Module 3: Transaction-Level Simulation

Introduction

Transaction-level simulation is critical for accurate MEV strategy backtesting. This module covers realistic transaction execution modeling, gas price dynamics, slippage calculation, and mempool-aware simulation. You'll learn to build sophisticated transaction simulators that account for network congestion, execution failures, and competitive dynamics.

Learning Objectives

By the end of this module, you will be able to:
- Build realistic transaction execution models with gas price simulation
- Implement mempool-aware simulation for competitive MEV strategies
- Calculate accurate slippage and price impact for different order types
- Simulate network congestion and transaction failures
- Model front-running and sandwich attacks with realistic dynamics

3.1 Transaction Execution Modeling

3.1.1 Basic Transaction Simulator

```
from dataclasses import dataclass
from typing import Dict, List, Optional, Any, Tuple
from enum import Enum
import numpy as np
from datetime import datetime, timedelta
import hashlib

class TransactionStatus(Enum):
    PENDING = "pending"
    INCLUDED = "included"
    FAILED = "failed"
    REPLACED = "replaced"
    CANCELLED = "cancelled"

class TransactionType(Enum):
    SWAP = "swap"
    MINT = "mint"
    BURN = "burn"
    LIQUIDATION = "liquidation"
    FLASH_LOAN = "flash_loan"
    ARBITRAGE = "arbitrage"

@dataclass
class TransactionRequest:
    """Transaction request before execution"""
    from_address: str
    to_address: str
    data: str
    gas_limit: int
    gas_price: int
    value: int = 0
    nonce: int = 0
    max_priority_fee: int = 0
    max_fee_per_gas: int = 0

@dataclass
class SimulatedTransaction:
    """Simulated transaction with realistic execution details"""
    tx_hash: str
    from_address: str
    to_address: str
    gas_used: int
```

```
    gas_price: int
    effective_gas_price: int
    status: TransactionStatus
    block_number: int
    timestamp: datetime
    nonce: int
    data: str
    value: int
    gas_limit: int
    effective_fee: int
    base_fee: int
    priority_fee: int

class TransactionSimulator:
    """Simulates realistic transaction execution"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.base_fee_per_gas = 20_000_000_000  # 20 gwei
        self.block_time = 12  # seconds
        self.max_priority_fee_per_gas = 5_000_000_000  # 5 gwei

    def simulate_transaction(self, tx_request: TransactionRequest,
                             current_base_fee: int = None,
                             network_conditions: Dict[str, Any] = None) -
> SimulatedTransaction:
        """Simulate single transaction execution"""

        if current_base_fee is None:
            current_base_fee = self.base_fee_per_gas

        if network_conditions is None:
            network_conditions = self._get_default_network_conditions()

        # Generate transaction hash
        tx_hash = self._generate_tx_hash(tx_request)

        # Calculate actual gas price
        effective_gas_price = self._calculate_effective_gas_price(
            tx_request.gas_price, current_base_fee, network_conditions
        )
```

```
        # Check if transaction would be included
        inclusion_probability = self._calculate_inclusion_probability(
            effective_gas_price, network_conditions
        )

        status = TransactionStatus.INCLUDED if np.random.random() <
inclusion_probability else TransactionStatus.FAILED

        # Simulate gas usage with realistic variation
        gas_used = self._simulate_gas_usage(tx_request.gas_limit,
tx_request.data)

        # Calculate inclusion timing
        block_number = self._simulate_inclusion_block(
            effective_gas_price, current_base_fee, network_conditions
        )

        timestamp = datetime.now() + timedelta(seconds=block_number *
self.block_time)

        return SimulatedTransaction(
            tx_hash=tx_hash,
            from_address=tx_request.from_address,
            to_address=tx_request.to_address,
            gas_used=gas_used,
            gas_price=tx_request.gas_price,
            effective_gas_price=effective_gas_price,
            status=status,
            block_number=block_number,
            timestamp=timestamp,
            nonce=tx_request.nonce,
            data=tx_request.data,
            value=tx_request.value,
            gas_limit=tx_request.gas_limit,
            effective_fee=gas_used * effective_gas_price,
            base_fee=current_base_fee,
            priority_fee=effective_gas_price - current_base_fee
        )

    def _generate_tx_hash(self, tx_request: TransactionRequest) -> str:
        """Generate realistic transaction hash"""
        tx_data = (
```

```
            f"{tx_request.from_address}{tx_request.to_address}"
            f"{tx_request.data}{tx_request.nonce}
{tx_request.gas_price}"
        ).encode()
        return "0x" + hashlib.sha256(tx_data).hexdigest()[:64]

    def _calculate_effective_gas_price(self, requested_gas_price: int,
                                       base_fee: int, network_conditions:
Dict[str, Any]) -> int:
        """Calculate effective gas price considering network
conditions"""
        # Base EIP-1559 calculation
        max_fee_per_gas = max(requested_gas_price, base_fee)
        priority_fee = min(
            requested_gas_price - base_fee,
            max_fee_per_gas - base_fee
        )

        # Apply network congestion adjustments
        congestion_multiplier =
network_conditions.get('congestion_multiplier', 1.0)
        effective_gas_price = int((base_fee + priority_fee) *
congestion_multiplier)

        return effective_gas_price

    def _calculate_inclusion_probability(self, gas_price: int,
                                         network_conditions: Dict[str,
Any]) -> float:
        """Calculate probability of transaction inclusion"""

        median_gas_price = network_conditions.get('median_gas_price',
20_000_000_000)
        inclusion_threshold =
network_conditions.get('inclusion_threshold', 0.5)

        if gas_price >= median_gas_price * 2:
            return 0.98  # Very high probability
        elif gas_price >= median_gas_price * 1.5:
            return 0.9   # High probability
        elif gas_price >= median_gas_price * 1.2:
            return 0.8   # Medium-high probability
```

```
        elif gas_price >= median_gas_price:
            return 0.6   # Medium probability
        elif gas_price >= median_gas_price * 0.8:
            return 0.4   # Low probability
        else:
            return 0.1   # Very low probability

    def _simulate_gas_usage(self, gas_limit: int, tx_data: str) -> int:
        """Simulate realistic gas usage with variations"""
        # Base gas estimation
        base_gas = len(tx_data) // 2 * 68  # Rough estimation

        # Add transaction-specific gas
        if 'swap' in tx_data.lower():
            base_gas += 150000  # DEX swap
        elif 'liquidation' in tx_data.lower():
            base_gas += 250000  # Liquidation
        elif 'flash' in tx_data.lower():
            base_gas += 300000  # Flash loan
        else:
            base_gas += 21000   # Simple transfer

        # Add realistic variation (±20%)
        variation = np.random.uniform(0.8, 1.2)
        gas_used = int(base_gas * variation)

        return min(gas_used, gas_limit)  # Cannot exceed gas limit

    def _simulate_inclusion_block(self, gas_price: int, base_fee: int,
                                  network_conditions: Dict[str, Any]) ->
int:
        """Simulate which block transaction would be included in"""

        median_gas_price = network_conditions.get('median_gas_price',
20_000_000_000)
        block_utilization = network_conditions.get('block_utilization',
0.5)

        # Calculate expected inclusion delay
        if gas_price >= median_gas_price * 2:
            expected_blocks = 1
        elif gas_price >= median_gas_price * 1.5:
```

```
            expected_blocks = 2
        elif gas_price >= median_gas_price:
            expected_blocks = 3
        else:
            expected_blocks = int(10 * block_utilization) + 1

        # Add randomness
        actual_blocks = max(1, int(np.random.poisson(expected_blocks)))

        return 18000000 + actual_blocks  # Current Ethereum block
number

    def _get_default_network_conditions(self) -> Dict[str, Any]:
        """Get default network conditions"""
        return {
            'median_gas_price': self.base_fee_per_gas,
            'congestion_multiplier': 1.0,
            'block_utilization': 0.5,
            'mempool_size': 1000,
            'inclusion_threshold': 0.5
        }

class BatchTransactionSimulator:
    """Simulates batch transaction execution with dependencies"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.tx_simulator = TransactionSimulator(config)
        self.nonces = {}  # Track nonces per address
        self.pending_transactions = []

    def simulate_batch_execution(self, transactions:
List[TransactionRequest],
                                 network_conditions: Dict[str, Any] =
None) -> List[SimulatedTransaction]:
        """Simulate execution of multiple transactions with
dependencies"""

        results = []

        # Sort transactions by gas price (highest first for MEV
strategies)
```

```
        sorted_txs = sorted(transactions, key=lambda x: x.gas_price,
reverse=True)

        for tx_request in sorted_txs:
            # Check nonce requirements
            if not self._validate_nonce(tx_request):
                # Create failed transaction
                failed_tx = self._create_failed_transaction(tx_request,
"Invalid nonce")
                results.append(failed_tx)
                continue

            # Simulate transaction
            simulated_tx = self.tx_simulator.simulate_transaction(
                tx_request, network_conditions=network_conditions
            )

            # Check for nonce conflicts with pending transactions
            if simulated_tx.status == TransactionStatus.INCLUDED:
                if simulated_tx.from_address in self.nonces:
                    if self.nonces[simulated_tx.from_address] !=
simulated_tx.nonce:
                        simulated_tx.status = TransactionStatus.FAILED
                        simulated_tx.effective_fee = 0

            results.append(simulated_tx)

            # Update nonce tracking
            if simulated_tx.status == TransactionStatus.INCLUDED:
                self.nonces[simulated_tx.from_address] =
simulated_tx.nonce + 1
            else:
                self.nonces[simulated_tx.from_address] =
simulated_tx.nonce + 1

        return results

    def _validate_nonce(self, tx_request: TransactionRequest) -> bool:
        """Validate transaction nonce"""
        if tx_request.from_address not in self.nonces:
            self.nonces[tx_request.from_address] = tx_request.nonce
            return True
```

```
        expected_nonce = self.nonces[tx_request.from_address]
        return tx_request.nonce == expected_nonce

    def _create_failed_transaction(self, tx_request:
TransactionRequest,
                                   failure_reason: str) ->
SimulatedTransaction:
        """Create a failed transaction result"""
        return SimulatedTransaction(
            tx_hash="0x" + "0" * 64,
            from_address=tx_request.from_address,
            to_address=tx_request.to_address,
            gas_used=0,
            gas_price=tx_request.gas_price,
            effective_gas_price=0,
            status=TransactionStatus.FAILED,
            block_number=0,
            timestamp=datetime.now(),
            nonce=tx_request.nonce,
            data=tx_request.data,
            value=tx_request.value,
            gas_limit=tx_request.gas_limit,
            effective_fee=0,
            base_fee=0,
            priority_fee=0
        )
```

