Strategy Diversification

Duration: 65 minutes
Level: Intermediate
Author: Obelisk Core

Learning Objectives

By the end of this module, you will be able to:

- Build diversified portfolios of MEV strategies
- Analyze correlation between different strategies
- Optimize capital allocation across strategies
- Implement dynamic rebalancing systems
- Monitor and manage portfolio correlation risk

Introduction to Strategy Diversification

Diversification is one of the most powerful tools in risk management, allowing you to reduce portfolio risk without proportionally reducing returns. In MEV trading, where strategies can be highly correlated during market stress, effective diversification requires careful selection and ongoing monitoring.

Why Diversification Matters in MEV

MEV strategies face unique correlation challenges:

- **Market regime dependencies**: All strategies may suffer during bear markets
- **Network effects**: Gas price spikes affect all on-chain strategies
- **Protocol correlations**: Lending protocol liquidations correlate with DeFi yields
- **MEV competition**: Increased competition reduces profitability across all strategies
- **Infrastructure risks**: Chain upgrades, outages affect all strategies simultaneously

Types of Diversification

1. **Strategy Diversification**: Different MEV approaches (arbitrage, liquidation, sandwich)
2. **Market Diversification**: Different blockchain networks and protocols
3. **Time Diversification**: Strategies active at different times
4. **Risk Factor Diversification**: Different risk exposures and sensitivities

5. **Geographic Diversification**: Different time zones and trading sessions

---

Correlation Analysis

1. Calculating Correlation Coefficients

```
import numpy as np
import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt

def calculate_correlation_matrix(returns_data):
    """
    Calculate correlation matrix for multiple strategy returns
    """
    correlation_matrix = returns_data.corr()
    return correlation_matrix

def calculate_rolling_correlation(series1, series2, window=30):
    """
    Calculate rolling correlation between two return series
    """
    rolling_corr =
series1.rolling(window=window).corr(series2.rolling(window=window))
    return rolling_corr

# Example with simulated MEV strategy returns
np.random.seed(42)

# Simulate returns for different MEV strategies
days = 252  # One year of daily returns
dates = pd.date_range('2023-01-01', periods=days)

# Arbitrage strategy - generally low correlation with others
arbitrage_returns = np.random.normal(0.001, 0.005, days)

# Liquidation strategy - moderate correlation with arbitrage during
stress
liquidation_returns = (arbitrage_returns * 0.3 +
                       np.random.normal(0, 0.008, days))

# Sandwich strategy - higher correlation during high MEV periods
sandwich_returns = (arbitrage_returns * 0.5 +
                    np.random.normal(0, 0.012, days))

# Front-running strategy - lower correlation
frontrun_returns = np.random.normal(0.0008, 0.006, days)
```

```
# Create DataFrame
strategy_returns = pd.DataFrame({
    'arbitrage': arbitrage_returns,
    'liquidation': liquidation_returns,
    'sandwich': sandwich_returns,
    'frontrun': frontrun_returns
}, index=dates)

# Calculate correlation matrix
corr_matrix = calculate_correlation_matrix(strategy_returns)
print("Strategy Correlation Matrix:")
print(corr_matrix.round(3))

# Calculate rolling correlations
rolling_corr_arbitrage_liquidation = calculate_rolling_correlation(
    strategy_returns['arbitrage'],
    strategy_returns['liquidation'],
    window=30
)

print(f"\nAverage Correlation (Arbitrage vs Liquidation):
{rolling_corr_arbitrage_liquidation.mean():.3f}")
print(f"Correlation Range: {rolling_corr_arbitrage_liquidation.min():.
3f} to {rolling_corr_arbitrage_liquidation.max():.3f}")
```

