Drawdown Control

Duration: 70 minutes
Level: Intermediate
Author: Obelisk Core

Learning Objectives

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

- Analyze drawdown patterns and identify root causes
- Build automated drawdown detection and response systems
- Implement capital preservation strategies during drawdowns
- Create recovery plans and back-to-profitability frameworks
- Design crisis management procedures for severe market stress

---

Introduction to Drawdown Control

Drawdowns are inevitable in trading, but their impact on long-term performance depends heavily on how they're managed. In MEV trading, where strategies can experience rapid drawdowns due to market volatility, network issues, or increased competition, effective drawdown control is crucial for capital preservation.

Understanding Drawdowns in MEV Context

MEV strategies face unique drawdown challenges:

- **MEV Competition**: Increased competition can rapidly erode profitability
- **Gas Cost Spikes**: Sudden gas price increases can turn profitable trades into losses
- **Protocol Risks**: Smart contract exploits or governance changes can invalidate strategies
- **Market Regime Changes**: Bull-to-bear transitions can affect all MEV strategies simultaneously
- **Infrastructure Failures**: Network outages or delays can prevent profitable execution

Types of Drawdowns

1. **Volatility Drawdowns**: Temporary increases in market volatility
2. **Competition Drawdowns**: Increased MEV competition reducing opportunity profitability
3. **Structural Drawdowns**: Fundamental changes in market structure or protocols

4. **Execution Drawdowns**: Technical issues preventing strategy execution
5. **Liquidity Drawdowns**: Reduced market liquidity affecting trade profitability

---

Drawdown Analysis and Detection

1. Comprehensive Drawdown Calculator

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime, timedelta

class DrawdownAnalyzer:
    def __init__(self, returns_data, initial_capital=100000):
        self.returns_data = returns_data
        self.initial_capital = initial_capital
        self.portfolio_values = self.calculate_portfolio_values()

    def calculate_portfolio_values(self):
        """
        Calculate cumulative portfolio values
        """
        cumulative_returns = (1 + self.returns_data).cumprod()
        portfolio_values = self.initial_capital * cumulative_returns
        return portfolio_values

    def calculate_drawdowns(self):
        """
        Calculate drawdown series
        """
        running_max = self.portfolio_values.expanding().max()
        drawdowns = (self.portfolio_values - running_max) / running_max
        return drawdowns

    def identify_drawdown_periods(self, min_drawdown_threshold=-0.01):
        """
        Identify individual drawdown periods
        """
        drawdowns = self.calculate_drawdowns()

        # Find start and end of drawdowns
        in_drawdown = drawdowns < min_drawdown_threshold

        periods = []
        start_idx = None

        for i, (is_dd, dd_value) in enumerate(zip(in_drawdown,
drawdowns)):
            if is_dd and start_idx is None:
```

```
                start_idx = i
            elif not is_dd and start_idx is not None:
                # Drawdown ended
                end_idx = i - 1

                period_drawdown = drawdowns.iloc[start_idx:end_idx+1]
                max_dd = period_drawdown.min()
                max_dd_idx = period_drawdown.idxmin()

                period = {
                    'start_date': drawdowns.index[start_idx],
                    'end_date': drawdowns.index[end_idx],
                    'duration_days': (drawdowns.index[end_idx] -
drawdowns.index[start_idx]).days,
                    'max_drawdown': max_dd,
                    'max_drawdown_date': max_dd_idx,
                    'recovery_date': None,
                    'recovery_days': None,
                    'total_return':
(self.portfolio_values.iloc[end_idx] /
self.portfolio_values.iloc[start_idx] - 1)
                }

                # Find recovery date
                peak_value = self.portfolio_values.loc[max_dd_idx]
                recovery_series =
self.portfolio_values.loc[max_dd_idx:]
                recovery_mask = recovery_series >= peak_value

                if recovery_mask.any():
                    recovery_idx = recovery_mask.idxmax()
                    period['recovery_date'] = recovery_idx
                    period['recovery_days'] = (recovery_idx -
max_dd_idx).days

                periods.append(period)
                start_idx = None

        # Handle ongoing drawdown
        if start_idx is not None:
            current_idx = len(drawdowns) - 1
            period_drawdown = drawdowns.iloc[start_idx:current_idx+1]
```

```
            max_dd = period_drawdown.min()
            max_dd_idx = period_drawdown.idxmin()

            period = {
                'start_date': drawdowns.index[start_idx],
                'end_date': None,  # Ongoing
                'duration_days': (drawdowns.index[current_idx] -
drawdowns.index[start_idx]).days,
                'max_drawdown': max_dd,
                'max_drawdown_date': max_dd_idx,
                'recovery_date': None,
                'recovery_days': None,
                'total_return':
(self.portfolio_values.iloc[current_idx] /
self.portfolio_values.iloc[start_idx] - 1)
            }
            periods.append(period)

        return periods

    def calculate_drawdown_statistics(self):
        """
        Calculate comprehensive drawdown statistics
        """
        drawdowns = self.calculate_drawdowns()
        periods = self.identify_drawdown_periods()

        if not periods:
            return {}

        # Extract metrics from periods
        max_drawdowns = [p['max_drawdown'] for p in periods if
p['max_drawdown'] is not None]
        durations = [p['duration_days'] for p in periods if
p['duration_days'] is not None]
        recovery_days = [p['recovery_days'] for p in periods if
p['recovery_days'] is not None]
        total_returns = [p['total_return'] for p in periods if
p['total_return'] is not None]

        # Current drawdown
        current_drawdown = drawdowns.iloc[-1]
```

```
        # Statistics
        stats = {
            'current_drawdown': current_drawdown,
            'max_drawdown': min(max_drawdowns) if max_drawdowns else 0,
            'avg_drawdown': np.mean(max_drawdowns) if max_drawdowns
else 0,
            'drawdown_volatility': np.std(max_drawdowns) if
max_drawdowns else 0,
            'avg_duration_days': np.mean(durations) if durations else
0,
            'max_duration_days': max(durations) if durations else 0,
            'avg_recovery_days': np.mean(recovery_days) if
recovery_days else 0,
            'total_drawdown_periods': len(periods),
            'recovery_rate': len([r for r in recovery_days if r is not
None]) / len(periods) if periods else 0,
            'avg_return_during_dd': np.mean(total_returns) if
total_returns else 0,
            'drawdown_frequency': len(periods) / len(drawdowns) * 252
if len(drawdowns) > 252 else 0  # Annualized
        }

        return stats, periods

    def analyze_drawdown_causes(self):
        """
        Analyze potential causes of drawdowns
        """
        drawdowns = self.calculate_drawdowns()

        # Calculate rolling volatility
        rolling_vol = self.returns_data.rolling(30).std() *
np.sqrt(252)

        # Identify high volatility periods
        vol_threshold = rolling_vol.quantile(0.8)
        high_vol_periods = rolling_vol > vol_threshold