3.1.2 Gas Price Dynamics Simulation

```
import pandas as pd
from typing import Dict, List, Any
from dataclasses import dataclass
from datetime import datetime, timedelta

@dataclass
class GasPriceSnapshot:
    """Gas price snapshot at a point in time"""
    timestamp: datetime
    base_fee_per_gas: int
    priority_fee_per_gas: int
    gas_price: int
    block_utilization: float
    pending_transactions: int
    median_gas_price: int

class GasPriceSimulator:
    """Simulates realistic gas price dynamics"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.base_fee_adjustment_factor = 0.125  # EIP-1559 base fee
adjustment
        self.history = []

    def generate_gas_price_series(self, duration_hours: int = 24,
                                  start_base_fee: int = 20_000_000_000) -
> pd.DataFrame:
        """Generate realistic gas price time series"""

        num_points = duration_hours * 60 * 5  # 5-minute intervals
        timestamps = pd.date_range(
            start=datetime.now() - timedelta(hours=duration_hours),
            periods=num_points,
            freq='5min'
        )

        # Initialize with base fee
        base_fees = [start_base_fee]

        # Simulate base fee changes (EIP-1559)
        for i in range(1, num_points):
```

```
            previous_base_fee = base_fees[-1]

            # Base fee adjusts by max ±12.5%
            adjustment_factor = np.random.uniform(0.875, 1.125)
            new_base_fee = int(previous_base_fee * adjustment_factor)

            # Add some mean reversion
            target_base_fee = 20_000_000_000  # 20 gwei
            reversion_factor = 0.01  # 1% reversion per step
            new_base_fee = int(
                new_base_fee * (1 - reversion_factor) + target_base_fee
* reversion_factor
            )

            base_fees.append(new_base_fee)

        # Generate priority fees
        priority_fees = []
        median_gas_prices = []
        block_utilizations = []

        for base_fee in base_fees:
            # Priority fee follows a distribution around the base fee
            priority_fee = np.random.lognormal(
                mean=np.log(base_fee * 1.1),  # Mean slightly above
base fee
                sigma=0.3  # Standard deviation
            )
            priority_fees.append(int(priority_fee))

            # Median gas price (some transactions pay more than base
fee)
            median_gas_price = base_fee + int(priority_fee * 0.5)
            median_gas_prices.append(median_gas_price)

            # Block utilization (affects gas prices)
            if base_fee > 50_000_000_000:  # High gas price
                utilization = np.random.uniform(0.7, 0.95)
            elif base_fee > 30_000_000_000:  # Medium gas price
                utilization = np.random.uniform(0.4, 0.8)
            else:  # Low gas price
                utilization = np.random.uniform(0.2, 0.6)
```

```
            block_utilizations.append(utilization)

        return pd.DataFrame({
            'timestamp': timestamps,
            'base_fee_per_gas': base_fees,
            'priority_fee_per_gas': priority_fees,
            'gas_price': [bf + pf for bf, pf in zip(base_fees,
priority_fees)],
            'median_gas_price': median_gas_prices,
            'block_utilization': block_utilizations
        })

    def simulate_network_congestion(self, gas_price_series:
pd.DataFrame) -> pd.DataFrame:
        """Simulate network congestion effects on gas prices"""

        df = gas_price_series.copy()

        # Identify congestion periods
        congestion_threshold = 0.8
        df['is_congested'] = df['block_utilization'] >
congestion_threshold

        # Gas price spikes during congestion
        spike_multiplier = np.where(
            df['is_congested'],
            np.random.uniform(2.0, 5.0, len(df)),
            np.random.uniform(0.8, 1.2, len(df))
        )

        # Apply congestion effects
        df['congested_gas_price'] = (df['gas_price'] *
spike_multiplier).astype(int)
        df['congestion_multiplier'] = spike_multiplier

        # Mempool size affects priority fees
        df['estimated_mempool_size'] = np.where(
            df['is_congested'],
            np.random.uniform(15000, 30000, len(df)),
            np.random.uniform(1000, 8000, len(df))
        )
```

```
        return df

    def simulate_gas_price_auction(self, pending_transactions:
List[Dict[str, Any]]) -> Dict[str, int]:
        """Simulate gas price auction within a block"""

        if not pending_transactions:
            return {}

        # Sort transactions by gas price (highest first)
        sorted_txs = sorted(pending_transactions, key=lambda x:
x['gas_price'], reverse=True)

        # Calculate gas limit of block
        block_gas_limit = 30_000_000  # 30M gas limit

        # Simulate block filling
        gas_used = 0
        inclusion_order = {}

        for tx in sorted_txs:
            if gas_used + tx['gas_limit'] <= block_gas_limit:
                # Transaction can fit in block
                inclusion_order[tx['tx_id']] = gas_used
                gas_used += tx['gas_limit']
            else:
                # Transaction doesn't fit, would need higher gas price
                break

        # Calculate effective gas prices based on inclusion position
        effective_gas_prices = {}
        for tx_id, gas_position in inclusion_order.items():
            # Slightly reduce gas price based on position
            original_gas_price = next(tx['gas_price'] for tx in
sorted_txs if tx['tx_id'] == tx_id)
            position_penalty = 0.95 ** (gas_position // 1_000_000)  #
Small penalty for position
            effective_gas_prices[tx_id] = int(original_gas_price *
position_penalty)

        return effective_gas_prices
```

```
    def add_flash_crash_events(self, gas_price_series: pd.DataFrame,
                               crash_probability: float = 0.05) ->
pd.DataFrame:
        """Add flash crash events to gas price series"""

        df = gas_price_series.copy()

        for i in range(len(df)):
            if np.random.random() < crash_probability:
                # Create a gas price crash (rapid drop)
                crash_start = i
                crash_duration = np.random.randint(10, 60)  # 10-60
periods

                for j in range(crash_duration):
                    if crash_start + j < len(df):
                        # Rapid drop followed by gradual recovery
                        if j < 5:
                            # Sharp drop
                            multiplier = 0.3 ** (j + 1)
                        else:
                            # Gradual recovery
                            recovery_factor = (j - 4) / (crash_duration
- 4)
                            multiplier = 0.3 + recovery_factor * 0.7

                        df.loc[crash_start + j, 'gas_price'] = int(
                            df.loc[crash_start + j, 'gas_price'] *
multiplier
                        )

        return df

# Usage example
gas_simulator = GasPriceSimulator({})
gas_series = gas_simulator.generate_gas_price_series(duration_hours=48,
start_base_fee=25_000_000_000)
congested_series =
gas_simulator.simulate_network_congestion(gas_series)
final_series = gas_simulator.add_flash_crash_events(congested_series)
```

```
print(f"Generated {len(final_series)} gas price data points")
print(final_series.head())
```

