Module 1: Backtesting Framework Design

Introduction

Backtesting is the cornerstone of MEV strategy development, providing a controlled environment to validate trading ideas, optimize parameters, and assess risk before deploying capital in production. This module will teach you to build robust, comprehensive backtesting frameworks specifically designed for MEV strategies.

Learning Objectives

By the end of this module, you will be able to:
- Design and implement modular backtesting frameworks
- Create realistic market simulation environments
- Build transaction-level backtesting with accurate gas modeling
- Implement performance metrics and statistical analysis tools
- Design extensible architectures for different MEV strategy types

1.1 Backtesting Framework Architecture

1.1.1 Core Components

A robust MEV backtesting framework consists of several interconnected components:

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Dict, List, Optional, Any
from datetime import datetime, timedelta
import pandas as pd
import numpy as np
from enum import Enum

class ExecutionType(Enum):
    MARKET_ORDER = "market_order"
    LIMIT_ORDER = "limit_order"
    INSTANT = "instant"

@dataclass
class Transaction:
    """Represents a transaction in the simulation"""
    timestamp: datetime
    tx_hash: str
    sender: str
    receiver: str
    token_in: str
    token_out: str
    amount_in: float
    amount_out: float
    gas_used: int
    gas_price: int
    execution_type: ExecutionType
    block_number: int
    success: bool

@dataclass
class MarketData:
    """Represents market data point"""
    timestamp: datetime
    price: float
    volume: float
    gas_price: int
    block_time: float
    liquidity: float

class BacktestEngine:
    """Core backtesting engine"""
```

```
    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.market_data = []
        self.transactions = []
        self.portfolio = Portfolio()
        self.metrics_collector = MetricsCollector()

    def add_market_data(self, data: MarketData):
        """Add market data point to the simulation"""
        self.market_data.append(data)

    def execute_transaction(self, tx: Transaction) -> bool:
        """Execute a transaction in the simulation"""
        # Check if transaction would succeed
        success = self._validate_transaction(tx)
        self.transactions.append(tx)

        if success:
            self.portfolio.update(tx)
            self.metrics_collector.record_transaction(tx)

        return success

    def _validate_transaction(self, tx: Transaction) -> bool:
        """Validate if transaction would succeed"""
        # Check gas price competitiveness
        latest_gas = self.market_data[-1].gas_price if self.market_data
else 20_000_000_000
        if tx.gas_price < latest_gas * 0.9:  # 10% below current
            return False

        # Check slippage tolerance
        if hasattr(tx, 'max_slippage') and tx.max_slippage:
            price_impact = self._calculate_price_impact(tx)
            if price_impact > tx.max_slippage:
                return False

        return True

    def _calculate_price_impact(self, tx: Transaction) -> float:
        """Calculate estimated price impact"""
```

```
        # Simplified price impact calculation
        order_size_ratio = tx.amount_in /
self.market_data[-1].liquidity
        return order_size_ratio * 0.5  # 50% impact coefficient

    def run_backtest(self) -> Dict[str, Any]:
        """Execute the complete backtest"""
        results = {
            'transactions': self.transactions,
            'portfolio_evolution': self.portfolio.evolution,
            'metrics': self.metrics_collector.get_results()
        }
        return results

@dataclass
class Portfolio:
    """Portfolio state and evolution tracking"""
    initial_balance: float = 1000.0
    balances: Dict[str, float] = None
    evolution: List[Dict[str, Any]] = None

    def __post_init__(self):
        if self.balances is None:
            self.balances = {'ETH': self.initial_balance, 'USDC': 0}
        if self.evolution is None:
            self.evolution = []

    def update(self, tx: Transaction):
        """Update portfolio based on transaction"""
        # Apply transaction to balances
        if tx.success:
            # Update token balances based on transaction
            self._apply_transaction_to_balances(tx)

        # Record portfolio state
        self.evolution.append({
            'timestamp': tx.timestamp,
            'balances': self.balances.copy(),
            'total_value_usd': self._calculate_total_value()
        })

    def _apply_transaction_to_balances(self, tx: Transaction):
```

```
        """Apply transaction effects to portfolio balances"""
        # This is a simplified implementation
        # In practice, you'd need full DeFi protocol integration
        if tx.token_out == 'USDC' and tx.token_in == 'ETH':
            self.balances['ETH'] -= tx.amount_in
            self.balances['USDC'] += tx.amount_out
        # Add more token pairs as needed

    def _calculate_total_value(self) -> float:
        """Calculate total portfolio value in USD"""
        # Simplified - would need price data for each token
        eth_price = 2000  # Placeholder
        return self.balances.get('ETH', 0) * eth_price +
self.balances.get('USDC', 0)

class MetricsCollector:
    """Collects and calculates performance metrics"""

    def __init__(self):
        self.transactions = []
        self.returns = []
        self.timestamps = []

    def record_transaction(self, tx: Transaction):
        """Record transaction for metrics calculation"""
        self.transactions.append(tx)

    def get_results(self) -> Dict[str, Any]:
        """Calculate and return performance metrics"""
        if not self.transactions:
            return {}

        return {
            'total_transactions': len(self.transactions),
            'successful_transactions': sum(1 for tx in
self.transactions if tx.success),
            'success_rate': sum(1 for tx in self.transactions if
tx.success) / len(self.transactions),
            'total_gas_used': sum(tx.gas_used for tx in
self.transactions if tx.success),
            'avg_gas_per_transaction': np.mean([tx.gas_used for tx in
self.transactions if tx.success]) if any(tx.success for tx in
```

