Module 5: Stress Testing & Scenario Analysis

Introduction

Stress testing and scenario analysis are critical for understanding how MEV strategies perform under extreme market conditions. This module covers Monte Carlo simulations, stress testing frameworks, extreme event modeling, and risk scenario generation to prepare strategies for unexpected market disruptions.

Learning Objectives

By the end of this module, you will be able to:
- Build Monte Carlo simulation frameworks for MEV strategy testing
- Design stress testing scenarios for extreme market conditions
- Model correlation breakdowns and systemic risk events
- Implement adaptive scenario generation based on market regimes
- Analyze strategy performance under stress conditions

5.1 Monte Carlo Simulation Framework

5.1.1 Basic Monte Carlo Engine

```
from typing import Dict, List, Any, Optional, Callable, Tuple
from dataclasses import dataclass
from enum import Enum
import numpy as np
import pandas as pd
from datetime import datetime, timedelta
import random
from concurrent.futures import ProcessPoolExecutor
import json

class SimulationScenario(Enum):
    NORMAL = "normal"
    VOLATILE = "volatile"
    CRASH = "crash"
    LIQUIDITY_CRISIS = "liquidity_crisis"
    GAS_SPIKE = "gas_spike"
    CORRELATION_BREAKDOWN = "correlation_breakdown"

@dataclass
class SimulationParameters:
    """Parameters for Monte Carlo simulation"""
    num_simulations: int
    time_horizon_days: int
    initial_capital: float
    strategy_config: Dict[str, Any]
    market_parameters: Dict[str, Any]
    correlation_matrix: Optional[np.ndarray] = None
    random_seed: Optional[int] = None

@dataclass
class SimulationResult:
    """Results from a single simulation run"""
    simulation_id: int
    final_value: float
    total_return: float
    max_drawdown: float
    sharpe_ratio: float
    win_rate: float
    total_trades: int
    gas_costs: float
    failure_count: int
    timeline: List[Dict[str, Any]]
```

```
    risk_metrics: Dict[str, float]

class MonteCarloEngine:
    """Monte Carlo simulation engine for MEV strategies"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.simulation_cache = {}
        self.market_models = {}

    def run_monte_carlo_simulation(self, params: SimulationParameters)
-> Dict[str, Any]:
        """Run comprehensive Monte Carlo simulation"""

        print(f"Starting Monte Carlo simulation with
{params.num_simulations} runs...")

        # Set random seed for reproducibility
        if params.random_seed:
            np.random.seed(params.random_seed)
            random.seed(params.random_seed)

        # Initialize market model
        market_model = self._initialize_market_model(params)

        # Run simulations
        results = []
        failed_simulations = 0

        for i in range(params.num_simulations):
            try:
                result = self._run_single_simulation(i, params,
market_model)
                results.append(result)

                if i % 100 == 0:
                    print(f"Completed {i}/{params.num_simulations}
simulations")

            except Exception as e:
                failed_simulations += 1
                print(f"Simulation {i} failed: {e}")
```

```
        print(f"Completed {params.num_simulations} simulations
({failed_simulations} failed)")

        # Aggregate results
        aggregate_results = self._aggregate_simulation_results(results)

        return {
            'individual_results': results,
            'aggregate_results': aggregate_results,
            'simulation_params': params,
            'success_rate': (params.num_simulations -
failed_simulations) / params.num_simulations
        }

    def _initialize_market_model(self, params: SimulationParameters) ->
Callable:
        """Initialize market model for simulation"""

        # Get market parameters
        market_params = params.market_parameters

        # Initialize price model
        def simulate_market_path(time_horizon_days: int, initial_price:
float) -> np.ndarray:
            """Simulate market price path using geometric Brownian
motion"""

            dt = 1/24  # Hourly steps
            num_steps = time_horizon_days * 24

            # Base parameters
            mu = market_params.get('drift', 0.0)  # Expected return
            sigma = market_params.get('volatility', 0.02)  # Volatility

            # Generate random shocks
            shocks = np.random.normal(0, 1, num_steps)

            # Simulate price path
            prices = [initial_price]
            for i in range(num_steps):
                # Add regime-switching effects
```

```
                if i % (24 * 7) == 0:  # Weekly regime changes
                    regime_factor = np.random.choice([0.5, 1.0, 2.0],
p=[0.1, 0.8, 0.1])
                    sigma_regime = sigma * regime_factor
                else:
                    sigma_regime = sigma

                # Geometric Brownian motion
                dW = shocks[i] * np.sqrt(dt)
                dS = prices[-1] * (mu * dt + sigma_regime * dW)
                new_price = prices[-1] + dS

                # Ensure price doesn't go negative
                prices.append(max(new_price, 0.01))

            return np.array(prices)

        # Initialize gas price model
        def simulate_gas_prices(time_horizon_days: int) -> np.ndarray:
            """Simulate gas price dynamics"""

            dt = 1  # Daily steps
            num_steps = time_horizon_days

            base_gas = market_params.get('base_gas_price',
20_000_000_000)

            # Gas prices with mean reversion and spikes
            gas_prices = [base_gas]

            for i in range(num_steps):
                current_gas = gas_prices[-1]

                # Mean reversion
                reversion = 0.1 * (base_gas - current_gas) * dt

                # Random walk
                random_change = np.random.normal(0, 0.2) * current_gas
* dt

                # Occasional spikes
                spike_probability = 0.05
```

```
                if np.random.random() < spike_probability:
                    spike_factor = np.random.uniform(2, 5)
                    spike = current_gas * spike_factor
                else:
                    spike = 0

                new_gas = current_gas + reversion + random_change +
spike
                gas_prices.append(max(new_gas, 1_000_000_000))  #
Minimum 1 gwei

            return np.array(gas_prices)

        # Initialize liquidity model
        def simulate_liquidity(time_horizon_days: int,
initial_liquidity: float) -> np.ndarray:
            """Simulate liquidity dynamics"""

            dt = 1  # Daily steps
            num_steps = time_horizon_days

            # Liquidity with mean reversion and volatility clustering
            liquidity_values = [initial_liquidity]

            for i in range(num_steps):
                current_liq = liquidity_values[-1]

                # Mean reversion to baseline
                baseline_liq = market_params.get('baseline_liquidity',
initial_liquidity)
                reversion = 0.05 * (baseline_liq - current_liq) * dt

                # Volatility clustering (GARCH-like)
                if i > 0:
                    volatility_factor = abs(liquidity_values[-1] -
liquidity_values[-2]) / liquidity_values[-2]
                    volatility_adjustment = np.random.normal(0,
volatility_factor * 0.1)
                else:
                    volatility_adjustment = np.random.normal(0, 0.05)

                new_liq = current_liq + reversion +
```

```
volatility_adjustment
                liquidity_values.append(max(new_liq, initial_liquidity
* 0.1))  # Minimum 10% of initial

            return np.array(liquidity_values)

        return {
            'price_path': simulate_market_path,
            'gas_prices': simulate_gas_prices,
            'liquidity_path': simulate_liquidity
        }

    def _run_single_simulation(self, sim_id: int, params:
SimulationParameters,
                               market_model: Dict) -> SimulationResult:
        """Run a single Monte Carlo simulation"""

        # Initialize simulation state
        capital = params.initial_capital
        position_size = 0
        trade_history = []
        timeline = []
        current_drawdown = 0
        peak_value = capital

        # Generate market scenarios
        time_horizon_days = params.time_horizon_days
        initial_price = params.market_parameters.get('initial_price',
2000.0)
        initial_liquidity =
params.market_parameters.get('initial_liquidity', 10000.0)

        price_path = market_model['price_path'](time_horizon_days,
initial_price)
        gas_prices = market_model['gas_prices'](time_horizon_days)
        liquidity_path = market_model['liquidity_path']
(time_horizon_days, initial_liquidity)

        # Simulate strategy execution
        for day in range(time_horizon_days):
            current_price = price_path[day * 24]  # Daily price
snapshot
```

```
            current_gas = gas_prices[day]
            current_liquidity = liquidity_path[day]

            # Generate opportunities based on market conditions
            opportunities = self._generate_opportunities(
                current_price, current_gas, current_liquidity, day,
params.strategy_config
            )

            daily_pnl = 0
            daily_trades = 0
            daily_gas_cost = 0

            for opportunity in opportunities:
                # Execute opportunity with market impact
                trade_result = self._execute_opportunity(
                    opportunity, current_price, current_liquidity,
current_gas
                )

                if trade_result['success']:
                    daily_pnl += trade_result['pnl']
                    daily_trades += 1
                    daily_gas_cost += trade_result['gas_cost']

                    trade_history.append({
                        'day': day,
                        'opportunity_type': opportunity['type'],
                        'size': trade_result['size'],
                        'pnl': trade_result['pnl'],
                        'gas_cost': trade_result['gas_cost'],
                        'success': True
                    })
                else:
                    # Failed trade (gas cost still incurred)
                    daily_gas_cost += trade_result['gas_cost']