2. Advanced Correlation Metrics

```
class CorrelationAnalyzer:
    def __init__(self, returns_data, window=30):
        self.returns_data = returns_data
        self.window = window
        self.correlation_matrix = returns_data.corr()

    def calculate_distance_correlation(self, series1, series2):
        """
        Calculate distance correlation (non-linear correlation measure)
        """
        def distance_matrix(x):
            n = len(x)
            x = np.array(x).reshape(-1, 1)
            distances = np.zeros((n, n))
            for i in range(n):
                for j in range(n):
                    distances[i, j] = np.abs(x[i] - x[j])
            return distances

        def center_distance_matrix(dist_matrix):
            n = dist_matrix.shape[0]
            row_means = np.mean(dist_matrix, axis=1)
            col_means = np.mean(dist_matrix, axis=0)
            grand_mean = np.mean(dist_matrix)
            centered = np.zeros((n, n))
            for i in range(n):
                for j in range(n):
                    centered[i, j] = (dist_matrix[i, j] - row_means[i]
-
                                      col_means[j] + grand_mean)
            return centered

        # Calculate distance matrices
        dist_x = distance_matrix(series1)
        dist_y = distance_matrix(series2)

        # Center matrices
        centered_x = center_distance_matrix(dist_x)
        centered_y = center_distance_matrix(dist_y)

        # Calculate distance covariances and variances
        dcov_xy = np.sqrt(np.mean(centered_x * centered_y))
```

```
        dcov_xx = np.sqrt(np.mean(centered_x * centered_x))
        dcov_yy = np.sqrt(np.mean(centered_y * centered_y))

        # Distance correlation
        if dcov_xx * dcov_yy > 0:
            dcor = dcov_xy / np.sqrt(dcov_xx * dcov_yy)
        else:
            dcor = 0

        return dcor

    def calculate_maximum_drawdown_correlation(self, strategy1,
strategy2):
        """
        Calculate correlation during drawdown periods
        """
        # Calculate cumulative returns
        cum_returns1 = (1 + strategy1).cumprod()
        cum_returns2 = (1 + strategy2).cumprod()

        # Calculate drawdowns
        peak1 = cum_returns1.expanding().max()
        peak2 = cum_returns2.expanding().max()
        drawdown1 = (cum_returns1 - peak1) / peak1
        drawdown2 = (cum_returns2 - peak2) / peak2

        # Get returns during drawdowns
        dd_returns1 = strategy1[drawdown1 < -0.05]  # 5%+ drawdowns
        dd_returns2 = strategy2[drawdown2 < -0.05]

        if len(dd_returns1) > 10 and len(dd_returns2) > 10:
            # Align the return series
            min_len = min(len(dd_returns1), len(dd_returns2))
            aligned_returns1 = dd_returns1.tail(min_len)
            aligned_returns2 = dd_returns2.tail(min_len)

            correlation = aligned_returns1.corr(aligned_returns2)
            return correlation
        else:
            return np.nan

    def calculate_tail_correlation(self, strategy1, strategy2,
```

```
percentile=5):
        """
        Calculate correlation during extreme negative returns
        """
        threshold1 = np.percentile(strategy1, percentile)
        threshold2 = np.percentile(strategy2, percentile)

        tail_returns1 = strategy1[strategy1 <= threshold1]
        tail_returns2 = strategy2[strategy2 <= threshold2]

        # Find common tail events
        common_dates = strategy1.index.intersection(strategy2.index)
        common_tail1 = strategy1[common_dates][strategy1[common_dates]
<= threshold1]
        common_tail2 = strategy2[common_dates][strategy2[common_dates]
<= threshold2]

        if len(common_tail1) > 5 and len(common_tail2) > 5:
            return common_tail1.corr(common_tail2)
        else:
            return np.nan

    def analyze_correlation_regimes(self):
        """
        Analyze how correlations change over time
        """
        correlations_over_time = {}

        for strategy1 in self.returns_data.columns:
            for strategy2 in self.returns_data.columns:
                if strategy1 < strategy2:  # Avoid duplicates
                    rolling_corr =
self.returns_data[strategy1].rolling(
                        self.window).corr(self.returns_data[strategy2])

                    correlations_over_time[f"{strategy1}_{strategy2}"]
= {
                        'rolling_correlation': rolling_corr,
                        'mean_correlation': rolling_corr.mean(),
                        'correlation_volatility': rolling_corr.std(),
                        'max_correlation': rolling_corr.max(),
                        'min_correlation': rolling_corr.min()
```

```
                    }

        return correlations_over_time

# Example usage
analyzer = CorrelationAnalyzer(strategy_returns)

# Calculate advanced correlations
distance_corr = analyzer.calculate_distance_correlation(
    strategy_returns['arbitrage'],
    strategy_returns['liquidation']
)

max_dd_corr = analyzer.calculate_maximum_drawdown_correlation(
    strategy_returns['arbitrage'],
    strategy_returns['sandwich']
)

tail_corr = analyzer.calculate_tail_correlation(
    strategy_returns['arbitrage'],
    strategy_returns['liquidation']
)

print("Advanced Correlation Metrics:")
print(f"Distance Correlation (Arbitrage vs Liquidation):
{distance_corr:.3f}")
print(f"Drawdown Correlation (Arbitrage vs Sandwich): {max_dd_corr:.
3f}")
print(f"Tail Correlation (Arbitrage vs Liquidation): {tail_corr:.3f}")

# Analyze correlation regimes
regime_analysis = analyzer.analyze_correlation_regimes()

print("\nCorrelation Regime Analysis:")
for pair, analysis in regime_analysis.items():
    print(f"{pair}:")
    print(f"  Mean: {analysis['mean_correlation']:.3f}")
    print(f"  Volatility: {analysis['correlation_volatility']:.3f}")
    print(f"  Range: {analysis['min_correlation']:.3f} to
{analysis['max_correlation']:.3f}")
```