        # Identify drawdown periods
        drawdown_periods = drawdowns < -0.05  # 5%+ drawdowns
```

```
        # Calculate correlation between drawdowns and high volatility
        correlation =
drawdown_periods.astype(int).corr(high_vol_periods.astype(int))

        return {
            'volatility_correlation': correlation,
            'high_vol_drawdown_rate': (drawdown_periods &
high_vol_periods).mean() / drawdown_periods.mean() if
drawdown_periods.any() else 0,
            'avg_vol_during_dd': rolling_vol[drawdown_periods].mean()
if drawdown_periods.any() else 0,
            'avg_vol_normal': rolling_vol[~drawdown_periods].mean() if
(1-drawdown_periods.astype(int)).any() else 0
        }

# Example usage with MEV strategy simulation
np.random.seed(42)

# Simulate 2 years of daily returns with various drawdown patterns
days = 504  # 2 years
dates = pd.date_range('2022-01-01', periods=days)

# Create returns with different market regimes
returns = []
base_vol = 0.015

for i in range(days):
    if i < 100:  # Bull market
        daily_return = np.random.normal(0.0015, base_vol)
    elif i < 200:  # High volatility period
        daily_return = np.random.normal(-0.0005, base_vol * 1.5)
    elif i < 300:  # Recovery
        daily_return = np.random.normal(0.001, base_vol * 0.8)
    elif i < 400:  # MEV competition increase
        daily_return = np.random.normal(-0.0002, base_vol * 1.2)
    else:  # Market stress
        daily_return = np.random.normal(-0.001, base_vol * 2)

    returns.append(daily_return)

mev_returns = pd.Series(returns, index=dates)
```

```
# Analyze drawdowns
analyzer = DrawdownAnalyzer(mev_returns, initial_capital=100000)

# Calculate statistics
stats, periods = analyzer.calculate_drawdown_statistics()
cause_analysis = analyzer.analyze_drawdown_causes()

print("Drawdown Analysis Results:")
print("=" * 50)
print(f"Current Drawdown: {stats['current_drawdown']:.2%}")
print(f"Maximum Drawdown: {stats['max_drawdown']:.2%}")
print(f"Average Drawdown: {stats['avg_drawdown']:.2%}")
print(f"Average Duration: {stats['avg_duration_days']:.1f} days")
print(f"Maximum Duration: {stats['max_duration_days']} days")
print(f"Recovery Rate: {stats['recovery_rate']:.1%}")
print(f"Drawdown Frequency: {stats['drawdown_frequency']:.1f} per
year")

print("\nDrawdown Periods:")
print("-" * 70)
print("Start Date\t\tEnd Date\t\tDuration\tMax DD\t\tRecovery")
print("-" * 70)

for period in periods[:5]:  # Show first 5 periods
    start = period['start_date'].strftime('%Y-%m-%d')
    end = period['end_date'].strftime('%Y-%m-%d') if period['end_date']
else 'Ongoing'
    duration = f"{period['duration_days']} days"
    max_dd = f"{period['max_drawdown']:.2%}"
    recovery = f"{period['recovery_days']} days" if
period['recovery_days'] else 'N/A'

    print(f"{start}\t{end}\t{duration:<12}\t{max_dd}\t{recovery}")

print("\nDrawdown Cause Analysis:")
print("-" * 30)
print(f"Volatility Correlation:
{cause_analysis['volatility_correlation']:.3f}")
print(f"High Vol Drawdown Rate:
{cause_analysis['high_vol_drawdown_rate']:.2%}")
print(f"Avg Vol During DD: {cause_analysis['avg_vol_during_dd']:.1%}")
print(f"Avg Vol Normal: {cause_analysis['avg_vol_normal']:.1%}")
```

2. Real-Time Drawdown Monitoring

```
class RealTimeDrawdownMonitor:
    def __init__(self, initial_capital, max_drawdown_limit=0.15):
        self.initial_capital = initial_capital
        self.max_drawdown_limit = max_drawdown_limit
        self.peak_value = initial_capital
        self.current_value = initial_capital
        self.drawdown_history = []
        self.alerts = []

    def update_portfolio_value(self, new_value, timestamp=None):
        """
        Update portfolio value and check for drawdown
        """
        if timestamp is None:
            timestamp = datetime.now()

        self.current_value = new_value

        # Update peak value
        if new_value > self.peak_value:
            self.peak_value = new_value

        # Calculate current drawdown
        current_drawdown = (self.peak_value - new_value) /
self.peak_value

        # Record drawdown
        drawdown_record = {
            'timestamp': timestamp,
            'portfolio_value': new_value,
            'peak_value': self.peak_value,
            'drawdown': current_drawdown,
            'drawdown_amount': self.peak_value - new_value
        }

        self.drawdown_history.append(drawdown_record)

        # Check for alerts
        self._check_drawdown_alerts(current_drawdown, timestamp)

        return current_drawdown
```

```
    def _check_drawdown_alerts(self, current_drawdown, timestamp):
        """
        Check for various drawdown alert levels
        """
        # Alert levels
        warning_level = 0.05    # 5%
        danger_level = 0.10     # 10%
        critical_level = 0.15   # 15%
        emergency_level = 0.20  # 20%

        alerts_to_add = []

        if current_drawdown >= emergency_level:
            alerts_to_add.append({
                'timestamp': timestamp,
                'level': 'EMERGENCY',
                'message': f"Emergency drawdown level reached:
{current_drawdown:.1%}",
                'action': 'IMMEDIATE_EXIT_REQUIRED'
            })
        elif current_drawdown >= critical_level:
            alerts_to_add.append({
                'timestamp': timestamp,
                'level': 'CRITICAL',
                'message': f"Critical drawdown level reached:
{current_drawdown:.1%}",
                'action': 'HALT_NEW_TRADES_REVIEW_REQUIRED'
            })
        elif current_drawdown >= danger_level:
            alerts_to_add.append({
                'timestamp': timestamp,
                'level': 'DANGER',
                'message': f"Dangerous drawdown level reached:
{current_drawdown:.1%}",
                'action': 'REDUCE_POSITION_SIZES'
            })
        elif current_drawdown >= warning_level:
            alerts_to_add.append({
                'timestamp': timestamp,
                'level': 'WARNING',
                'message': f"Drawdown warning level reached:
{current_drawdown:.1%}",
```

```
                'action': 'MONITOR_CLOSELY'
            })

        # Check drawdown velocity (rate of change)
        if len(self.drawdown_history) >= 2:
            recent_drawdowns = [dd['drawdown'] for dd in
self.drawdown_history[-5:]]
            if len(recent_drawdowns) >= 2:
                velocity = recent_drawdowns[-1] - recent_drawdowns[-2]
                if velocity < -0.02:  # Drawdown increasing rapidly
(>2% in one period)
                    alerts_to_add.append({
                        'timestamp': timestamp,
                        'level': 'RAPID_DRAWDOWN',
                        'message': f"Rapid drawdown detected:
{velocity:.2%} in one period",
                        'action': 'IMMEDIATE_REVIEW_REQUIRED'
                    })

        self.alerts.extend(alerts_to_add)
        return alerts_to_add

    def get_current_status(self):
        """
        Get current drawdown status and statistics
        """
        if not self.drawdown_history:
            return {
                'current_drawdown': 0,
                'drawdown_status': 'HEALTHY',
                'days_in_drawdown': 0,
                'active_alerts': 0
            }

        current_record = self.drawdown_history[-1]