                    trade_history.append({
                        'day': day,
                        'opportunity_type': opportunity['type'],
                        'size': trade_result['size'],
                        'pnl': -trade_result['gas_cost'],
```

```
                        'gas_cost': trade_result['gas_cost'],
                        'success': False
                    })

            # Update capital
            capital += daily_pnl
            peak_value = max(peak_value, capital)
            current_drawdown = (peak_value - capital) / peak_value

            # Record daily state
            timeline.append({
                'day': day,
                'capital': capital,
                'daily_pnl': daily_pnl,
                'daily_trades': daily_trades,
                'daily_gas_cost': daily_gas_cost,
                'current_drawdown': current_drawdown,
                'price': current_price,
                'gas_price': current_gas,
                'liquidity': current_liquidity
            })

        # Calculate final metrics
        total_return = (capital - params.initial_capital) /
params.initial_capital
        returns = [day['daily_pnl'] / params.initial_capital for day in
timeline]

        # Risk metrics
        if len(returns) > 1:
            sharpe_ratio = np.mean(returns) / np.std(returns) *
np.sqrt(365) if np.std(returns) > 0 else 0
        else:
            sharpe_ratio = 0

        max_drawdown = max(day['current_drawdown'] for day in timeline)

        # Trading statistics
        successful_trades = [t for t in trade_history if t['success']]
        win_rate = len(successful_trades) / len(trade_history) if
trade_history else 0
```

```
        return SimulationResult(
            simulation_id=sim_id,
            final_value=capital,
            total_return=total_return,
            max_drawdown=max_drawdown,
            sharpe_ratio=sharpe_ratio,
            win_rate=win_rate,
            total_trades=len(trade_history),
            gas_costs=sum(t['gas_cost'] for t in trade_history),
            failure_count=sum(1 for t in trade_history if not
t['success']),
            timeline=timeline,
            risk_metrics={
                'total_return': total_return,
                'volatility': np.std(returns) * np.sqrt(365),
                'calmar_ratio': total_return / max_drawdown if
max_drawdown > 0 else 0,
                'var_95': np.percentile(returns, 5),
                'cvar_95': np.mean([r for r in returns if r <=
np.percentile(returns, 5)])
            }
        )

    def _generate_opportunities(self, price: float, gas_price: int,
liquidity: float,
                                day: int, strategy_config: Dict[str,
Any]) -> List[Dict[str, Any]]:
        """Generate MEV opportunities based on market conditions"""

        opportunities = []

        # Base opportunity generation rate
        base_opportunities_per_day =
strategy_config.get('base_opportunities_per_day', 10)

        # Adjust for market conditions
        volatility_factor = min(3.0, 1.0 + (abs(price - 2000) / 2000) *
2)
        liquidity_factor = min(2.0, 1.0 + (10000 / liquidity))

        adjusted_rate = base_opportunities_per_day * volatility_factor
* liquidity_factor
```

```
        num_opportunities = np.random.poisson(adjusted_rate)

        for _ in range(num_opportunities):
            # Choose opportunity type based on market conditions
            if gas_price > 50_000_000_000:  # High gas prices
                # Fewer opportunities, but larger
                opportunity_type = np.random.choice(['arbitrage',
'liquidation'], p=[0.6, 0.4])
                size_factor = np.random.uniform(1.5, 3.0)
            else:
                opportunity_type = np.random.choice(['arbitrage',
'liquidation', 'sandwich'], p=[0.4, 0.4, 0.2])
                size_factor = np.random.uniform(0.5, 2.0)

            # Base size
            base_size = strategy_config.get('base_opportunity_size',
1000)
            opportunity_size = base_size * size_factor

            # Success probability based on competition and gas prices
            success_probability = 0.8
            if gas_price > 100_000_000_000:  # Very high gas
                success_probability *= 0.5
            if opportunity_size > liquidity * 0.1:  # Very large
relative to liquidity
                success_probability *= 0.7

            opportunity = {
                'type': opportunity_type,
                'size': opportunity_size,
                'success_probability': success_probability,
                'expected_profit':
self._calculate_expected_profit(opportunity_type, opportunity_size),
                'gas_requirement':
self._estimate_gas_requirement(opportunity_type)
            }

            opportunities.append(opportunity)

        return opportunities

    def _calculate_expected_profit(self, opportunity_type: str, size:
```

```
float) -> float:
        """Calculate expected profit for opportunity type"""

        base_profits = {
            'arbitrage': 0.005,  # 0.5% of size
            'liquidation': 0.08,  # 8% of size (liquidation bonus)
            'sandwich': 0.002   # 0.2% of size
        }

        base_profit_rate = base_profits.get(opportunity_type, 0.001)

        # Size scaling (diminishing returns for larger sizes)
        size_factor = min(1.0, (size / 1000) ** 0.5)

        return size * base_profit_rate * size_factor

    def _estimate_gas_requirement(self, opportunity_type: str) -> int:
        """Estimate gas requirement for opportunity type"""

        gas_estimates = {
            'arbitrage': 150000,
            'liquidation': 250000,
            'sandwich': 200000
        }

        return gas_estimates.get(opportunity_type, 150000)

    def _execute_opportunity(self, opportunity: Dict[str, Any], price:
float,
                             liquidity: float, gas_price: int) ->
Dict[str, Any]:
        """Execute a single opportunity"""

        # Check if opportunity is viable
        if np.random.random() > opportunity['success_probability']:
            return {
                'success': False,
                'size': opportunity['size'],
                'pnl': -opportunity['gas_requirement'] * gas_price /
1e18 * price / 2000,
                'gas_cost': opportunity['gas_requirement'] *
gas_price / 1e18 * price / 2000
```

```
            }

        # Calculate market impact
        size_ratio = opportunity['size'] / liquidity
        impact_rate = 0.001 * (size_ratio ** 1.2)  # Quadratic impact

        # Calculate profit
        gross_profit = opportunity['expected_profit']
        impact_cost = opportunity['size'] * impact_rate
        gas_cost = opportunity['gas_requirement'] * gas_price / 1e18 *
price / 2000

        net_profit = gross_profit - impact_cost - gas_cost

        return {
            'success': True,
            'size': opportunity['size'],
            'pnl': net_profit,
            'gas_cost': gas_cost
        }

    def _aggregate_simulation_results(self, results:
List[SimulationResult]) -> Dict[str, Any]:
        """Aggregate results across all simulations"""

        if not results:
            return {}

        # Extract metrics
        final_values = [r.final_value for r in results]
        total_returns = [r.total_return for r in results]
        max_drawdowns = [r.max_drawdown for r in results]
        sharpe_ratios = [r.sharpe_ratio for r in results]
        win_rates = [r.win_rate for r in results]
        total_trades = [r.total_trades for r in results]
        gas_costs = [r.gas_costs for r in results]
        failure_counts = [r.failure_count for r in results]

        # Aggregate statistics
        aggregate = {
            'mean_final_value': np.mean(final_values),
            'median_final_value': np.median(final_values),
```

```
            'std_final_value': np.std(final_values),
            'percentile_5': np.percentile(final_values, 5),
            'percentile_95': np.percentile(final_values, 95),
            'probability_of_loss': sum(1 for r in final_values if r <
1000000) / len(final_values),
            'expected_return': np.mean(total_returns),
            'return_volatility': np.std(total_returns),
            'worst_case_drawdown': np.max(max_drawdowns),
            'average_sharpe': np.mean(sharpe_ratios),
            'average_win_rate': np.mean(win_rates),
            'total_trades_mean': np.mean(total_trades),
            'total_trades_std': np.std(total_trades),
            'gas_costs_mean': np.mean(gas_costs),
            'failure_rate': np.mean([f/t if t > 0 else 0 for f, t in
zip(failure_counts, total_trades)])
        }

        # Value at Risk and Expected Shortfall
        sorted_returns = sorted(total_returns)
        var_95 = np.percentile(sorted_returns, 5)
        var_99 = np.percentile(sorted_returns, 1)

        tail_returns = [r for r in sorted_returns if r <= var_95]
        expected_shortfall = np.mean(tail_returns) if tail_returns else
var_95

        aggregate.update({
            'var_95': var_95,
            'var_99': var_99,
            'expected_shortfall': expected_shortfall
        })

        return aggregate

# Example usage
mc_engine = MonteCarloEngine({})

# Define simulation parameters
sim_params = SimulationParameters(
    num_simulations=1000,
    time_horizon_days=30,
    initial_capital=1_000_000,  # $1M
```

```
    strategy_config={
        'base_opportunities_per_day': 15,
        'base_opportunity_size': 2000,
        'max_position_size': 0.1
    },
    market_parameters={
        'initial_price': 2000.0,
        'initial_liquidity': 15000.0,
        'drift': 0.0,
        'volatility': 0.03,
        'base_gas_price': 25_000_000_000,
        'baseline_liquidity': 15000.0
    },
    random_seed=42
)

# Run Monte Carlo simulation
mc_results = mc_engine.run_monte_carlo_simulation(sim_params)

print(f"Simulation completed with {mc_results['success_rate']:.1%}
success rate")
print(f"Expected return: {mc_results['aggregate_results']
['expected_return']:.2%}")
print(f"95% VaR: {mc_results['aggregate_results']['var_95']:.2%}")
print(f"Expected Shortfall: {mc_results['aggregate_results']
['expected_shortfall']:.2%}")
print(f"Probability of loss: {mc_results['aggregate_results']
['probability_of_loss']:.2%}")
```

