```
from dataclasses import dataclass
from typing import List, Optional, Dict, Any
from datetime import datetime
import asyncio
from loguru import logger
import json
import base58
from decimal import Decimal

# Swarms imports
from swarms import Agent

# Solana imports
from solders.rpc.responses import GetTransactionResp
from solders.transaction import Transaction
from anchorpy import Provider, Wallet
from solders.keypair import Keypair
import aiohttp

# Specialized Solana Analysis System Prompt
SOLANA_ANALYSIS_PROMPT = """You are a specialized Solana blockchain analyst agent. Your role is to:

1. Analyze real-time Solana transactions for patterns and anomalies
2. Identify potential market-moving transactions and whale movements
3. Detect important DeFi interactions across major protocols
```

4. Monitor program interactions for suspicious or notable activity

5. Track token movements across significant protocols like:

   - Serum DEX

   - Raydium

   - Orca

   - Marinade

   - Jupiter

   - Other major Solana protocols

When analyzing transactions, consider:

- Transaction size relative to protocol norms

- Historical patterns for involved addresses

- Impact on protocol liquidity

- Relationship to known market events

- Potential wash trading or suspicious patterns

- MEV opportunities and arbitrage patterns

- Program interaction sequences

Provide analysis in the following format:

```
{
    "analysis_type": "[whale_movement|program_interaction|defi_trade|suspicious_activity]",
    "severity": "[high|medium|low]",
    "details": {
        "transaction_context": "...",
        "market_impact": "...",
        "recommended_actions": "...",
```

```
        "related_patterns": "..."
    }
}

Focus on actionable insights that could affect:
1. Market movements
2. Protocol stability
3. Trading opportunities
4. Risk management
"""



@dataclass
class TransactionData:
    """Data structure for parsed Solana transaction information"""

    signature: str
    block_time: datetime
    slot: int
    fee: int
    lamports: int
    from_address: str
    to_address: str
    program_id: str
    instruction_data: Optional[str] = None
    program_logs: List[str] = None
```

```
    @property
    def sol_amount(self) -> Decimal:
        """Convert lamports to SOL"""
        return Decimal(self.lamports) / Decimal(1e9)

    def to_dict(self) -> Dict[str, Any]:
        """Convert transaction data to dictionary for agent analysis"""
        return {
            "signature": self.signature,
            "timestamp": self.block_time.isoformat(),
            "slot": self.slot,
            "fee": self.fee,
            "amount_sol": str(self.sol_amount),
            "from_address": self.from_address,
            "to_address": self.to_address,
            "program_id": self.program_id,
            "instruction_data": self.instruction_data,
            "program_logs": self.program_logs,
        }


class SolanaSwarmAgent:
    """Intelligent agent for analyzing Solana transactions using swarms"""

    def __init__(
```

```
    self,
    agent_name: str = "Solana-Analysis-Agent",
    model_name: str = "gpt-4",
):
    self.agent = Agent(
        agent_name=agent_name,
        system_prompt=SOLANA_ANALYSIS_PROMPT,
        model_name=model_name,
        max_loops=1,
        autosave=True,
        dashboard=False,
        verbose=True,
        dynamic_temperature_enabled=True,
        saved_state_path="solana_agent.json",
        user_name="solana_analyzer",
        retry_attempts=3,
        context_length=4000,
    )

    # Initialize known patterns database
    self.known_patterns = {
        "whale_addresses": set(),
        "program_interactions": {},
        "recent_transactions": [],
    }
    logger.info(
```

```
            f"Initialized {agent_name} with specialized Solana analysis capabilities"
        )

    async def analyze_transaction(
        self, tx_data: TransactionData
    ) -> Dict[str, Any]:
        """Analyze a transaction using the specialized agent"""
        try:
            # Update recent transactions for pattern analysis
            self.known_patterns["recent_transactions"].append(
                tx_data.signature
            )
            if len(self.known_patterns["recent_transactions"]) > 1000:
                self.known_patterns["recent_transactions"].pop(0)

            # Prepare context for agent
            context = {
                "transaction": tx_data.to_dict(),
                "known_patterns": {
                    "recent_similar_transactions": [
                        tx
                        for tx in self.known_patterns[
                            "recent_transactions"
                        ][-5:]
                        if abs(
                            TransactionData(tx).sol_amount
```

```
                - tx_data.sol_amount
            )
            < 1
        ],
        "program_statistics": self.known_patterns[
            "program_interactions"
        ].get(tx_data.program_id, {}),
    },
}

# Get analysis from agent
analysis = await self.agent.run_async(
    f"Analyze the following Solana transaction and provide insights: {json.dumps(context, indent=2)}"
)

# Update pattern database
if tx_data.sol_amount > 1000:  # Track whale addresses
    self.known_patterns["whale_addresses"].add(
        tx_data.from_address
    )

# Update program interaction statistics
if (
    tx_data.program_id
    not in self.known_patterns["program_interactions"]