        # Determine status
        current_dd = current_record['drawdown']
        if current_dd >= 0.20:
            status = 'EMERGENCY'
        elif current_dd >= 0.15:
            status = 'CRITICAL'
```

```
        elif current_dd >= 0.10:
            status = 'DANGER'
        elif current_dd >= 0.05:
            status = 'WARNING'
        else:
            status = 'HEALTHY'

        # Count days in current drawdown
        days_in_drawdown = 0
        peak_date = current_record['peak_value']

        for record in reversed(self.drawdown_history):
            if record['drawdown'] > 0:
                days_in_drawdown += 1
            else:
                break

        # Count active alerts (last 24 hours)
        cutoff_time = datetime.now() - timedelta(hours=24)
        active_alerts = len([a for a in self.alerts if a['timestamp'] >
cutoff_time])

        return {
            'current_drawdown': current_dd,
            'drawdown_status': status,
            'days_in_drawdown': days_in_drawdown,
            'active_alerts': active_alerts,
            'peak_value': self.peak_value,
            'current_value': self.current_value,
            'drawdown_amount': current_record['drawdown_amount']
        }

    def generate_drawdown_report(self):
        """
        Generate comprehensive drawdown report
        """
        if not self.drawdown_history:
            return "No data available for analysis"

        # Calculate statistics
        drawdowns = [record['drawdown'] for record in
self.drawdown_history]
```

```
        max_drawdown = min(drawdowns)
        current_drawdown = drawdowns[-1]

        # Count alert levels
        alert_counts = {}
        for alert in self.alerts:
            level = alert['level']
            alert_counts[level] = alert_counts.get(level, 0) + 1

        # Calculate drawdown frequency
        drawdown_days = len([d for d in drawdowns if d > 0])
        total_days = len(drawdowns)
        drawdown_frequency = drawdown_days / total_days * 100 if
total_days > 0 else 0

        report = f"""
Drawdown Monitoring Report
==========================

Current Status:
- Current Drawdown: {current_drawdown:.2%}
- Maximum Drawdown: {max_drawdown:.2%}
- Peak Portfolio Value: ${self.peak_value:,.2f}
- Current Portfolio Value: ${self.current_value:,.2f}

Alert Summary:
"""

        for level, count in alert_counts.items():
            report += f"- {level}: {count} alerts\n"

        report += f"""
Statistics:
- Total Days Monitored: {total_days}
- Days in Drawdown: {drawdown_days} ({drawdown_frequency:.1f}%)
- Drawdown Limit: {self.max_drawdown_limit:.1%}

Recent Alerts (Last 24 Hours):
"""

        # Recent alerts
        recent_alerts = [a for a in self.alerts if a['timestamp'] >
```

```
datetime.now() - timedelta(hours=24)]
        if recent_alerts:
            for alert in recent_alerts[-5:]:  # Last 5 alerts
                report += f"- {alert['timestamp'].strftime('%Y-%m-%d
%H:%M')}: {alert['message']}\n"
        else:
            report += "No recent alerts\n"

        return report

# Example usage
monitor = RealTimeDrawdownMonitor(initial_capital=100000,
max_drawdown_limit=0.15)

# Simulate portfolio value changes over time
simulated_values = [
    100000,  # Start
    102000,  # Gain
    105000,  # More gain
    103000,  # Small loss
    99000,   # First drawdown
    95000,   # More drawdown (5% - warning)
    90000,   # 10% drawdown (danger)
    85000,   # 15% drawdown (critical)
    80000,   # 20% drawdown (emergency)
]

print("Real-Time Drawdown Monitoring Example:")
print("=" * 50)

for i, value in enumerate(simulated_values):
    alerts = monitor.update_portfolio_value(value)
    status = monitor.get_current_status()

    print(f"Update {i+1}: ${value:,}")
    print(f"  Status: {status['drawdown_status']}")
    print(f"  Current DD: {status['current_drawdown']:.2%}")
    print(f"  Days in DD: {status['days_in_drawdown']}")

    if alerts:
        print(f"  New Alerts: {len(alerts)}")
        for alert in alerts:
```

```
            print(f"    {alert['level']}: {alert['message']}")
    print()

# Generate final report
report = monitor.generate_drawdown_report()
print(report)
```

Drawdown Prevention Strategies

1. Dynamic Position Sizing Based on Drawdown

```
class DrawdownAwarePositionSizer:
    def __init__(self, initial_capital, base_position_size,
max_drawdown_limit=0.15):
        self.initial_capital = initial_capital
        self.base_position_size = base_position_size
        self.max_drawdown_limit = max_drawdown_limit
        self.current_capital = initial_capital
        self.peak_capital = initial_capital

    def calculate_drawdown_multiplier(self, current_drawdown):
        """
        Calculate position size multiplier based on current drawdown
        """
        if current_drawdown <= 0:
            return 1.0  # No reduction for profitable periods

        # Gradual reduction as drawdown increases
        if current_drawdown < 0.05:  # 0-5% drawdown
            multiplier = 1.0 - (current_drawdown / 0.05) * 0.2  #
Reduce up to 20%
        elif current_drawdown < 0.10:  # 5-10% drawdown
            multiplier = 0.8 - ((current_drawdown - 0.05) / 0.05) * 0.3
# Reduce 20% to 50%
        elif current_drawdown < 0.15:  # 10-15% drawdown
            multiplier = 0.5 - ((current_drawdown - 0.10) / 0.05) * 0.4
# Reduce 50% to 90%
        else:  # >15% drawdown
            multiplier = 0.1  # Maximum 90% reduction

        return max(0.1, multiplier)  # Minimum 10% of normal size

    def calculate_position_size(self, opportunity_data,
current_capital=None):
        """
        Calculate position size considering drawdown
        """
        if current_capital is not None:
            self.current_capital = current_capital

        # Calculate current drawdown
        self.peak_capital = max(self.peak_capital,
self.current_capital)
```

```
        current_drawdown = (self.peak_capital - self.current_capital) /
self.peak_capital

        # Calculate base position size
        base_size = self.base_position_size * (self.current_capital /
self.initial_capital)

        # Apply drawdown multiplier
        dd_multiplier =
self.calculate_drawdown_multiplier(current_drawdown)

        # Additional adjustments based on opportunity quality
        opportunity_adjustment =
self._assess_opportunity_quality(opportunity_data)

        final_size = base_size * dd_multiplier * opportunity_adjustment

        return {
            'position_size': final_size,
            'base_size': base_size,
            'drawdown_multiplier': dd_multiplier,
            'opportunity_adjustment': opportunity_adjustment,
            'current_drawdown': current_drawdown,
            'capital_ratio': self.current_capital /
self.initial_capital
        }

    def _assess_opportunity_quality(self, opportunity_data):
        """
        Adjust position size based on opportunity quality
        """
        quality_score = 1.0

        # Factor in win rate
        if 'win_rate' in opportunity_data:
            quality_score *= min(1.2, opportunity_data['win_rate'] *
1.5)

        # Factor in expected profit
        if 'expected_profit' in opportunity_data:
            profit_multiplier = min(1.5,
opportunity_data['expected_profit'] / 100)
```