5.1.2 Parallel Monte Carlo Execution

```
import multiprocessing as mp
from concurrent.futures import ProcessPoolExecutor, as_completed
import pickle
import os

class ParallelMonteCarloEngine:
    """Parallel Monte Carlo simulation engine"""

    def __init__(self, config: Dict[str, Any], num_processes: int =
None):
        self.config = config
        self.num_processes = num_processes or mp.cpu_count()

    def run_parallel_simulation(self, params: SimulationParameters) ->
Dict[str, Any]:
        """Run Monte Carlo simulation in parallel"""

        print(f"Starting parallel Monte Carlo simulation with
{self.num_processes} processes...")

        # Split simulation into chunks
        chunk_size = params.num_simulations // self.num_processes
        simulation_chunks = []

        for i in range(self.num_processes):
            start_idx = i * chunk_size
            end_idx = start_idx + chunk_size if i < self.num_processes
- 1 else params.num_simulations

            chunk_params = SimulationParameters(
                num_simulations=end_idx - start_idx,
                time_horizon_days=params.time_horizon_days,
                initial_capital=params.initial_capital,
                strategy_config=params.strategy_config,
                market_parameters=params.market_parameters,
                correlation_matrix=params.correlation_matrix,
                random_seed=params.random_seed + i if
params.random_seed else None
            )

            simulation_chunks.append(chunk_params)
```

```
        # Run simulations in parallel
        all_results = []

        with ProcessPoolExecutor(max_workers=self.num_processes) as
executor:
            # Submit all simulation tasks
            future_to_chunk = {
                executor.submit(self._run_simulation_chunk,
chunk_params, i): i
                for i, chunk_params in enumerate(simulation_chunks)
            }

            # Collect results
            for future in as_completed(future_to_chunk):
                chunk_id = future_to_chunk[future]
                try:
                    chunk_results = future.result()
                    all_results.extend(chunk_results)
                    print(f"Completed chunk {chunk_id + 1}/
{len(simulation_chunks)}")
                except Exception as e:
                    print(f"Chunk {chunk_id} failed: {e}")

        # Aggregate results
        mc_engine = MonteCarloEngine(self.config)
        aggregate_results =
mc_engine._aggregate_simulation_results(all_results)

        return {
            'individual_results': all_results,
            'aggregate_results': aggregate_results,
            'simulation_params': params,
            'num_processes_used': self.num_processes,
            'success_rate': len(all_results) / params.num_simulations
        }

    def _run_simulation_chunk(self, chunk_params: SimulationParameters,
                              chunk_id: int) -> List[SimulationResult]:
        """Run a chunk of simulations in a separate process"""

        # Reinitialize the engine for this process
        mc_engine = MonteCarloEngine(self.config)
```

```
        market_model = mc_engine._initialize_market_model(chunk_params)

        results = []

        for i in range(chunk_params.num_simulations):
            sim_id = chunk_id * (chunk_params.num_simulations //
len([chunk_params])) + i
            result = mc_engine._run_single_simulation(sim_id,
chunk_params, market_model)
            results.append(result)

        return results

# Performance comparison example
def compare_simulation_performance():
    """Compare performance of sequential vs parallel Monte Carlo"""

    # Define test parameters
    test_params = SimulationParameters(
        num_simulations=500,
        time_horizon_days=30,
        initial_capital=1_000_000,
        strategy_config={'base_opportunities_per_day': 10},
        market_parameters={
            'initial_price': 2000.0,
            'initial_liquidity': 10000.0,
            'volatility': 0.03
        },
        random_seed=42
    )

    # Sequential execution
    print("Running sequential Monte Carlo...")
    import time
    start_time = time.time()

    mc_engine = MonteCarloEngine({})
    sequential_results =
mc_engine.run_monte_carlo_simulation(test_params)

    sequential_time = time.time() - start_time
    print(f"Sequential execution time: {sequential_time:.2f} seconds")
```

```
    # Parallel execution
    print("\nRunning parallel Monte Carlo...")
    start_time = time.time()

    parallel_engine = ParallelMonteCarloEngine({}, num_processes=4)
    parallel_results =
parallel_engine.run_parallel_simulation(test_params)

    parallel_time = time.time() - start_time
    print(f"Parallel execution time: {parallel_time:.2f} seconds")

    print(f"\nSpeedup: {sequential_time / parallel_time:.2f}x")
    print(f"Sequential results:
{len(sequential_results['individual_results'])}")
    print(f"Parallel results:
{len(parallel_results['individual_results'])}")

    return {
        'sequential_time': sequential_time,
        'parallel_time': parallel_time,
        'speedup': sequential_time / parallel_time
    }

# Run performance comparison
perf_results = compare_simulation_performance()
```

