```python
import os
import threading
import time
from collections import deque
from dataclasses import dataclass
from datetime import datetime
from queue import Queue
from typing import Any, Dict, List, Optional, Tuple

import ccxt
import numpy as np
import pandas as pd
from dotenv import load_dotenv
from loguru import logger
from scipy import stats
from swarm_models import OpenAIChat

from swarms import Agent

logger.enable("")


@dataclass
class MarketSignal:
    timestamp: datetime
    signal_type: str
```

```
    source: str
    data: Dict[str, Any]
    confidence: float
    metadata: Dict[str, Any]


class MarketDataBuffer:
    def __init__(self, max_size: int = 10000):
        self.max_size = max_size
        self.data = deque(maxlen=max_size)
        self.lock = threading.Lock()

    def add(self, item: Any) -> None:
        with self.lock:
            self.data.append(item)

    def get_latest(self, n: int = None) -> List[Any]:
        with self.lock:
            if n is None:
                return list(self.data)
            return list(self.data)[-n:]


class SignalCSVWriter:
    def __init__(self, output_dir: str = "market_data"):
        self.output_dir = output_dir
```

```
        self.ensure_output_dir()
        self.files = {}

    def ensure_output_dir(self):
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)

    def get_filename(self, signal_type: str, symbol: str) -> str:
        date_str = datetime.now().strftime("%Y%m%d")
        return (
            f"{self.output_dir}/{signal_type}_{symbol}_{date_str}.csv"
        )

    def write_order_book_signal(self, signal: MarketSignal):
        symbol = signal.data["symbol"]
        metrics = signal.data["metrics"]
        filename = self.get_filename("order_book", symbol)

        # Create header if file doesn't exist
        if not os.path.exists(filename):
            header = [
                "timestamp",
                "symbol",
                "bid_volume",
                "ask_volume",
                "mid_price",
```

```
        "bid_vwap",
        "ask_vwap",
        "spread",
        "depth_imbalance",
        "confidence",
    ]
    with open(filename, "w") as f:
        f.write(",".join(header) + "\n")

# Write data
data = [
    str(signal.timestamp),
    symbol,
    str(metrics["bid_volume"]),
    str(metrics["ask_volume"]),
    str(metrics["mid_price"]),
    str(metrics["bid_vwap"]),
    str(metrics["ask_vwap"]),
    str(metrics["spread"]),
    str(metrics["depth_imbalance"]),
    str(signal.confidence),
]

with open(filename, "a") as f:
    f.write(",".join(data) + "\n")
```

```
def write_tick_signal(self, signal: MarketSignal):
    symbol = signal.data["symbol"]
    metrics = signal.data["metrics"]
    filename = self.get_filename("tick_data", symbol)

    if not os.path.exists(filename):
        header = [
            "timestamp",
            "symbol",
            "vwap",
            "price_momentum",
            "volume_mean",
            "trade_intensity",
            "kyle_lambda",
            "roll_spread",
            "confidence",
        ]
        with open(filename, "w") as f:
            f.write(",".join(header) + "\n")

    data = [
        str(signal.timestamp),
        symbol,
        str(metrics["vwap"]),
        str(metrics["price_momentum"]),
        str(metrics["volume_mean"]),
```

```
        str(metrics["trade_intensity"]),
        str(metrics["kyle_lambda"]),
        str(metrics["roll_spread"]),
        str(signal.confidence),
    ]

    with open(filename, "a") as f:
        f.write(",".join(data) + "\n")

def write_arbitrage_signal(self, signal: MarketSignal):
    if (
        "best_opportunity" not in signal.data
        or not signal.data["best_opportunity"]
    ):
        return

    symbol = signal.data["symbol"]
    opp = signal.data["best_opportunity"]
    filename = self.get_filename("arbitrage", symbol)

    if not os.path.exists(filename):
        header = [
            "timestamp",
            "symbol",
            "buy_venue",
            "sell_venue",
```

```
        "spread",
        "return",
        "buy_price",
        "sell_price",
        "confidence",
    ]
    with open(filename, "w") as f:
        f.write(",".join(header) + "\n")

data = [
    str(signal.timestamp),
    symbol,
    opp["buy_venue"],
    opp["sell_venue"],
    str(opp["spread"]),
    str(opp["return"]),
    str(opp["buy_price"]),
    str(opp["sell_price"]),
    str(signal.confidence),
]

with open(filename, "a") as f:
    f.write(",".join(data) + "\n")


class ExchangeManager:
```

```
def __init__(self):
    self.available_exchanges = {
        "kraken": ccxt.kraken,
        "coinbase": ccxt.coinbase,
        "kucoin": ccxt.kucoin,
        "bitfinex": ccxt.bitfinex,
        "gemini": ccxt.gemini,
    }
    self.active_exchanges = {}
    self.test_exchanges()

def test_exchanges(self):
    """Test each exchange and keep only the accessible ones"""
    for name, exchange_class in self.available_exchanges.items():
        try:
            exchange = exchange_class()
            exchange.load_markets()
            self.active_exchanges[name] = exchange
