value_wei'])

            # Check for MEV-relevant characteristics
            if analysis['value_usd'] >= self.value_thresholds['large']:
                analysis['is_mev_relevant'] = True

            if analysis['gas_price'] > 100e9:  # 100 gwei
                analysis['urgency'] = 'high'

            # Decode method name (simplified)
            input_data = tx_data.get('input', '0x')
            if len(input_data) > 10:
                method_signature = input_data[:10]
                analysis['method_name'] =
self._decode_method_name(method_signature)

            # Detect DEX interactions
            analysis['dex_interactions'] =
self._detect_dex_interactions(analysis['to_address'])

            # Detect MEV opportunities
            mev_detection = self._detect_mev_opportunity(tx_data,
analysis)
            analysis.update(mev_detection)

        except Exception as e:
            self.logger.error(f"Transaction analysis error: {e}")

        return analysis

    def _estimate_usd_value(self, value_wei: int) -> float:
        """Estimate USD value of transaction"""
        # Simplified - would need real price feeds
        eth_price = 2000  # Mock ETH price
```

```
        eth_value = value_wei / 1e18
        return eth_value * eth_price

    def _decode_method_name(self, method_signature: str) -> str:
        """Decode method signature to readable name"""
        # This is simplified - real implementation would use ABI
decoding
        method_mapping = {
            '0x38ed1739': 'swapExactTokensForTokens',
            '0x7ff36ab5': 'swapExactETHForTokens',
            '0x18cbafe5': 'swapExactTokensForETH',
            '0x5ae401dc': 'multicall'
        }

        return method_mapping.get(method_signature, 'unknown_method')

    def _detect_dex_interactions(self, to_address: str) -> List[str]:
        """Detect DEX protocol interactions"""

        dex_addresses = {
            '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D': 'Uniswap V2',
            '0xE592427A0AEce92De3E3461a08Cc53721Faff32e': 'Uniswap V3',
            '0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f': 'SushiSwap',
            '0x10ED43C718714eb63d5aA57B78B54704E256024E': 'PancakeSwap'
        }

        return [name for addr, name in dex_addresses.items() if
to_address == addr.lower()]

    def _detect_mev_opportunity(self, tx_data: Dict[str, Any],
analysis: Dict[str, Any]) -> Dict[str, Any]:
        """Detect MEV opportunities in transaction"""

        result = {
            'is_mev_opportunity': False,
            'opportunity_type': None,
            'estimated_profit': 0.0
        }

        try:
            # Check for arbitrage opportunities
            if analysis['dex_interactions'] and
```

```
len(analysis['dex_interactions']) >= 2:
                result['is_mev_opportunity'] = True
                result['opportunity_type'] = 'arbitrage'
                result['estimated_profit'] =
self._estimate_arbitrage_profit(analysis)

            # Check for front-running opportunities
            elif analysis['value_usd'] >=
self.value_thresholds['very_large']:
                result['is_mev_opportunity'] = True
                result['opportunity_type'] = 'front_running'
                result['estimated_profit'] = analysis['value_usd'] *
0.001  # 0.1% of value

            # Check for sandwich opportunities
            elif self._detect_sandwich_pattern(tx_data):
                result['is_mev_opportunity'] = True
                result['opportunity_type'] = 'sandwich_attack'
                result['estimated_profit'] =
self._estimate_sandwich_profit(analysis)

        except Exception as e:
            self.logger.error(f"MEV detection error: {e}")

        return result

    def _estimate_arbitrage_profit(self, analysis: Dict[str, Any]) ->
float:
        """Estimate potential arbitrage profit"""
        # Simplified arbitrage profit calculation
        base_value = analysis['value_usd']
        arbitrage_percentage = 0.002  # 0.2%
        return base_value * arbitrage_percentage

    def _detect_sandwich_pattern(self, tx_data: Dict[str, Any]) ->
bool:
        """Detect potential sandwich attack pattern"""
        # Simplified detection - would need more sophisticated analysis
        return False

    def _estimate_sandwich_profit(self, analysis: Dict[str, Any]) ->
float:
```

```
        """Estimate potential sandwich attack profit"""
        base_value = analysis['value_usd']
        sandwich_percentage = 0.001  # 0.1%
        return base_value * sandwich_percentage

class MempoolAnalyzer(DataProcessor):
    """Analyze mempool for MEV opportunities"""

    def __init__(self, config: StreamConfig):
        super().__init__(config)