5.2 Stress Testing Scenarios

5.2.1 Market Crash Scenarios

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

class StressScenario(Enum):
    BLACK_MONDAY = "black_monday"
    FLASH_CRASH = "flash_crash"
    LIQUIDITY_VANISHING = "liquidity_vanishing"
    GAS_PRICE_SPIKE = "gas_price_spike"
    CORRELATION_BREAKDOWN = "correlation_breakdown"
    REGULATORY_SHOCK = "regulatory_shock"
    CROSS_CHAIN_CRISIS = "cross_chain_crisis"

@dataclass
class StressTestConfig:
    """Configuration for stress testing"""
    scenario: StressScenario
    severity: float  # 0.0 to 1.0
    duration_days: int
    start_day: int
    strategy_params: Dict[str, Any]
    market_params: Dict[str, Any]

class StressTestingFramework:
    """Framework for running stress tests on MEV strategies"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.scenario_models = {}
        self._initialize_scenario_models()

    def _initialize_scenario_models(self):
        """Initialize stress scenario models"""

        # Black Monday scenario - simultaneous market crash
        self.scenario_models[StressScenario.BLACK_MONDAY] =
self._black_monday_model

        # Flash crash scenario - rapid price drop and recovery
        self.scenario_models[StressScenario.FLASH_CRASH] =
```

```
self._flash_crash_model

        # Liquidity vanishing scenario - sudden liquidity withdrawal
        self.scenario_models[StressScenario.LIQUIDITY_VANISHING] =
self._liquidity_vanishing_model

        # Gas price spike scenario - network congestion
        self.scenario_models[StressScenario.GAS_PRICE_SPIKE] =
self._gas_price_spike_model

        # Correlation breakdown scenario - normally correlated assets
move independently
        self.scenario_models[StressScenario.CORRELATION_BREAKDOWN] =
self._correlation_breakdown_model

        # Regulatory shock scenario - sudden regulatory intervention
        self.scenario_models[StressScenario.REGULATORY_SHOCK] =
self._regulatory_shock_model

        # Cross-chain crisis scenario - bridge and cross-chain
disruptions
        self.scenario_models[StressScenario.CROSS_CHAIN_CRISIS] =
self._cross_chain_crisis_model

    def run_stress_test(self, config: StressTestConfig,
                        baseline_performance: Optional[Dict[str, Any]] =
None) -> Dict[str, Any]:
        """Run a single stress test scenario"""

        print(f"Running stress test:
{config.scenario.value} (severity: {config.severity:.2f})")

        # Initialize Monte Carlo engine
        mc_engine = MonteCarloEngine(self.config)

        # Modify market parameters for stress scenario
        stress_market_params = self._apply_stress_scenario(
            config.market_params, config
        )

        # Create stress test parameters
        stress_params = SimulationParameters(
```

```
            num_simulations=200,  # Reduced for faster stress testing
            time_horizon_days=config.duration_days,
            initial_capital=1_000_000,
            strategy_config=config.strategy_params,
            market_parameters=stress_market_params,
            random_seed=42
        )

        # Run stress test
        stress_results =
mc_engine.run_monte_carlo_simulation(stress_params)

        # Calculate stress test metrics
        stress_metrics = self._calculate_stress_metrics(stress_results,
baseline_performance)

        return {
            'scenario': config.scenario.value,
            'severity': config.severity,
            'duration_days': config.duration_days,
            'stress_results': stress_results,
            'stress_metrics': stress_metrics,
            'impact_assessment':
self._assess_stress_impact(stress_metrics)
        }

    def run_multiple_stress_tests(self, scenarios:
List[StressTestConfig],
                                  baseline_performance:
Optional[Dict[str, Any]] = None) -> Dict[str, Any]:
        """Run multiple stress test scenarios"""

        results = {}

        for scenario_config in scenarios:
            try:
                result = self.run_stress_test(scenario_config,
baseline_performance)
                results[scenario_config.scenario.value] = result

            except Exception as e:
                print(f"Stress test failed for
```

```
{scenario_config.scenario.value}: {e}")
                results[scenario_config.scenario.value] = {
                    'error': str(e),
                    'scenario': scenario_config.scenario.value
                }

        # Aggregate stress test results
        aggregate_results = self._aggregate_stress_results(results)

        return {
            'individual_results': results,
            'aggregate_results': aggregate_results,
            'baseline_performance': baseline_performance
        }

    def _apply_stress_scenario(self, base_params: Dict[str, Any],
                               config: StressTestConfig) -> Dict[str,
Any]:
        """Apply stress scenario to market parameters"""

        scenario_model = self.scenario_models[config.scenario]
        stressed_params = scenario_model(base_params, config)

        return stressed_params

    def _black_monday_model(self, base_params: Dict[str, Any],
                            config: StressTestConfig) -> Dict[str, Any]:
        """Black Monday scenario - severe market crash"""

        stressed_params = base_params.copy()

        # Increase volatility significantly
        base_volatility = stressed_params.get('volatility', 0.03)
        stressed_params['volatility'] = base_volatility * (1 +
config.severity * 5)

        # Add negative drift
        stressed_params['drift'] = -0.2 * config.severity  # 20% daily
decline max

        # Reduce liquidity significantly
        base_liquidity = stressed_params.get('initial_liquidity',
```

```
10000)
        stressed_params['initial_liquidity'] = base_liquidity * (1 -
config.severity * 0.8)

        # Spike gas prices
        base_gas = stressed_params.get('base_gas_price',
25_000_000_000)
        stressed_params['base_gas_price'] = base_gas * (1 +
config.severity * 10)