3.2 Slippage and Price Impact Modeling

3.2.1 AMM Price Impact Model

```
from typing import Dict, List, Any, Optional
from dataclasses import dataclass
from enum import Enum
import numpy as np

class AMMType(Enum):
    CONSTANT_PRODUCT = "constant_product"
    STABLE_SWAP = "stable_swap"
    CONCENTRATED_LIQUIDITY = "concentrated_liquidity"

@dataclass
class PoolState:
    """AMM pool state"""
    pool_address: str
    token0: str
    token1: str
    reserve0: float
    reserve1: float
    fee_tier: int
    amm_type: AMMType
    total_liquidity: float

@dataclass
class SwapResult:
    """Swap calculation result"""
    amount_in: float
    amount_out: float
    price_impact: float
    effective_price: float
    slippage: float
    fee_paid: float
    new_reserve0: float
    new_reserve1: float

class AMMSwapSimulator:
    """Simulates AMM swaps with realistic price impact"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.fee_tiers = {
            500: 0.0005,   # 0.05%
            3000: 0.003,   # 0.3%
```

```
            10000: 0.01    # 1%
        }

    def simulate_swap(self, pool: PoolState, amount_in: float,
                      token_in: str, slippage_tolerance: float = 0.005) -
> SwapResult:
        """Simulate swap in AMM pool"""

        # Determine pool parameters based on AMM type
        if pool.amm_type == AMMType.CONSTANT_PRODUCT:
            return self._simulate_constant_product_swap(
                pool, amount_in, token_in, slippage_tolerance
            )
        elif pool.amm_type == AMMType.STABLE_SWAP:
            return self._simulate_stable_swap(
                pool, amount_in, token_in, slippage_tolerance
            )
        elif pool.amm_type == AMMType.CONCENTRATED_LIQUIDITY:
            return self._simulate_concentrated_liquidity_swap(
                pool, amount_in, token_in, slippage_tolerance
            )
        else:
            raise ValueError(f"Unsupported AMM type: {pool.amm_type}")

    def _simulate_constant_product_swap(self, pool: PoolState,
amount_in: float,
                                        token_in: str,
slippage_tolerance: float) -> SwapResult:
        """Simulate swap in constant product AMM (Uniswap V2 style)"""

        # Get current reserves
        if token_in == pool.token0:
            reserve_in = pool.reserve0
            reserve_out = pool.reserve1
        else:
            reserve_in = pool.reserve1
            reserve_out = pool.reserve0

        # Calculate fee
        fee_rate = self.fee_tiers.get(pool.fee_tier, 0.003)
        amount_in_after_fee = amount_in * (1 - fee_rate)
```

```
        # Calculate amount out using constant product formula
        # (x + Δx)(y - Δy) = xy
        # Δy = y * Δx / (x + Δx)
        amount_out_before_slippage = reserve_out *
amount_in_after_fee / (reserve_in + amount_in_after_fee)

        # Calculate price impact
        price_before = reserve_out / reserve_in
        price_after = (reserve_out - amount_out_before_slippage) /
(reserve_in + amount_in_after_fee)
        price_impact = (price_before - price_after) / price_before

        # Apply slippage tolerance
        min_amount_out = amount_out_before_slippage * (1 -
slippage_tolerance)
        actual_amount_out = amount_out_before_slippage

        if actual_amount_out < min_amount_out:
            # Transaction would fail due to slippage
            actual_amount_out = 0

        # Calculate effective price
        effective_price = amount_in / actual_amount_out if
actual_amount_out > 0 else float('inf')

        # Calculate fee paid
        fee_paid = amount_in * fee_rate

        # Update reserves
        if token_in == pool.token0:
            new_reserve0 = reserve_in + amount_in
            new_reserve1 = reserve_out - actual_amount_out
        else:
            new_reserve0 = reserve_out - actual_amount_out
            new_reserve1 = reserve_in + amount_in

        return SwapResult(
            amount_in=amount_in,
            amount_out=actual_amount_out,
            price_impact=price_impact,
            effective_price=effective_price,
            slippage=slippage_tolerance,
```

```
            fee_paid=fee_paid,
            new_reserve0=new_reserve0,
            new_reserve1=new_reserve1
        )

    def _simulate_stable_swap(self, pool: PoolState, amount_in: float,
                              token_in: str, slippage_tolerance: float) -
> SwapResult:
        """Simulate swap in stable swap AMM (Curve style)"""

        # Stable swaps have lower price impact
        # Use a modified constant product with amplification factor

        amplification_factor = 10  # Example amplification factor

        if token_in == pool.token0:
            reserve_in = pool.reserve0
            reserve_out = pool.reserve1
        else:
            reserve_in = pool.reserve1
            reserve_out = pool.reserve0

        fee_rate = self.fee_tiers.get(pool.fee_tier, 0.0004)  # Lower
fee for stable swaps
        amount_in_after_fee = amount_in * (1 - fee_rate)

        # Stable swap formula with amplification
        k = amplification_factor
        amount_out_before_slippage = reserve_out *
amount_in_after_fee / (
            (reserve_in + amount_in_after_fee) * (1 + k) / (k *
reserve_in)
        )

        # Much lower price impact for stable swaps
        price_before = reserve_out / reserve_in
        price_after = (reserve_out - amount_out_before_slippage) /
(reserve_in + amount_in_after_fee)
        price_impact = (price_before - price_after) / price_before *
0.1  # Reduce by 90%

        # Apply slippage
```

```
        min_amount_out = amount_out_before_slippage * (1 -
slippage_tolerance)
        actual_amount_out = amount_out_before_slippage if
amount_out_before_slippage >= min_amount_out else 0

        effective_price = amount_in / actual_amount_out if
actual_amount_out > 0 else float('inf')
        fee_paid = amount_in * fee_rate

        # Update reserves
        if token_in == pool.token0:
            new_reserve0 = reserve_in + amount_in
            new_reserve1 = reserve_out - actual_amount_out