        # Mempool analysis window
        self.analysis_window = 300  # 5 minutes
        self.recent_txs = deque(maxlen=self.analysis_window)

        # Price impact tracking
        self.price_tracking = defaultdict(float)

        # Gas price analysis
        self.gas_percentiles = {
            'p50': [],
            'p90': [],
            'p99': []
        }

    async def process_event(self, event: DataEvent) ->
Optional[DataEvent]:
        """Analyze mempool transaction"""

        start_time = time.time()

        try:
            if event.event_type != DataType.MEMPOOL:
                return event

            tx_data = event.data

            # Store for analysis
            self.recent_txs.append({
                'timestamp': datetime.utcnow(),
                'tx_hash': tx_data.get('hash', ''),
                'gas_price': int(tx_data.get('gasPrice', '0'), 16) if
```

```
tx_data.get('gasPrice') else 0,
                'value': int(tx_data.get('value', '0'), 16) if
tx_data.get('value') else 0,
                'to': tx_data.get('to', ''),
                'data': tx_data.get('input', '')
            })

            # Update gas price percentiles
            self._update_gas_percentiles()

            # Analyze for opportunities
            opportunities = self._analyze_mempool_opportunities()

            if opportunities:
                # Add opportunity data to event
                event.data['mempool_analysis'] = opportunities
                event.metadata.update({
                    'opportunity_count': len(opportunities),
                    'total_potential_profit':
sum(op.get('estimated_profit', 0) for op in opportunities)
                })

                # High priority for significant opportunities
                if any(op.get('estimated_profit', 0) > 100 for op in
opportunities):
                    event.priority = EventPriority.HIGH

            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, True)

            return event

        except Exception as e:
            self.logger.error(f"Mempool analysis error: {e}")
            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, False)
            return event

    def _update_gas_percentiles(self):
        """Update gas price percentiles"""

        if len(self.recent_txs) < 10:
```

```
            return

        gas_prices = [tx['gas_price'] for tx in self.recent_txs if
tx['gas_price'] > 0]

        if not gas_prices:
            return

        gas_prices.sort()

        self.gas_percentiles['p50'] = gas_prices[int(len(gas_prices) *
0.5)]
        self.gas_percentiles['p90'] = gas_prices[int(len(gas_prices) *
0.9)]
        self.gas_percentiles['p99'] = gas_prices[int(len(gas_prices) *
0.99)]

    def _analyze_mempool_opportunities(self) -> List[Dict[str, Any]]:
        """Analyze mempool for MEV opportunities"""

        opportunities = []
        current_time = datetime.utcnow()

        for tx in self.recent_txs:
            tx_age = (current_time - tx['timestamp']).total_seconds()

            if tx_age > 60:  # Skip transactions older than 1 minute
                continue

            # Check for arbitrage opportunities
            if self._is_arbitrage_opportunity(tx):
                opportunity = {
                    'type': 'arbitrage',
                    'tx_hash': tx['tx_hash'],
                    'estimated_profit':
self._estimate_arbitrage_profit(tx),
                    'confidence': 0.7,
                    'urgency': 'high' if tx['gas_price'] >
self.gas_percentiles['p90'] else 'normal',
                    'time_remaining': 60 - tx_age
                }
                opportunities.append(opportunity)
```

```
            # Check for liquidation opportunities
            if self._is_liquidation_opportunity(tx):
                opportunity = {
                    'type': 'liquidation',
                    'tx_hash': tx['tx_hash'],
                    'estimated_profit':
self._estimate_liquidation_profit(tx),
                    'confidence': 0.8,
                    'urgency': 'critical',
                    'time_remaining': 60 - tx_age
                }
                opportunities.append(opportunity)

        return opportunities

    def _is_arbitrage_opportunity(self, tx: Dict[str, Any]) -> bool:
        """Check if transaction represents arbitrage opportunity"""

        # High value transactions to DEX routers
        dex_routers = [
            '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
            '0xE592427A0AEce92De3E3461a08Cc53721Faff32e',
            '0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f'
        ]

        return (tx['to'].lower() in [addr.lower() for addr in
dex_routers] and
                tx['value'] > 1e18)  # > 1 ETH

    def _is_liquidation_opportunity(self, tx: Dict[str, Any]) -> bool:
        """Check if transaction represents liquidation opportunity"""