```
            quality_score *= profit_multiplier

        # Factor in time sensitivity (higher sensitivity = lower size
due to risk)
        if 'time_sensitivity' in opportunity_data:
            quality_score *= (2 - opportunity_data['time_sensitivity'])
# Inverse relationship

        return quality_score

# Example usage
dd_position_sizer = DrawdownAwarePositionSizer(
    initial_capital=100000,
    base_position_size=10000,
    max_drawdown_limit=0.15
)

# Test different drawdown levels
test_scenarios = [
    {'capital': 100000, 'win_rate': 0.7, 'expected_profit': 150,
'time_sensitivity': 1.0},  # No drawdown
    {'capital': 95000, 'win_rate': 0.7, 'expected_profit': 150,
'time_sensitivity': 1.0},   # 5% drawdown
    {'capital': 85000, 'win_rate': 0.7, 'expected_profit': 150,
'time_sensitivity': 1.0},   # 15% drawdown
    {'capital': 75000, 'win_rate': 0.7, 'expected_profit': 150,
'time_sensitivity': 1.0},   # 25% drawdown (severe)
]

print("Drawdown-Aware Position Sizing:")
print("=" * 60)
print("Capital\t\tDrawdown\tBase Size\tFinal Size\tMultiplier")
print("-" * 60)

for scenario in test_scenarios:
    result = dd_position_sizer.calculate_position_size(scenario)

    print(f"<span class="math-inline" style="display: inline;"><math
xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>s</mi><mi>c</mi><mi>e</mi><mi>n</
mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>o</mi><msup><mo
stretchy="false">[</mo><mi>&#x02032;</mi></msup><mi>c</mi><mi>a</
```

```
mi><mi>p</mi><mi>i</mi><mi>t</mi><mi>a</mi><msup><mi>l</
mi><mi>&#x02032;</mi></msup><mo stretchy="false">]</mo><mi>:</
mi><mo>&#x0002C;</mo></mrow><mi>\t</mi><mi>\t</mi><mrow><mi>r</
mi><mi>e</mi><mi>s</mi><mi>u</mi><mi>l</mi><mi>t</mi><msup><mo
stretchy="false">[</mo><mi>&#x02032;</mi></msup><mi>c</mi><mi>u</
mi><mi>r</mi><mi>r</mi><mi>e</mi><mi>n</mi><msub><mi>t</mi><mi>d</mi></
msub><mi>r</mi><mi>a</mi><mi>w</mi><mi>d</mi><mi>o</mi><mi>w</
mi><msup><mi>n</mi><mi>&#x02032;</mi></msup><mo stretchy="false">]</
mo><mi>:</mi><mi>.1</mi></mrow></mrow></math></
span>{result['base_size']:,.0f}\t\t${result['position_size']:,.0f}
\t\t{result['drawdown_multiplier']:.2f}")
```

2. Correlation-Based Position Reduction

```
class CorrelationAwareRiskManager:
    def __init__(self, portfolio_strategies, correlation_thresholds):
        """
        Reduce position sizes when portfolio correlation increases
        correlation_thresholds = {
            'high_correlation': 0.8,
            'correlation_increase': 0.2
        }
        """
        self.portfolio_strategies = portfolio_strategies
        self.correlation_thresholds = correlation_thresholds
        self.correlation_history = []

    def calculate_portfolio_correlation(self, returns_data):
        """
        Calculate current portfolio correlation
        """
        if len(returns_data.columns) < 2:
            return 0

        correlation_matrix = returns_data.corr()

        # Calculate average off-diagonal correlation
        correlations = []
        for i in range(len(correlation_matrix)):
            for j in range(i + 1, len(correlation_matrix)):
                correlations.append(correlation_matrix.iloc[i, j])

        return np.mean(correlations) if correlations else 0

    def calculate_correlation_multiplier(self, current_correlation,
previous_correlation=None):
        """
        Calculate position size multiplier based on correlation
        """
        # Base reduction for high correlation
        correlation_multiplier = 1.0

        if current_correlation >
self.correlation_thresholds['high_correlation']:
            excess_correlation = current_correlation -
self.correlation_thresholds['high_correlation']
```

```
            correlation_multiplier -= excess_correlation * 0.5  #
Reduce up to 50%

        # Additional reduction for correlation spikes
        if previous_correlation is not None:
            correlation_increase = current_correlation -
previous_correlation

            if correlation_increase >
self.correlation_thresholds['correlation_increase']:
                spike_reduction = (correlation_increase -
self.correlation_thresholds['correlation_increase']) * 0.3
                correlation_multiplier -= spike_reduction

        return max(0.5, correlation_multiplier)  # Minimum 50% of
normal size

    def adjust_position_sizes(self, base_positions, returns_data):
        """
        Adjust all position sizes based on correlation
        """
        current_correlation =
self.calculate_portfolio_correlation(returns_data)

        # Store correlation history
        self.correlation_history.append(current_correlation)
        if len(self.correlation_history) > 30:
            self.correlation_history.pop(0)

        # Get previous correlation
        previous_correlation = self.correlation_history[-2] if
len(self.correlation_history) > 1 else None

        # Calculate correlation multiplier
        correlation_multiplier =
self.calculate_correlation_multiplier(current_correlation,
previous_correlation)

        # Adjust each position
        adjusted_positions = {}
        for strategy, base_size in base_positions.items():
            adjusted_size = base_size * correlation_multiplier
```

```
            adjusted_positions[strategy] = {
                'base_size': base_size,
                'adjusted_size': adjusted_size,
                'reduction': (base_size - adjusted_size) / base_size
            }

        return {
            'adjusted_positions': adjusted_positions,
            'correlation_multiplier': correlation_multiplier,
            'current_correlation': current_correlation,
            'correlation_change': current_correlation -
previous_correlation if previous_correlation else 0
        }

# Example usage
portfolio_strategies = ['arbitrage', 'liquidation', 'sandwich',
'frontrun']
correlation_manager = CorrelationAwareRiskManager(
    portfolio_strategies,
    {'high_correlation': 0.7, 'correlation_increase': 0.2}
)

# Base position sizes
base_positions = {
    'arbitrage': 30000,
    'liquidation': 25000,
    'sandwich': 25000,
    'frontrun': 20000
}

# Simulate different correlation scenarios
correlation_scenarios = [
    {'correlation': 0.3, 'description': 'Normal correlation'},
    {'correlation': 0.6, 'description': 'Moderate correlation'},
    {'correlation': 0.8, 'description': 'High correlation'},
    {'correlation': 0.9, 'description': 'Very high correlation'},
]

print("\nCorrelation-Aware Risk Management:")
print("=" * 80)
print("Scenario\t\t\tCurrent
Corr\tMultiplier\tArbitrage\tLiquidation\tSandwich\tFrontrun")
```

```
print("-" * 80)

for scenario in correlation_scenarios:
    # Simulate correlation matrix with target correlation
    corr_matrix = np.eye(4)
    np.fill_diagonal(corr_matrix, 1.0)
    corr_value = scenario['correlation']

    # Set all off-diagonal elements to target correlation
    for i in range(4):
        for j in range(4):
            if i != j:
                corr_matrix[i, j] = corr_value

    # Create mock returns data
    mock_returns = pd.DataFrame(
        np.random.multivariate_normal(
            mean=[0.001]*4,
            cov=corr_matrix * 0.000225,  # Approximate 15% annual
volatility
            size=252
        ),
        columns=portfolio_strategies
    )