```
self.transactions) else 0
        }
```

1.1.2 Strategy Base Classes

```
from abc import ABC, abstractmethod
from typing import List, Tuple

class MEVStrategy(ABC):
    """Abstract base class for MEV strategies"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.name = self.__class__.__name__
        self.is_active = True

    @abstractmethod
    def analyze_market(self, market_data: MarketData) -> List[Dict[str,
Any]]:
        """Analyze market conditions and identify opportunities"""
        pass

    @abstractmethod
    def should_execute(self, opportunity: Dict[str, Any], market_data:
MarketData) -> bool:
        """Determine if opportunity should be executed"""
        pass

    @abstractmethod
    def execute_opportunity(self, opportunity: Dict[str, Any],
market_data: MarketData) -> Transaction:
        """Execute the opportunity and return transaction"""
        pass

    def before_backtest_start(self):
        """Called once at the start of backtest"""
        pass

    def after_backtest_end(self):
        """Called once at the end of backtest"""
        pass

class ArbitrageStrategy(MEVStrategy):
    """Example arbitrage strategy implementation"""

    def analyze_market(self, market_data: MarketData) -> List[Dict[str,
Any]]:
```

```
        """Scan for arbitrage opportunities"""
        opportunities = []

        # Simulate checking multiple DEXs for price differences
        # In reality, this would query real DEX data
        dexs = ['uniswap', 'sushiswap', 'curve']
        prices = {dex: np.random.normal(2000, 10) for dex in dexs}

        # Find arbitrage opportunities
        min_price = min(prices.values())
        max_price = max(prices.values())

        if (max_price - min_price) / min_price > 0.005:  # 0.5%
threshold
            opportunities.append({
                'type': 'arbitrage',
                'buy_dex': min(prices, key=prices.get),
                'sell_dex': max(prices, key=prices.get),
                'buy_price': min_price,
                'sell_price': max_price,
                'profit_potential': max_price - min_price,
                'timestamp': market_data.timestamp
            })

        return opportunities

    def should_execute(self, opportunity: Dict[str, Any], market_data:
MarketData) -> bool:
        """Determine if arbitrage opportunity is worth executing"""
        # Check minimum profit threshold
        min_profit = self.config.get('min_profit_usd', 10)
        return opportunity['profit_potential'] > min_profit

    def execute_opportunity(self, opportunity: Dict[str, Any],
market_data: MarketData) -> Transaction:
        """Execute arbitrage opportunity"""
        return Transaction(
            timestamp=opportunity['timestamp'],

tx_hash=f"0x{''.join(np.random.choice(list('0123456789abcdef'), 64))}",
            sender="mev_bot",
            receiver=opportunity['buy_dex'],
```

```
            token_in='ETH',
            token_out='USDC',
            amount_in=1.0,
            amount_out=opportunity['buy_price'],
            gas_used=150000,
            gas_price=market_data.gas_price,
            execution_type=ExecutionType.MARKET_ORDER,
            block_number=18000000 +
len(opportunity['timestamp'].strftime('%s')),
            success=True
        )

class LiquidationStrategy(MEVStrategy):
    """Example liquidation strategy implementation"""

    def analyze_market(self, market_data: MarketData) -> List[Dict[str,
Any]]:
        """Scan for liquidation opportunities"""
        opportunities = []

        # Simulate checking lending protocols for undercollateralized
positions
        protocols = ['aave', 'compound', 'maker']

        for protocol in protocols:
            # Simulate finding liquidation opportunities
            if np.random.random() < 0.1:  # 10% chance
                opportunities.append({
                    'type': 'liquidation',
                    'protocol': protocol,
                    'borrower_address':
f"0x{''.join(np.random.choice(list('0123456789abcdef'), 40))}",
                    'collateral_token': 'ETH',
                    'debt_token': 'USDC',
                    'debt_amount': np.random.uniform(1000, 10000),
                    'liquidation_bonus': 0.08,  # 8% bonus
                    'timestamp': market_data.timestamp
                })

        return opportunities

    def should_execute(self, opportunity: Dict[str, Any], market_data:
```

```
MarketData) -> bool:
        """Determine if liquidation opportunity should be executed"""
        # Check minimum liquidation amount
        min_liquidation = self.config.get('min_liquidation_usd', 500)
        return opportunity['debt_amount'] > min_liquidation

    def execute_opportunity(self, opportunity: Dict[str, Any],
market_data: MarketData) -> Transaction:
        """Execute liquidation opportunity"""
        return Transaction(
            timestamp=opportunity['timestamp'],

tx_hash=f"0x{''.join(np.random.choice(list('0123456789abcdef'), 64))}",
            sender="mev_bot",
            receiver=opportunity['protocol'],
            token_in='USDC',
            token_out='ETH',
            amount_in=opportunity['debt_amount'],
            amount_out=opportunity['debt_amount'] * (1 +
opportunity['liquidation_bonus']) / market_data.price,
            gas_used=200000,
            gas_price=market_data.gas_price,
            execution_type=ExecutionType.MARKET_ORDER,
            block_number=18000000 +
len(opportunity['timestamp'].strftime('%s')),
            success=True
        )
```

1.2 Market Data Simulation

1.2.1 Historical Data Generator