        # High gas price transactions to liquidation contracts
        liquidation_contracts = [
            '0x7d2768de32b0b80b7a3454c06bdac94a69ddc7a9',  # Aave
            '0x3d9819210a31b4961b30ef54be2aed79b9c4483e'   # Compound
        ]

        high_gas_threshold = self.gas_percentiles['p99']

        return (tx['to'].lower() in [addr.lower() for addr in
```

```
liquidation_contracts] and
                tx['gas_price'] > high_gas_threshold)

    def _estimate_arbitrage_profit(self, tx: Dict[str, Any]) -> float:
        """Estimate arbitrage profit potential"""

        eth_value = tx['value'] / 1e18
        estimated_profit_percentage = 0.005  # 0.5%

        return eth_value * estimated_profit_percentage * 2000  # $2000
ETH price

    def _estimate_liquidation_profit(self, tx: Dict[str, Any]) ->
float:
        """Estimate liquidation profit potential"""

        # Liquidations typically have 5-10% bonus
        liquidation_bonus = 0.075  # 7.5%

        eth_value = tx['value'] / 1e18
        return eth_value * liquidation_bonus * 2000  # $2000 ETH price

class PriceDataProcessor(DataProcessor):
    """Process real-time price data"""

    def __init__(self, config: StreamConfig):
        super().__init__(config)

        # Price tracking
        self.price_history = defaultdict(lambda: deque(maxlen=1000))
        self.price_alerts = {}

        # Price volatility analysis
        self.volatility_window = 50

        # Cross-exchange price comparison
        self.exchange_prices = defaultdict(dict)

    async def process_event(self, event: DataEvent) ->
Optional[DataEvent]:
        """Process price data event"""
```

```
        start_time = time.time()

        try:
            if event.event_type != DataType.PRICE:
                return event

            price_data = event.data

            # Extract price information
            symbol = price_data.get('s', '')  # Symbol
            price = float(price_data.get('c', 0))  # Close price
            volume = float(price_data.get('v', 0))  # Volume

            # Store price data
            self.price_history[symbol].append({
                'timestamp': datetime.utcnow(),
                'price': price,
                'volume': volume
            })

            # Update exchange price tracking
            exchange = event.source
            self.exchange_prices[symbol][exchange] = {
                'price': price,
                'timestamp': datetime.utcnow()
            }

            # Analyze for arbitrage opportunities
            arbitrage_opportunities =
self._find_arbitrage_opportunities(symbol)

            if arbitrage_opportunities:
                event.data['price_analysis'] = {
                    'arbitrage_opportunities': arbitrage_opportunities,
                    'price_volatility':
self._calculate_volatility(symbol),
                    'price_trend': self._calculate_trend(symbol)
                }

                # High priority for significant arbitrage
                max_profit = max(op.get('profit_percentage', 0) for op
in arbitrage_opportunities)
```

```
                if max_profit > 0.5:  # 0.5% profit
                    event.priority = EventPriority.HIGH

            # Check for price alerts
            alerts = self._check_price_alerts(symbol, price)
            if alerts:
                event.data['price_alerts'] = alerts

            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, True)

            return event

        except Exception as e:
            self.logger.error(f"Price data processing error: {e}")
            processing_time = (time.time() - start_time) * 1000
            self.update_stats(processing_time, False)
            return event

    def _find_arbitrage_opportunities(self, symbol: str) ->
List[Dict[str, Any]]:
        """Find arbitrage opportunities across exchanges"""

        opportunities = []
        exchange_data = self.exchange_prices.get(symbol, {})

        if len(exchange_data) < 2:
            return opportunities

        # Compare prices across exchanges
        prices = [(exchange, data['price']) for exchange, data in
exchange_data.items()]

        for i, (exchange1, price1) in enumerate(prices):
            for exchange2, price2 in prices[i+1:]:

                price_diff = abs(price1 - price2)
                avg_price = (price1 + price2) / 2
                profit_percentage = (price_diff / avg_price) * 100

                if profit_percentage > 0.1:  # Only significant
differences
```

```
                    opportunities.append({
                        'buy_exchange': exchange1 if price1 < price2
else exchange2,
                        'sell_exchange': exchange2 if price1 < price2
else exchange1,
                        'buy_price': min(price1, price2),
                        'sell_price': max(price1, price2),
                        'profit_percentage': profit_percentage,
                        'estimated_profit': price_diff * 1000
# Assume $1000 trade
                    })

        return opportunities

    def _calculate_volatility(self, symbol: str) -> float:
        """Calculate price volatility"""

        prices = list(self.price_history[symbol])
        if len(prices) < 2:
            return 0.0

        price_values = [p['price'] for p in prices[-
self.volatility_window:]]

        if len(price_values) < 2:
            return 0.0

        # Calculate standard deviation
        mean_price = sum(price_values) / len(price_values)
        variance = sum((price - mean_price) ** 2 for price in
price_values) / len(price_values)
        std_dev = variance ** 0.5