Portfolio Optimization

1. Markowitz Mean-Variance Optimization

```
from scipy.optimize import minimize

class MevPortfolioOptimizer:
    def __init__(self, returns_data):
        self.returns_data = returns_data
        self.expected_returns = returns_data.mean()
        self.covariance_matrix = returns_data.cov()
        self.n_assets = len(returns_data.columns)

    def calculate_portfolio_metrics(self, weights):
        """
        Calculate portfolio return, risk, and Sharpe ratio
        """
        portfolio_return = np.dot(weights, self.expected_returns) * 252
# Annualized
        portfolio_risk = np.sqrt(np.dot(weights.T,
np.dot(self.covariance_matrix, weights)) * 252)
        sharpe_ratio = portfolio_return / portfolio_risk if
portfolio_risk > 0 else 0

        return portfolio_return, portfolio_risk, sharpe_ratio

    def optimize_max_sharpe(self, risk_free_rate=0.02):
        """
        Optimize portfolio for maximum Sharpe ratio
        """
        def negative_sharpe(weights):
            portfolio_return, portfolio_risk, sharpe_ratio =
self.calculate_portfolio_metrics(weights)
            return -sharpe_ratio  # Negative because we minimize

        # Constraints: weights sum to 1
        constraints = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}

        # Bounds: each weight between 0 and 1 (long-only)
        bounds = tuple((0, 1) for _ in range(self.n_assets))

        # Initial guess: equal weights
        initial_weights = np.array([1/self.n_assets] * self.n_assets)

        result = minimize(negative_sharpe, initial_weights,
method='SLSQP',
```

```
                        bounds=bounds, constraints=constraints)

        return result.x

    def optimize_minimum_variance(self):
        """
        Optimize portfolio for minimum variance
        """
        def portfolio_variance(weights):
            return np.dot(weights.T, np.dot(self.covariance_matrix,
weights))

        # Constraints: weights sum to 1
        constraints = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}

        # Bounds: each weight between 0 and 1
        bounds = tuple((0, 1) for _ in range(self.n_assets))

        # Initial guess
        initial_weights = np.array([1/self.n_assets] * self.n_assets)

        result = minimize(portfolio_variance, initial_weights,
method='SLSQP',
                          bounds=bounds, constraints=constraints)

        return result.x

    def optimize_maximum_diversification(self):
        """
        Optimize portfolio for maximum diversification ratio
        """
        def negative_diversification_ratio(weights):
            portfolio_return, portfolio_risk, sharpe_ratio =
self.calculate_portfolio_metrics(weights)

            # Calculate weighted average of individual volatilities
            individual_vols = np.sqrt(np.diag(self.covariance_matrix))
            weighted_avg_vol = np.dot(weights, individual_vols) *
np.sqrt(252)

            # Diversification ratio
            div_ratio = weighted_avg_vol / portfolio_risk if
```

```
portfolio_risk > 0 else 0
            return -div_ratio  # Negative because we minimize

        # Constraints and bounds
        constraints = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}
        bounds = tuple((0, 1) for _ in range(self.n_assets))
        initial_weights = np.array([1/self.n_assets] * self.n_assets)

        result = minimize(negative_diversification_ratio,
initial_weights, method='SLSQP',
                          bounds=bounds, constraints=constraints)

        return result.x

    def calculate_efficient_frontier(self, num_portfolios=100):
        """
        Calculate efficient frontier
        """
        # Target returns from minimum to maximum expected return
        min_ret = self.expected_returns.min() * 252
        max_ret = self.expected_returns.max() * 252
        target_returns = np.linspace(min_ret, max_ret, num_portfolios)

        efficient_portfolios = []

        for target_return in target_returns:
            def portfolio_variance(weights):
                return np.dot(weights.T, np.dot(self.covariance_matrix,
weights))

            def return_constraint(weights):
                return np.dot(weights, self.expected_returns) * 252 -
target_return

            # Constraints
            constraints = [
                {'type': 'eq', 'fun': lambda x: np.sum(x) - 1},
                {'type': 'eq', 'fun': return_constraint}
            ]

            # Bounds
            bounds = tuple((0, 1) for _ in range(self.n_assets))
```

```
            initial_weights = np.array([1/self.n_assets] *
self.n_assets)

            try:
                result = minimize(portfolio_variance, initial_weights,
method='SLSQP',
                                  bounds=bounds, constraints=constraints)

                if result.success:
                    weights = result.x
                    portfolio_return, portfolio_risk, sharpe_ratio =
self.calculate_portfolio_metrics(weights)

                    efficient_portfolios.append({
                        'weights': weights,
                        'return': portfolio_return,
                        'risk': portfolio_risk,
                        'sharpe_ratio': sharpe_ratio
                    })
            except:
                continue

        return efficient_portfolios

# Example usage
optimizer = MevPortfolioOptimizer(strategy_returns)

# Optimize for different objectives
max_sharpe_weights = optimizer.optimize_max_sharpe()
min_var_weights = optimizer.optimize_minimum_variance()
max_div_weights = optimizer.optimize_maximum_diversification()

# Calculate portfolio metrics
strategies = strategy_returns.columns.tolist()

print("Portfolio Optimization Results:")
print("\n1. Maximum Sharpe Ratio Portfolio:")
print("=" * 40)
for i, strategy in enumerate(strategies):
    print(f"{strategy}: {max_sharpe_weights[i]:.1%}")
max_sharpe_ret, max_sharpe_risk, max_sharpe_ratio =
optimizer.calculate_portfolio_metrics(max_sharpe_weights)
```

```
print(f"Expected Return: {max_sharpe_ret:.2%}")
print(f"Risk (Volatility): {max_sharpe_risk:.2%}")
print(f"Sharpe Ratio: {max_sharpe_ratio:.2f}")

print("\n2. Minimum Variance Portfolio:")
print("=" * 40)
for i, strategy in enumerate(strategies):
    print(f"{strategy}: {min_var_weights[i]:.1%}")
min_var_ret, min_var_risk, min_var_ratio =
optimizer.calculate_portfolio_metrics(min_var_weights)
print(f"Expected Return: {min_var_ret:.2%}")
print(f"Risk (Volatility): {min_var_risk:.2%}")
print(f"Sharpe Ratio: {min_var_ratio:.2f}")

print("\n3. Maximum Diversification Portfolio:")
print("=" * 40)
for i, strategy in enumerate(strategies):
    print(f"{strategy}: {max_div_weights[i]:.1%}")
max_div_ret, max_div_risk, max_div_ratio =
optimizer.calculate_portfolio_metrics(max_div_weights)
print(f"Expected Return: {max_div_ret:.2%}")
print(f"Risk (Volatility): {max_div_risk:.2%}")
print(f"Sharpe Ratio: {max_div_ratio:.2f}")
```