```
import numpy as np
from scipy import stats
from typing import Iterator, List, Tuple

class MarketDataGenerator:
    """Generates realistic market data for backtesting"""

    def __init__(self, start_date: datetime, end_date: datetime,
                 initial_price: float = 2000, volatility: float =
0.02):
        self.start_date = start_date
        self.end_date = end_date
        self.initial_price = initial_price
        self.volatility = volatility

    def generate_price_series(self, num_points: int = None) ->
pd.DataFrame:
        """Generate realistic price series using geometric Brownian
motion"""
        if num_points is None:
            num_points = int((self.end_date -
self.start_date).total_seconds() / 12)  # 12-second intervals

        dt = (self.end_date - self.start_date).total_seconds() /
num_points
        dates = pd.date_range(start=self.start_date,
periods=num_points, freq='12S')

        # Generate random walk with realistic characteristics
        np.random.seed(42)  # For reproducible results
        returns = np.random.normal(0, self.volatility * np.sqrt(dt),
num_points)

        # Add mean reversion and volatility clustering
        for i in range(1, len(returns)):
            returns[i] += -0.1 * np.log(returns[i-1]) * dt  # Mean
reversion
            volatility_multiplier = 1 + 0.5 * abs(returns[i-1]) /
self.volatility
            returns[i] *= volatility_multiplier

        # Calculate prices
```

```
        prices = [self.initial_price]
        for ret in returns:
            prices.append(prices[-1] * (1 + ret))

        # Generate volume data with realistic patterns
        base_volume = 1000
        volume_multiplier = np.random.lognormal(0, 1, num_points)
        volumes = [base_volume * vol for vol in volume_multiplier]

        # Generate gas price data with realistic patterns
        base_gas = 20_000_000_000  # 20 gwei
        gas_prices = []
        for i in range(num_points):
            # Gas prices follow a more complex pattern with spikes
            if np.random.random() < 0.05:  # 5% chance of spike
                gas_price = base_gas * np.random.uniform(2, 10)
            else:
                gas_price = base_gas * np.random.uniform(0.5, 2)
            gas_prices.append(int(gas_price))

        # Block time data
        block_times = np.random.normal(12, 2, num_points)
# ~12 second blocks with some variance

        # Liquidity data
        liquidity = np.random.uniform(1e6, 1e7, num_points)

        return pd.DataFrame({
            'timestamp': dates,
            'price': prices[1:],  # Remove initial price
            'volume': volumes,
            'gas_price': gas_prices,
            'block_time': block_times,
            'liquidity': liquidity
        })

class EventMarketDataGenerator(MarketDataGenerator):
    """Generates market data with specific events for stress testing"""

    def generate_with_events(self) -> pd.DataFrame:
        """Generate price series with specific market events"""
        base_data = self.generate_price_series()
```

```
        # Add flash crash event
        crash_time = int(len(base_data) * 0.6)
        base_data.loc[crash_time:crash_time+5, 'price'] *= 0.85  # 15%
drop

        # Add volatility spike event
        vol_time = int(len(base_data) * 0.3)
        base_data.loc[vol_time:vol_time+10, 'gas_price'] *= 3
# Triple gas prices

        # Add liquidity withdrawal event
        liq_time = int(len(base_data) * 0.8)
        base_data.loc[liq_time:liq_time+20, 'liquidity'] *= 0.3  # 70%
liquidity drop

        return base_data

# Example usage
start_date = datetime(2024, 1, 1)
end_date = datetime(2024, 12, 31)

generator = EventMarketDataGenerator(
    start_date=start_date,
    end_date=end_date,
    initial_price=2000,
    volatility=0.025
)

market_data = generator.generate_with_events()
print(f"Generated {len(market_data)} data points")
print(market_data.head())
```

1.2.2 Realistic Transaction Simulation

```
from dataclasses import dataclass
from typing import Dict, List, Optional
import hashlib

@dataclass
class SwapTransaction:
    """Simulates a DEX swap transaction"""
    token_in: str
    token_out: str
    amount_in: float
    expected_min_out: float
    deadline: datetime
    recipient: str

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

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.success_rate = config.get('success_rate', 0.95)
        self.gas_limit_factor = config.get('gas_limit_factor', 1.2)

    def simulate_swap(self, swap_tx: SwapTransaction, market_data:
MarketData) -> Transaction:
        """Simulate a DEX swap transaction"""
        # Check if transaction would be successful
        success = np.random.random() < self.success_rate

        # Calculate gas usage based on complexity
        base_gas = 100000
        if swap_tx.token_in == 'ETH' and swap_tx.token_out == 'USDC':
            gas_used = base_gas * 1.2  # More complex
        else:
            gas_used = base_gas