        # Return coefficient of variation
        return std_dev / mean_price if mean_price > 0 else 0.0

    def _calculate_trend(self, symbol: str) -> str:
        """Calculate price trend"""

        prices = list(self.price_history[symbol])
        if len(prices) < 10:
            return 'insufficient_data'
```

```
        # Simple moving average crossover
        short_avg = sum(p['price'] for p in prices[-5:]) / 5
        long_avg = sum(p['price'] for p in prices[-20:]) / 20

        if short_avg > long_avg * 1.001:  # 0.1% threshold
            return 'bullish'
        elif short_avg < long_avg * 0.999:
            return 'bearish'
        else:
            return 'sideways'

    def _check_price_alerts(self, symbol: str, current_price: float) ->
List[Dict[str, Any]]:
        """Check for price alerts"""

        alerts = []

        # Check predefined alerts
        for alert_id, alert_config in self.price_alerts.items():
            if alert_config['symbol'] == symbol:

                if alert_config['type'] == 'above' and current_price >
alert_config['price']:
                    alerts.append({
                        'alert_id': alert_id,
                        'type': 'price_above',
                        'threshold': alert_config['price'],
                        'current_price': current_price,
                        'message': f"{symbol} price above
{alert_config['price']}"
                    })

                elif alert_config['type'] == 'below' and current_price
< alert_config['price']:
                    alerts.append({
                        'alert_id': alert_id,
                        'type': 'price_below',
                        'threshold': alert_config['price'],
                        'current_price': current_price,
                        'message': f"{symbol} price below
{alert_config['price']}"
```

```
                    })

        return alerts
```

---

4. Real-time Price Feeds

Aggregated Price Feed System

```
class PriceFeedAggregator:
    """Aggregate price data from multiple sources"""

    def __init__(self):
        self.logger = logging.getLogger(__name__)

        # Price data storage
        self.price_sources = {}
        self.aggregated_prices = {}
        self.price_history = defaultdict(lambda: deque(maxlen=1000))

        # Source reliability tracking
        self.source_stats = defaultdict(lambda: {
            'last_update': None,
            'update_count': 0,
            'error_count': 0,
            'reliability_score': 1.0
        })

        # Price deviation thresholds
        self.max_deviation_pct = 0.05  # 5% max deviation between
sources

        # Update intervals
        self.update_intervals = {
            'high_frequency': 1.0,   # 1 second
            'normal': 5.0,           # 5 seconds
            'low_frequency': 30.0    # 30 seconds
        }

    def add_price_source(self,
                         source_id: str,
                         source_type: str,
                         update_frequency: str = 'normal'):
        """Add a price data source"""

        self.price_sources[source_id] = {
            'type': source_type,
            'update_frequency': update_frequency,
            'last_update': None,
            'is_active': True
        }
```

```
        self.logger.info(f"Added price source: {source_id}
({source_type})")

    async def update_price(self,
                           source_id: str,
                           symbol: str,
                           price: float,
                           volume: Optional[float] = None,
                           timestamp: Optional[datetime] = None):
        """Update price from a source"""

        if timestamp is None:
            timestamp = datetime.utcnow()

        try:
            # Update source statistics
            source_stats = self.source_stats[source_id]
            source_stats['last_update'] = timestamp
            source_stats['update_count'] += 1

            # Store raw price data
            price_data = {
                'symbol': symbol,
                'price': price,
                'volume': volume,
                'timestamp': timestamp,
                'source': source_id
            }

            # Update price history
            self.price_history[symbol].append(price_data)

            # Aggregate price
            await self._aggregate_price(symbol, source_id, price_data)

            self.logger.debug(f"Updated price from {source_id}:
{symbol} = ${price}")

        except Exception as e:
            self.logger.error(f"Error updating price from {source_id}:
{e}")
```