            logger.info(f"Successfully connected to {name}")
        except Exception as e:
            logger.warning(f"Could not connect to {name}: {e}")

def get_primary_exchange(self) -> Optional[ccxt.Exchange]:
    """Get the first available exchange"""
    if not self.active_exchanges:
        raise RuntimeError("No exchanges available")
```

```python
        return next(iter(self.active_exchanges.values()))


    def get_all_active_exchanges(self) -> Dict[str, ccxt.Exchange]:
        """Get all active exchanges"""
        return self.active_exchanges



class BaseMarketAgent(Agent):
    def __init__(
        self,
        agent_name: str,
        system_prompt: str,
        api_key: str,
        model_name: str = "gpt-4-0125-preview",
        temperature: float = 0.1,
    ):
        model = OpenAIChat(
            openai_api_key=api_key,
            model_name=model_name,
            temperature=temperature,
        )
        super().__init__(
            agent_name=agent_name,
            system_prompt=system_prompt,
            llm=model,
            max_loops=1,
```

```
            autosave=True,
            dashboard=False,
            verbose=True,
            dynamic_temperature_enabled=True,
            context_length=200000,
            streaming_on=True,
            output_type="str",
        )
        self.signal_queue = Queue()
        self.is_running = False
        self.last_update = datetime.now()
        self.update_interval = 1.0  # seconds

    def rate_limit_check(self) -> bool:
        current_time = datetime.now()
        if (
            current_time - self.last_update
        ).total_seconds() < self.update_interval:
            return False
        self.last_update = current_time
        return True


class OrderBookAgent(BaseMarketAgent):
    def __init__(self, api_key: str):
        system_prompt = """
```

```
        You are an Order Book Analysis Agent specialized in detecting institutional flows.
        Monitor order book depth and changes to identify potential large trades and institutional activity.
        Analyze patterns in order placement and cancellation rates.
        """
        super().__init__("OrderBookAgent", system_prompt, api_key)
        exchange_manager = ExchangeManager()
        self.exchange = exchange_manager.get_primary_exchange()
        self.order_book_buffer = MarketDataBuffer(max_size=100)
        self.vwap_window = 20

    def calculate_order_book_metrics(
        self, order_book: Dict
    ) -> Dict[str, float]:
        bids = np.array(order_book["bids"])
        asks = np.array(order_book["asks"])

        # Calculate key metrics
        bid_volume = np.sum(bids[:, 1])
        ask_volume = np.sum(asks[:, 1])
        mid_price = (bids[0][0] + asks[0][0]) / 2

        # Calculate VWAP
        bid_vwap = (
            np.sum(
                bids[: self.vwap_window, 0]
```

```
        * bids[: self.vwap_window, 1]
    )
    / bid_volume
    if bid_volume > 0
    else 0
)
ask_vwap = (
    np.sum(
        asks[: self.vwap_window, 0]
        * asks[: self.vwap_window, 1]
    )
    / ask_volume
    if ask_volume > 0
    else 0
)

# Calculate order book slope
bid_slope = np.polyfit(
    range(len(bids[:10])), bids[:10, 0], 1
)[0]
ask_slope = np.polyfit(
    range(len(asks[:10])), asks[:10, 0], 1
)[0]

return {
    "bid_volume": bid_volume,
```

```
        "ask_volume": ask_volume,
        "mid_price": mid_price,
        "bid_vwap": bid_vwap,
        "ask_vwap": ask_vwap,
        "bid_slope": bid_slope,
        "ask_slope": ask_slope,
        "spread": asks[0][0] - bids[0][0],
        "depth_imbalance": (bid_volume - ask_volume)
        / (bid_volume + ask_volume),
    }

def detect_large_orders(
    self, metrics: Dict[str, float], threshold: float = 2.0
) -> bool:
    historical_books = self.order_book_buffer.get_latest(20)
    if not historical_books:
        return False

    # Calculate historical volume statistics
    hist_volumes = [
        book["bid_volume"] + book["ask_volume"]
        for book in historical_books
    ]
    volume_mean = np.mean(hist_volumes)
    volume_std = np.std(hist_volumes)
```

```
        current_volume = metrics["bid_volume"] + metrics["ask_volume"]
        z_score = (current_volume - volume_mean) / (
            volume_std if volume_std > 0 else 1
        )

        return abs(z_score) > threshold

    def analyze_order_book(self, symbol: str) -> MarketSignal:
        if not self.rate_limit_check():
            return None

        try:
            order_book = self.exchange.fetch_order_book(
                symbol, limit=100
            )
            metrics = self.calculate_order_book_metrics(order_book)
            self.order_book_buffer.add(metrics)

            # Format data for LLM analysis
            analysis_prompt = f"""
            Analyze this order book for {symbol}:
            Bid Volume: {metrics['bid_volume']}
            Ask Volume: {metrics['ask_volume']}
            Mid Price: {metrics['mid_price']}
            Spread: {metrics['spread']}
            Depth Imbalance: {metrics['depth_imbalance']}
```

```
What patterns do you see? Is there evidence of institutional activity?