        # Add crash event
        crash_day = config.start_day + int(config.duration_days * 0.1)
# Crash early in period
        stressed_params['crash_event'] = {
            'day': crash_day,
            'magnitude': -0.3 * config.severity,  # 30% max crash
            'duration': int(config.duration_days * 0.05)
        }

        return stressed_params

    def _flash_crash_model(self, base_params: Dict[str, Any],
                           config: StressTestConfig) -> Dict[str, Any]:
        """Flash crash scenario - rapid drop and recovery"""

        stressed_params = base_params.copy()

        # Extreme volatility
        base_volatility = stressed_params.get('volatility', 0.03)
        stressed_params['volatility'] = base_volatility * (1 +
config.severity * 15)

        # Flash crash event
        crash_day = config.start_day + int(config.duration_days * 0.2)
        stressed_params['flash_crash'] = {
            'day': crash_day,
            'magnitude': -0.5 * config.severity,  # 50% max flash crash
            'recovery_hours': int(config.severity * 4) + 1  # Recovery
in hours
        }

        # Temporary liquidity reduction
```

```
        base_liquidity = stressed_params.get('initial_liquidity',
10000)
        stressed_params['initial_liquidity'] = base_liquidity * (1 -
config.severity * 0.9)

        return stressed_params

    def _liquidity_vanishing_model(self, base_params: Dict[str, Any],
                                   config: StressTestConfig) -> Dict[str,
Any]:
        """Liquidity vanishing scenario"""

        stressed_params = base_params.copy()

        # Severe liquidity reduction
        base_liquidity = stressed_params.get('initial_liquidity',
10000)
        liquidity_reduction = config.severity * 0.95  # Up to 95%
reduction
        stressed_params['initial_liquidity'] = base_liquidity * (1 -
liquidity_reduction)

        # Volatile liquidity
        stressed_params['liquidity_volatility'] = config.severity * 0.5

        # Gas price spike due to increased competition
        base_gas = stressed_params.get('base_gas_price',
25_000_000_000)
        stressed_params['base_gas_price'] = base_gas * (1 +
config.severity * 20)

        return stressed_params

    def _gas_price_spike_model(self, base_params: Dict[str, Any],
                               config: StressTestConfig) -> Dict[str,
Any]:
        """Gas price spike scenario"""

        stressed_params = base_params.copy()

        # Extreme gas price spike
        base_gas = stressed_params.get('base_gas_price',
```

```
25_000_000_000)
        spike_factor = 1 + config.severity * 50  # Up to 51x increase
        stressed_params['base_gas_price'] = int(base_gas *
spike_factor)

        # Network congestion effects
        stressed_params['congestion_level'] = config.severity

        # Reduced opportunity generation
        opportunity_multiplier = 1 - config.severity * 0.8
        stressed_params['opportunity_multiplier'] =
opportunity_multiplier

        return stressed_params

    def _correlation_breakdown_model(self, base_params: Dict[str, Any],
                                     config: StressTestConfig) ->
Dict[str, Any]:
        """Correlation breakdown scenario"""

        stressed_params = base_params.copy()

        # Random walk becomes more random
        stressed_params['mean_reversion_strength'] = config.severity *
0.9

        # Cross-asset correlations break down
        stressed_params['correlation_strength'] = 1 - config.severity

        # Increased idiosyncratic risk
        base_volatility = stressed_params.get('volatility', 0.03)
        stressed_params['volatility'] = base_volatility * (1 +
config.severity)

        return stressed_params

    def _regulatory_shock_model(self, base_params: Dict[str, Any],
                                config: StressTestConfig) -> Dict[str,
Any]:
        """Regulatory shock scenario"""

        stressed_params = base_params.copy()
```

```
        # Reduced opportunity generation due to regulation
        stressed_params['opportunity_multiplier'] = 1 - config.severity
* 0.7

        # Increased compliance costs
        stressed_params['compliance_overhead'] = config.severity * 0.3

        # Market uncertainty increases volatility
        base_volatility = stressed_params.get('volatility', 0.03)
        stressed_params['volatility'] = base_volatility * (1 +
config.severity * 2)

        return stressed_params

    def _cross_chain_crisis_model(self, base_params: Dict[str, Any],
                                  config: StressTestConfig) -> Dict[str,
Any]:
        """Cross-chain crisis scenario"""

        stressed_params = base_params.copy()

        # Bridge reliability issues
        stressed_params['bridge_failure_probability'] = config.severity
* 0.3

        # Cross-chain arbitrage opportunities reduced
        stressed_params['cross_chain_opportunity_multiplier'] = 1 -
config.severity * 0.6

        # Increased execution times
        stressed_params['execution_delay_multiplier'] = 1 +
config.severity * 3

        return stressed_params

    def _calculate_stress_metrics(self, stress_results: Dict[str, Any],
                                  baseline_performance:
Optional[Dict[str, Any]]) -> Dict[str, Any]:
        """Calculate stress test metrics"""

        stress_agg = stress_results['aggregate_results']
```

```
        if baseline_performance:
            # Compare against baseline
            baseline_return =
baseline_performance.get('expected_return', 0)
            baseline_var = baseline_performance.get('var_95', 0)
            baseline_sharpe =
baseline_performance.get('average_sharpe', 0)

            return_stress = stress_agg['expected_return']
            var_stress = stress_agg['var_95']
            sharpe_stress = stress_agg['average_sharpe']

            metrics = {
                'return_impact': return_stress - baseline_return,
                'var_increase': var_stress - baseline_var,
                'sharpe_degradation': sharpe_stress - baseline_sharpe,
                'prob_loss_increase': stress_agg['probability_of_loss']
- baseline_performance.get('probability_of_loss', 0),
                'stress_sharpe_ratio': return_stress /
abs(stress_agg['worst_case_drawdown']) if
stress_agg['worst_case_drawdown'] > 0 else 0,
                'stress_var_95': var_stress,
                'stress_expected_return': return_stress,
                'stress_max_drawdown':
stress_agg['worst_case_drawdown']
            }
        else:
            # Absolute stress metrics
            metrics = {
                'stress_expected_return':
stress_agg['expected_return'],
                'stress_var_95': stress_agg['var_95'],
                'stress_max_drawdown':
stress_agg['worst_case_drawdown'],
                'stress_sharpe_ratio': stress_agg['expected_return'] /
stress_agg['return_volatility'] if stress_agg['return_volatility'] > 0
else 0,
                'stress_probability_of_loss':
stress_agg['probability_of_loss'],
                'stress_gas_costs': stress_agg['gas_costs_mean']
            }
```

```
        return metrics

    def _assess_stress_impact(self, stress_metrics: Dict[str, Any]) ->
str:
        """Assess the overall impact of stress scenario"""

        # Simple scoring system
        impact_score = 0

        # Return impact
        return_impact = stress_metrics.get('return_impact', 0)
        if return_impact < -0.2:
            impact_score += 3
        elif return_impact < -0.1:
            impact_score += 2
        elif return_impact < 0:
            impact_score += 1

        # VaR increase
        var_increase = stress_metrics.get('var_increase', 0)
        if var_increase > 0.1:
            impact_score += 3
        elif var_increase > 0.05:
            impact_score += 2
        elif var_increase > 0:
            impact_score += 1

        # Probability of loss increase
        prob_loss_increase = stress_metrics.get('prob_loss_increase',
0)
        if prob_loss_increase > 0.2:
            impact_score += 3
        elif prob_loss_increase > 0.1:
            impact_score += 2
        elif prob_loss_increase > 0:
            impact_score += 1

        # Determine overall assessment
        if impact_score >= 7:
            return "SEVERE - Strategy shows significant vulnerability"
        elif impact_score >= 5:
```

```
            return "HIGH - Strategy shows material degradation"
        elif impact_score >= 3:
            return "MODERATE - Strategy shows some vulnerability"
        elif impact_score >= 1:
            return "LOW - Strategy shows minimal impact"
        else:
            return "MINIMAL - Strategy is resilient"

    def _aggregate_stress_results(self, results: Dict[str, Any]) ->
Dict[str, Any]:
        """Aggregate results across multiple stress scenarios"""

        if not results:
            return {}

        successful_scenarios = {k: v for k, v in results.items() if
'error' not in v}

        if not successful_scenarios:
            return {'error': 'All stress scenarios failed'}

        # Aggregate metrics across scenarios
        all_metrics = [result['stress_metrics'] for result in
successful_scenarios.values()]

        # Calculate average impact
        avg_return_impact = np.mean([m.get('return_impact', 0) for m in
all_metrics])
        avg_var_increase = np.mean([m.get('var_increase', 0) for m in
all_metrics])
        avg_sharpe_degradation = np.mean([m.get('sharpe_degradation',
0) for m in all_metrics])

        # Find worst-case scenarios
        worst_return_impact = min([m.get('return_impact', 0) for m in
all_metrics])
        worst_var_increase = max([m.get('var_increase', 0) for m in
all_metrics])

        # Count scenarios by impact level
        impact_levels = {
            'SEVERE': 0,
```

```
            'HIGH': 0,
            'MODERATE': 0,
            'LOW': 0,
            'MINIMAL': 0
        }

        for result in successful_scenarios.values():
            impact_level = result['impact_assessment'].split(' - ')[0]
            if impact_level in impact_levels:
                impact_levels[impact_level] += 1

        return {
            'average_return_impact': avg_return_impact,
            'average_var_increase': avg_var_increase,
            'average_sharpe_degradation': avg_sharpe_degradation,
            'worst_case_return_impact': worst_return_impact,
            'worst_case_var_increase': worst_var_increase,
            'scenario_impact_distribution': impact_levels,
            'total_scenarios_tested': len(successful_scenarios),
            'most_vulnerable_scenario': max(
                successful_scenarios.items(),
                key=lambda x: abs(x[1]
['stress_metrics'].get('return_impact', 0))
            )[0] if successful_scenarios else None
        }