```
            # Update error statistics
            source_stats = self.source_stats[source_id]
            source_stats['error_count'] += 1

            # Calculate reliability score
            total_attempts = source_stats['update_count'] +
source_stats['error_count']
            if total_attempts > 0:
                source_stats['reliability_score'] = (
                    source_stats['update_count'] / total_attempts
                )

    async def _aggregate_price(self, symbol: str, source_id: str,
price_data: Dict[str, Any]):
        """Aggregate prices from multiple sources"""

        current_prices = []

        # Get recent prices from all active sources
        for source in self.price_sources:
            if not self.price_sources[source]['is_active']:
                continue

            source_data = self._get_recent_price(symbol, source,
max_age_seconds=60)
            if source_data:
                current_prices.append(source_data)

        if len(current_prices) == 0:
            return

        # Filter outliers
        filtered_prices = self._filter_outliers(current_prices)

        # Calculate aggregated price
        if filtered_prices:
            aggregated_price =
self._calculate_weighted_average(filtered_prices)

            # Store aggregated price
            self.aggregated_prices[symbol] = {
```

```
                'price': aggregated_price['price'],
                'confidence': aggregated_price['confidence'],
                'sources_count': len(filtered_prices),
                'sources': [p['source'] for p in filtered_prices],
                'timestamp': datetime.utcnow(),
                'method': 'weighted_average'
            }

            # Trigger price update events
            await self._on_price_update(symbol,
self.aggregated_prices[symbol])

    def _get_recent_price(self, symbol: str, source_id: str,
max_age_seconds: float = 60) -> Optional[Dict[str, Any]]:
        """Get most recent price for symbol from source"""

        recent_prices = list(self.price_history[symbol])

        # Find most recent price from this source
        for price_data in reversed(recent_prices):
            if (price_data['source'] == source_id and
                (datetime.utcnow() -
price_data['timestamp']).total_seconds() < max_age_seconds):
                return price_data

        return None

    def _filter_outliers(self, prices: List[Dict[str, Any]]) ->
List[Dict[str, Any]]:
        """Filter out outlier prices using statistical methods"""

        if len(prices) <= 2:
            return prices

        # Calculate price statistics
        price_values = [p['price'] for p in prices]
        mean_price = sum(price_values) / len(price_values)

        # Calculate standard deviation
        variance = sum((price - mean_price) ** 2 for price in
price_values) / len(price_values)
        std_dev = variance ** 0.5
```

```
        # Filter outliers (prices beyond 2 standard deviations)
        filtered_prices = []
        for price_data in prices:
            deviation = abs(price_data['price'] - mean_price)
            if deviation <= 2 * std_dev:
                filtered_prices.append(price_data)

        # If we filtered out too many, keep all
        if len(filtered_prices) < len(prices) // 2:
            self.logger.warning(f"Outlier filtering too aggressive,
keeping all prices")
            filtered_prices = prices

        return filtered_prices

    def _calculate_weighted_average(self, prices: List[Dict[str, Any]])
-> Dict[str, Any]:
        """Calculate weighted average price"""

        if not prices:
            return {'price': 0, 'confidence': 0}

        total_weight = 0
        weighted_sum = 0
        confidence_sum = 0

        for price_data in prices:
            source_id = price_data['source']
            source_stats = self.source_stats[source_id]

            # Weight by source reliability and freshness
            reliability_weight = source_stats['reliability_score']
            freshness_weight = 1.0
# Could be adjusted based on update frequency

            weight = reliability_weight * freshness_weight

            weighted_sum += price_data['price'] * weight
            total_weight += weight
            confidence_sum += reliability_weight
```

```
        if total_weight > 0:
            aggregated_price = weighted_sum / total_weight
            confidence = confidence_sum / len(prices)
        else:
            aggregated_price = 0
            confidence = 0

        return {
            'price': aggregated_price,
            'confidence': confidence
        }

    async def _on_price_update(self, symbol: str, price_info: Dict[str,
Any]):
        """Handle price update events"""