Are there any significant imbalances that could lead to price movement?
"""

# Get LLM analysis
llm_analysis = self.run(analysis_prompt)

# Original signal creation with added LLM analysis
return MarketSignal(
    timestamp=datetime.now(),
    signal_type="order_book_analysis",
    source="OrderBookAgent",
    data={
        "metrics": metrics,
        "large_order_detected": self.detect_large_orders(
            metrics
        ),
        "symbol": symbol,
        "llm_analysis": llm_analysis,  # Add LLM insights
    },
    confidence=min(
        abs(metrics["depth_imbalance"]) * 0.7
        + (
            1.0
            if self.detect_large_orders(metrics)
```

```
                else 0.0
            )
            * 0.3,
            1.0,
        ),
        metadata={
            "update_latency": (
                datetime.now() - self.last_update
            ).total_seconds(),
            "buffer_size": len(
                self.order_book_buffer.get_latest()
            ),
        },
    )
except Exception as e:
    logger.error(f"Error in order book analysis: {str(e)}")
    return None


class TickDataAgent(BaseMarketAgent):
    def __init__(self, api_key: str):
        system_prompt = """
        You are a Tick Data Analysis Agent specialized in analyzing high-frequency price movements.
        Monitor tick-by-tick data for patterns indicating short-term price direction.
        Analyze trade size distribution and execution speed.
        """
```

```
    super().__init__("TickDataAgent", system_prompt, api_key)
    self.tick_buffer = MarketDataBuffer(max_size=5000)
    exchange_manager = ExchangeManager()
    self.exchange = exchange_manager.get_primary_exchange()

def calculate_tick_metrics(
    self, ticks: List[Dict]
) -> Dict[str, float]:
    df = pd.DataFrame(ticks)
    df["price"] = pd.to_numeric(df["price"])
    df["volume"] = pd.to_numeric(df["amount"])

    # Calculate key metrics
    metrics = {}

    # Volume-weighted average price (VWAP)
    metrics["vwap"] = (df["price"] * df["volume"]).sum() / df[
        "volume"
    ].sum()

    # Price momentum
    metrics["price_momentum"] = df["price"].diff().mean()

    # Volume profile
    metrics["volume_mean"] = df["volume"].mean()
    metrics["volume_std"] = df["volume"].std()
```

```
    # Trade intensity
    time_diff = (
        df["timestamp"].max() - df["timestamp"].min()
    ) / 1000  # Convert to seconds
    metrics["trade_intensity"] = (
        len(df) / time_diff if time_diff > 0 else 0
    )

    # Microstructure indicators
    metrics["kyle_lambda"] = self.calculate_kyle_lambda(df)
    metrics["roll_spread"] = self.calculate_roll_spread(df)

    return metrics

def calculate_kyle_lambda(self, df: pd.DataFrame) -> float:
    """Calculate Kyle's Lambda (price impact coefficient)"""
    try:
        price_changes = df["price"].diff().dropna()
        volume_changes = df["volume"].diff().dropna()

        if len(price_changes) > 1 and len(volume_changes) > 1:
            slope, _, _, _, _ = stats.linregress(
                volume_changes, price_changes
            )
            return abs(slope)
```

```
    except Exception as e:
        logger.warning(f"Error calculating Kyle's Lambda: {e}")
    return 0.0

def calculate_roll_spread(self, df: pd.DataFrame) -> float:
    """Calculate Roll's implied spread"""
    try:
        price_changes = df["price"].diff().dropna()
        if len(price_changes) > 1:
            autocov = np.cov(
                price_changes[:-1], price_changes[1:]
            )[0][1]
            return 2 * np.sqrt(-autocov) if autocov < 0 else 0.0
    except Exception as e:
        logger.warning(f"Error calculating Roll spread: {e}")
    return 0.0

def calculate_tick_metrics(
    self, ticks: List[Dict]
) -> Dict[str, float]:
    try:
        # Debug the incoming data structure
        logger.info(
            f"Raw tick data structure: {ticks[0] if ticks else 'No ticks'}"
        )
```

```
# Convert trades to proper format
formatted_trades = []
for trade in ticks:
    formatted_trade = {
        "price": float(
            trade.get("price", trade.get("last", 0))
        ),  # Handle different exchange formats