# Example usage
stress_framework = StressTestingFramework({})

# Define baseline performance (from normal Monte Carlo)
baseline_performance = {
    'expected_return': 0.15,
    'var_95': -0.05,
    'average_sharpe': 1.2,
    'probability_of_loss': 0.1
}

# Define stress test scenarios
stress_scenarios = [
    StressTestConfig(
        scenario=StressScenario.BLACK_MONDAY,
        severity=0.5,
```

```
        duration_days=30,
        start_day=5,
        strategy_params={'base_opportunities_per_day': 10},
        market_params={
            'initial_price': 2000.0,
            'initial_liquidity': 15000.0,
            'volatility': 0.03
        }
    ),
    StressTestConfig(
        scenario=StressScenario.FLASH_CRASH,
        severity=0.7,
        duration_days=7,
        start_day=2,
        strategy_params={'base_opportunities_per_day': 10},
        market_params={
            'initial_price': 2000.0,
            'initial_liquidity': 15000.0,
            'volatility': 0.03
        }
    ),
    StressTestConfig(
        scenario=StressScenario.GAS_PRICE_SPIKE,
        severity=0.8,
        duration_days=14,
        start_day=0,
        strategy_params={'base_opportunities_per_day': 10},
        market_params={
            'initial_price': 2000.0,
            'initial_liquidity': 15000.0,
            'volatility': 0.03
        }
    )
]

# Run stress tests
stress_results =
stress_framework.run_multiple_stress_tests(stress_scenarios,
baseline_performance)

print("Stress Testing Summary:")
print(f"Scenarios tested: {stress_results['aggregate_results']
```

```
['total_scenarios_tested']}")
print(f"Average return impact: {stress_results['aggregate_results']
['average_return_impact']:.2%}")
print(f"Worst case return impact: {stress_results['aggregate_results']
['worst_case_return_impact']:.2%}")
print(f"Most vulnerable scenario: {stress_results['aggregate_results']
['most_vulnerable_scenario']}")
```

5.2.2 Scenario Generation Engine

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

class MarketRegime(Enum):
    BULL = "bull"
    BEAR = "bear"
    SIDEWAYS = "sideways"
    CRISIS = "crisis"
    RECOVERY = "recovery"

@dataclass
class ScenarioParameters:
    """Parameters for scenario generation"""
    regime: MarketRegime
    severity: float  # 0-1
    duration: int
    correlations: Dict[str, float]
    volatilities: Dict[str, float]
    jumps: List[Dict[str, Any]]
    structural_changes: List[Dict[str, Any]]

class AdaptiveScenarioGenerator:
    """Generates adaptive stress testing scenarios"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.regime_models = {}
        self.correlation_matrices = {}
        self._initialize_regime_models()

    def _initialize_regime_models(self):
        """Initialize models for different market regimes"""

        # Bull market regime
        self.regime_models[MarketRegime.BULL] = {
            'drift': 0.0005,  # Daily positive drift
            'volatility_multiplier': 0.8,
            'liquidity_multiplier': 1.2,
```

```
            'gas_price_multiplier': 0.9,
            'opportunity_multiplier': 1.1
        }

        # Bear market regime
        self.regime_models[MarketRegime.BEAR] = {
            'drift': -0.0003,  # Daily negative drift
            'volatility_multiplier': 1.5,
            'liquidity_multiplier': 0.7,
            'gas_price_multiplier': 1.3,
            'opportunity_multiplier': 0.8
        }

        # Crisis regime
        self.regime_models[MarketRegime.CRISIS] = {
            'drift': -0.002,  # Strong negative drift
            'volatility_multiplier': 3.0,
            'liquidity_multiplier': 0.3,
            'gas_price_multiplier': 5.0,
            'opportunity_multiplier': 0.4
        }

        # Recovery regime
        self.regime_models[MarketRegime.RECOVERY] = {
            'drift': 0.001,  # Strong positive drift
            'volatility_multiplier': 1.8,
            'liquidity_multiplier': 0.8,
            'gas_price_multiplier': 2.0,
            'opportunity_multiplier': 1.3
        }

    def generate_scenario_matrix(self, base_params: Dict[str, Any],
                                 num_scenarios: int = 100) ->
List[ScenarioParameters]:
        """Generate a matrix of scenarios for comprehensive testing"""

        scenarios = []

        # Define scenario grid
        regimes = list(MarketRegime)
        severities = [0.1, 0.3, 0.5, 0.7, 0.9]
        durations = [7, 14, 30, 60]
```

```
        # Generate combinations
        combinations = list(itertools.product(regimes, severities,
durations))

        # Sample scenarios to reach target number
        if len(combinations) > num_scenarios:
            selected_combinations = np.random.choice(
                len(combinations), size=num_scenarios, replace=False
            )
            combinations = [combinations[i] for i in
selected_combinations]

        for regime, severity, duration in combinations:

            # Generate scenario-specific parameters
            scenario = self._generate_single_scenario(base_params,
regime, severity, duration)
            scenarios.append(scenario)

        return scenarios

    def _generate_single_scenario(self, base_params: Dict[str, Any],
                                  regime: MarketRegime, severity: float,
                                  duration: int) -> ScenarioParameters:
        """Generate a single scenario with specific parameters"""

        # Get regime model
        model = self.regime_models[regime]

        # Calculate adjusted parameters
        adjusted_drift = model['drift'] * severity
        adjusted_volatility = base_params.get('volatility', 0.03) *
model['volatility_multiplier']

        # Generate correlation structure
        correlations = self._generate_correlation_structure(regime,
severity)

        # Generate volatilities
        volatilities = {
            'price_volatility': adjusted_volatility,
```

```
            'liquidity_volatility': adjusted_volatility *
model.get('liquidity_volatility', 1.5),
            'gas_volatility': adjusted_volatility *
model.get('gas_volatility', 2.0),
            'volume_volatility': adjusted_volatility * 1.2
        }

        # Generate jumps and structural changes
        jumps = self._generate_jump_events(regime, severity, duration)
        structural_changes = self._generate_structural_changes(regime,
severity, duration)

        return ScenarioParameters(
            regime=regime,
            severity=severity,
            duration=duration,
            correlations=correlations,
            volatilities=volatilities,
            jumps=jumps,
            structural_changes=structural_changes
        )

    def _generate_correlation_structure(self, regime: MarketRegime,
severity: float) -> Dict[str, float]:
        """Generate correlation structure for the scenario"""

        # Base correlations
        base_correlations = {
            'price_liquidity': -0.6,  # Negative correlation
            'price_gas': 0.3,         # Positive correlation
            'liquidity_gas': -0.4,    # Negative correlation
            'volume_gas': 0.5         # Positive correlation
        }

        # Adjust correlations based on regime and severity
        adjusted_correlations = {}

        if regime == MarketRegime.CRISIS:
            # During crisis, correlations break down
            correlation_breakdown = severity * 0.8
            for key, base_corr in base_correlations.items():
                adjusted_correlations[key] = base_corr * (1 -
```

```
correlation_breakdown)

        elif regime == MarketRegime.RECOVERY:
            # During recovery, correlations strengthen
            correlation_strengthening = severity * 0.3
            for key, base_corr in base_correlations.items():
                adjusted_correlations[key] = base_corr * (1 +
correlation_strengthening)

        else:
            # Normal adjustment
            adjustment_factor = 1 + (severity - 0.5) * 0.4
            for key, base_corr in base_correlations.items():
                adjusted_correlations[key] = base_corr *
adjustment_factor

        return adjusted_correlations

    def _generate_jump_events(self, regime: MarketRegime, severity:
float,
                              duration: int) -> List[Dict[str, Any]]:
        """Generate jump events for the scenario"""

        jumps = []

        # Number of jumps depends on regime and severity
        if regime == MarketRegime.CRISIS:
            jump_probability = 0.8 * severity
        elif regime == MarketRegime.BEAR:
            jump_probability = 0.3 * severity
        else:
            jump_probability = 0.1 * severity

        # Generate jumps
        for day in range(duration):
            if np.random.random() < jump_probability:
                jump_magnitude = np.random.normal(
                    regime == MarketRegime.CRISIS and -0.1 or 0,
                    0.05 * (1 + severity)
                )

                jumps.append({
```

```
                    'day': day,
                    'magnitude': jump_magnitude,
                    'duration': np.random.randint(1, max(2,
int(duration * 0.1))),
                    'affected_assets': ['price', 'liquidity', 'gas']
                })

        return jumps

    def _generate_structural_changes(self, regime: MarketRegime,
severity: float,
                                     duration: int) -> List[Dict[str,
Any]]:
        """Generate structural changes for the scenario"""

        structural_changes = []

        # Probability of structural changes
        if regime == MarketRegime.CRISIS:
            change_probability = 0.6 * severity
        elif regime == MarketRegime.RECOVERY:
            change_probability = 0.4 * severity
        else:
            change_probability = 0.1 * severity

        # Generate structural changes
        for day in range(duration):
            if np.random.random() < change_probability and day <
duration - 7:  # Leave time for effect

                change_type = np.random.choice([
                    'liquidity_withdrawal',
                    'regulatory_shock',
                    'technology_failure',
                    'market_maker_exit'
                ])

                change_magnitude = severity * np.random.uniform(0.2,
0.8)

                structural_changes.append({
                    'day': day,
```

```
                    'type': change_type,
                    'magnitude': change_magnitude,
                    'recovery_time': np.random.randint(7, min(30,
duration - day))
                })

        return structural_changes

    def generate_extreme_tail_scenarios(self, base_params: Dict[str,
Any],
                                        num_extreme_scenarios: int = 20)
-> List[ScenarioParameters]:
        """Generate extreme tail scenarios for stress testing"""

        extreme_scenarios = []

        # Define extreme combinations
        extreme_combinations = [
            (MarketRegime.CRISIS, 0.9, 30),
            (MarketRegime.CRISIS, 0.95, 14),
            (MarketRegime.BEAR, 0.8, 60),
            (MarketRegime.RECOVERY, 0.8, 30)  # Recovery can be
volatile
        ]

        # Generate scenarios
        for i in range(num_extreme_scenarios):
            regime, severity, duration = extreme_combinations[i %
len(extreme_combinations)]

            # Add some randomness
            severity = min(1.0, severity + np.random.uniform(-0.05,
0.05))
            duration = max(7, int(duration + np.random.randint(-7, 8)))

            scenario = self._generate_single_scenario(base_params,
regime, severity, duration)
            extreme_scenarios.append(scenario)

        return extreme_scenarios

    def optimize_scenario_selection(self, scenario_pool:
```

```
List[ScenarioParameters],
                                    selection_criteria: Dict[str, float])
-> List[ScenarioParameters]:
        """Optimize scenario selection based on criteria"""

        if not selection_criteria:
            return scenario_pool

        # Score each scenario
        scenario_scores = []

        for scenario in scenario_pool:
            score = 0

            # Diversity score (regime variety)
            regime_diversity = len(set(s.regime for s in scenario_pool
if s.regime == scenario.regime))
            score += regime_diversity * 0.3

            # Severity coverage
            severity_coverage = abs(scenario.severity -
selection_criteria.get('target_severity', 0.5))
            score += (1 - severity_coverage) * 0.3

            # Duration variety
            duration_coverage = abs(scenario.duration -
selection_criteria.get('target_duration', 30))
            score += (1 - duration_coverage / 60) * 0.2
# Normalize by max duration

            # Complexity score
            complexity = len(scenario.jumps) +
len(scenario.structural_changes)
            score += min(1.0, complexity / 10) * 0.2

            scenario_scores.append((scenario, score))

        # Sort by score and select top scenarios
        scenario_scores.sort(key=lambda x: x[1], reverse=True)

        # Select diverse scenarios
        selected_scenarios = []
```

```
        selected_regimes = set()

        for scenario, score in scenario_scores:
            if len(selected_scenarios) >=
selection_criteria.get('num_scenarios', 50):
                break

            # Ensure regime diversity
            if scenario.regime not in selected_regimes or
len(selected_scenarios) > 20:
                selected_scenarios.append(scenario)
                selected_regimes.add(scenario.regime)

        return selected_scenarios

# Example usage
scenario_generator = AdaptiveScenarioGenerator({})

# Generate comprehensive scenario matrix
base_params = {
    'initial_price': 2000.0,
    'initial_liquidity': 15000.0,
    'volatility': 0.03
}

# Generate 100 scenarios for testing
scenario_matrix =
scenario_generator.generate_scenario_matrix(base_params,
num_scenarios=100)

print(f"Generated {len(scenario_matrix)} scenarios")

# Analyze scenario distribution
regime_counts = {}
severity_distribution = []
duration_distribution = []

for scenario in scenario_matrix:
    regime_counts[scenario.regime.value] =
regime_counts.get(scenario.regime.value, 0) + 1
    severity_distribution.append(scenario.severity)
    duration_distribution.append(scenario.duration)
```

```
print(f"Regime distribution: {regime_counts}")
print(f"Severity range: {min(severity_distribution):.1f} -
{max(severity_distribution):.1f}")
print(f"Duration range: {min(duration_distribution)} -
{max(duration_distribution)} days")

# Generate extreme tail scenarios
extreme_scenarios =
scenario_generator.generate_extreme_tail_scenarios(base_params, 20)

print(f"Generated {len(extreme_scenarios)} extreme tail scenarios")

# Optimize scenario selection
selection_criteria = {
    'target_severity': 0.7,
    'target_duration': 30,
    'num_scenarios': 25
}

optimized_scenarios =
scenario_generator.optimize_scenario_selection(scenario_matrix,
selection_criteria)

print(f"Selected {len(optimized_scenarios)} optimized scenarios")

# Analyze optimized selection
optimized_regimes = {}
for scenario in optimized_scenarios:
    optimized_regimes[scenario.regime.value] =
optimized_regimes.get(scenario.regime.value, 0) + 1

print(f"Optimized regime distribution: {optimized_regimes}")
```