        # This would trigger events to the data stream
        # For now, just log the update
        self.logger.info(f"Price update: {symbol} = $
{price_info['price']:.4f} "
                         f"(confidence: {price_info['confidence']:.2f},
"
                         f"sources: {price_info['sources_count']})")

    async def get_price(self, symbol: str) -> Optional[Dict[str, Any]]:
        """Get current aggregated price"""
        return self.aggregated_prices.get(symbol)

    async def get_price_history(self, symbol: str, limit: int = 100) ->
List[Dict[str, Any]]:
        """Get price history"""
        return list(self.price_history[symbol])[-limit:]

    async def get_source_stats(self) -> Dict[str, Any]:
        """Get source reliability statistics"""
        return dict(self.source_stats)

    async def get_price_analysis(self, symbol: str) -> Dict[str, Any]:
        """Get comprehensive price analysis"""

        if symbol not in self.aggregated_prices:
            return {'error': 'No price data available'}
```

```
        current_price = self.aggregated_prices[symbol]
        history = list(self.price_history[symbol])

        # Calculate additional metrics
        analysis = {
            'current_price': current_price,
            'sources_count': len(history),
            'price_volatility':
self._calculate_price_volatility(history),
            'price_trend': self._calculate_price_trend(history),
            'source_reliability':
self._calculate_source_reliability(symbol),
            'last_update_age': (datetime.utcnow() -
current_price['timestamp']).total_seconds()
        }

        return analysis

    def _calculate_price_volatility(self, history: List[Dict[str,
Any]]) -> float:
        """Calculate price volatility"""

        if len(history) < 2:
            return 0.0

        prices = [p['price'] for p in history]
        mean_price = sum(prices) / len(prices)

        variance = sum((price - mean_price) ** 2 for price in prices) /
len(prices)
        std_dev = variance ** 0.5

        return std_dev / mean_price if mean_price > 0 else 0.0

    def _calculate_price_trend(self, history: List[Dict[str, Any]]) ->
str:
        """Calculate price trend"""

        if len(history) < 10:
            return 'insufficient_data'
```

```
        # Simple moving average analysis
        recent_prices = [p['price'] for p in history[-20:]]
        older_prices = [p['price'] for p in history[-50:-20]]

        recent_avg = sum(recent_prices) / len(recent_prices)
        older_avg = sum(older_prices) / len(older_prices) if
older_prices else recent_avg

        change_pct = (recent_avg - older_avg) / older_avg if older_avg
> 0 else 0

        if change_pct > 0.01:  # 1% increase
            return 'bullish'
        elif change_pct < -0.01:  # 1% decrease
            return 'bearish'
        else:
            return 'sideways'

    def _calculate_source_reliability(self, symbol: str) -> Dict[str,
float]:
        """Calculate reliability metrics for price sources"""

        reliability = {}

        for source_id in self.price_sources:
            source_stats = self.source_stats[source_id]

            # Get recent updates for this symbol
            recent_updates = [
                p for p in self.price_history[symbol]
                if p['source'] == source_id and
                (datetime.utcnow() - p['timestamp']).total_seconds() <
3600  # Last hour
            ]

            update_frequency = len(recent_updates) / 3600  # Updates
per second
            reliability[source_id] = {
                'reliability_score': source_stats['reliability_score'],
                'update_frequency': update_frequency,
                'last_update_age': (
                    datetime.utcnow() - source_stats['last_update']
```

```
                ).total_seconds() if source_stats['last_update'] else
float('inf')
            }

        return reliability

class PriceFeedWebSocketHandler:
    """Handle real-time price feed updates via WebSocket"""

    def __init__(self, aggregator: PriceFeedAggregator):
        self.aggregator = aggregator
        self.logger = logging.getLogger(__name__)

        # WebSocket connections for price feeds
        self.price_feeds = {}

    async def connect_binance_stream(self, symbol: str):
        """Connect to Binance WebSocket price stream"""

        endpoint = f"wss://stream.binance.com:9443/ws/{symbol.lower()}
@ticker"

        try:
            websocket = await websockets.connect(endpoint)

            self.price_feeds[f'binance_{symbol}'] = {
                'websocket': websocket,
                'endpoint': endpoint,
                'symbol': symbol,
                'source': 'binance'
            }

            # Start listening to price updates
            asyncio.create_task(self._listen_to_binance_stream(symbol,
websocket))

            self.logger.info(f"Connected to Binance price stream:
{symbol}")

        except Exception as e:
            self.logger.error(f"Failed to connect to Binance stream:
{e}")
```

```
    async def _listen_to_binance_stream(self, symbol: str, websocket):
        """Listen to Binance price stream"""

        try:
            async for message in websocket:
                try:
                    data = json.loads(message)