        "amount": float(
            trade.get(
                "amount",
                trade.get(
                    "size", trade.get("quantity", 0)
                ),
            )
        ),
        "timestamp": trade.get(
            "timestamp", int(time.time() * 1000)
        ),
    }
    formatted_trades.append(formatted_trade)

df = pd.DataFrame(formatted_trades)

if df.empty:
    logger.warning("No valid trades to analyze")
    return {
```

```
        "vwap": 0.0,
        "price_momentum": 0.0,
        "volume_mean": 0.0,
        "volume_std": 0.0,
        "trade_intensity": 0.0,
        "kyle_lambda": 0.0,
        "roll_spread": 0.0,
    }

# Calculate metrics with the properly formatted data
metrics = {}
metrics["vwap"] = (
    (df["price"] * df["amount"]).sum()
    / df["amount"].sum()
    if not df.empty
    else 0
)
metrics["price_momentum"] = (
    df["price"].diff().mean() if len(df) > 1 else 0
)
metrics["volume_mean"] = df["amount"].mean()
metrics["volume_std"] = df["amount"].std()

time_diff = (
    (df["timestamp"].max() - df["timestamp"].min()) / 1000
    if len(df) > 1
```

```
        else 1
    )
    metrics["trade_intensity"] = (
        len(df) / time_diff if time_diff > 0 else 0
    )

    metrics["kyle_lambda"] = self.calculate_kyle_lambda(df)
    metrics["roll_spread"] = self.calculate_roll_spread(df)

    logger.info(f"Calculated metrics: {metrics}")
    return metrics

except Exception as e:
    logger.error(
        f"Error in calculate_tick_metrics: {str(e)}",
        exc_info=True,
    )
    # Return default metrics on error
    return {
        "vwap": 0.0,
        "price_momentum": 0.0,
        "volume_mean": 0.0,
        "volume_std": 0.0,
        "trade_intensity": 0.0,
        "kyle_lambda": 0.0,
        "roll_spread": 0.0,
```

```
        }


    def analyze_ticks(self, symbol: str) -> MarketSignal:
        if not self.rate_limit_check():
            return None


        try:
            # Fetch recent trades
            trades = self.exchange.fetch_trades(symbol, limit=100)


            # Debug the raw trades data
            logger.info(f"Fetched {len(trades)} trades for {symbol}")
            if trades:
                logger.info(f"Sample trade: {trades[0]}")


            self.tick_buffer.add(trades)
            recent_ticks = self.tick_buffer.get_latest(1000)
            metrics = self.calculate_tick_metrics(recent_ticks)


            # Only proceed with LLM analysis if we have valid metrics
            if metrics["vwap"] > 0:
                analysis_prompt = f"""
                Analyze these trading patterns for {symbol}:
                VWAP: {metrics['vwap']:.2f}
                Price Momentum: {metrics['price_momentum']:.2f}
                Trade Intensity: {metrics['trade_intensity']:.2f}
```

```
        Kyle's Lambda: {metrics['kyle_lambda']:.2f}

        What does this tell us about:
        1. Current market sentiment
        2. Potential price direction
        3. Trading activity patterns
        """
        llm_analysis = self.run(analysis_prompt)
    else:
        llm_analysis = "Insufficient data for analysis"

    return MarketSignal(
        timestamp=datetime.now(),
        signal_type="tick_analysis",
        source="TickDataAgent",
        data={
            "metrics": metrics,
            "symbol": symbol,
            "prediction": np.sign(metrics["price_momentum"]),
            "llm_analysis": llm_analysis,
        },
        confidence=min(metrics["trade_intensity"] / 100, 1.0)
        * 0.4
        + min(metrics["kyle_lambda"], 1.0) * 0.6,
        metadata={
            "update_latency": (
```

```
                datetime.now() - self.last_update
            ).total_seconds(),
            "buffer_size": len(self.tick_buffer.get_latest()),
        },
    )

except Exception as e:
    logger.error(
        f"Error in tick analysis: {str(e)}", exc_info=True
    )
    return None


class LatencyArbitrageAgent(BaseMarketAgent):
    def __init__(self, api_key: str):
        system_prompt = """
        You are a Latency Arbitrage Agent specialized in detecting price discrepancies across venues.