    # Adjust positions
    result = correlation_manager.adjust_position_sizes(base_positions,
mock_returns)

    scenario_name = scenario['description']
    current_corr = result['current_correlation']
    multiplier = result['correlation_multiplier']

    arbitrage_new = result['adjusted_positions']['arbitrage']
['adjusted_size']
    liquidation_new = result['adjusted_positions']['liquidation']
['adjusted_size']
    sandwich_new = result['adjusted_positions']['sandwich']
['adjusted_size']
    frontrun_new = result['adjusted_positions']['frontrun']
['adjusted_size']
```

```
    print(f"{scenario_name:<20}\t{current_corr:.2f}\t\t{multiplier:.2f}
\t\t<span class="math-inline" style="display: inline;"><math
xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>a</mi><mi>r</mi><mi>b</mi><mi>i</
mi><mi>t</mi><mi>r</mi><mi>a</mi><mi>g</mi><msub><mi>e</mi><mi>n</mi></
msub><mi>e</mi><mi>w</mi><mi>:</mi><mo>&#x0003E;</mo><mn>6</
mn><mo>&#x0002C;</mo><mi>.0</mi><mi>f</mi></mrow><mi>\t</mi><mi>\t</
mi></mrow></math></span>{liquidation_new:>6,.0f}\t\t<span class="math-
inline" style="display: inline;"><math xmlns="http://www.w3.org/1998/
Math/MathML" display="inline"><mrow><mrow><mi>s</mi><mi>a</mi><mi>n</
mi><mi>d</mi><mi>w</mi><mi>i</mi><mi>c</mi><msub><mi>h</mi><mi>n</mi></
msub><mi>e</mi><mi>w</mi><mi>:</mi><mo>&#x0003E;</mo><mn>6</
mn><mo>&#x0002C;</mo><mi>.0</mi><mi>f</mi></mrow><mi>\t</mi><mi>\t</
mi></mrow></math></span>{frontrun_new:>6,.0f}")
```

Recovery Strategies

1. Drawdown Recovery Framework

```
class DrawdownRecoveryFramework:
    def __init__(self, recovery_targets, risk_adjustments):
        """
        recovery_targets = {
            'partial_recovery': 0.5,  # Recover 50% of drawdown
            'full_recovery': 1.0,     # Recover 100% of drawdown
        }
        risk_adjustments = {
            'aggressive': {'position_multiplier': 1.2,
'target_recovery': 0.8},
            'moderate': {'position_multiplier': 1.0, 'target_recovery':
0.5},
            'conservative': {'position_multiplier': 0.8,
'target_recovery': 0.3}
        }
        """
        self.recovery_targets = recovery_targets
        self.risk_adjustments = risk_adjustments
        self.recovery_phases = []

    def calculate_recovery_plan(self, initial_capital, current_capital,
drawdown_amount, risk_profile='moderate'):
        """
        Create a recovery plan based on current drawdown
        """
        drawdown_percentage = drawdown_amount / initial_capital
        current_drawdown_percentage = (initial_capital -
current_capital) / initial_capital

        # Get risk adjustment parameters
        risk_params = self.risk_adjustments[risk_profile]
        position_multiplier = risk_params['position_multiplier']
        target_recovery = risk_params['target_recovery']

        # Calculate recovery targets
        target_recovery_amount = drawdown_amount * target_recovery
        target_capital = current_capital + target_recovery_amount

        # Calculate required return
        required_return = target_recovery_amount / current_capital

        # Adjust position sizing for recovery
```

```
        recovery_position_size = current_capital * 0.05 *
position_multiplier  # Risk 5% per trade

        # Timeline estimates (simplified)
        if risk_profile == 'aggressive':
            timeline_days = max(10, int(required_return * 100))  #
Faster recovery
        elif risk_profile == 'moderate':
            timeline_days = max(20, int(required_return * 150))
        else:  # conservative
            timeline_days = max(30, int(required_return * 200))

        # Create recovery milestones
        milestones = self._create_recovery_milestones(
            current_capital, target_capital, timeline_days,
risk_profile
        )

        recovery_plan = {
            'risk_profile': risk_profile,
            'current_drawdown': current_drawdown_percentage,
            'drawdown_amount': drawdown_amount,
            'target_recovery_amount': target_recovery_amount,
            'target_capital': target_capital,
            'required_return': required_return,
            'timeline_days': timeline_days,
            'position_size': recovery_position_size,
            'position_multiplier': position_multiplier,
            'milestones': milestones,
            'success_probability':
self._estimate_success_probability(risk_profile, required_return)
        }

        return recovery_plan

    def _create_recovery_milestones(self, current_capital,
target_capital, timeline_days, risk_profile):
        """
        Create milestone targets for recovery
        """
        milestone_dates = []
        milestone_targets = []
```

```
        # Create 4 milestone checkpoints
        num_milestones = 4
        milestone_interval = timeline_days // num_milestones

        for i in range(1, num_milestones + 1):
            days_from_now = milestone_interval * i
            target_value = current_capital + ((target_capital -
current_capital) * i / num_milestones)

            milestone_dates.append(days_from_now)
            milestone_targets.append(target_value)

        return list(zip(milestone_dates, milestone_targets))

    def _estimate_success_probability(self, risk_profile,
required_return):
        """
        Estimate probability of successful recovery (simplified model)
        """
        # Base probability by risk profile
        base_probabilities = {
            'aggressive': 0.6,
            'moderate': 0.75,
            'conservative': 0.85
        }

        base_prob = base_probabilities[risk_profile]

        # Adjust based on required return
        if required_return > 0.5:  # >50% required return
            base_prob *= 0.7
        elif required_return > 0.3:  # >30% required return
            base_prob *= 0.85
        elif required_return > 0.1:  # >10% required return
            base_prob *= 0.95

        return min(0.95, base_prob)

    def track_recovery_progress(self, recovery_plan, current_capital,
days_elapsed):
        """
```

```
        Track progress towards recovery goals
        """
        initial_target = recovery_plan['target_capital']
        progress_percentage = (current_capital - (initial_target -
recovery_plan['target_recovery_amount'])) /
recovery_plan['target_recovery_amount']

        # Check milestone progress
        milestones_hit = 0
        next_milestone = None

        for milestone_days, milestone_target in
recovery_plan['milestones']:
            if days_elapsed >= milestone_days and current_capital >=
milestone_target:
                milestones_hit += 1
            elif next_milestone is None and days_elapsed <
milestone_days:
                next_milestone = (milestone_days, milestone_target)

        # Calculate projected timeline
        if progress_percentage > 0 and days_elapsed > 0:
            projected_total_days = days_elapsed / progress_percentage
            days_behind_schedule = projected_total_days -
recovery_plan['timeline_days']
        else:
            projected_total_days = None
            days_behind_schedule = None

        return {
            'progress_percentage': progress_percentage,
            'milestones_hit': milestones_hit,
            'total_milestones': len(recovery_plan['milestones']),
            'next_milestone': next_milestone,
            'days_elapsed': days_elapsed,
            'projected_total_days': projected_total_days,
            'days_behind_schedule': days_behind_schedule,
            'recovery_status':
self._determine_recovery_status(progress_percentage, days_elapsed,
recovery_plan['timeline_days'])
        }
```