        else:
            new_reserve0 = reserve_out - actual_amount_out
            new_reserve1 = reserve_in + amount_in

        return SwapResult(
            amount_in=amount_in,
            amount_out=actual_amount_out,
            price_impact=price_impact,
            effective_price=effective_price,
            slippage=slippage_tolerance,
            fee_paid=fee_paid,
            new_reserve0=new_reserve0,
            new_reserve1=new_reserve1
        )

    def _simulate_concentrated_liquidity_swap(self, pool: PoolState,
amount_in: float,
                                              token_in: str,
slippage_tolerance: float) -> SwapResult:
        """Simulate swap in concentrated liquidity AMM (Uniswap V3
style)"""

        # Concentrated liquidity has higher price impact but only in
specific price ranges

        if token_in == pool.token0:
            reserve_in = pool.reserve0
            reserve_out = pool.reserve1
        else:
```

```
            reserve_in = pool.reserve1
            reserve_out = pool.reserve0

        fee_rate = self.fee_tiers.get(pool.fee_tier, 0.003)
        amount_in_after_fee = amount_in * (1 - fee_rate)

        # Concentrated liquidity has variable price impact based on
current price vs tick ranges
        # Simulate this by using different coefficients based on
position in range

        # Assume current price is in the middle of a range (normal
conditions)
        base_impact_factor = 0.5  # Lower base impact

        # Price impact calculation with concentrated liquidity effects
        amount_out_before_slippage = reserve_out *
amount_in_after_fee / (reserve_in + amount_in_after_fee)

        # Add concentrated liquidity adjustments
        base_price_impact = amount_in_after_fee / (reserve_in +
amount_in_after_fee)
        concentrated_impact = base_price_impact * base_impact_factor *
2  # Variable impact

        price_impact = concentrated_impact

        # Apply slippage
        min_amount_out = amount_out_before_slippage * (1 -
slippage_tolerance)
        actual_amount_out = amount_out_before_slippage if
amount_out_before_slippage >= min_amount_out else 0

        effective_price = amount_in / actual_amount_out if
actual_amount_out > 0 else float('inf')
        fee_paid = amount_in * fee_rate

        # Update reserves
        if token_in == pool.token0:
            new_reserve0 = reserve_in + amount_in
            new_reserve1 = reserve_out - actual_amount_out
        else:
```

```
            new_reserve0 = reserve_out - actual_amount_out
            new_reserve1 = reserve_in + amount_in

        return SwapResult(
            amount_in=amount_in,
            amount_out=actual_amount_out,
            price_impact=price_impact,
            effective_price=effective_price,
            slippage=slippage_tolerance,
            fee_paid=fee_paid,
            new_reserve0=new_reserve0,
            new_reserve1=new_reserve1
        )

    def simulate_multi_hop_swap(self, pools: List[PoolState],
amount_in: float,
                                path: List[str], slippage_tolerance:
float = 0.005) -> SwapResult:
        """Simulate multi-hop swap across multiple pools"""

        current_amount = amount_in
        current_token = path[0]

        total_price_impact = 0
        total_slippage = 0
        total_fee_paid = 0

        # Execute swaps through the path
        for i, token_out in enumerate(path[1:], 1):
            # Find pool for this token pair
            pool = None
            for p in pools:
                if ((p.token0 == current_token and p.token1 ==
token_out) or
                    (p.token1 == current_token and p.token0 ==
token_out)):
                    pool = p
                    break

            if pool is None:
                # Pool not found, swap fails
                return SwapResult(
```

```
                    amount_in=amount_in,
                    amount_out=0,
                    price_impact=1.0,
                    effective_price=float('inf'),
                    slippage=1.0,
                    fee_paid=0,
                    new_reserve0=0,
                    new_reserve1=0
                )

            # Execute swap
            swap_result = self.simulate_swap(pool, current_amount,
current_token, slippage_tolerance)

            if swap_result.amount_out == 0:
                # Swap failed
                return swap_result

            # Accumulate metrics
            total_price_impact += swap_result.price_impact
            total_slippage = max(total_slippage, swap_result.slippage)
            total_fee_paid += swap_result.fee_paid

            # Update for next hop
            current_amount = swap_result.amount_out
            current_token = token_out

            # Update pool reserves for next iteration
            if current_token == pool.token0:
                pool.reserve0 = swap_result.new_reserve0
                pool.reserve1 = swap_result.new_reserve1
            else:
                pool.reserve0 = swap_result.new_reserve1
                pool.reserve1 = swap_result.new_reserve0

        # Return final result
        return SwapResult(
            amount_in=amount_in,
            amount_out=current_amount,
            price_impact=total_price_impact / len(path),  # Average
price impact
            effective_price=amount_in / current_amount,
```

```
            slippage=total_slippage,
            fee_paid=total_fee_paid,
            new_reserve0=0,  # Not applicable for multi-hop
            new_reserve1=0
        )

# Example usage
pool = PoolState(
    pool_address="0x1234",
    token0="WETH",
    token1="USDC",
    reserve0=1000.0,
    reserve1=2000000.0,
    fee_tier=3000,
    amm_type=AMMType.CONSTANT_PRODUCT,
    total_liquidity=2000000.0
)

simulator = AMMSwapSimulator({})
swap_result = simulator.simulate_swap(pool, 10.0, "WETH", 0.01)

print(f"Swap result: {swap_result.amount_out:.2f} USDC")
print(f"Price impact: {swap_result.price_impact:.4f}")
print(f"Effective price: {swap_result.effective_price:.2f}")
print(f"Fee paid: {swap_result.fee_paid:.4f} WETH")
```