        Monitor multiple exchanges for price differences exceeding transaction costs.
        Calculate optimal trade sizes and routes.
        """
        super().__init__(
            "LatencyArbitrageAgent", system_prompt, api_key
        )
        exchange_manager = ExchangeManager()
        self.exchanges = exchange_manager.get_all_active_exchanges()
        self.fee_structure = {
```

```
        "kraken": 0.0026,  # 0.26% taker fee
        "coinbase": 0.006,  # 0.6% taker fee
        "kucoin": 0.001,  # 0.1% taker fee
        "bitfinex": 0.002,  # 0.2% taker fee
        "gemini": 0.003,  # 0.3% taker fee
    }
    self.price_buffer = {
        ex: MarketDataBuffer(max_size=100)
        for ex in self.exchanges
    }

def calculate_effective_prices(
    self, ticker: Dict, venue: str
) -> Tuple[float, float]:
    """Calculate effective prices including fees"""
    fee = self.fee_structure[venue]
    return (
        ticker["bid"] * (1 - fee),  # Effective sell price
        ticker["ask"] * (1 + fee),  # Effective buy price
    )

def calculate_arbitrage_metrics(
    self, prices: Dict[str, Dict]
) -> Dict:
    opportunities = []
```

```
for venue1 in prices:
    for venue2 in prices:
        if venue1 != venue2:
            sell_price, _ = self.calculate_effective_prices(
                prices[venue1], venue1
            )
            _, buy_price = self.calculate_effective_prices(
                prices[venue2], venue2
            )

            spread = sell_price - buy_price
            if spread > 0:
                opportunities.append(
                    {
                        "sell_venue": venue1,
                        "buy_venue": venue2,
                        "spread": spread,
                        "return": spread / buy_price,
                        "buy_price": buy_price,
                        "sell_price": sell_price,
                    }
                )

return {
    "opportunities": opportunities,
    "best_opportunity": (
```

```
            max(opportunities, key=lambda x: x["return"])
            if opportunities
            else None
        ),
    }

def find_arbitrage(self, symbol: str) -> MarketSignal:
    """
    Find arbitrage opportunities across exchanges with LLM analysis
    """
    if not self.rate_limit_check():
        return None

    try:
        prices = {}
        timestamps = {}

        for name, exchange in self.exchanges.items():
            try:
                ticker = exchange.fetch_ticker(symbol)
                prices[name] = {
                    "bid": ticker["bid"],
                    "ask": ticker["ask"],
                }
                timestamps[name] = ticker["timestamp"]
                self.price_buffer[name].add(prices[name])
```

```
    except Exception as e:
        logger.warning(
            f"Error fetching {name} price: {e}"
        )

if len(prices) < 2:
    return None

metrics = self.calculate_arbitrage_metrics(prices)

if not metrics["best_opportunity"]:
    return None

# Calculate confidence based on spread and timing
opp = metrics["best_opportunity"]
timing_factor = 1.0 - min(
    abs(
        timestamps[opp["sell_venue"]]
        - timestamps[opp["buy_venue"]]
    )
    / 1000,
    1.0,
)
spread_factor = min(
    opp["return"] * 5, 1.0
)  # Scale return to confidence
```

```
confidence = timing_factor * 0.4 + spread_factor * 0.6

# Format price data for LLM analysis
price_summary = "\n".join(
    [
        f"{venue}: Bid ${prices[venue]['bid']:.2f}, Ask ${prices[venue]['ask']:.2f}"
        for venue in prices.keys()
    ]
)

# Create detailed analysis prompt
analysis_prompt = f"""
Analyze this arbitrage opportunity for {symbol}:

Current Prices:
{price_summary}

Best Opportunity Found:
Buy Venue: {opp['buy_venue']} at ${opp['buy_price']:.2f}
Sell Venue: {opp['sell_venue']} at ${opp['sell_price']:.2f}
Spread: ${opp['spread']:.2f}
Expected Return: {opp['return']*100:.3f}%
Time Difference: {abs(timestamps[opp['sell_venue']] - timestamps[opp['buy_venue']])}ms

Consider:
```