2. Risk Parity Portfolio

```
class RiskParityPortfolio:
    def __init__(self, covariance_matrix):
        self.covariance_matrix = covariance_matrix
        self.n_assets = len(covariance_matrix)

    def calculate_risk_contributions(self, weights):
        """
        Calculate risk contribution of each asset
        """
        portfolio_variance = np.dot(weights.T,
np.dot(self.covariance_matrix, weights))
        marginal_contrib = np.dot(self.covariance_matrix, weights)
        risk_contrib = weights * marginal_contrib / portfolio_variance

        return risk_contrib

    def optimize_risk_parity(self, max_iterations=1000,
tolerance=1e-6):
        """
        Optimize portfolio for equal risk contributions
        """
        # Initial equal weights
        weights = np.array([1/self.n_assets] * self.n_assets)

        for iteration in range(max_iterations):
            # Calculate risk contributions
            risk_contrib = self.calculate_risk_contributions(weights)
            target_contrib = 1/self.n_assets

            # Check convergence
            max_deviation = np.max(np.abs(risk_contrib -
target_contrib))
            if max_deviation < tolerance:
                break

            # Update weights (simplified Newton-Raphson)
            for i in range(self.n_assets):
                if risk_contrib[i] > 0:
                    adjustment = target_contrib / risk_contrib[i]
                    weights[i] *= adjustment

            # Normalize weights
```

```
            weights = weights / np.sum(weights)

        return weights

    def calculate_diversification_ratio(self, weights):
        """
        Calculate diversification ratio
        """
        # Weighted average of individual volatilities
        individual_vols = np.sqrt(np.diag(self.covariance_matrix))
        weighted_avg_vol = np.dot(weights, individual_vols)

        # Portfolio volatility
        portfolio_vol = np.sqrt(np.dot(weights.T,
np.dot(self.covariance_matrix, weights)))

        return weighted_avg_vol / portfolio_vol if portfolio_vol > 0
else 1

# Example usage
risk_parity = RiskParityPortfolio(optimizer.covariance_matrix)

risk_parity_weights = risk_parity.optimize_risk_parity()
risk_contributions =
risk_parity.calculate_risk_contributions(risk_parity_weights)
diversification_ratio =
risk_parity.calculate_diversification_ratio(risk_parity_weights)

print("\n4. Risk Parity Portfolio:")
print("=" * 40)
for i, strategy in enumerate(strategies):
    print(f"{strategy}: {risk_parity_weights[i]:.1%} (Risk:
{risk_contributions[i]:.1%})")

print(f"Diversification Ratio: {diversification_ratio:.2f}")
print(f"Target risk contribution per asset: {1/len(strategies):.1%}")

# Verify risk parity
risk_parity_ret, risk_parity_risk, risk_parity_sharpe =
optimizer.calculate_portfolio_metrics(risk_parity_weights)
print(f"Expected Return: {risk_parity_ret:.2%}")
```

```
print(f"Risk (Volatility): {risk_parity_risk:.2%}")
print(f"Sharpe Ratio: {risk_parity_sharpe:.2f}")
```

Dynamic Allocation Strategies

1. Correlation-Adjusted Allocation

```
class CorrelationAdjustedAllocator:
    def __init__(self, base_allocation, correlation_matrix):
        self.base_allocation = base_allocation  # Dict of {strategy:
weight}
        self.correlation_matrix = correlation_matrix
        self.correlation_threshold = 0.7  # Above this correlation,
reduce allocation

    def calculate_correlation_penalty(self, strategy_i, strategy_j):
        """
        Calculate allocation penalty based on correlation
        """
        correlation = self.correlation_matrix.loc[strategy_i,
strategy_j]
        if correlation > self.correlation_threshold:
            # Reduce allocation for highly correlated strategies
            penalty = (correlation - self.correlation_threshold) / (1 -
self.correlation_threshold)
            return penalty
        return 0

    def adjust_allocation_for_correlation(self):
        """
        Adjust base allocation to reduce correlation risk
        """
        adjusted_allocation = self.base_allocation.copy()
        strategies = list(self.base_allocation.keys())

        # First pass: identify highly correlated pairs
        correlation_penalties = {}
        for i, strategy_i in enumerate(strategies):
            for j, strategy_i in enumerate(strategies):
                if i < j:  # Avoid double counting
                    penalty =
self.calculate_correlation_penalty(strategy_i, strategy_j)
                    if penalty > 0:
                        correlation_penalties[(strategy_i, strategy_j)]
= penalty

        # Second pass: apply penalties
        for (strategy_i, strategy_j), penalty in
correlation_penalties.items():
```

```
            original_weight_i = adjusted_allocation[strategy_i]
            original_weight_j = adjusted_allocation[strategy_j]

            # Reduce both strategies by penalty percentage
            reduction_i = original_weight_i * penalty * 0.5
            reduction_j = original_weight_j * penalty * 0.5

            adjusted_allocation[strategy_i] = original_weight_i -
reduction_i
            adjusted_allocation[strategy_j] = original_weight_j -
reduction_j

        # Normalize weights to sum to 1
        total_weight = sum(adjusted_allocation.values())
        if total_weight > 0:
            adjusted_allocation = {k: v/total_weight for k, v in
adjusted_allocation.items()}

        return adjusted_allocation

    def calculate_portfolio_correlation(self, allocation):
        """
        Calculate effective portfolio correlation
        """
        strategies = list(allocation.keys())
        if len(strategies) < 2:
            return 0

        # Calculate weighted average correlation
        weighted_correlation = 0
        total_weight = sum(allocation.values())

        for i, strategy_i in enumerate(strategies):
            for j, strategy_j in enumerate(strategies):
                if i < j:  # Avoid double counting
                    weight_i = allocation[strategy_i] / total_weight
                    weight_j = allocation[strategy_j] / total_weight
                    correlation =
self.correlation_matrix.loc[strategy_i, strategy_j]
                    weighted_correlation += weight_i * weight_j *
correlation
```

```
        return weighted_correlation * 2
# Multiply by 2 to account for symmetric pairs

# Example usage
base_allocation = {
    'arbitrage': 0.30,
    'liquidation': 0.25,
    'sandwich': 0.25,
    'frontrun': 0.20
}

correlation_adj_allocator =
CorrelationAdjustedAllocator(base_allocation, corr_matrix)

adjusted_allocation =
correlation_adj_allocator.adjust_allocation_for_correlation()
effective_correlation =
correlation_adj_allocator.calculate_portfolio_correlation(adjusted_allocation)

print("\n5. Correlation-Adjusted Allocation:")
print("=" * 40)
print("Strategy\tBase\tAdjusted\tDifference")
for strategy in base_allocation.keys():
    base_weight = base_allocation[strategy]
    adj_weight = adjusted_allocation[strategy]
    diff = adj_weight - base_weight
    print(f"{strategy}\t{base_weight:.1%}\t{adj_weight:.1%}\t{diff:+.
1%}")

print(f"\nEffective Portfolio Correlation: {effective_correlation:.
3f}")
```