3.2.2 Mempool and Front-Running Simulation

```
from typing import Dict, List, Any, Optional
from dataclasses import dataclass
from datetime import datetime, timedelta
import heapq

@dataclass
class MempoolTransaction:
    """Transaction in the mempool"""
    tx_id: str
    from_address: str
    to_address: str
    data: str
    gas_price: int
    gas_limit: int
    value: int
    timestamp: datetime
    priority: int  # Higher priority = better gas price
    estimated_inclusion_time: int  # Blocks until inclusion

class MempoolSimulator:
    """Simulates mempool dynamics and front-running"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.mempool = []
        self.processed_transactions = set()

    def add_transaction(self, tx: MempoolTransaction):
        """Add transaction to mempool"""
        # Add to heap for priority ordering
        heapq.heappush(self.mempool, (-tx.priority, tx.timestamp, tx))

    def simulate_block_formation(self, block_gas_limit: int =
30_000_000) -> List[MempoolTransaction]:
        """Simulate which transactions get included in a block"""

        included_txs = []
        gas_used = 0

        # Process transactions in priority order
        while self.mempool and gas_used < block_gas_limit:
            priority, timestamp, tx = heapq.heappop(self.mempool)
```

```
            # Check if transaction fits in remaining gas
            if gas_used + tx.gas_limit <= block_gas_limit:
                tx.estimated_inclusion_time = 0  # Included in current
block
                included_txs.append(tx)
                self.processed_transactions.add(tx.tx_id)
                gas_used += tx.gas_limit
            else:
                # Transaction doesn't fit, re-add with increased
inclusion time
                tx.estimated_inclusion_time += 1
                heapq.heappush(self.mempool, (-tx.priority, timestamp,
tx))

        return included_txs

    def simulate_front_running(self, victim_tx: MempoolTransaction,
                               attacker_gas_price_multiplier: float =
1.1) -> Dict[str, Any]:
        """Simulate front-running attack"""

        # Create front-running transaction
        front_run_tx = MempoolTransaction(
            tx_id=f"fr_{victim_tx.tx_id}",
            from_address="attacker_address",
            to_address=victim_tx.to_address,
            data=victim_tx.data,
            gas_price=int(victim_tx.gas_price *
attacker_gas_price_multiplier),
            gas_limit=victim_tx.gas_limit,
            value=victim_tx.value,
            timestamp=victim_tx.timestamp +
timedelta(milliseconds=100),  # Slightly later
            priority=victim_tx.priority + 1,  # Higher priority
            estimated_inclusion_time=0
        )

        # Create back-running transaction
        back_run_tx = MempoolTransaction(
            tx_id=f"br_{victim_tx.tx_id}",
            from_address="attacker_address",
```

```
            to_address=victim_tx.to_address,
            data=victim_tx.data,
            gas_price=victim_tx.gas_price,  # Same gas price
            gas_limit=victim_tx.gas_limit,
            value=victim_tx.value,
            timestamp=victim_tx.timestamp + timedelta(seconds=2),  #
Later
            priority=victim_tx.priority,  # Same priority
            estimated_inclusion_time=1  # Next block
        )

        # Add attack transactions to mempool
        self.add_transaction(front_run_tx)
        self.add_transaction(back_run_tx)

        # Simulate block inclusion
        block_txs = self.simulate_block_formation()

        # Determine if attack was successful
        attack_successful = (
            front_run_tx in block_txs and
            back_run_tx in block_txs and
            victim_tx not in block_txs
        )

        return {
            'attack_successful': attack_successful,
            'front_run_included': front_run_tx in block_txs,
            'back_run_included': back_run_tx in block_txs,
            'victim_blocked': victim_tx not in block_txs,
            'gas_cost': (front_run_tx.gas_limit +
back_run_tx.gas_limit) * front_run_tx.gas_price,
            'expected_profit': self._calculate_attack_profit(victim_tx,
block_txs)
        }

    def _calculate_attack_profit(self, victim_tx: MempoolTransaction,
                                 block_txs: List[MempoolTransaction]) ->
float:
        """Calculate expected profit from front-running attack"""

        # This is a simplified calculation
```

```
        # In reality, you'd need to simulate the actual trade execution

        # Assume victim transaction would have generated $100 profit
        base_profit = 100.0

        # Reduce profit based on competition
        competition_factor = 1.0 - (len([tx for tx in block_txs if
tx.tx_id != victim_tx.tx_id]) * 0.1)

        # Calculate gas costs
        gas_costs = 0
        for tx in block_txs:
            if tx.tx_id.startswith(('fr_', 'br_')):  # Attack
transactions
                gas_costs += tx.gas_limit * tx.gas_price * 2e-9  #
Convert to ETH, then to USD (assume 1 ETH = $2000)

        return base_profit * competition_factor - gas_costs * 2000

    def simulate_sandwich_attack(self, victim_tx: MempoolTransaction,
                                 sandwich_margin: float = 0.02) ->
Dict[str, Any]:
        """Simulate sandwich attack"""

        # Create buy transaction (slightly higher gas price)
        buy_tx = MempoolTransaction(
            tx_id=f"buy_{victim_tx.tx_id}",
            from_address="sandwich_bot",
            to_address=victim_tx.to_address,
            data=victim_tx.data,
            gas_price=int(victim_tx.gas_price * 1.05),  # 5% higher
            gas_limit=victim_tx.gas_limit,
            value=victim_tx.value,
            timestamp=victim_tx.timestamp + timedelta(milliseconds=50),
            priority=victim_tx.priority + 1,
            estimated_inclusion_time=0
        )

        # Create sell transaction (same gas price as victim)
        sell_tx = MempoolTransaction(
            tx_id=f"sell_{victim_tx.tx_id}",
            from_address="sandwich_bot",
```

```
            to_address=victim_tx.to_address,
            data=victim_tx.data,
            gas_price=victim_tx.gas_price,
            gas_limit=victim_tx.gas_limit,
            value=victim_tx.value,
            timestamp=victim_tx.timestamp + timedelta(seconds=2),
            priority=victim_tx.priority,
            estimated_inclusion_time=1
        )

        # Add sandwich transactions
        self.add_transaction(buy_tx)
        self.add_transaction(sell_tx)

        # Simulate block inclusion
        block_txs = self.simulate_block_formation()

        sandwich_success = buy_tx in block_txs and sell_tx in block_txs

        # Calculate sandwich profit
        sandwich_profit = self._calculate_sandwich_profit(victim_tx,
sandwich_margin)
        sandwich_gas_cost = (buy_tx.gas_limit * buy_tx.gas_price +
                             sell_tx.gas_limit * sell_tx.gas_price) *
2e-9 * 2000

        return {
            'sandwich_successful': sandwich_success,
            'buy_included': buy_tx in block_txs,
            'sell_included': sell_tx in block_txs,
            'expected_profit': sandwich_profit - sandwich_gas_cost,
            'sandwich_margin': sandwich_margin
        }

    def _calculate_sandwich_profit(self, victim_tx: MempoolTransaction,
                                   margin: float) -> float:
        """Calculate profit from sandwich attack"""

        # Estimate victim's trade impact
        trade_size = victim_tx.value * 2e-9  # Convert to ETH
(simplified)
```

```
        # Assume victim trade causes 2% price movement
        price_impact = 0.02

        # Sandwich bot profits from both sides of the spread
        buy_profit = trade_size * price_impact * margin
        sell_profit = trade_size * price_impact * margin

        return buy_profit + sell_profit

    def simulate_mev_competition(self, opportunities: List[Dict[str,
Any]],
                                 participants: List[str]) -> Dict[str,
Any]:
        """Simulate competition for MEV opportunities"""

        results = {
            'successful_attacks': [],
            'failed_attacks': [],
            'competition_metrics': {}
        }

        # Create transactions for each participant for each opportunity
        all_transactions = []

        for i, opportunity in enumerate(opportunities):
            for participant in participants:
                # Each participant tries to capture the opportunity
                tx = MempoolTransaction(
                    tx_id=f"mev_{participant}_{i}",
                    from_address=participant,
                    to_address=opportunity.get('target_contract',
'0x1234'),
                    data=opportunity.get('calldata', '0x'),
                    gas_price=np.random.uniform(20_000_000_000,
50_000_000_000),  # High gas prices
                    gas_limit=opportunity.get('gas_limit', 200_000),
                    value=opportunity.get('value', 0),
                    timestamp=datetime.now() + timedelta(milliseconds=i
* 100),
                    priority=0,
                    estimated_inclusion_time=0
                )
```

```
                all_transactions.append(tx)

        # Add all transactions to mempool
        for tx in all_transactions:
            self.add_transaction(tx)

        # Simulate competition over multiple blocks
        block_number = 0
        captured_opportunities = set()

        while self.mempool and block_number < 10:  # Max 10 blocks
            block_txs = self.simulate_block_formation()

            # Check which opportunities were captured
            for tx in block_txs:
                opportunity_id = tx.tx_id.split('_')[-1]
                if opportunity_id.isdigit() and int(opportunity_id) not
in captured_opportunities:
                    captured_opportunities.add(int(opportunity_id))
                    results['successful_attacks'].append({
                        'participant': tx.from_address,
                        'opportunity_id': int(opportunity_id),
                        'gas_cost': tx.gas_limit * tx.gas_price * 2e-9
* 2000,
                        'block_number': block_number
                    })

            block_number += 1

        # Calculate competition metrics
        total_opportunities = len(opportunities)
        successful_captures = len(captured_opportunities)

        results['competition_metrics'] = {
            'total_opportunities': total_opportunities,
            'successful_captures': successful_captures,
            'capture_rate': successful_captures / total_opportunities
if total_opportunities > 0 else 0,
            'average_blocks_to_capture': np.mean([att['block_number']
for att in results['successful_attacks']]) if
results['successful_attacks'] else 0
        }
```

```
        return results

# Example usage
mempool = MempoolSimulator({})

# Create a victim transaction
victim_tx = MempoolTransaction(
    tx_id="victim_123",
    from_address="user_abc",
    to_address="uniswap_v3",
    data="swap_eth_for_usdc",
    gas_price=25_000_000_000,
    gas_limit=150_000,
    value=1_000_000_000_000_000_000,  # 1 ETH
    timestamp=datetime.now(),
    priority=50,
    estimated_inclusion_time=2
)

# Simulate front-running
fr_result = mempool.simulate_front_running(victim_tx,
attacker_gas_price_multiplier=1.2)
print(f"Front-running successful: {fr_result['attack_successful']}")
print(f"Expected profit: ${fr_result['expected_profit']:.2f}")

# Simulate sandwich attack
sandwich_result = mempool.simulate_sandwich_attack(victim_tx,
sandwich_margin=0.02)
print(f"Sandwich successful: {sandwich_result['sandwich_successful']}")
print(f"Expected profit: ${sandwich_result['expected_profit']:.2f}")
```