```
1. Is this opportunity likely to be profitable after execution costs?
2. What risks might prevent successful execution?
3. What market conditions might have created this opportunity?
4. How does the timing difference affect execution probability?
"""

# Get LLM analysis
llm_analysis = self.run(analysis_prompt)

# Create comprehensive signal
return MarketSignal(
    timestamp=datetime.now(),
    signal_type="arbitrage_opportunity",
    source="LatencyArbitrageAgent",
    data={
        "metrics": metrics,
        "symbol": symbol,
        "best_opportunity": metrics["best_opportunity"],
        "all_prices": prices,
        "llm_analysis": llm_analysis,
        "timing": {
            "time_difference_ms": abs(
                timestamps[opp["sell_venue"]]
                - timestamps[opp["buy_venue"]]
            ),
            "timestamps": timestamps,
```

```
                },
            },
            confidence=confidence,
            metadata={
                "update_latency": (
                    datetime.now() - self.last_update
                ).total_seconds(),
                "timestamp_deltas": timestamps,
                "venue_count": len(prices),
                "execution_risk": 1.0
                - timing_factor,  # Higher time difference = higher risk
            },
        )

    except Exception as e:
        logger.error(f"Error in arbitrage analysis: {str(e)}")
        return None


class SwarmCoordinator:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.agents = {
            "order_book": OrderBookAgent(api_key),
            "tick_data": TickDataAgent(api_key),
            "latency_arb": LatencyArbitrageAgent(api_key),
```

```
    }
    self.signal_processors = []
    self.signal_history = MarketDataBuffer(max_size=1000)
    self.running = False
    self.lock = threading.Lock()
    self.csv_writer = SignalCSVWriter()

def register_signal_processor(self, processor):
    """Register a new signal processor function"""
    with self.lock:
        self.signal_processors.append(processor)

def process_signals(self, signals: List[MarketSignal]):
    """Process signals through all registered processors"""
    if not signals:
        return

    self.signal_history.add(signals)

    try:
        for processor in self.signal_processors:
            processor(signals)
    except Exception as e:
        logger.error(f"Error in signal processing: {e}")

def aggregate_signals(
```

```
    self, signals: List[MarketSignal]
) -> Dict[str, Any]:
    """Aggregate multiple signals into a combined market view"""
    if not signals:
        return {}

    self.signal_history.add(signals)

    aggregated = {
        "timestamp": datetime.now(),
        "symbols": set(),
        "agent_signals": {},
        "combined_confidence": 0,
        "market_state": {},
    }

    for signal in signals:
        symbol = signal.data.get("symbol")
        if symbol:
            aggregated["symbols"].add(symbol)

        agent_type = signal.source
        if agent_type not in aggregated["agent_signals"]:
            aggregated["agent_signals"][agent_type] = []
        aggregated["agent_signals"][agent_type].append(signal)
```

```
# Update market state based on signal type
if signal.signal_type == "order_book_analysis":
    metrics = signal.data.get("metrics", {})
    aggregated["market_state"].update(
        {
            "order_book_imbalance": metrics.get(
                "depth_imbalance"
            ),
            "spread": metrics.get("spread"),
            "large_orders_detected": signal.data.get(
                "large_order_detected"
            ),
        }
    )
elif signal.signal_type == "tick_analysis":
    metrics = signal.data.get("metrics", {})
    aggregated["market_state"].update(
        {
            "price_momentum": metrics.get(
                "price_momentum"
            ),
            "trade_intensity": metrics.get(
                "trade_intensity"
            ),
            "kyle_lambda": metrics.get("kyle_lambda"),
        }
```

```
        )
      elif signal.signal_type == "arbitrage_opportunity":
        opp = signal.data.get("best_opportunity")
        if opp:
          aggregated["market_state"].update(
            {
              "arbitrage_spread": opp.get("spread"),
              "arbitrage_return": opp.get("return"),
            }
          )

    # Calculate combined confidence as weighted average
    confidences = [s.confidence for s in signals]
    if confidences:
      aggregated["combined_confidence"] = np.mean(confidences)

    return aggregated

  def start(self, symbols: List[str], interval: float = 1.0):
    """Start the swarm monitoring system"""
    if self.running:
      logger.warning("Swarm is already running")