2. Regime-Based Allocation

```
class RegimeBasedAllocator:
    def __init__(self, allocation_rules):
        """
        allocation_rules = {
            'bull_market': {strategy: weight},
            'bear_market': {strategy: weight},
            'high_volatility': {strategy: weight},
            'normal_market': {strategy: weight}
        }
        """
        self.allocation_rules = allocation_rules

    def detect_market_regime(self, market_data):
        """
        Detect current market regime based on data
        """
        # Simple regime detection (can be made more sophisticated)
        price_change_1d = market_data.get('price_change_1d', 0)
        volatility_1d = market_data.get('volatility_1d', 0.02)

        # Define thresholds (these should be calibrated)
        if price_change_1d > 0.05:  # 5%+ gain
            return 'bull_market'
        elif price_change_1d < -0.05:  # 5%+ loss
            return 'bear_market'
        elif volatility_1d > 0.05:  # High volatility
            return 'high_volatility'
        else:
            return 'normal_market'

    def get_allocation_for_regime(self, market_data):
        """
        Get optimal allocation for current market regime
        """
        current_regime = self.detect_market_regime(market_data)

        if current_regime in self.allocation_rules:
            return self.allocation_rules[current_regime]
        else:
            return self.allocation_rules['normal_market']

    def calculate_transition_cost(self, current_allocation,
```

```
target_allocation):
        """
        Calculate cost of transitioning between allocations
        """
        # Assume 0.1% transaction cost per 1% of portfolio turnover
        transaction_cost_rate = 0.001

        turnover = 0
        for strategy in current_allocation.keys():
            if strategy in target_allocation:
                turnover += abs(target_allocation[strategy] -
current_allocation[strategy])

        transition_cost = turnover * transaction_cost_rate
        return transition_cost, turnover

# Example usage
regime_allocation_rules = {
    'bull_market': {
        'arbitrage': 0.20,
        'liquidation': 0.35,
        'sandwich': 0.30,
        'frontrun': 0.15
    },
    'bear_market': {
        'arbitrage': 0.40,
        'liquidation': 0.20,
        'sandwich': 0.15,
        'frontrun': 0.25
    },
    'high_volatility': {
        'arbitrage': 0.50,
        'liquidation': 0.15,
        'sandwich': 0.10,
        'frontrun': 0.25
    },
    'normal_market': {
        'arbitrage': 0.30,
        'liquidation': 0.25,
        'sandwich': 0.25,
        'frontrun': 0.20
    }
```

```
}

regime_allocator = RegimeBasedAllocator(regime_allocation_rules)

# Test regime detection and allocation
test_scenarios = [
    {'price_change_1d': 0.08, 'volatility_1d': 0.03},  # Bull market
    {'price_change_1d': -0.06, 'volatility_1d': 0.04}, # Bear market
    {'price_change_1d': 0.02, 'volatility_1d': 0.08},  # High
volatility
    {'price_change_1d': 0.01, 'volatility_1d': 0.02},  # Normal market
]

current_allocation = regime_allocation_rules['normal_market']

print("\n6. Regime-Based Allocation:")
print("=" * 60)
print("Scenario\t\tRegime\t\t\tArbitrage\tLiquidation\tSandwich\tFrontrun")

for i, scenario in enumerate(test_scenarios):
    regime = regime_allocator.detect_market_regime(scenario)
    allocation = regime_allocator.get_allocation_for_regime(scenario)

    transition_cost, turnover =
regime_allocator.calculate_transition_cost(
        current_allocation, allocation
    )

    scenario_name = f"Scenario {i+1}"
    regime_name = regime.replace('_', ' ').title()

    print(f"{scenario_name:<12}\t{regime_name:<12}
\t{allocation['arbitrage']:.1%}\t\t{allocation['liquidation']:.1%}
\t\t{allocation['sandwich']:.1%}\t\t{allocation['frontrun']:.1%}")
    print(f"\t\t\tTransition Cost: {transition_cost:.2%} (Turnover:
{turnover:.1%})")

current_allocation = allocation  # Update for next iteration
```