        # Apply gas limit factor
        gas_used = int(gas_used * self.gas_limit_factor)

        # Calculate actual output based on slippage and market
conditions
        price_impact = self._calculate_price_impact(swap_tx,
market_data)
```

```
        actual_out = swap_tx.expected_min_out * (1 - price_impact)

        # Add random slippage
        slippage = np.random.uniform(-0.01, 0.01)  # ±1% slippage
        actual_out *= (1 + slippage)

        # Generate realistic tx hash
        tx_data = f"{swap_tx.token_in}{swap_tx.amount_in}
{market_data.timestamp}".encode()
        tx_hash = hashlib.sha256(tx_data).hexdigest()[:64]

        return Transaction(
            timestamp=market_data.timestamp,
            tx_hash=f"0x{tx_hash}",
            sender="mev_bot",
            receiver="uniswap_v3",
            token_in=swap_tx.token_in,
            token_out=swap_tx.token_out,
            amount_in=swap_tx.amount_in,
            amount_out=max(0, actual_out),  # Ensure non-negative
            gas_used=gas_used,
            gas_price=market_data.gas_price,
            execution_type=ExecutionType.LIMIT_ORDER,
            block_number=18000000 +
int(market_data.timestamp.timestamp()),
            success=success
        )

    def _calculate_price_impact(self, swap_tx: SwapTransaction,
market_data: MarketData) -> float:
        """Calculate realistic price impact for swap"""
        # Simplified price impact calculation
        swap_value_usd = swap_tx.amount_in * market_data.price
        liquidity_ratio = swap_value_usd / market_data.liquidity

        # Quadratic price impact curve
        impact = 0.001 * liquidity_ratio + 0.0001 * liquidity_ratio **
2
        return min(impact, 0.05)  # Cap at 5%

    def simulate_batch_execution(self, transactions: List[Transaction],
                                 market_data: MarketData) ->
```

```
List[Transaction]:
        """Simulate batch execution with dependency constraints"""
        executed = []

        for tx in transactions:
            # Check if dependencies are satisfied
            if self._check_dependencies(tx, executed):
                executed_tx = self.simulate_individual_transaction(tx,
market_data)
                executed.append(executed_tx)

        return executed

    def _check_dependencies(self, tx: Transaction, already_executed:
List[Transaction]) -> bool:
        """Check if transaction dependencies are satisfied"""
        # For MEV strategies, check if prerequisites are met
        # This is a simplified implementation
        return True

    def simulate_individual_transaction(self, tx: Transaction,
market_data: MarketData) -> Transaction:
        """Simulate individual transaction with all effects"""
        # Apply all realistic effects
        tx.success = np.random.random() < self.success_rate

        if tx.success:
            # Apply random variations to gas usage
            gas_variation = np.random.uniform(0.8, 1.2)
            tx.gas_used = int(tx.gas_used * gas_variation)

            # Apply partial fills for limit orders
            if tx.execution_type == ExecutionType.LIMIT_ORDER:
                if np.random.random() < 0.1:  # 10% chance of partial
fill
                    tx.amount_out *= np.random.uniform(0.5, 1.0)