```
    def _determine_recovery_status(self, progress_percentage,
days_elapsed, target_days):
        """
        Determine current recovery status
        """
        if progress_percentage >= 1.0:
            return 'RECOVERED'
        elif progress_percentage >= 0.8:
            return 'ON_TRACK'
        elif progress_percentage >= 0.5:
            return 'SLOW_PROGRESS'
        elif days_elapsed > target_days * 1.5:
            return 'BEHIND_SCHEDULE'
        else:
            return 'IN_PROGRESS'

# Example usage
recovery_framework = DrawdownRecoveryFramework(
    recovery_targets={'partial_recovery': 0.5, 'full_recovery': 1.0},
    risk_adjustments={
        'aggressive': {'position_multiplier': 1.2, 'target_recovery':
0.8},
        'moderate': {'position_multiplier': 1.0, 'target_recovery':
0.5},
        'conservative': {'position_multiplier': 0.8, 'target_recovery':
0.3}
    }
)

# Test different drawdown scenarios
initial_capital = 100000
current_capital = 85000  # 15% drawdown
drawdown_amount = initial_capital - current_capital

print("Drawdown Recovery Framework:")
print("=" * 70)

for risk_profile in ['conservative', 'moderate', 'aggressive']:
    recovery_plan = recovery_framework.calculate_recovery_plan(
        initial_capital, current_capital, drawdown_amount, risk_profile
    )
```

```
    print(f"\n{risk_profile.upper()} Recovery Plan:")
    print(f"  Current Drawdown: {recovery_plan['current_drawdown']:.
1%}")
    print(f"  Target Recovery:
{recovery_plan['target_recovery_amount']:,.0f}
({recovery_plan['required_return']:.1%} return needed)")
    print(f"  Timeline: {recovery_plan['timeline_days']} days")
    print(f"  Position Size: ${recovery_plan['position_size']:,.0f}")
    print(f"  Success Probability:
{recovery_plan['success_probability']:.1%}")

    print(f"  Milestones:")
    for days, target in recovery_plan['milestones']:
        print(f"    Day {days}: ${target:,.0f}")

# Simulate recovery progress
print(f"\nRecovery Progress Tracking:")
print("-" * 40)

# Test progress at different time points
test_days = [15, 30, 45, 60]
current_capital_progress = 86500  # Some recovery

for days in test_days:
    progress = recovery_framework.track_recovery_progress(
        recovery_plan, current_capital_progress, days
    )

    print(f"Day {days}: {progress['recovery_status']} -
{progress['progress_percentage']:.1%} complete")
    if progress['days_behind_schedule']:
        print(f"  {progress['days_behind_schedule']:.0f} days behind
schedule")
```

2. Capital Preservation Strategies

```
class CapitalPreservationSystem:
    def __init__(self, preservation_thresholds):
        """
        preservation_thresholds = {
            'reduce_risk': 0.05,    # 5% drawdown
            'minimum_risk': 0.10,   # 10% drawdown
            'survival_mode': 0.15,  # 15% drawdown
            'emergency_exit': 0.20  # 20% drawdown
        }
        """
        self.preservation_thresholds = preservation_thresholds
        self.preservation_mode = 'NORMAL'

    def determine_preservation_mode(self, current_drawdown):
        """
        Determine appropriate preservation mode based on drawdown
        """
        if current_drawdown >=
self.preservation_thresholds['emergency_exit']:
            return 'EMERGENCY_EXIT'
        elif current_drawdown >=
self.preservation_thresholds['survival_mode']:
            return 'SURVIVAL_MODE'
        elif current_drawdown >=
self.preservation_thresholds['minimum_risk']:
            return 'MINIMUM_RISK'
        elif current_drawdown >=
self.preservation_thresholds['reduce_risk']:
            return 'REDUCE_RISK'
        else:
            return 'NORMAL'

    def get_preservation_actions(self, preservation_mode):
        """
        Get specific actions for each preservation mode
        """
        actions = {
            'NORMAL': {
                'position_size_multiplier': 1.0,
                'allowed_strategies': 'ALL',
                'max_position_risk': 0.02,
                'new_trades': 'ALLOWED',
```

```
                'review_frequency': 'WEEKLY'
            },
            'REDUCE_RISK': {
                'position_size_multiplier': 0.75,
                'allowed_strategies': 'HIGH_CONVICTION_ONLY',
                'max_position_risk': 0.015,
                'new_trades': 'RESTRICTED',
                'review_frequency': 'DAILY'
            },
            'MINIMUM_RISK': {
                'position_size_multiplier': 0.5,
                'allowed_strategies': 'SAFEST_ONLY',
                'max_position_risk': 0.01,
                'new_trades': 'APPROVAL_REQUIRED',
                'review_frequency': 'DAILY'
            },
            'SURVIVAL_MODE': {
                'position_size_multiplier': 0.25,
                'allowed_strategies': 'ARBITRAGE_ONLY',
                'max_position_risk': 0.005,
                'new_trades': 'FORBIDDEN',
                'review_frequency': 'CONTINUOUS'
            },
            'EMERGENCY_EXIT': {
                'position_size_multiplier': 0.0,
                'allowed_strategies': 'NONE',
                'max_position_risk': 0.0,
                'new_trades': 'FORBIDDEN',
                'review_frequency': 'IMMEDIATE'
            }
        }

        return actions.get(preservation_mode, actions['NORMAL'])

    def calculate_emergency_exit_plan(self, current_positions,
current_drawdown):
        """
        Calculate emergency exit plan for critical drawdowns
        """
        if current_drawdown <
self.preservation_thresholds['emergency_exit']:
            return None
```

```
        exit_plan = {
            'exit_order': [],
            'estimated_time': '2-6 hours',
            'estimated_cost': 0,
            'risk_reduction': 0
        }

        # Prioritize exits by risk level and liquidity
        exit_priorities = [
            ('high_risk_low_liquidity', 'IMMEDIATE'),
            ('high_risk_high_liquidity', 'URGENT'),
            ('low_risk_low_liquidity', 'SCHEDULED'),
            ('low_risk_high_liquidity', 'ROUTINE')
        ]

        total_position_value = sum(pos['value'] for pos in
current_positions.values())

        for priority, urgency in exit_priorities:
            for position_id, position in current_positions.items():
                position_priority =
self._classify_position_priority(position)

                if position_priority == priority:
                    exit_amount = position['value']
                    estimated_cost = exit_amount * 0.002  # 0.2%
estimated exit cost

                    exit_plan['exit_order'].append({
                        'position_id': position_id,
                        'amount': exit_amount,
                        'urgency': urgency,
                        'estimated_cost': estimated_cost,
                        'priority_rank':
exit_priorities.index((priority, urgency))
                    })

                    exit_plan['estimated_cost'] += estimated_cost

        exit_plan['risk_reduction'] = min(0.95, (total_position_value -
exit_plan['estimated_cost']) / total_position_value)
```

```
        return exit_plan

    def _classify_position_priority(self, position):
        """
        Classify position for exit priority
        """
        risk_level = position.get('risk_level', 'medium')
        liquidity_level = position.get('liquidity', 'medium')

        priority_key = f"{risk_level}_risk_{liquidity_level}_liquidity"