5.3 Advanced Risk Analysis

5.3.1 Tail Risk Analysis

```
from typing import Dict, List, Any, Optional, Tuple
from dataclasses import dataclass
import numpy as np
from scipy import stats
from scipy.optimize import minimize
import pandas as pd

@dataclass
class TailRiskMetrics:
    """Tail risk measurement metrics"""
    var_99: float
    var_99_9: float
    expected_shortfall_99: float
    extreme_value_index: float
    tail_dependency: float
    max_drawdown: float
    drawdown_duration: int
    recovery_time: float

class TailRiskAnalyzer:
    """Advanced tail risk analysis for MEV strategies"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.extreme_value_models = {}

    def analyze_tail_risk(self, simulation_results:
List[SimulationResult],
                          confidence_levels: List[float] = [0.95, 0.99,
0.999]) -> Dict[str, Any]:
        """Comprehensive tail risk analysis"""

        # Extract returns
        returns = [r.total_return for r in simulation_results]

        # Basic tail metrics
        tail_metrics = self._calculate_tail_metrics(returns,
confidence_levels)

        # Extreme Value Theory analysis
        evt_analysis = self._extreme_value_analysis(returns)
```

```
        # Drawdown analysis
        drawdown_analysis = self._analyze_drawdowns(simulation_results)

        # Tail dependence analysis
        tail_dependence =
self._analyze_tail_dependence(simulation_results)

        # Stress scenario analysis
        stress_analysis =
self._stress_scenario_tail_analysis(simulation_results)

        return {
            'tail_metrics': tail_metrics,
            'extreme_value_analysis': evt_analysis,
            'drawdown_analysis': drawdown_analysis,
            'tail_dependence': tail_dependence,
            'stress_analysis': stress_analysis,
            'risk_recommendations':
self._generate_risk_recommendations(tail_metrics, evt_analysis)
        }

    def _calculate_tail_metrics(self, returns: List[float],
                                confidence_levels: List[float]) ->
Dict[str, float]:
        """Calculate Value at Risk and Expected Shortfall"""

        returns_array = np.array(returns)

        metrics = {}

        for confidence in confidence_levels:
            # Value at Risk
            var = np.percentile(returns_array, (1 - confidence) * 100)

            # Expected Shortfall (Conditional VaR)
            tail_returns = returns_array[returns_array <= var]
            es = np.mean(tail_returns) if len(tail_returns) > 0 else
var

            metrics[f'var_{int(confidence * 100)}'] = var
            metrics[f'es_{int(confidence * 100)}'] = es
```

```
            # Return period (expected frequency of such losses)
            exceedance_prob = (1 - confidence)
            return_period = 1 / exceedance_prob if exceedance_prob > 0
else float('inf')
            metrics[f'return_period_{int(confidence * 100)}'] =
return_period

        return metrics

    def _extreme_value_analysis(self, returns: List[float]) ->
Dict[str, Any]:
        """Extreme Value Theory analysis using Block Maxima method"""

        if len(returns) < 100:
            return {'error': 'Insufficient data for EVT analysis'}

        # Block maxima approach (monthly blocks)
        block_size = 30  # 30-day blocks
        returns_array = np.array(returns)

        # Reshape into blocks
        num_blocks = len(returns_array) // block_size
        if num_blocks < 10:
            return {'error': 'Insufficient blocks for EVT analysis'}

        blocks = returns_array[:num_blocks *
block_size].reshape(num_blocks, block_size)
        block_maxima = np.min(blocks, axis=1)
# Minimum returns (worst case)

        # Fit Generalized Extreme Value distribution
        try:
            from scipy.stats import genextreme
            params = genextreme.fit(block_maxima)
            shape, loc, scale = params

            # Calculate tail index (ξ)
            tail_index = shape

            # Determine distribution type
            if shape > 0:
                distribution_type = "Fréchet (Heavy-tailed)"
```

```
            elif shape < 0:
                distribution_type = "Weibull (Light-tailed)"
            else:
                distribution_type = "Gumbel (Exponential tails)"

            # Calculate extreme quantiles
            extreme_vars = {}
            for level in [0.95, 0.99, 0.999, 0.9999]:
                extreme_var = genextreme.ppf(1 - level, shape, loc=loc,
scale=scale)
                extreme_vars[f'evt_var_{int(level * 10000)}'] =
extreme_var

            return {
                'distribution_type': distribution_type,
                'tail_index': tail_index,
                'shape_parameter': shape,
                'location_parameter': loc,
                'scale_parameter': scale,
                'num_blocks': num_blocks,
                'block_maxima_mean': np.mean(block_maxima),
                'block_maxima_std': np.std(block_maxima),
                'extreme_quantiles': extreme_vars,
                'tail_risk_assessment':
self._assess_tail_risk_from_evt(tail_index)
            }

        except Exception as e:
            return {'error': f'EVT fitting failed: {str(e)}'}

    def _assess_tail_risk_from_evt(self, tail_index: float) -> str:
        """Assess tail risk based on extreme value index"""

        if tail_index > 0.5:
            return "EXTREME - Heavy-tailed distribution indicates
extreme tail risk"
        elif tail_index > 0.2:
            return "HIGH - Significant tail risk present"
        elif tail_index > 0:
            return "MODERATE - Moderate tail risk"
        elif tail_index > -0.2:
            return "LOW - Light tails, limited tail risk"
```

```
        else:
            return "MINIMAL - Very light tails"

    def _analyze_drawdowns(self, simulation_results:
List[SimulationResult]) -> Dict[str, Any]:
        """Analyze drawdown characteristics"""

        drawdowns = []
        durations = []
        recovery_times = []

        for result in simulation_results:
            # Extract capital timeline
            timeline = result.timeline
            values = [day['capital'] for day in timeline]

            if len(values) < 2:
                continue

            # Calculate drawdown series
            peak = values[0]
            drawdown_series = []

            for value in values:
                if value > peak:
                    peak = value
                drawdown = (peak - value) / peak
                drawdown_series.append(drawdown)

            # Find maximum drawdown
            max_dd = max(drawdown_series)
            drawdowns.append(max_dd)

            # Find drawdown duration
            in_drawdown = False
            dd_duration = 0
            max_dd_duration = 0

            for dd in drawdown_series:
                if dd > 0.01:  # 1% threshold
                    if not in_drawdown:
                        in_drawdown = True
```

```
                    dd_duration = 0
                dd_duration += 1
                max_dd_duration = max(max_dd_duration, dd_duration)
            else:
                if in_drawdown:
                    durations.append(dd_duration)
                    in_drawdown = False

        # Recovery time (time to reach previous peak)
        peak_value = values[0]
        recovery_time = None

        for i, value in enumerate(values):
            if value >= peak_value and i > 0:
                recovery_time = i
                break

        if recovery_time is not None:
            recovery_times.append(recovery_time)

    # Aggregate drawdown statistics
    if drawdowns:
        drawdown_analysis = {
            'mean_max_drawdown': np.mean(drawdowns),
            'median_max_drawdown': np.median(drawdowns),
            'worst_drawdown': max(drawdowns),
            'drawdown_percentile_95': np.percentile(drawdowns, 95),
            'probability_of_large_drawdown': sum(1 for dd in
drawdowns if dd > 0.2) / len(drawdowns),
            'mean_drawdown_duration': np.mean(durations) if
durations else 0,
            'worst_drawdown_duration': max(durations) if durations
else 0,
            'mean_recovery_time': np.mean(recovery_times) if
recovery_times else 0,
            'unrecovered_simulations': len(drawdowns) -
len(recovery_times)
        }
    else:
        drawdown_analysis = {'error': 'No drawdowns found'}

    return drawdown_analysis
```

```
    def _analyze_tail_dependence(self, simulation_results:
List[SimulationResult]) -> Dict[str, Any]:
        """Analyze tail dependence between different risk factors"""

        if len(simulation_results) < 50:
            return {'error': 'Insufficient data for tail dependence
analysis'}

        # Extract relevant metrics for correlation analysis
        returns = [r.total_return for r in simulation_results]
        max_drawdowns = [r.max_drawdown for r in simulation_results]
        sharpe_ratios = [r.sharpe_ratio for r in simulation_results]
        win_rates = [r.win_rate for r in simulation_results]

        # Calculate correlations
        correlation_matrix = np.corrcoef([returns, max_drawdowns,
sharpe_ratios, win_ratios])

        # Calculate tail dependence coefficients
        tail_dependence = {}

        # Upper tail dependence (extreme positive outcomes)
        upper_tail_dep = self._calculate_tail_dependence_coeff(
            np.array(returns), np.array(sharpe_ratios), upper=True
        )

        # Lower tail dependence (extreme negative outcomes)
        lower_tail_dep = self._calculate_tail_dependence_coeff(
            np.array(returns), np.array(max_drawdowns), upper=False
        )

        tail_dependence.update({
            'upper_tail_dependence': upper_tail_dep,
            'lower_tail_dependence': lower_tail_dep,
            'correlation_matrix': correlation_matrix.tolist(),
            'risk_factor_correlations': {
                'return_drawdown_corr': correlation_matrix[0, 1],
                'return_sharpe_corr': correlation_matrix[0, 2],
                'drawdown_sharpe_corr': correlation_matrix[1, 2]
            }
        })
```

```
        return tail_dependence

    def _calculate_tail_dependence_coeff(self, x: np.ndarray, y:
np.ndarray,
                                         upper: bool = True) -> float:
        """Calculate tail dependence coefficient"""

        # Calculate empirical copula
        n = len(x)

        # Transform to uniform margins
        u = np.argsort(np.argsort(x)) / n
        v = np.argsort(np.argsort(y)) / n

        # Define threshold for tail analysis
        threshold = 0.95 if upper else 0.05

        if upper:
            # Upper tail dependence
            u_exceed = u >= threshold
            v_exceed = v >= threshold
        else:
            # Lower tail dependence
            u_exceed = u <= threshold
            v_exceed = v <= threshold

        # Calculate tail dependence coefficient
        if np.sum(u_exceed) == 0:
            return 0.0

        joint_exceed = np.sum(u_exceed & v_exceed)
        individual_exceed = np.sum(u_exceed)

        return joint_exceed / individual_exceed if individual_exceed >
0 else 0.0

    def _stress_scenario_tail_analysis(self, simulation_results:
List[SimulationResult]) -> Dict[str, Any]:
        """Analyze tail behavior under stress scenarios"""

        # This would integrate with stress testing framework
```

```
        # For now, we'll analyze the extremes in regular simulations

        if len(simulation_results) < 100:
            return {'error': 'Insufficient data for stress scenario
analysis'}

        # Extract worst-case scenarios
        returns = [r.total_return for r in simulation_results]

        # Define extreme thresholds
        var_1_percent = np.percentile(returns, 1)
        extreme_returns = [r for r in returns if r <= var_1_percent]

        # Analyze characteristics of extreme scenarios
        extreme_scenarios = [r for r in simulation_results if
r.total_return <= var_1_percent]

        if not extreme_scenarios:
            return {'message': 'No extreme scenarios found'}

        # Extract common characteristics
        max_drawdowns = [r.max_drawdown for r in extreme_scenarios]
        sharpe_ratios = [r.sharpe_ratio for r in extreme_scenarios]
        win_rates = [r.win_rate for r in extreme_scenarios]
        total_trades = [r.total_trades for r in extreme_scenarios]
        gas_costs = [r.gas_costs for r in extreme_scenarios]

        return {
            'extreme_scenario_count': len(extreme_scenarios),
            'extreme_scenario_percentage': len(extreme_scenarios) /
len(simulation_results) * 100,
            'common_characteristics': {
                'average_max_drawdown': np.mean(max_drawdowns),
                'average_sharpe_ratio': np.mean(sharpe_ratios),
                'average_win_rate': np.mean(win_rates),
                'average_total_trades': np.mean(total_trades),
                'average_gas_costs': np.mean(gas_costs)
            },
            'extreme_scenario_variability': {
                'drawdown_std': np.std(max_drawdowns),
                'sharpe_std': np.std(sharpe_ratios),
                'trades_std': np.std(total_trades)
```

```
            }
        }

    def _generate_risk_recommendations(self, tail_metrics: Dict[str,
float],
                                       evt_analysis: Dict[str, Any]) ->
List[str]:
        """Generate risk management recommendations"""

        recommendations = []

        # VaR-based recommendations
        var_99 = tail_metrics.get('var_99', 0)
        if var_99 < -0.3:
            recommendations.append("CRITICAL: 99% VaR exceeds 30%.
Consider reducing position sizes immediately.")
        elif var_99 < -0.2:
            recommendations.append("HIGH: 99% VaR exceeds 20%.
Implement strict risk limits.")
        elif var_99 < -0.1:
            recommendations.append("MODERATE: Monitor VaR levels
closely.")

        # Expected Shortfall recommendations
        es_99 = tail_metrics.get('es_99', 0)
        if es_99 < -0.4:
            recommendations.append("EXTREME: Expected Shortfall
indicates severe tail risk. Reduce exposure.")

        # EVT-based recommendations
        tail_index = evt_analysis.get('tail_index', 0)
        if tail_index > 0.5:
            recommendations.append("Extreme Value Theory indicates
heavy-tailed risk. Expect frequent extreme events.")
        elif tail_index > 0.2:
            recommendations.append("Moderate tail risk detected.
Prepare for occasional extreme moves.")

        # Drawdown recommendations
        if len(recommendations) == 0:
            recommendations.append("Tail risk appears manageable under
current parameters.")
```

```
        return recommendations

# Example usage
tail_analyzer = TailRiskAnalyzer({})

# Using simulation results from earlier example
# tail_results =
tail_analyzer.analyze_tail_risk(mc_results['individual_results'])

# print(f"99% VaR: {tail_results['tail_metrics']['var_99']:.2%}")
# print(f"Expected Shortfall (99%): {tail_results['tail_metrics']
['es_99']:.2%}")
# print(f"Tail Index:
{tail_results['extreme_value_analysis'].get('tail_index', 'N/A')}")
```