        return tx
```

1.3 Performance Metrics and Analysis

1.3.1 Comprehensive Metrics Calculator

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

@dataclass
class PerformanceMetrics:
    """Comprehensive performance metrics"""
    total_return: float
    annualized_return: float
    volatility: float
    sharpe_ratio: float
    max_drawdown: float
    calmar_ratio: float
    sortino_ratio: float
    beta: float
    alpha: float
    var_95: float
    cvar_95: float
    win_rate: float
    profit_factor: float
    avg_trade: float
    largest_win: float
    largest_loss: float
    total_trades: int
    gas_cost_pct: float

class PerformanceAnalyzer:
    """Analyzes strategy performance with comprehensive metrics"""

    def __init__(self, risk_free_rate: float = 0.02):
        self.risk_free_rate = risk_free_rate

    def calculate_metrics(self, portfolio_evolution: List[Dict[str,
Any]],
                          transactions: List[Transaction],
                          benchmark_returns: pd.Series = None) ->
PerformanceMetrics:
        """Calculate comprehensive performance metrics"""

        # Convert portfolio evolution to DataFrame for analysis
        df = pd.DataFrame(portfolio_evolution)
```

```
        df['timestamp'] = pd.to_datetime([x['timestamp'] for x in
portfolio_evolution])
        df.set_index('timestamp', inplace=True)

        # Calculate returns
        total_value = df['total_value_usd'].values
        returns = np.diff(total_value) / total_value[:-1]

        # Basic metrics
        total_return = (total_value[-1] / total_value[0]) - 1

        # Annualized metrics
        days = (df.index[-1] - df.index[0]).days
        years = days / 365.25
        annualized_return = (1 + total_return) ** (1/years) - 1

        # Risk metrics
        volatility = np.std(returns) * np.sqrt(365) if len(returns) > 1
else 0

        # Risk-adjusted metrics
        excess_returns = returns - (self.risk_free_rate / 365)
        sharpe_ratio = np.mean(excess_returns) / np.std(returns) *
np.sqrt(365) if np.std(returns) > 0 else 0

        # Drawdown metrics
        cumulative_returns = (1 + returns).cumprod()
        rolling_max = np.maximum.accumulate(cumulative_returns)
        drawdowns = (cumulative_returns - rolling_max) / rolling_max
        max_drawdown = np.min(drawdowns)
        calmar_ratio = annualized_return / abs(max_drawdown) if
max_drawdown < 0 else 0

        # Downside deviation (Sortino ratio)
        downside_returns = returns[returns < 0]
        downside_deviation = np.std(downside_returns) * np.sqrt(365) if
len(downside_returns) > 0 else 0
        sortino_ratio = annualized_return / downside_deviation if
downside_deviation > 0 else 0

        # Value at Risk and Conditional VaR
        var_95 = np.percentile(returns, 5)
```

```
        cvar_95 = np.mean(returns[returns <= var_95])

        # Trade analysis
        profitable_trades = [tx for tx in transactions if tx.success
and tx.amount_out > tx.amount_in]
        win_rate = len(profitable_trades) / len(transactions) if
transactions else 0

        # Gas cost analysis
        total_gas_cost = sum(tx.gas_used * tx.gas_price for tx in
transactions if tx.success)
        total_profit = sum(tx.amount_out - tx.amount_in for tx in
profitable_trades) * 2000  # Convert to USD
        gas_cost_pct = total_gas_cost / total_profit if total_profit >
0 else 0

        return PerformanceMetrics(
            total_return=total_return,
            annualized_return=annualized_return,
            volatility=volatility,
            sharpe_ratio=sharpe_ratio,
            max_drawdown=max_drawdown,
            calmar_ratio=calmar_ratio,
            sortino_ratio=sortino_ratio,
            beta=0.0,  # Would need benchmark data
            alpha=0.0,  # Would need benchmark data
            var_95=var_95,
            cvar_95=cvar_95,
            win_rate=win_rate,
            profit_factor=total_profit / abs(total_profit -
total_gas_cost) if total_profit != total_gas_cost else 0,
            avg_trade=np.mean([tx.amount_out - tx.amount_in for tx in
profitable_trades]) if profitable_trades else 0,
            largest_win=max([tx.amount_out - tx.amount_in for tx in
profitable_trades]) if profitable_trades else 0,
            largest_loss=min([tx.amount_out - tx.amount_in for tx in
transactions if tx.success]) if transactions else 0,
            total_trades=len(transactions),
            gas_cost_pct=gas_cost_pct
        )

    def generate_report(self, metrics: PerformanceMetrics) -> str:
```

```
        """Generate human-readable performance report"""
        report = f"""
BACKTESTING RESULTS
==================

Returns Analysis
----------------
Total Return: {metrics.total_return:.2%}
Annualized Return: {metrics.annualized_return:.2%}
Volatility: {metrics.volatility:.2%}
Sharpe Ratio: {metrics.sharpe_ratio:.2f}
Sortino Ratio: {metrics.sortino_ratio:.2f}

Risk Analysis
-------------
Maximum Drawdown: {metrics.max_drawdown:.2%}
Calmar Ratio: {metrics.calmar_ratio:.2f}
Value at Risk (95%): {metrics.var_95:.2%}
Conditional VaR (95%): {metrics.cvar_95:.2%}

Trading Statistics
------------------
Total Trades: {metrics.total_trades}
Win Rate: {metrics.win_rate:.2%}
Average Trade: {metrics.avg_trade:.4f}
Profit Factor: {metrics.profit_factor:.2f}
Largest Win: {metrics.largest_win:.4f}
Largest Loss: {metrics.largest_loss:.4f}

Cost Analysis
-------------
Gas Cost as % of Profit: {metrics.gas_cost_pct:.2%}
"""
        return report

    def plot_performance(self, portfolio_evolution: List[Dict[str,
Any]],
                         transactions: List[Transaction]) -> str:
        """Generate performance visualization"""
        try:
            import matplotlib.pyplot as plt
            import seaborn as sns
```

```
            # Set up plotting style
            plt.style.use('seaborn-v0_8')
            sns.set_palette("husl")

            # Convert data
            df = pd.DataFrame(portfolio_evolution)
            df['timestamp'] = pd.to_datetime([x['timestamp'] for x in
portfolio_evolution])
            df.set_index('timestamp', inplace=True)

            # Create subplots
            fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2,
figsize=(15, 10))

            # Portfolio value
            ax1.plot(df.index, df['total_value_usd'], linewidth=2)
            ax1.set_title('Portfolio Value Over Time')
            ax1.set_ylabel('Portfolio Value (USD)')
            ax1.grid(True)

            # Drawdown
            total_value = df['total_value_usd'].values
            cumulative_returns = np.cumprod([1] +
list(np.diff(total_value) / total_value[:-1]))
            rolling_max = np.maximum.accumulate(cumulative_returns)
            drawdowns = (cumulative_returns - rolling_max) /
rolling_max

            ax2.fill_between(df.index, drawdowns, 0, alpha=0.3,
color='red')
            ax2.set_title('Drawdown Over Time')
            ax2.set_ylabel('Drawdown')
            ax2.grid(True)

            # Trade distribution
            profits = [tx.amount_out - tx.amount_in for tx in
transactions if tx.success]
            if profits:
                ax3.hist(profits, bins=20, alpha=0.7,
edgecolor='black')
                ax3.set_title('Trade Profit Distribution')
```

```
                ax3.set_xlabel('Profit per Trade')
                ax3.set_ylabel('Frequency')
                ax3.grid(True)

            # Cumulative returns
            cumulative_pnl = np.cumsum(profits) if profits else [0]
            ax4.plot(range(len(cumulative_pnl)), cumulative_pnl,
linewidth=2)
            ax4.set_title('Cumulative P&L')
            ax4.set_xlabel('Trade Number')
            ax4.set_ylabel('Cumulative P&L')
            ax4.grid(True)

            plt.tight_layout()
            plt.savefig('/workspace/performance_analysis.png', dpi=300,
bbox_inches='tight')
            plt.close()

            return "Performance plots saved to
performance_analysis.png"

        except ImportError:
            return "matplotlib not available for plotting"
```