        # Map to exit priorities
        if risk_level == 'high' and liquidity_level == 'low':
            return 'high_risk_low_liquidity'
        elif risk_level == 'high':
            return 'high_risk_high_liquidity'
        elif liquidity_level == 'low':
            return 'low_risk_low_liquidity'
        else:
            return 'low_risk_high_liquidity'

# Example usage
preservation_system = CapitalPreservationSystem({
    'reduce_risk': 0.05,
    'minimum_risk': 0.10,
    'survival_mode': 0.15,
    'emergency_exit': 0.20
})

# Test different drawdown levels
drawdown_levels = [0.03, 0.08, 0.12, 0.17, 0.22]

print("Capital Preservation System:")
print("=" * 80)
print("Drawdown\tMode\t\t\tPos Size\tAllowed Strategies\tNew
Trades\tReview Freq")
print("-" * 80)

for drawdown in drawdown_levels:
    mode = preservation_system.determine_preservation_mode(drawdown)
    actions = preservation_system.get_preservation_actions(mode)
```

```
    print(f"{drawdown:.1%}\t\t{mode.replace('_', ' '):<15}
\t{actions['position_size_multiplier']:.2f}
x\t\t{actions['allowed_strategies']:<15}\t{actions['new_trades']:<15}
\t{actions['review_frequency']}")

# Test emergency exit plan
print(f"\nEmergency Exit Plan Example:")
print("-" * 40)

current_positions = {
    'ETH_USDC': {'value': 50000, 'risk_level': 'high', 'liquidity':
'medium'},
    'BTC_Arbitrage': {'value': 30000, 'risk_level': 'low', 'liquidity':
'high'},
    'DeFi_Lending': {'value': 20000, 'risk_level': 'high', 'liquidity':
'low'}
}

emergency_plan =
preservation_system.calculate_emergency_exit_plan(current_positions,
0.22)

if emergency_plan:
    print(f"Estimated Time: {emergency_plan['estimated_time']}")
    print(f"Total Estimated Cost: ${emergency_plan['estimated_cost']:,.
2f}")
    print(f"Risk Reduction: {emergency_plan['risk_reduction']:.1%}")

    print(f"\nExit Order:")
    for exit_item in sorted(emergency_plan['exit_order'], key=lambda x:
x['priority_rank']):
        print(f"  {exit_item['urgency']}: {exit_item['position_id']} -
<span class="math-inline" style="display: inline;"><math xmlns="http://
www.w3.org/1998/Math/MathML" display="inline"><mrow><mrow><mi>e</
mi><mi>x</mi><mi>i</mi><msub><mi>t</mi><mi>i</mi></msub><mi>t</
mi><mi>e</mi><mi>m</mi><msup><mo stretchy="false">[</mo><mi>&#x02032;</
mi></msup><mi>a</mi><mi>m</mi><mi>o</mi><mi>u</mi><mi>n</
mi><msup><mi>t</mi><mi>&#x02032;</mi></msup><mo stretchy="false">]</
mo><mi>:</mi><mo>&#x0002C;</mo><mi>.0</mi><mi>f</mi></mrow><mo
stretchy="false">&#x00028;</mo></mrow></math></
span>{exit_item['estimated_cost']:,.2f} cost)")
```

Crisis Management Procedures

1. Comprehensive Crisis Response System

```
class CrisisResponseSystem:
    def __init__(self, crisis_thresholds, response_protocols):
        """
        crisis_thresholds = {
            'market_stress': {'volatility_spike': 3.0, 'volume_spike':
2.0},
            'operational_risk': {'error_rate': 0.05,
'downtime_minutes': 30},
            'portfolio_risk': {'correlation_spike': 0.3,
'drawdown_acceleration': 0.05}
        }
        """
        self.crisis_thresholds = crisis_thresholds
        self.response_protocols = response_protocols
        self.crisis_level = 'NORMAL'
        self.active_protocols = []

    def assess_crisis_conditions(self, market_data, operational_data,
portfolio_data):
        """
        Assess current crisis conditions across multiple dimensions
        """
        crisis_indicators = {
            'market_stress': self._assess_market_stress(market_data),
            'operational_risk':
self._assess_operational_risk(operational_data),
            'portfolio_risk':
self._assess_portfolio_risk(portfolio_data)
        }

        # Determine crisis level
        crisis_score = sum(crisis_indicators.values())

        if crisis_score >= 9:
            crisis_level = 'SEVERE'
        elif crisis_score >= 6:
            crisis_level = 'HIGH'
        elif crisis_score >= 3:
            crisis_level = 'MEDIUM'
        else:
            crisis_level = 'LOW'
```

```
        return crisis_level, crisis_indicators

    def _assess_market_stress(self, market_data):
        """
        Assess market stress conditions
        """
        score = 0

        # Volatility spike
        if market_data.get('volatility_multiplier', 1) >
self.crisis_thresholds['market_stress']['volatility_spike']:
            score += 3

        # Volume spike
        if market_data.get('volume_multiplier', 1) >
self.crisis_thresholds['market_stress']['volume_spike']:
            score += 2

        # Price gaps
        if market_data.get('large_price_gaps', 0) > 2:
            score += 2

        return min(score, 5)  # Cap at 5

    def _assess_operational_risk(self, operational_data):
        """
        Assess operational risk
        """
        score = 0

        # Error rate
        if operational_data.get('error_rate', 0) >
self.crisis_thresholds['operational_risk']['error_rate']:
            score += 3

        # System downtime
        if operational_data.get('downtime_minutes', 0) >
self.crisis_thresholds['operational_risk']['downtime_minutes']:
            score += 2

        # Failed transactions
        if operational_data.get('failed_transaction_rate', 0) > 0.1:
```

```
            score += 2

        return min(score, 5)

    def _assess_portfolio_risk(self, portfolio_data):
        """
        Assess portfolio risk
        """
        score = 0

        # Correlation spike
        if portfolio_data.get('correlation_increase', 0) >
self.crisis_thresholds['portfolio_risk']['correlation_spike']:
            score += 3

        # Drawdown acceleration
        if portfolio_data.get('drawdown_acceleration', 0) >
self.crisis_thresholds['portfolio_risk']['drawdown_acceleration']:
            score += 2

        # Concentration risk
        if portfolio_data.get('concentration_risk', 0) > 0.7:
            score += 2

        return min(score, 5)

    def activate_crisis_protocols(self, crisis_level):
        """
        Activate appropriate crisis response protocols
        """
        if crisis_level == 'SEVERE':
            protocols = [
                'EMERGENCY_STOP_ALL_TRADING',
                'LIQUIDATE_ALL_POSITIONS',
                'ALERT_ALL_STAKEHOLDERS',
                'ACTIVATE_BACKUP_SYSTEMS'
            ]
        elif crisis_level == 'HIGH':
            protocols = [
                'SUSPEND_NEW_TRADES',
                'REDUCE_POSITION_SIZES',
                'INCREASE_MONITORING',
```