Rebalancing Strategies

1. Threshold-Based Rebalancing

```
class ThresholdRebalancer:
    def __init__(self, target_allocation, rebalance_threshold=0.05):
        """
        Rebalance when any asset deviates from target by more than
threshold
        """
        self.target_allocation = target_allocation
        self.rebalance_threshold = rebalance_threshold

    def check_rebalance_needed(self, current_allocation):
        """
        Check if rebalancing is needed based on deviation thresholds
        """
        rebalance_needed = False
        deviations = {}

        for strategy in self.target_allocation.keys():
            target_weight = self.target_allocation[strategy]
            current_weight = current_allocation.get(strategy, 0)
            deviation = abs(current_weight - target_weight)

            deviations[strategy] = deviation

            if deviation > self.rebalance_threshold:
                rebalance_needed = True

        return rebalance_needed, deviations

    def calculate_rebalance_trades(self, current_allocation,
portfolio_value):
        """
        Calculate trades needed to rebalance to target allocation
        """
        trades = {}

        for strategy in self.target_allocation.keys():
            target_weight = self.target_allocation[strategy]
            current_weight = current_allocation.get(strategy, 0)

            # Calculate desired dollar amount
            target_dollar = target_weight * portfolio_value
            current_dollar = current_weight * portfolio_value
```

```
            # Trade amount (positive = buy, negative = sell)
            trade_amount = target_dollar - current_dollar
            trades[strategy] = trade_amount

        return trades

    def calculate_rebalance_cost(self, trades,
transaction_cost_rate=0.001):
        """
        Calculate cost of rebalancing
        """
        total_traded = sum(abs(trade) for trade in trades.values())
        rebalance_cost = total_traded * transaction_cost_rate
        return rebalance_cost

# Example usage
target_allocation = {
    'arbitrage': 0.30,
    'liquidation': 0.25,
    'sandwich': 0.25,
    'frontrun': 0.20
}

threshold_rebalancer = ThresholdRebalancer(target_allocation,
rebalance_threshold=0.05)

# Simulate different current allocations
test_allocations = [
    {'arbitrage': 0.35, 'liquidation': 0.25, 'sandwich': 0.25,
'frontrun': 0.15},  # Needs rebalance
    {'arbitrage': 0.29, 'liquidation': 0.26, 'sandwich': 0.25,
'frontrun': 0.20},  # Close enough
    {'arbitrage': 0.20, 'liquidation': 0.35, 'sandwich': 0.30,
'frontrun': 0.15},  # Needs rebalance
]

portfolio_value = 100000

print("\n7. Threshold-Based Rebalancing:")
print("=" * 80)
```

```
for i, current_allocation in enumerate(test_allocations):
    needs_rebalance, deviations =
threshold_rebalancer.check_rebalance_needed(current_allocation)
    trades =
threshold_rebalancer.calculate_rebalance_trades(current_allocation,
portfolio_value)
    rebalance_cost =
threshold_rebalancer.calculate_rebalance_cost(trades)

    print(f"\nScenario {i+1} - Rebalance Needed: {needs_rebalance}")
    print("Strategy\tTarget\tCurrent\tDeviation\tTrade Amount")

    for strategy in target_allocation.keys():
        target = target_allocation[strategy]
        current = current_allocation.get(strategy, 0)
        deviation = deviations[strategy]
        trade = trades.get(strategy, 0)

        status = "NEEDS REBALANCE" if deviation > 0.05 else "OK"

        print(f"{strategy:<10}\t{target:.1%}\t{current:.1%}
\t{deviation:+.1%}\t\t${trade:+,.0f} {status}")

    print(f"Total Rebalance Cost: ${rebalance_cost:,.2f}")
```

2. Time-Based Rebalancing

```
import calendar
from datetime import datetime, timedelta

class TimeBasedRebalancer:
    def __init__(self, target_allocation,
rebalance_frequency='monthly'):
        """
        Rebalance on a fixed schedule
        """
        self.target_allocation = target_allocation
        self.rebalance_frequency = rebalance_frequency
        self.last_rebalance_date = None
        self.rebalance_dates = self._generate_rebalance_schedule()

    def _generate_rebalance_schedule(self):
        """
        Generate schedule of rebalance dates
        """
        schedule = []
        current_date = datetime.now()

        if self.rebalance_frequency == 'weekly':
            # Rebalance every Monday
            days_ahead = 7 - current_date.weekday()
            if days_ahead == 0:  # Today is Monday
                days_ahead = 7
            next_rebalance = current_date + timedelta(days=days_ahead)

            # Generate schedule for next 12 months
            for week in range(52):
                schedule.append(next_rebalance + timedelta(weeks=week))

        elif self.rebalance_frequency == 'monthly':
            # Rebalance on 1st of each month
            if current_date.day == 1:
                next_rebalance = current_date
            else:
                if current_date.month == 12:
                    next_rebalance = datetime(current_date.year + 1, 1,
1)
                else:
                    next_rebalance = datetime(current_date.year,
```

```
current_date.month + 1, 1)

            # Generate schedule for next 12 months
            for month in range(12):
                if current_date.month + month <= 12:
                    schedule.append(datetime(current_date.year,
current_date.month + month, 1))
                else:
                    schedule.append(datetime(current_date.year + 1,
                                             current_date.month + month -
12, 1))

        return schedule

    def is_rebalance_due(self, current_date=None):
        """
        Check if rebalance is due based on schedule
        """
        if current_date is None:
            current_date = datetime.now()

        if not self.rebalance_dates:
            return False

        next_rebalance_date = min(self.rebalance_dates)
        return current_date >= next_rebalance_date

    def get_days_until_rebalance(self, current_date=None):
        """
        Calculate days until next scheduled rebalance
        """
        if current_date is None:
            current_date = datetime.now()

        if not self.rebalance_dates:
            return None

        next_rebalance_date = min(self.rebalance_dates)
        days_until = (next_rebalance_date - current_date).days

        return days_until
```

```
# Example usage
time_rebalancer = TimeBasedRebalancer(target_allocation,
rebalance_frequency='monthly')

print("\n8. Time-Based Rebalancing:")
print("=" * 50)
print(f"Rebalance Frequency: {time_rebalancer.rebalance_frequency}")
print(f"Next Rebalance Date:
{min(time_rebalancer.rebalance_dates).strftime('%Y-%m-%d')}")
print(f"Days Until Next Rebalance:
{time_rebalancer.get_days_until_rebalance()}")

# Test if rebalance is due
current_date = datetime(2023, 2, 1)  # February 1st (should trigger
monthly rebalance)
print(f"\nCurrent Date: {current_date.strftime('%Y-%m-%d')}")
print(f"Rebalance Due:
{time_rebalancer.is_rebalance_due(current_date)}")

current_date = datetime(2023, 2, 15)  # February 15th (should not
trigger)
print(f"Current Date: {current_date.strftime('%Y-%m-%d')}")
print(f"Rebalance Due:
{time_rebalancer.is_rebalance_due(current_date)}")
```