```

```
            ):
                self.known_patterns["program_interactions"][
                    tx_data.program_id
                ] = {"total_interactions": 0, "total_volume": 0}
            self.known_patterns["program_interactions"][
                tx_data.program_id
            ]["total_interactions"] += 1
            self.known_patterns["program_interactions"][
                tx_data.program_id
            ]["total_volume"] += float(tx_data.sol_amount)

            return json.loads(analysis)

        except Exception as e:
            logger.error(f"Error in agent analysis: {str(e)}")
            return {
                "analysis_type": "error",
                "severity": "low",
                "details": {
                    "error": str(e),
                    "transaction": tx_data.signature,
                },
            }


class SolanaTransactionMonitor:
```

```
"""Main class for monitoring and analyzing Solana transactions"""


def __init__(
    self,
    rpc_url: str,
    swarm_agent: SolanaSwarmAgent,
    min_sol_threshold: Decimal = Decimal("100"),
):
    self.rpc_url = rpc_url
    self.swarm_agent = swarm_agent
    self.min_sol_threshold = min_sol_threshold
    self.wallet = Wallet(Keypair())
    self.provider = Provider(rpc_url, self.wallet)
    logger.info("Initialized Solana transaction monitor")


async def parse_transaction(
    self, tx_resp: GetTransactionResp
) -> Optional[TransactionData]:
    """Parse transaction response into TransactionData object"""
    try:
        if not tx_resp.value:
            return None


        tx_value = tx_resp.value
        meta = tx_value.transaction.meta
        if not meta:
```

```
    return None

tx: Transaction = tx_value.transaction.transaction

# Extract transaction details
from_pubkey = str(tx.message.account_keys[0])
to_pubkey = str(tx.message.account_keys[1])
program_id = str(tx.message.account_keys[-1])

# Calculate amount from balance changes
amount = abs(meta.post_balances[0] - meta.pre_balances[0])

return TransactionData(
    signature=str(tx_value.transaction.signatures[0]),
    block_time=datetime.fromtimestamp(
        tx_value.block_time or 0
    ),
    slot=tx_value.slot,
    fee=meta.fee,
    lamports=amount,
    from_address=from_pubkey,
    to_address=to_pubkey,
    program_id=program_id,
    program_logs=(
        meta.log_messages if meta.log_messages else []
    ),
```

```
            )
        except Exception as e:
            logger.error(f"Failed to parse transaction: {str(e)}")
            return None


async def start_monitoring(self):
    """Start monitoring for new transactions"""
    logger.info(
        "Starting transaction monitoring with swarm agent analysis"
    )

    async with aiohttp.ClientSession() as session:
        async with session.ws_connect(self.rpc_url) as ws:
            await ws.send_json(
                {
                    "jsonrpc": "2.0",
                    "id": 1,
                    "method": "transactionSubscribe",
                    "params": [
                        {"commitment": "finalized"},
                        {
                            "encoding": "jsonParsed",
                            "commitment": "finalized",
                        },
                    ],
                }
```

```
)


async for msg in ws:
    if msg.type == aiohttp.WSMsgType.TEXT:
        try:
            data = json.loads(msg.data)
            if "params" in data:
                signature = data["params"]["result"][
                    "value"
                ]["signature"]

                # Fetch full transaction data
                tx_response = await self.provider.connection.get_transaction(
                    base58.b58decode(signature)
                )

                if tx_response:
                    tx_data = (
                        await self.parse_transaction(
                            tx_response
                        )
                    )
                    if (
                        tx_data
                        and tx_data.sol_amount
                        >= self.min_sol_threshold
```

```
                ):
                    # Get agent analysis
                    analysis = await self.swarm_agent.analyze_transaction(
                        tx_data
                    )

                    logger.info(
                        f"Transaction Analysis:\n"
                        f"Signature: {tx_data.signature}\n"
                        f"Amount: {tx_data.sol_amount} SOL\n"
                        f"Analysis: {json.dumps(analysis, indent=2)}"
                    )

            except Exception as e:
                logger.error(
                    f"Error processing message: {str(e)}"
                )
                continue


async def main():
    """Example usage"""

    # Start monitoring
    try:
        # Initialize swarm agent
```

```
        swarm_agent = SolanaSwarmAgent(
            agent_name="Solana-Whale-Detector", model_name="gpt-4"
        )

        # Initialize monitor
        monitor = SolanaTransactionMonitor(
            rpc_url="wss://api.mainnet-beta.solana.com",
            swarm_agent=swarm_agent,
            min_sol_threshold=Decimal("100"),
        )

        await monitor.start_monitoring()
    except KeyboardInterrupt:
        logger.info("Shutting down gracefully...")


if __name__ == "__main__":
    asyncio.run(main())
```