                    # Extract price data
                    price_data = {
                        'symbol': symbol,
                        'price': float(data['c']),  # Current price
                        'volume': float(data['v']),  # Volume
                        'timestamp': datetime.utcnow()
                    }

                    # Update price in aggregator
                    await self.aggregator.update_price(
                        source_id='binance',
                        symbol=symbol,
                        price=price_data['price'],
                        volume=price_data['volume'],
                        timestamp=price_data['timestamp']
                    )

                except (json.JSONDecodeError, KeyError, ValueError) as
e:
                    self.logger.error(f"Error parsing Binance message:
{e}")

        except websockets.exceptions.ConnectionClosed:
            self.logger.warning(f"Binance stream closed for {symbol}")
        except Exception as e:
            self.logger.error(f"Binance stream error for {symbol}:
{e}")

    async def connect_coinbase_stream(self, symbol: str):
        """Connect to Coinbase WebSocket price stream"""

        endpoint = "wss://ws-feed.exchange.coinbase.com"
```

```
        try:
            websocket = await websockets.connect(endpoint)

            # Subscribe to ticker
            subscribe_msg = {
                "type": "subscribe",
                "product_ids": [symbol],
                "channels": ["ticker"]
            }

            await websocket.send(json.dumps(subscribe_msg))

            self.price_feeds[f'coinbase_{symbol}'] = {
                'websocket': websocket,
                'endpoint': endpoint,
                'symbol': symbol,
                'source': 'coinbase'
            }

            # Start listening
            asyncio.create_task(self._listen_to_coinbase_stream(symbol,
websocket))

            self.logger.info(f"Connected to Coinbase price stream:
{symbol}")

        except Exception as e:
            self.logger.error(f"Failed to connect to Coinbase stream:
{e}")

    async def _listen_to_coinbase_stream(self, symbol: str, websocket):
        """Listen to Coinbase price stream"""

        try:
            async for message in websocket:
                try:
                    data = json.loads(message)

                    # Check if it's a ticker message
                    if data.get('type') == 'ticker' and
data.get('price'):
```

```
                        price_data = {
                            'symbol': symbol,
                            'price': float(data['price']),
                            'volume': float(data.get('last_size', 0)),
                            'timestamp': datetime.utcnow()
                        }

                        await self.aggregator.update_price(
                            source_id='coinbase',
                            symbol=symbol,
                            price=price_data['price'],
                            volume=price_data['volume'],
                            timestamp=price_data['timestamp']
                        )

                except (json.JSONDecodeError, KeyError, ValueError) as
e:

self.logger.error(f"Error parsing Coinbase message: {e}")

        except websockets.exceptions.ConnectionClosed:
            self.logger.warning(f"Coinbase stream closed for {symbol}")
        except Exception as e:
            self.logger.error(f"Coinbase stream error for {symbol}:
{e}")
```

Summary

This comprehensive module on real-time data streaming provides essential skills for building high-performance MEV monitoring and execution systems. Key achievements:

✅ What You Learned:

- **High-performance data streaming architecture** with priority queues and concurrent processing
- **Advanced WebSocket clients** with automatic reconnection and heartbeat monitoring
- **Sophisticated event processing systems** with filtering, analysis, and enrichment
- **Real-time price feed aggregation** from multiple exchanges with outlier detection
- **Mempool analysis and transaction monitoring** for MEV opportunity detection

- **Production-grade connection management** with failover and performance monitoring

🚀 Production Implementation:

- **Multi-source price aggregation** with reliability scoring and confidence intervals
- **Real-time MEV opportunity detection** in mempool transactions and blocks
- **High-frequency data processing** with sub-millisecond latency optimization
- **Robust WebSocket management** with automatic reconnection and error recovery
- **Comprehensive monitoring and alerting** for data quality and system health

💡 Performance Optimizations:

- **Priority-based event queuing** for critical MEV opportunities
- **Concurrent event processing** with processor chaining and timeout management
- **Intelligent outlier filtering** across price sources and transaction data
- **Memory-efficient data structures** with automatic cleanup and retention policies
- **Connection pooling and reuse** for optimal resource utilization

The real-time data streaming skills from this module enable building ultra-low-latency MEV systems that can monitor blockchain activity, detect opportunities, and execute trades with institutional-grade speed and reliability.

Next: In Module 4, we'll explore infrastructure automation, covering Docker, Kubernetes, cloud deployment, and scaling strategies for production MEV systems.