1.4 Advanced Backtesting Features

1.4.1 Multi-Strategy Framework

```
from typing import Dict, List, Any
from dataclasses import dataclass
import asyncio

@dataclass
class StrategyPerformance:
    """Individual strategy performance metrics"""
    name: str
    allocation: float
    return_contribution: float
    risk_contribution: float
    sharpe_ratio: float
    correlation_with_portfolio: float

class MultiStrategyPortfolio:
    """Portfolio of multiple MEV strategies"""

    def __init__(self, strategies: List[MEVStrategy], allocations:
Dict[str, float]):
        self.strategies = {s.name: s for s in strategies}
        self.allocations = allocations
        self.portfolio_value = 1000000  # $1M starting capital
        self.strategy_values = {name: allocation * self.portfolio_value
                                for name, allocation in
allocations.items()}

    def run_backtest(self, market_data: List[MarketData]) -> Dict[str,
Any]:
        """Run backtest across all strategies"""
        strategy_results = {name: [] for name in
self.strategies.keys()}

        for data_point in market_data:
            # Distribute opportunities to strategies
            for strategy_name, strategy in self.strategies.items():
                opportunities = strategy.analyze_market(data_point)

                # Allocate capital based on strategy allocation
                available_capital = self.strategy_values[strategy_name]

                for opportunity in opportunities:
                    if strategy.should_execute(opportunity,
```

```
data_point):
                        tx = strategy.execute_opportunity(opportunity,
data_point)

                        # Scale transaction based on available capital
                        max_trade_size = available_capital * 0.1
# Max 10% per trade
                        tx.amount_in = min(tx.amount_in,
max_trade_size / data_point.price)

                        if tx.success:
                            # Update strategy-specific portfolio
                            profit = tx.amount_out - tx.amount_in
                            self.strategy_values[strategy_name] +=
profit

                        strategy_results[strategy_name].append({
                            'timestamp': data_point.timestamp,
                            'transaction': tx,
                            'strategy_value':
self.strategy_values[strategy_name]
                        })

        # Calculate portfolio-level metrics
        total_value = sum(self.strategy_values.values())
        total_return = (total_value - self.portfolio_value) /
self.portfolio_value

        return {
            'strategy_results': strategy_results,
            'portfolio_performance': {
                'total_return': total_return,
                'final_value': total_value,
                'strategy_values': self.strategy_values
            }
        }

# Example usage
arbitrage_strategy = ArbitrageStrategy({'min_profit_usd': 5})
liquidation_strategy = LiquidationStrategy({'min_liquidation_usd':
200})
```

```
portfolio = MultiStrategyPortfolio(
    strategies=[arbitrage_strategy, liquidation_strategy],
    allocations={'ArbitrageStrategy': 0.7, 'LiquidationStrategy': 0.3}
)
```