      return

    self.running = True
```

```
def agent_loop(agent, symbol):
    while self.running:
        try:
            if isinstance(agent, OrderBookAgent):
                signal = agent.analyze_order_book(symbol)
            elif isinstance(agent, TickDataAgent):
                signal = agent.analyze_ticks(symbol)
            elif isinstance(agent, LatencyArbitrageAgent):
                signal = agent.find_arbitrage(symbol)

            if signal:
                agent.signal_queue.put(signal)
        except Exception as e:
            logger.error(
                f"Error in {agent.agent_name} loop: {e}"
            )

        time.sleep(interval)


def signal_collection_loop():
    while self.running:
        try:
            current_signals = []

            # Collect signals from all agents
            for agent in self.agents.values():
```

```
                while not agent.signal_queue.empty():
                    signal = agent.signal_queue.get_nowait()
                    if signal:
                        current_signals.append(signal)

            if current_signals:
                # Process current signals
                self.process_signals(current_signals)

                # Aggregate and analyze
                aggregated = self.aggregate_signals(
                    current_signals
                )
                logger.info(
                    f"Aggregated market view: {aggregated}"
                )

        except Exception as e:
            logger.error(
                f"Error in signal collection loop: {e}"
            )

        time.sleep(interval)

# Start agent threads
self.threads = []
```

```
        for symbol in symbols:
            for agent in self.agents.values():
                thread = threading.Thread(
                    target=agent_loop,
                    args=(agent, symbol),
                    daemon=True,
                )
                thread.start()
                self.threads.append(thread)

        # Start signal collection thread
        collection_thread = threading.Thread(
            target=signal_collection_loop, daemon=True
        )
        collection_thread.start()
        self.threads.append(collection_thread)

    def stop(self):
        """Stop the swarm monitoring system"""
        self.running = False
        for thread in self.threads:
            thread.join(timeout=5.0)
        logger.info("Swarm stopped")


def market_making_processor(signals: List[MarketSignal]):
```

```
"""Enhanced signal processor with LLM analysis integration"""
for signal in signals:
    if signal.confidence > 0.8:
        if signal.signal_type == "arbitrage_opportunity":
            opp = signal.data.get("best_opportunity")
            if (
                opp and opp["return"] > 0.001
            ):  # 0.1% return threshold
                logger.info(
                    "\nSignificant arbitrage opportunity detected:"
                )
                logger.info(f"Return: {opp['return']*100:.3f}%")
                logger.info(f"Spread: ${opp['spread']:.2f}")
                if "llm_analysis" in signal.data:
                    logger.info("\nLLM Analysis:")
                    logger.info(signal.data["llm_analysis"])

        elif signal.signal_type == "order_book_analysis":
            imbalance = signal.data["metrics"]["depth_imbalance"]
            if abs(imbalance) > 0.3:
                logger.info(
                    f"\nSignificant order book imbalance detected: {imbalance:.3f}"
                )
                if "llm_analysis" in signal.data:
                    logger.info("\nLLM Analysis:")
                    logger.info(signal.data["llm_analysis"])
```

```python
        elif signal.signal_type == "tick_analysis":
            momentum = signal.data["metrics"]["price_momentum"]
            if abs(momentum) > 0:
                logger.info(
                    f"\nSignificant price momentum detected: {momentum:.3f}"
                )
                if "llm_analysis" in signal.data:
                    logger.info("\nLLM Analysis:")
                    logger.info(signal.data["llm_analysis"])


load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")

coordinator = SwarmCoordinator(api_key)
coordinator.register_signal_processor(market_making_processor)

symbols = ["BTC/USDT", "ETH/USDT"]

logger.info(
    "Starting market microstructure analysis with LLM integration..."
)
logger.info(f"Monitoring symbols: {symbols}")
logger.info(
    f"CSV files will be written to: {os.path.abspath('market_data')}"
```

```
)


try:
    coordinator.start(symbols)
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    logger.info("Gracefully shutting down...")
    coordinator.stop()
```