3.3 Network Congestion Simulation

3.3.1 Block Space Competition Model

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

@dataclass
class Block:
    """Represents a blockchain block"""
    block_number: int
    timestamp: datetime
    gas_limit: int
    gas_used: int
    base_fee_per_gas: int
    transactions: List['NetworkTransaction']

@dataclass
class NetworkTransaction:
    """Transaction in the network"""
    tx_id: str
    sender: str
    recipient: str
    gas_limit: int
    gas_price: int
    priority_fee: int
    max_fee_per_gas: int
    value: int
    data: str
    arrival_time: datetime
    estimated_inclusion_time: int

class NetworkCongestionSimulator:
    """Simulates network congestion and block space competition"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.block_gas_limit = 30_000_000
        self.target_gas_per_block = 15_000_000  # 50% utilization
target
        self.base_fee_adjustment_factor = 0.125  # EIP-1559

        self.pending_transactions = []
```

```
        self.block_history = []

    def simulate_network_load(self, duration_blocks: int = 100) ->
List[Block]:
        """Simulate network load over multiple blocks"""

        blocks = []
        current_base_fee = 20_000_000_000  # Starting at 20 gwei

        for block_num in range(18000000, 18000000 + duration_blocks):

            # Generate transactions for this block
            new_transactions =
self._generate_transactions_for_block(block_num)
            self.pending_transactions.extend(new_transactions)

            # Sort transactions by effective gas price
            sorted_txs = sorted(
                self.pending_transactions,
                key=lambda x: (x.max_fee_per_gas, x.gas_limit),
                reverse=True
            )

            # Create block
            block = self._create_block(block_num, sorted_txs,
current_base_fee)
            blocks.append(block)

            # Remove included transactions
            included_tx_ids = {tx.tx_id for tx in block.transactions}
            self.pending_transactions = [
                tx for tx in self.pending_transactions
                if tx.tx_id not in included_tx_ids
            ]

            # Update base fee for next block
            current_base_fee = self._update_base_fee(
                block.gas_used, current_base_fee
            )

        return blocks
```

```
    def _generate_transactions_for_block(self, block_number: int) ->
List[NetworkTransaction]:
        """Generate realistic transactions for a block"""

        num_transactions = np.random.poisson(150)  # Average 150
transactions per block

        transactions = []

        for i in range(num_transactions):
            # Transaction types with realistic probabilities
            tx_type = np.random.choice(['transfer', 'swap', 'defi',
'mev'],
                                       p=[0.6, 0.25, 0.10, 0.05])

            if tx_type == 'transfer':
                gas_limit = 21000
                base_gas_price = np.random.normal(22_000_000_000,
3_000_000_000)
            elif tx_type == 'swap':
                gas_limit = np.random.randint(100_000, 200_000)
                base_gas_price = np.random.normal(30_000_000_000,
8_000_000_000)
            elif tx_type == 'defi':
                gas_limit = np.random.randint(200_000, 500_000)
                base_gas_price = np.random.normal(40_000_000_000,
15_000_000_000)
            else:  # MEV
                gas_limit = np.random.randint(150_000, 300_000)
                base_gas_price = np.random.normal(50_000_000_000,
20_000_000_000)

            # Apply congestion effects
            congestion_multiplier = 1.0
            if len(transactions) > 100:  # High load
                congestion_multiplier = 1.5
            elif len(transactions) > 50:  # Medium load
                congestion_multiplier = 1.2

            gas_price = int(base_gas_price * congestion_multiplier)

            tx = NetworkTransaction(
```

```
tx_id=f"0x{''.join(np.random.choice(list('0123456789abcdef'), 64))}",

sender=f"0x{''.join(np.random.choice(list('0123456789abcdef'), 40))}",

recipient=f"0x{''.join(np.random.choice(list('0123456789abcdef'),
40))}",
                gas_limit=gas_limit,
                gas_price=gas_price,
                priority_fee=gas_price * 0.3,  # 30% of gas price as
priority fee
                max_fee_per_gas=gas_price * 2,  # Max 2x gas price
                value=np.random.randint(0, 10_000_000_000_000_000_000),
# Up to 10 ETH
                data="0x" +
"".join(np.random.choice(list('0123456789abcdef'), 64)),
                arrival_time=datetime.now() +
timedelta(seconds=block_number * 12 + i * 0.1),
                estimated_inclusion_time=0
            )

            transactions.append(tx)

        return transactions

    def _create_block(self, block_number: int, sorted_txs:
List[NetworkTransaction],
                      base_fee: int) -> Block:
        """Create a block from sorted transactions"""

        gas_used = 0
        included_transactions = []

        for tx in sorted_txs:
            if gas_used + tx.gas_limit <= self.block_gas_limit:
                included_transactions.append(tx)
                gas_used += tx.gas_limit

                # Update estimated inclusion time
                tx.estimated_inclusion_time = 0
            else:
                # Transaction doesn't fit, will be included in future
```

```
blocks
                tx.estimated_inclusion_time += 1

        # Calculate average effective gas price
        if included_transactions:
            effective_gas_prices = [
                min(tx.gas_price, base_fee + tx.priority_fee) *
tx.gas_limit
                for tx in included_transactions
            ]
            avg_effective_gas_price = sum(effective_gas_prices) /
sum(tx.gas_limit for tx in included_transactions)
        else:
            avg_effective_gas_price = base_fee

        return Block(
            block_number=block_number,
            timestamp=datetime.now() + timedelta(seconds=block_number *
12),
            gas_limit=self.block_gas_limit,
            gas_used=gas_used,
            base_fee_per_gas=base_fee,
            transactions=included_transactions
        )

    def _update_base_fee(self, gas_used: int, current_base_fee: int) ->
int:
        """Update base fee according to EIP-1559"""

        if gas_used > self.target_gas_per_block:
            # Gas used exceeds target, increase base fee
            usage_ratio = gas_used / self.target_gas_per_block
            adjustment = current_base_fee * usage_ratio *
self.base_fee_adjustment_factor
            new_base_fee = int(current_base_fee + adjustment)
        else:
            # Gas used below target, decrease base fee
            usage_ratio = gas_used / self.target_gas_per_block
            adjustment = current_base_fee * usage_ratio *
self.base_fee_adjustment_factor
            new_base_fee = int(current_base_fee - adjustment)
```