Portfolio Monitoring and Risk Management

1. Correlation Monitoring System

```
class CorrelationMonitor:
    def __init__(self, strategy_returns, alert_thresholds):
        """
        Monitor correlation levels and generate alerts
        alert_thresholds = {
            'high_correlation': 0.8,     # Alert when correlation
exceeds this
            'correlation_spike': 0.3,    # Alert when correlation
increases by this amount
            'low_diversification': 0.6   # Alert when effective
diversification falls below this
        }
        """
        self.strategy_returns = strategy_returns
        self.alert_thresholds = alert_thresholds
        self.correlation_history = []
        self.alerts = []

    def calculate_current_correlations(self):
        """
        Calculate current correlation matrix
        """
        return self.strategy_returns.corr()

    def calculate_effective_correlation(self, correlation_matrix):
        """
        Calculate effective correlation (average off-diagonal
correlation)
        """
        n = len(correlation_matrix)
        correlations = []

        for i in range(n):
            for j in range(i + 1, n):  # Only upper triangle
                correlations.append(correlation_matrix.iloc[i, j])

        return np.mean(correlations) if correlations else 0

    def calculate_correlation_spike(self, current_correlations,
previous_correlations):
        """
        Calculate the largest increase in correlation
```

```
        """
        if previous_correlations is None:
            return 0

        max_increase = 0
        strategies = current_correlations.columns

        for i, strategy1 in enumerate(strategies):
            for j, strategy2 in enumerate(strategies):
                if i < j:
                    current_corr = current_correlations.loc[strategy1,
strategy2]
                    previous_corr =
previous_correlations.loc[strategy1, strategy2]
                    increase = current_corr - previous_corr
                    max_increase = max(max_increase, increase)

        return max_increase

    def check_correlation_alerts(self):
        """
        Check for correlation-based alerts
        """
        current_correlations = self.calculate_current_correlations()

        # Store history
        self.correlation_history.append(current_correlations.copy())
        if len(self.correlation_history) > 30:  # Keep last 30 days
            self.correlation_history.pop(0)

        # Calculate current metrics
        effective_correlation =
self.calculate_effective_correlation(current_correlations)

        # Check against thresholds
        new_alerts = []

        # High correlation alert
        if effective_correlation >
self.alert_thresholds['high_correlation']:
            new_alerts.append({
                'timestamp': datetime.now(),
```

```
                'type': 'HIGH_CORRELATION',
                'message': f"Effective correlation
{effective_correlation:.3f} exceeds threshold
{self.alert_thresholds['high_correlation']:.3f}",
                'severity': 'HIGH'
            })

        # Correlation spike alert
        if len(self.correlation_history) >= 2:
            previous_correlations = self.correlation_history[-2]
            correlation_spike =
self.calculate_correlation_spike(current_correlations,
previous_correlations)

            if correlation_spike >
self.alert_thresholds['correlation_spike']:
                new_alerts.append({
                    'timestamp': datetime.now(),
                    'type': 'CORRELATION_SPIKE',
                    'message': f"Correlation spike of
{correlation_spike:.3f} detected",
                    'severity': 'MEDIUM'
                })

        # Low diversification alert
        if effective_correlation >
self.alert_thresholds['low_diversification']:
            new_alerts.append({
                'timestamp': datetime.now(),
                'type': 'LOW_DIVERSIFICATION',
                'message': f"Effective correlation
{effective_correlation:.3f} suggests low diversification",
                'severity': 'MEDIUM'
            })

        self.alerts.extend(new_alerts)
        return new_alerts

    def get_correlation_summary(self):
        """
        Get summary of current correlation status
        """
```

```
        current_correlations = self.calculate_current_correlations()
        effective_correlation =
self.calculate_effective_correlation(current_correlations)

        # Find highest correlations
        high_correlations = []
        strategies = current_correlations.columns

        for i, strategy1 in enumerate(strategies):
            for j, strategy2 in enumerate(strategies):
                if i < j:
                    correlation = current_correlations.loc[strategy1,
strategy2]
                    high_correlations.append((strategy1, strategy2,
correlation))

        high_correlations.sort(key=lambda x: x[2], reverse=True)

        return {
            'effective_correlation': effective_correlation,
            'highest_correlations': high_correlations[:3],  # Top 3
            'correlation_matrix': current_correlations,
            'total_alerts': len([a for a in self.alerts if
a['severity'] in ['HIGH', 'MEDIUM']])
        }

# Example usage
alert_thresholds = {
    'high_correlation': 0.8,
    'correlation_spike': 0.3,
    'low_diversification': 0.6
}

correlation_monitor = CorrelationMonitor(strategy_returns,
alert_thresholds)

# Simulate monitoring over time
for day in range(5):
    # Simulate daily correlation changes (this would be real data in
practice)
    alerts = correlation_monitor.check_correlation_alerts()
```

```
    if alerts:
        print(f"\nDay {day + 1} - New Alerts:")
        for alert in alerts:
            print(f"  {alert['type']}: {alert['message']}")

# Get correlation summary
summary = correlation_monitor.get_correlation_summary()
print(f"\nCorrelation Monitoring Summary:")
print(f"Effective Correlation: {summary['effective_correlation']:.3f}")
print(f"Total Active Alerts: {summary['total_alerts']}")

print("\nHighest Correlations:")
for strategy1, strategy2, correlation in
summary['highest_correlations']:
    print(f"  {strategy1} - {strategy2}: {correlation:.3f}")
```