1.4.2 Parameter Optimization

```
from itertools import product
from typing import Dict, Any, List
import pandas as pd

class ParameterOptimizer:
    """Optimize strategy parameters using grid search or Bayesian
optimization"""

    def __init__(self, strategy_class, parameter_ranges: Dict[str,
List[Any]]):
        self.strategy_class = strategy_class
        self.parameter_ranges = parameter_ranges

    def grid_search(self, market_data: List[MarketData],
                    optimization_metric: str = 'sharpe_ratio') ->
Dict[str, Any]:
        """Perform grid search optimization"""

        # Generate all parameter combinations
        param_combinations =
list(product(*self.parameter_ranges.values()))
        param_names = list(self.parameter_ranges.keys())

        best_result = None
        best_params = None
        best_score = float('-inf')

        results = []

        for combination in param_combinations:
            # Create strategy with current parameters
            params = dict(zip(param_names, combination))
            strategy = self.strategy_class(params)

            # Run backtest
            engine = BacktestEngine({})
            self._run_strategy_backtest(engine, strategy, market_data)
            metrics = engine.metrics_collector.get_results()

            # Extract optimization metric
            if optimization_metric in metrics:
                score = metrics[optimization_metric]
```

```
            else:
                score = 0  # Default score

            results.append({
                'params': params,
                'score': score,
                'metrics': metrics
            })

            if score > best_score:
                best_score = score
                best_params = params
                best_result = results[-1]

        return {
            'best_params': best_params,
            'best_score': best_score,
            'best_result': best_result,
            'all_results': results
        }

    def _run_strategy_backtest(self, engine: BacktestEngine, strategy:
MEVStrategy,
                               market_data: List[MarketData]):
        """Run backtest for a single strategy"""
        engine.strategies = [strategy]

        for data_point in market_data:
            opportunities = strategy.analyze_market(data_point)

            for opportunity in opportunities:
                if strategy.should_execute(opportunity, data_point):
                    tx = strategy.execute_opportunity(opportunity,
data_point)
                    engine.execute_transaction(tx)

class WalkForwardOptimizer:
    """Walk-forward analysis for time-varying parameter optimization"""

    def __init__(self, train_period_days: int = 30, test_period_days:
int = 7):
        self.train_period_days = train_period_days
```

```
        self.test_period_days = test_period_days

    def optimize_and_validate(self, market_data: pd.DataFrame,
                              optimizer: ParameterOptimizer) -> Dict[str,
Any]:
        """Run walk-forward optimization"""

        results = []
        current_date = market_data['timestamp'].min()
        end_date = market_data['timestamp'].max()

        while current_date < end_date:
            # Define training period
            train_start = current_date
            train_end = current_date +
timedelta(days=self.train_period_days)

            # Define test period
            test_start = train_end
            test_end = train_end +
timedelta(days=self.test_period_days)

            if test_end > end_date:
                break

            # Split data
            train_data = market_data[
                (market_data['timestamp'] >= train_start) &
                (market_data['timestamp'] < train_end)
            ]
            test_data = market_data[
                (market_data['timestamp'] >= test_start) &
                (market_data['timestamp'] < test_end)
            ]

            if len(train_data) == 0 or len(test_data) == 0:
                current_date += timedelta(days=self.test_period_days)
                continue

            # Optimize on training data
            optimization_result =
optimizer.grid_search(train_data.to_dict('records'))
```

```
            # Validate on test data with optimized parameters
            strategy =
optimizer.strategy_class(optimization_result['best_params'])
            test_engine = BacktestEngine({})
            self._run_strategy_backtest(test_engine, strategy,
test_data.to_dict('records'))
            test_metrics = test_engine.metrics_collector.get_results()

            results.append({
                'train_period': {'start': train_start, 'end':
train_end},
                'test_period': {'start': test_start, 'end': test_end},
                'optimized_params': optimization_result['best_params'],
                'train_score': optimization_result['best_score'],
                'test_metrics': test_metrics
            })

            current_date += timedelta(days=self.test_period_days)

        return {'walk_forward_results': results}
```

1.5 Testing and Validation

1.5.1 Out-of-Sample Testing

```
class BacktestValidator:
    """Validates backtest results with out-of-sample testing"""

    def __init__(self, out_of_sample_ratio: float = 0.2):
        self.out_of_sample_ratio = out_of_sample_ratio

    def split_data(self, market_data: pd.DataFrame) -> tuple:
        """Split data into in-sample and out-of-sample periods"""
        split_point = int(len(market_data) * (1 -
self.out_of_sample_ratio))

        in_sample = market_data.iloc[:split_point]
        out_of_sample = market_data.iloc[split_point:]

        return in_sample, out_of_sample

    def validate_strategy(self, strategy_class, params: Dict[str, Any],
                          in_sample_data: pd.DataFrame,
                          out_of_sample_data: pd.DataFrame) -> Dict[str,
Any]:
        """Validate strategy on both in-sample and out-of-sample
data"""

        # Run on in-sample data
        in_sample_engine = BacktestEngine({})
        in_sample_strategy = strategy_class(params)
        self._run_strategy_backtest(in_sample_engine,
in_sample_strategy,
                                    in_sample_data.to_dict('records'))
        in_sample_metrics =
in_sample_engine.metrics_collector.get_results()

        # Run on out-of-sample data
        out_of_sample_engine = BacktestEngine({})
        out_of_sample_strategy = strategy_class(params)
        self._run_strategy_backtest(out_of_sample_engine,
out_sample_strategy,

out_of_sample_data.to_dict('records'))
        out_of_sample_metrics =
out_of_sample_engine.metrics_collector.get_results()
```

```
        # Compare results
        validation_result = {
            'in_sample': in_sample_metrics,
            'out_of_sample': out_of_sample_metrics,
            'performance_consistency':
self._check_consistency(in_sample_metrics, out_of_sample_metrics),
            'recommendation':
self._generate_recommendation(in_sample_metrics, out_of_sample_metrics)
        }

        return validation_result

    def _check_consistency(self, in_sample: Dict[str, Any],
                           out_of_sample: Dict[str, Any]) -> Dict[str,
Any]:
        """Check consistency between in-sample and out-of-sample
performance"""

        metrics_to_check = ['total_return', 'sharpe_ratio',
'max_drawdown']
        consistency = {}

        for metric in metrics_to_check:
            in_val = in_sample.get(metric, 0)
            out_val = out_of_sample.get(metric, 0)

            if metric == 'max_drawdown':  # For drawdown, smaller
absolute is better
                consistency[metric] = abs(out_val) <= abs(in_val) * 1.5
# Allow 50% worse
            else:
                consistency[metric] = out_val >= in_val * 0.7  # Allow
30% worse

        return consistency

    def _generate_recommendation(self, in_sample: Dict[str, Any],
                                 out_of_sample: Dict[str, Any]) -> str:
        """Generate recommendation based on validation results"""

        consistency = self._check_consistency(in_sample, out_of_sample)
```

```
        if all(consistency.values()):
            return "STRONG BUY: Strategy shows consistent performance
across time periods"
        elif sum(consistency.values()) >= len(consistency) * 0.7:
            return "BUY: Strategy shows good consistency with minor
degradation"
        elif sum(consistency.values()) >= len(consistency) * 0.5:
            return "HOLD: Strategy shows moderate consistency, monitor
closely"
        else:
            return "SELL: Strategy shows poor out-of-sample
performance"
```