5.4 Practical Exercise: Complete Stress Testing Framework

```
def build_complete_stress_testing_system():
    """Complete exercise to build a stress testing and scenario
analysis system"""

    print("Building Complete Stress Testing & Scenario Analysis
System...")

    # Initialize components
    config = {'stress_config': 'comprehensive'}

    # 1. Monte Carlo Engine
    print("\n1. Testing Monte Carlo Simulation...")
    mc_engine = MonteCarloEngine(config)

    # Basic Monte Carlo test
    basic_params = SimulationParameters(
        num_simulations=100,
        time_horizon_days=30,
        initial_capital=1_000_000,
        strategy_config={'base_opportunities_per_day': 10},
        market_parameters={
            'initial_price': 2000.0,
            'initial_liquidity': 10000.0,
            'volatility': 0.03
        },
        random_seed=42
    )

    basic_results = mc_engine.run_monte_carlo_simulation(basic_params)
    print(f"   Completed {len(basic_results['individual_results'])}
simulations")
    print(f"   Expected return: {basic_results['aggregate_results']
['expected_return']:.2%}")
    print(f"   95% VaR: {basic_results['aggregate_results']['var_95']:.
2%}")

    # 2. Parallel Monte Carlo
    print("\n2. Testing Parallel Monte Carlo...")
    parallel_engine = ParallelMonteCarloEngine(config, num_processes=2)

    parallel_results =
parallel_engine.run_parallel_simulation(basic_params)
```

```
    print(f"   Parallel processing: {parallel_results['success_rate']:.
1%} success rate")
    print(f"   Results: {len(parallel_results['individual_results'])}
simulations")

    # 3. Stress Testing Framework
    print("\n3. Testing Stress Testing Framework...")
    stress_framework = StressTestingFramework(config)

    # Define stress scenarios
    stress_scenarios = [
        StressTestConfig(
            scenario=StressScenario.BLACK_MONDAY,
            severity=0.6,
            duration_days=30,
            start_day=5,
            strategy_params={'base_opportunities_per_day': 10},
            market_parameters=basic_params.market_parameters
        ),
        StressTestConfig(
            scenario=StressScenario.GAS_PRICE_SPIKE,
            severity=0.8,
            duration_days=14,
            start_day=0,
            strategy_params={'base_opportunities_per_day': 10},
            market_parameters=basic_params.market_parameters
        )
    ]

    stress_results =
stress_framework.run_multiple_stress_tests(stress_scenarios,
basic_results['aggregate_results'])
    print(f"   Stress scenarios tested:
{stress_results['aggregate_results']['total_scenarios_tested']}")
    print(f"   Average return impact:
{stress_results['aggregate_results']['average_return_impact']:.2%}")

    # 4. Adaptive Scenario Generator
    print("\n4. Testing Scenario Generator...")
    scenario_generator = AdaptiveScenarioGenerator({})

    # Generate scenario matrix
```

```
    base_market_params = {
        'initial_price': 2000.0,
        'initial_liquidity': 15000.0,
        'volatility': 0.03
    }

    scenario_matrix =
scenario_generator.generate_scenario_matrix(base_market_params, 50)
    print(f"   Generated {len(scenario_matrix)} scenarios")

    # Analyze scenario diversity
    regime_counts = {}
    for scenario in scenario_matrix:
        regime = scenario.regime.value
        regime_counts[regime] = regime_counts.get(regime, 0) + 1

    print(f"   Scenario diversity: {regime_counts}")

    # Generate extreme scenarios
    extreme_scenarios =
scenario_generator.generate_extreme_tail_scenarios(base_market_params,
10)
    print(f"   Extreme scenarios: {len(extreme_scenarios)}")

    # 5. Tail Risk Analysis
    print("\n5. Testing Tail Risk Analysis...")
    tail_analyzer = TailRiskAnalyzer({})

    tail_results =
tail_analyzer.analyze_tail_risk(basic_results['individual_results'])
    print(f"   99% VaR: {tail_results['tail_metrics']['var_99']:.2%}")
    print(f"   Expected Shortfall: {tail_results['tail_metrics']
['es_99']:.2%}")

    if 'tail_index' in tail_results['extreme_value_analysis']:
        print(f"   EVT Tail Index:
{tail_results['extreme_value_analysis']['tail_index']:.3f}")
        print(f"   Distribution Type:
{tail_results['extreme_value_analysis']['distribution_type']}")

    # 6. Comprehensive Integration Test
    print("\n6. Testing Complete Integration...")
```

```
    # Run a comprehensive test combining multiple components
    comprehensive_params = SimulationParameters(
        num_simulations=200,
        time_horizon_days=60,
        initial_capital=1_000_000,
        strategy_config={'base_opportunities_per_day': 12},
        market_parameters={
            'initial_price': 2000.0,
            'initial_liquidity': 15000.0,
            'volatility': 0.04,
            'drift': 0.001
        },
        random_seed=123
    )

    comprehensive_results =
mc_engine.run_monte_carlo_simulation(comprehensive_params)

    # Comprehensive analysis
    comprehensive_tail_analysis =
tail_analyzer.analyze_tail_risk(comprehensive_results['individual_results'])

    # Generate recommendations
    recommendations =
comprehensive_tail_analysis['risk_recommendations']

    print(f"   Comprehensive test completed:")
    print(f"   - Simulations:
{len(comprehensive_results['individual_results'])}")
    print(f"   - Expected return:
{comprehensive_results['aggregate_results']['expected_return']:.2%}")
    print(f"   - Max drawdown:
{comprehensive_results['aggregate_results']['worst_case_drawdown']:.
2%}")
    print(f"   - Probability of loss:
{comprehensive_results['aggregate_results']['probability_of_loss']:.
1%}")

    print(f"\n   Risk Recommendations:")
    for i, rec in enumerate(recommendations, 1):
        print(f"   {i}. {rec}")
```

```
    # 7. Performance Benchmarking
    print("\n7. Performance Benchmarking...")

    # Benchmark sequential vs parallel
    perf_test_params = SimulationParameters(
        num_simulations=200,
        time_horizon_days=20,
        initial_capital=1_000_000,
        strategy_config={'base_opportunities_per_day': 5},
        market_parameters={'initial_price': 2000.0,
'initial_liquidity': 10000.0, 'volatility': 0.03},
        random_seed=999
    )

    # Sequential test
    import time
    start_time = time.time()
    seq_results =
mc_engine.run_monte_carlo_simulation(perf_test_params)
    seq_time = time.time() - start_time

    # Parallel test
    start_time = time.time()
    par_engine = ParallelMonteCarloEngine(config, num_processes=2)
    par_results = par_engine.run_parallel_simulation(perf_test_params)
    par_time = time.time() - start_time

    speedup = seq_time / par_time
    print(f"   Sequential time: {seq_time:.2f}s")
    print(f"   Parallel time: {par_time:.2f}s")
    print(f"   Speedup: {speedup:.2f}x")

    print("\n✅ Complete Stress Testing & Scenario Analysis System
Ready!")

    # Summary
    print("\n" + "="*60)
    print("STRESS TESTING & SCENARIO ANALYSIS SUMMARY")
    print("="*60)
    print(f"Monte Carlo Engine: ✅
{len(basic_results['individual_results'])} simulations completed")
```

```
    print(f"Parallel Processing: ✅ {speedup:.2f}x speedup achieved")
    print(f"Stress Testing: ✅
{len(stress_results['individual_results'])} scenarios tested")
    print(f"Scenario Generation: ✅ {len(scenario_matrix)} +
{len(extreme_scenarios)} extreme scenarios")
    print(f"Tail Risk Analysis: ✅ EVT analysis with
{comprehensive_tail_analysis['extreme_value_analysis'].get('num_blocks',
'N/A')} blocks")
    print(f"Risk Assessment: ✅ {len(recommendations)} recommendations
generated")
    print(f"Integration Test: ✅ Comprehensive analysis successful")

    return {
        'monte_carlo_engine': mc_engine,
        'parallel_engine': parallel_engine,
        'stress_framework': stress_framework,
        'scenario_generator': scenario_generator,
        'tail_analyzer': tail_analyzer,
        'test_results': {
            'basic_simulation': basic_results,
            'stress_scenarios': stress_results,
            'scenario_matrix': scenario_matrix,
            'extreme_scenarios': extreme_scenarios,
            'tail_analysis': comprehensive_tail_analysis,
            'performance_benchmark': {
                'sequential_time': seq_time,
                'parallel_time': par_time,
                'speedup': speedup
            }
        }
    }

# Run the complete exercise
if __name__ == "__main__":
    stress_testing_system = build_complete_stress_testing_system()
    print("\nComplete stress testing and scenario analysis framework
ready for production!")
```

Module Summary

In this module, you have built a comprehensive stress testing and scenario analysis system that includes:

- **Monte Carlo Simulation Engine**: Scalable parallel and sequential simulation frameworks
- **Stress Testing Scenarios**: Multiple market stress scenarios (crashes, volatility spikes, liquidity crises)
- **Adaptive Scenario Generation**: Dynamic scenario creation based on market regimes
- **Extreme Value Theory**: Advanced tail risk analysis using statistical methods
- **Tail Risk Analysis**: Comprehensive risk metrics including VaR, ES, and drawdown analysis
- **Performance Optimization**: Parallel processing for faster simulation execution

Key Takeaways

1. **Comprehensive Testing**: Stress testing should cover multiple failure modes and extreme conditions
2. **Parallel Processing**: Monte Carlo simulations benefit significantly from parallel execution
3. **Scenario Diversity**: Generate diverse scenarios covering different market regimes
4. **Tail Risk Focus**: Pay special attention to extreme outcomes and tail dependencies
5. **Adaptive Models**: Scenario generators should adapt to current market conditions
6. **Risk Management Integration**: Use stress test results for proactive risk management

Next Steps

1. **Calibration**: Calibrate models using historical crisis data
2. **Real-time Integration**: Connect with live market monitoring
3. **Portfolio-level Stress**: Extend to multi-strategy portfolio stress testing
4. **Regulatory Compliance**: Ensure stress testing meets regulatory requirements
5. **Automated Alerts**: Implement automated risk alerts based on stress metrics

This stress testing and scenario analysis framework provides the comprehensive risk assessment tools needed for robust MEV strategy development and risk management.