2. Diversification Score Calculator

```
class DiversificationScore:
    def __init__(self, correlation_matrix, weights):
        """
        Calculate various diversification metrics
        """
        self.correlation_matrix = correlation_matrix
        self.weights = weights

    def calculate_herfindahl_index(self):
        """
        Calculate Herfindahl-Hirschman Index for concentration
        """
        hhi = sum(w**2 for w in self.weights.values())
        return hhi  # 0 = perfect diversification, 1 = single asset

    def calculate_effective_number_of_assets(self):
        """
        Calculate effective number of assets (inverse HHI)
        """
        hhi = self.calculate_herfindahl_index()
        return 1 / hhi if hhi > 0 else 1

    def calculate_portfolio_correlation(self):
        """
        Calculate weighted average portfolio correlation
        """
        strategies = list(self.weights.keys())
        weighted_correlation = 0

        for i, strategy1 in enumerate(strategies):
            for j, strategy2 in enumerate(strategies):
                if i < j:
                    weight1 = self.weights[strategy1]
                    weight2 = self.weights[strategy2]
                    correlation =
self.correlation_matrix.loc[strategy1, strategy2]
                    weighted_correlation += weight1 * weight2 *
correlation

        return 2 * weighted_correlation  # Multiply by 2 for symmetric
pairs
```

```
    def calculate_diversification_ratio(self):
        """
        Calculate diversification ratio
        """
        # This is a simplified version - would need individual asset
volatilities for full calculation
        portfolio_correlation = self.calculate_portfolio_correlation()

        # Simple diversification score based on correlation

# Perfect diversification = 0 correlation, no diversification = 1
correlation
        diversification_score = 1 - portfolio_correlation

        return max(0, min(1, diversification_score))
# Clamp between 0 and 1

    def get_diversification_metrics(self):
        """
        Get comprehensive diversification metrics
        """
        return {
            'herfindahl_index': self.calculate_herfindahl_index(),
            'effective_assets':
self.calculate_effective_number_of_assets(),
            'portfolio_correlation':
self.calculate_portfolio_correlation(),
            'diversification_ratio':
self.calculate_diversification_ratio(),
            'concentration_risk': max(self.weights.values()) if
self.weights else 0
        }

# Example usage - test different allocations
test_allocations = [
    {'arbitrage': 0.40, 'liquidation': 0.30, 'sandwich': 0.20,
'frontrun': 0.10},  # Concentrated
    {'arbitrage': 0.25, 'liquidation': 0.25, 'sandwich': 0.25,
'frontrun': 0.25},  # Equal weights
    {'arbitrage': 0.35, 'liquidation': 0.35, 'sandwich': 0.15,
'frontrun': 0.15},  # Some concentration
]
```

```
print("\n9. Diversification Analysis:")
print("=" * 80)
print("Allocation\t\t\tHHI\tEff Assets\tPort Corr\tDiv
Ratio\tConcentration")

for allocation in test_allocations:
    div_calculator = DiversificationScore(corr_matrix, allocation)
    metrics = div_calculator.get_diversification_metrics()

    allocation_name = f"{allocation['arbitrage']:.1%}/
{allocation['liquidation']:.1%}/{allocation['sandwich']:.1%}/
{allocation['frontrun']:.1%}"

    print(f"{allocation_name:<20}\t{metrics['herfindahl_index']:.3f}
\t{metrics['effective_assets']:.1f}
\t\t{metrics['portfolio_correlation']:.3f}
\t\t{metrics['diversification_ratio']:.3f}
\t\t{metrics['concentration_risk']:.1%}")
```

Key Takeaways

1. **Correlation analysis is crucial** - High correlations can eliminate diversification benefits.
2. **Multiple correlation measures provide better insight** - Use distance correlation, tail correlation, and drawdown correlation.
3. **Optimization should consider practical constraints** - Real portfolios need limits on individual allocations and turnover.
4. **Dynamic allocation improves robustness** - Regime-based and correlation-adjusted allocations perform better in changing markets.
5. **Rebalancing strategy affects performance** - Threshold-based rebalancing often outperforms fixed schedules.
6. **Diversification is not just about correlation** - Consider factor exposures, time diversification, and risk concentration.
7. **Monitoring correlation changes is essential** - Markets evolve, and correlations can change quickly.
8. **Risk parity provides good balance** - Equal risk contribution often outperforms equal weight allocations.

Next Steps

In the next module, **Drawdown Control**, you'll learn to:

- Analyze drawdown patterns and causes
- Build drawdown recovery systems
- Implement capital preservation strategies
- Create drawdown early warning systems
- Design crisis management procedures

This will complete your core risk management education before moving to advanced automated systems.

Summary

Strategy diversification is the cornerstone of building robust MEV portfolios. By understanding correlation patterns, implementing smart allocation strategies, and maintaining disciplined rebalancing, you can significantly reduce portfolio risk while maintaining strong returns.

Remember that diversification is not a one-time exercise but an ongoing process. Market conditions change, correlations evolve, and new opportunities emerge. The most successful MEV portfolios are those that adapt their diversification strategies to changing market conditions while maintaining disciplined risk management principles.

The key is to start with a solid theoretical foundation, implement practical monitoring systems, and continuously refine your approach based on real-world performance data.