1.6 Practical Exercise: Build Your First Backtesting Framework

```
def create_custom_backtesting_framework():
    """Complete example of building a custom backtesting framework"""

    # 1. Define strategy configuration
    config = {
        'initial_balance': 10000,
        'min_profit_threshold': 0.005,  # 0.5%
        'max_slippage': 0.01,  # 1%
        'gas_price_multiplier': 1.1,  # 10% above market
        'success_rate': 0.95
    }

    # 2. Initialize backtesting engine
    engine = BacktestEngine(config)

    # 3. Generate market data
    generator = MarketDataGenerator(
        start_date=datetime(2024, 1, 1),
        end_date=datetime(2024, 1, 31),
        initial_price=2000,
        volatility=0.03
    )

    market_data = generator.generate_price_series()

    # 4. Initialize strategies
    strategies = [
        ArbitrageStrategy(config),
        LiquidationStrategy(config)
    ]

    # 5. Run backtest
    print("Running backtest simulation...")

    for i, data_point in market_data.iterrows():
        market_data_point = MarketData(
            timestamp=data_point['timestamp'],
            price=data_point['price'],
            volume=data_point['volume'],
            gas_price=data_point['gas_price'],
            block_time=data_point['block_time'],
            liquidity=data_point['liquidity']
```

```
        )

        # Analyze opportunities with each strategy
        for strategy in strategies:
            opportunities = strategy.analyze_market(market_data_point)

            for opportunity in opportunities:
                if strategy.should_execute(opportunity,
market_data_point):
                    tx = strategy.execute_opportunity(opportunity,
market_data_point)
                    engine.execute_transaction(tx)

    # 6. Get results
    results = engine.run_backtest()

    # 7. Analyze performance
    analyzer = PerformanceAnalyzer()
    metrics =
analyzer.calculate_metrics(results['portfolio_evolution'],
results['transactions'])

    # 8. Generate report
    report = analyzer.generate_report(metrics)
    print(report)

    return {
        'results': results,
        'metrics': metrics,
        'report': report
    }

# Run the exercise
if __name__ == "__main__":
    framework_results = create_custom_backtesting_framework()
    print("Custom backtesting framework completed successfully!")
```

Module Summary

In this module, you have learned to build a comprehensive backtesting framework for MEV strategies. The framework includes:

- **Modular Architecture**: Extensible design supporting multiple strategy types

- **Realistic Market Simulation**: Historical data generation with realistic market characteristics
- **Transaction-Level Backtesting**: Accurate simulation of DEX interactions, gas costs, and slippage
- **Performance Analysis**: Comprehensive metrics calculation and visualization
- **Parameter Optimization**: Grid search and walk-forward optimization techniques
- **Validation Framework**: Out-of-sample testing for strategy robustness

The framework you've built provides a solid foundation for developing and testing MEV strategies before production deployment. In the next module, we'll explore data collection and processing pipelines to feed high-quality market data into your backtesting system.

Key Takeaways

1. **Separation of Concerns**: Separate strategy logic from execution and data handling
2. **Realistic Simulation**: Include gas costs, slippage, and execution failures in backtests
3. **Performance Analysis**: Use comprehensive metrics, not just returns
4. **Validation is Critical**: Always test strategies on out-of-sample data
5. **Parameter Optimization**: Use proper validation techniques to avoid overfitting
6. **Modular Design**: Build frameworks that can be extended and reused

Next Steps

1. **Extend the Framework**: Add support for more complex transaction types (flash loans, multi-hop swaps)
2. **Historical Data Integration**: Connect to real blockchain data sources
3. **Benchmark Comparisons**: Compare strategy performance against relevant benchmarks
4. **Risk Management**: Implement position sizing and risk controls in the framework
5. **Real-Time Testing**: Develop paper trading capabilities for live strategy validation

This framework provides the foundation for rigorous MEV strategy development and validation. In the following modules, we'll dive deeper into specific aspects of backtesting and simulation.