```
        # Ensure base fee doesn't go negative or extreme
        new_base_fee = max(1_000_000_000, min(new_base_fee,
1_000_000_000_000))  # 1 gwei to 1000 gwei

        return new_base_fee

    def simulate_transaction_failure_scenarios(self, transactions:
List[NetworkTransaction]) -> Dict[str, Any]:
        """Simulate various transaction failure scenarios"""

        results = {
            'insufficient_gas': 0,
            'nonce_conflicts': 0,
            'replaced_by_higher_gas': 0,
            'out_of_gas': 0,
            'execution_failed': 0,
            'successful': 0
        }

        for tx in transactions:

            # Scenario 1: Insufficient gas price
            median_gas_price = 30_000_000_000  # Assume median gas
price
            if tx.gas_price < median_gas_price * 0.7:
                results['insufficient_gas'] += 1
                continue

            # Scenario 2: Nonce conflicts (simplified)
            if np.random.random() < 0.02:  # 2% chance of nonce
conflict
                results['nonce_conflicts'] += 1
                continue

            # Scenario 3: Replaced by higher gas price transaction
            if np.random.random() < 0.05:  # 5% chance of replacement
                results['replaced_by_higher_gas'] += 1
                continue

            # Scenario 4: Out of gas during execution
            if tx.gas_limit < 50000:  # Very low gas limit
                results['out_of_gas'] += 1
```

```
                continue

            # Scenario 5: Execution failed (revert, etc.)
            if np.random.random() < 0.01:  # 1% chance of execution
failure
                results['execution_failed'] += 1
                continue

            # Transaction successful
            results['successful'] += 1

        return results

    def analyze_congestion_patterns(self, blocks: List[Block]) ->
Dict[str, Any]:
        """Analyze congestion patterns in block data"""

        if not blocks:
            return {}

        gas_usage = [block.gas_used for block in blocks]
        base_fees = [block.base_fee_per_gas for block in blocks]
        transaction_counts = [len(block.transactions) for block in
blocks]

        analysis = {
            'average_gas_usage': np.mean(gas_usage),
            'gas_usage_volatility': np.std(gas_usage),
            'average_base_fee': np.mean(base_fees),
            'base_fee_volatility': np.std(base_fees),
            'average_transactions_per_block':
np.mean(transaction_counts),
            'congestion_periods':
self._identify_congestion_periods(blocks),
            'fee_spike_analysis': self._analyze_fee_spikes(blocks)
        }

        return analysis

    def _identify_congestion_periods(self, blocks: List[Block]) ->
List[Dict[str, Any]]:
        """Identify periods of network congestion"""
```

```
        congestion_periods = []
        current_period = None

        for block in blocks:
            utilization = block.gas_used / block.gas_limit

            if utilization > 0.9:  # High congestion threshold
                if current_period is None:
                    current_period = {
                        'start_block': block.block_number,
                        'start_time': block.timestamp,
                        'peak_utilization': utilization,
                        'max_base_fee': block.base_fee_per_gas
                    }
                else:
                    # Update current period
                    current_period['peak_utilization'] = max(
                        current_period['peak_utilization'], utilization
                    )
                    current_period['max_base_fee'] = max(
                        current_period['max_base_fee'],
block.base_fee_per_gas
                    )
            else:
                if current_period is not None:
                    # End current congestion period
                    current_period['end_block'] = block.block_number
                    current_period['end_time'] = block.timestamp
                    current_period['duration_blocks'] = (
                        current_period['end_block'] -
current_period['start_block']
                    )
                    congestion_periods.append(current_period)
                    current_period = None

        # Handle case where congestion period extends to end
        if current_period is not None:
            last_block = blocks[-1]
            current_period['end_block'] = last_block.block_number
            current_period['end_time'] = last_block.timestamp
            current_period['duration_blocks'] = (
```

```
                current_period['end_block'] -
current_period['start_block']
            )
            congestion_periods.append(current_period)

        return congestion_periods

    def _analyze_fee_spikes(self, blocks: List[Block]) -> Dict[str,
Any]:
        """Analyze gas fee spikes in block data"""

        base_fees = [block.base_fee_per_gas for block in blocks]

        # Calculate fee spikes (significant increases from previous
block)
        fee_increases = []
        for i in range(1, len(base_fees)):
            increase = (base_fees[i] - base_fees[i-1]) / base_fees[i-1]
            fee_increases.append(increase)

        # Identify spikes (>50% increase)
        spike_threshold = 0.5
        spikes = [inc for inc in fee_increases if inc >
spike_threshold]

        return {
            'total_spikes': len(spikes),
            'spike_rate': len(spikes) / len(fee_increases) if
fee_increases else 0,
            'average_spike_magnitude': np.mean(spikes) if spikes else
0,
            'max_spike_magnitude': max(spikes) if spikes else 0,
            'largest_fee': max(base_fees),
            'fee_volatility': np.std(base_fees)
        }

# Example usage
congestion_sim = NetworkCongestionSimulator({})

# Simulate network load
blocks = congestion_sim.simulate_network_load(duration_blocks=50)
```

```
print(f"Generated {len(blocks)} blocks")
print(f"Average gas usage: {np.mean([b.gas_used for b in blocks]):.
0f}")
print(f"Average base fee: {np.mean([b.base_fee_per_gas for b in
blocks]):.0f} gwei")

# Analyze congestion patterns
analysis = congestion_sim.analyze_congestion_patterns(blocks)
print(f"Identified {len(analysis.get('congestion_periods', []))}
congestion periods")
print(f"Fee spike rate: {analysis.get('fee_spike_analysis',
{}).get('spike_rate', 0):.1%}")
```

3.4 Practical Exercise: Complete Transaction Simulation System

```
def build_transaction_simulation_system():
    """Complete exercise to build a transaction simulation system"""

    print("Building Transaction-Level Simulation System...")

    # Initialize simulation components
    config = {
        'base_fee_per_gas': 20_000_000_000,
        'block_time': 12,
        'max_priority_fee': 5_000_000_000
    }

    # 1. Basic Transaction Simulator
    print("\n1. Initializing Transaction Simulator...")
    tx_simulator = TransactionSimulator(config)

    # Test single transaction
    tx_request = TransactionRequest(
        from_address="0x1234567890123456789012345678901234567890",
        to_address="0x0987654321098765432109876543210987654321",
        data="0x" + "1" * 100,  # Sample swap data
        gas_limit=200_000,
        gas_price=30_000_000_000,  # 30 gwei
        value=1_000_000_000_000_000_000  # 1 ETH
    )

    simulated_tx = tx_simulator.simulate_transaction(tx_request)
    print(f"   Transaction {simulated_tx.tx_hash[:10]}...")
    print(f"   Status: {simulated_tx.status.value}")
    print(f"   Gas used: {simulated_tx.gas_used}")
    print(f"   Effective gas price:
{simulated_tx.effective_gas_price / 1e9:.1f} gwei")

    # 2. AMM Swap Simulation
    print("\n2. Testing AMM Swap Simulation...")

    # Create Uniswap V2 pool
    uniswap_pool = PoolState(
        pool_address="0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
        token0="WETH",
        token1="USDC",
        reserve0=1000.0,
```