```
                'PREPARE_CONTINGENCY_PLANS'
            ]
        elif crisis_level == 'MEDIUM':
            protocols = [
                'INCREASE_RISK_MONITORING',
                'REVIEW_POSITION_SIZES',
                'ENHANCED_REPORTING'
            ]
        else:
            protocols = ['NORMAL_MONITORING']

        self.active_protocols = protocols
        return protocols

    def execute_crisis_response(self, crisis_level, portfolio_state):
        """
        Execute crisis response actions
        """
        protocols = self.activate_crisis_protocols(crisis_level)
        response_actions = []

        for protocol in protocols:
            if protocol == 'EMERGENCY_STOP_ALL_TRADING':
                response_actions.append({
                    'action': 'STOP_ALL_TRADING',
                    'priority': 'IMMEDIATE',
                    'estimated_time': '1 minute',
                    'impact': 'Complete trading halt'
                })

            elif protocol == 'LIQUIDATE_ALL_POSITIONS':
                response_actions.append({
                    'action': 'LIQUIDATE_POSITIONS',
                    'priority': 'URGENT',
                    'estimated_time': '30 minutes',
                    'impact': 'Emergency position liquidation',
                    'tolerance': 'High slippage tolerance'
                })

            elif protocol == 'SUSPEND_NEW_TRADES':
                response_actions.append({
                    'action': 'SUSPEND_NEW_POSITIONS',
```

```
                    'priority': 'URGENT',
                    'estimated_time': '5 minutes',
                    'impact': 'No new positions allowed'
                })

            elif protocol == 'REDUCE_POSITION_SIZES':
                response_actions.append({
                    'action': 'REDUCE_POSITION_SIZES',
                    'priority': 'HIGH',
                    'estimated_time': '15 minutes',
                    'impact': 'Reduce sizes by 50%'
                })

        return response_actions

    def generate_crisis_report(self, crisis_level, indicators,
response_actions):
        """
        Generate comprehensive crisis response report
        """
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        report = f"""
CRISIS RESPONSE REPORT
======================
Timestamp: {timestamp}
Crisis Level: {crisis_level}

CRISIS INDICATORS:
------------------
Market Stress: {indicators['market_stress']}/5
Operational Risk: {indicators['operational_risk']}/5
Portfolio Risk: {indicators['portfolio_risk']}/5

ACTIVE PROTOCOLS:
-----------------
"""

        for protocol in self.active_protocols:
            report += f"- {protocol}\n"

        report += """
```

```
RESPONSE ACTIONS:
-----------------
"""

        for action in response_actions:
            report += f"- {action['action']} ({action['priority']})\n"
            report += f"  Time: {action['estimated_time']}\n"
            report += f"  Impact: {action['impact']}\n"
            if 'tolerance' in action:
                report += f"  Special Instructions:
{action['tolerance']}\n"
            report += "\n"

        report += """
NEXT STEPS:
----------
1. Execute immediate actions
2. Monitor system status
3. Communicate with stakeholders
4. Prepare for recovery phase
"""

        return report

# Example usage
crisis_system = CrisisResponseSystem(
    crisis_thresholds={
        'market_stress': {'volatility_spike': 3.0, 'volume_spike':
2.0},
        'operational_risk': {'error_rate': 0.05, 'downtime_minutes':
30},
        'portfolio_risk': {'correlation_spike': 0.3,
'drawdown_acceleration': 0.05}
    },
    response_protocols={}
)

# Test crisis scenarios
crisis_scenarios = [
    {
        'name': 'Market Crash',
        'market_data': {'volatility_multiplier': 4.5,
```

```
'volume_multiplier': 3.2, 'large_price_gaps': 5},
        'operational_data': {'error_rate': 0.02, 'downtime_minutes': 0,
'failed_transaction_rate': 0.05},
        'portfolio_data': {'correlation_increase': 0.4,
'drawdown_acceleration': 0.08, 'concentration_risk': 0.8}
    },
    {
        'name': 'Technical Issues',
        'market_data': {'volatility_multiplier': 1.2,
'volume_multiplier': 1.1, 'large_price_gaps': 0},
        'operational_data': {'error_rate': 0.08, 'downtime_minutes':
45, 'failed_transaction_rate': 0.15},
        'portfolio_data': {'correlation_increase': 0.1,
'drawdown_acceleration': 0.02, 'concentration_risk': 0.5}
    },
    {
        'name': 'Portfolio Stress',
        'market_data': {'volatility_multiplier': 1.8,
'volume_multiplier': 1.5, 'large_price_gaps': 1},
        'operational_data': {'error_rate': 0.01, 'downtime_minutes': 0,
'failed_transaction_rate': 0.02},
        'portfolio_data': {'correlation_increase': 0.35,
'drawdown_acceleration': 0.06, 'concentration_risk': 0.75}
    }
]

print("Crisis Response System Examples:")
print("=" * 60)

for scenario in crisis_scenarios:
    crisis_level, indicators = crisis_system.assess_crisis_conditions(
        scenario['market_data'],
        scenario['operational_data'],
        scenario['portfolio_data']
    )

    response_actions =
crisis_system.execute_crisis_response(crisis_level, {})

    print(f"\n{scenario['name']} Scenario:")
    print(f"Crisis Level: {crisis_level}")
    print(f"Indicators: Market={indicators['market_stress']},
```

```
Ops={indicators['operational_risk']},
Portfolio={indicators['portfolio_risk']}")

    print("Response Actions:")
    for action in response_actions:
        print(f"  - {action['action']} ({action['priority']})")

    # Generate full report for first scenario
    if scenario == crisis_scenarios[0]:
        report = crisis_system.generate_crisis_report(crisis_level,
indicators, response_actions)
        print(f"\nSample Crisis Report Generated (first 500 chars):")
        print(report[:500] + "...")
```

Key Takeaways

1. **Early detection is crucial** - Implement multiple layers of drawdown monitoring and alerts.
2. **Progressive response system** - Have clear escalation procedures based on drawdown severity.
3. **Recovery planning is essential** - Don't just survive drawdowns, have a plan to recover from them.
4. **Preservation before recovery** - Protect capital first, optimize recovery second.
5. **Crisis preparedness matters** - Pre-planned responses prevent panic decisions during stress.
6. **Multiple metrics provide better insight** - Monitor drawdown amount, duration, velocity, and frequency.
7. **Correlation awareness reduces risk** - High correlations can amplify drawdowns across the portfolio.
8. **Risk-adjusted recovery approaches** - Faster recovery attempts carry higher risk of further losses.

Next Steps

In the final module, **Risk Management Systems**, you'll learn to:

- Build comprehensive automated risk monitoring systems
- Create integrated risk dashboards and alerting
- Implement machine learning-based risk prediction
- Design risk management APIs and infrastructure

- Build comprehensive risk reporting frameworks

This will complete your risk management education with practical implementation of automated systems.

Summary

Effective drawdown control is the difference between surviving market stress and thriving in all conditions. By implementing comprehensive monitoring, progressive response systems, and structured recovery plans, you can protect your capital while maintaining the ability to generate returns.

Remember that drawdowns are not failures—they are opportunities to demonstrate the robustness of your risk management systems. The traders who survive and prosper through market stress are those who have prepared for adverse conditions and execute their plans discipline when markets turn against them.

The key is to treat drawdown control as an ongoing process, not a reactive measure. Build systems, test them, and trust them when you need them most.