```
        reserve1=2_000_000.0,
        fee_tier=3000,
        amm_type=AMMType.CONSTANT_PRODUCT,
        total_liquidity=2_000_000.0
    )

    amm_simulator = AMMSwapSimulator({})
    swap_result = amm_simulator.simulate_swap(uniswap_pool, 10.0,
"WETH", 0.01)

    print(f"   Input: 10.0 WETH")
    print(f"   Output: {swap_result.amount_out:.2f} USDC")
    print(f"   Price impact: {swap_result.price_impact:.4f}")
    print(f"   Fee paid: {swap_result.fee_paid:.4f} WETH")

    # 3. Mempool Simulation
    print("\n3. Testing Mempool Simulation...")

    mempool = MempoolSimulator({})

    # Create victim transaction
    victim_tx = MempoolTransaction(
        tx_id="victim_123",
        from_address="user_123",
        to_address="uniswap_v3",
        data="swap_eth_for_usdc",
        gas_price=25_000_000_000,
        gas_limit=150_000,
        value=1_000_000_000_000_000_000,
        timestamp=datetime.now(),
        priority=50,
        estimated_inclusion_time=0
    )

    # Test front-running
    fr_result = mempool.simulate_front_running(victim_tx, 1.2)
    print(f"   Front-running successful:
{fr_result['attack_successful']}")
    print(f"   Expected profit: ${fr_result['expected_profit']:.2f}")

    # Test sandwich attack
    sandwich_result = mempool.simulate_sandwich_attack(victim_tx, 0.02)
```

```
    print(f"   Sandwich successful:
{sandwich_result['sandwich_successful']}")
    print(f"   Expected profit: ${sandwich_result['expected_profit']:.
2f}")

    # 4. Network Congestion Simulation
    print("\n4. Testing Network Congestion Simulation...")

    congestion_sim = NetworkCongestionSimulator({})
    blocks = congestion_sim.simulate_network_load(24)  # 24 blocks

    print(f"   Generated {len(blocks)} blocks")
    gas_usages = [b.gas_used for b in blocks]
    base_fees = [b.base_fee_per_gas for b in blocks]

    print(f"   Average gas usage: {np.mean(gas_usages):.0f} /
{congestion_sim.block_gas_limit:,}")
    print(f"   Average base fee: {np.mean(base_fees) / 1e9:.1f} gwei")
    print(f"   Max base fee: {max(base_fees) / 1e9:.1f} gwei")

    # Analyze congestion
    analysis = congestion_sim.analyze_congestion_patterns(blocks)
    print(f"   Congestion periods:
{len(analysis.get('congestion_periods', []))}")

    # 5. Batch Transaction Simulation
    print("\n5. Testing Batch Transaction Simulation...")

    batch_simulator = BatchTransactionSimulator({})

    # Create batch of transactions
    batch_txs = []
    for i in range(10):
        batch_txs.append(TransactionRequest(
            from_address=f"0x{'1234567890abcdef' * 4}",
            to_address=f"0x{'0987654321fedcba' * 4}",
            data=f"0x{i:064x}",
            gas_limit=150_000 + i * 10_000,
            gas_price=30_000_000_000 + i * 5_000_000_000,
            nonce=i
        ))
```

```
    batch_results = batch_simulator.simulate_batch_execution(batch_txs)

    successful_txs = sum(1 for tx in batch_results if tx.status ==
TransactionStatus.INCLUDED)
    print(f"   Batch processed: {successful_txs}/{len(batch_txs)}
successful")
    print(f"   Total gas cost: {sum(tx.effective_fee for tx in
batch_results if tx.status == TransactionStatus.INCLUDED) / 1e18:.4f}
ETH")

    # 6. Gas Price Dynamics
    print("\n6. Testing Gas Price Dynamics...")

    gas_sim = GasPriceSimulator({})
    gas_series = gas_sim.generate_gas_price_series(duration_hours=6)

    print(f"   Generated {len(gas_series)} gas price points")
    print(f"   Base fee range: {gas_series['base_fee_per_gas'].min() /
1e9:.1f} - {gas_series['base_fee_per_gas'].max() / 1e9:.1f} gwei")

    # Add congestion effects
    congested_series = gas_sim.simulate_network_congestion(gas_series)
    congestion_periods = (congested_series['is_congested'] ==
True).sum()
    print(f"   Congestion periods: {congestion_periods}")

    print("\n✅ Transaction Simulation System Test Complete!")

    # Summary statistics
    print("\n" + "="*50)
    print("SIMULATION SYSTEM SUMMARY")
    print("="*50)
    print(f"Transaction Success Rate: {(simulated_tx.status ==
TransactionStatus.INCLUDED) * 100:.1f}%")
    print(f"Swap Price Impact: {swap_result.price_impact:.4f}
({swap_result.price_impact * 100:.2f}%)")
    print(f"Front-running Success: {fr_result['attack_successful']}")
    print(f"Network Utilization: {np.mean(gas_usages) /
congestion_sim.block_gas_limit:.1%}")
    print(f"Gas Price Volatility: {np.std(base_fees) /
np.mean(base_fees):.2%}")
```

```
    return {
        'tx_simulator': tx_simulator,
        'amm_simulator': amm_simulator,
        'mempool': mempool,
        'congestion_sim': congestion_sim,
        'batch_simulator': batch_simulator,
        'gas_simulator': gas_sim,
        'test_results': {
            'single_tx_success': simulated_tx.status ==
TransactionStatus.INCLUDED,
            'swap_impact': swap_result.price_impact,
            'front_running_success': fr_result['attack_successful'],
            'sandwich_success': sandwich_result['sandwich_successful'],
            'avg_network_utilization': np.mean(gas_usages) /
congestion_sim.block_gas_limit,
            'batch_success_rate': successful_txs / len(batch_txs)
        }
    }

# Run the exercise
if __name__ == "__main__":
    simulation_results = build_transaction_simulation_system()
    print("\nTransaction-level simulation system is ready for MEV
strategy backtesting!")
```

Module Summary

In this module, you have built a comprehensive transaction-level simulation system that includes:

- **Realistic Transaction Execution**: Gas price dynamics, inclusion probability, and execution failures
- **AMM Price Impact Models**: Constant product, stable swap, and concentrated liquidity simulations
- **Mempool Simulation**: Front-running, sandwich attacks, and MEV competition modeling
- **Network Congestion**: Block space competition, gas fee dynamics, and failure scenarios
- **Batch Processing**: Multiple transaction simulation with dependencies and nonce tracking
- **Complete Integration**: All components working together for realistic backtesting

Key Takeaways

1. **Realistic Execution Modeling**: Account for gas price competition and network conditions
2. **Slippage Calculation**: Use proper AMM models for accurate price impact estimation
3. **Competitive Dynamics**: Model MEV competition and front-running scenarios
4. **Network Effects**: Simulate congestion and its impact on transaction inclusion
5. **Failure Scenarios**: Account for various transaction failure modes
6. **Performance Simulation**: Include gas costs and timing in strategy evaluation

Next Steps

1. **Integration with Backtesting Framework**: Connect transaction simulator with strategy backtesting
2. **Real-time Adaptation**: Implement adaptive gas pricing based on market conditions
3. **Multi-chain Support**: Extend to other blockchain networks
4. **Advanced MEV Models**: Add more sophisticated attack models
5. **Live Trading Integration**: Connect simulation with live trading execution

This transaction-level simulation provides the realistic execution environment needed for accurate MEV strategy evaluation and optimization.