# PRD 기반 암호화폐 자동거래 시스템 구현 가이드

**작성자:** Manus AI
**작성일:** 2025년 9월 1일
**문서 버전:** 1.0
**기반 문서:** 암호화폐 자동거래 PRD 최종완전판

## 목차



---

## 개요

본 문서는 PRD(Product Requirements Document)에 명시된 암호화폐 자동거래 시스템의 실제 구현을 위한 상세한 기술 가이드입니다. 파이썬을 기반으로 하여 바이낸스와 바이비트 거래소 API를 활용한 완전한 자동거래 시스템을 구축하는 방법을 단계별로 제시합니다.

PRD에서 정의된 핵심 기능들인 Price Channel 기반 진입, 호가 감지 진입, PCS(Price Channel System) 청산, 그리고 포괄적인 리스크 관리 시스템을 실제 코드로 구현하는 과정을 다룹니다. 특히 현물과 선물 거래를 모두 지원하는 듀얼 버전 시스템의 구현에 중점을 두어, 실제 운영 환경에서 안정적으로 작동할 수 있는 견고한 시스템 구축을 목표로 합니다.

이 가이드는 중급 이상의 파이썬 개발 경험과 암호화폐 거래에 대한 기본적인 이해를 전제로 작성되었습니다. 각 섹션은 이론적 배경, 구현 방법, 실제 코드 예제, 그리고 테스트 방법을 포함하여 개발자가 단계적으로 시스템을 구축할 수 있도록 구성되어 있습니다.

# 시스템 아키텍처 설계

## 전체 시스템 구조

암호화폐 자동거래 시스템의 아키텍처는 모듈화된 설계 원칙을 따라 구성됩니다. 각 모듈은 독립적으로 개발, 테스트, 배포가 가능하며, 시스템의 확장성과 유지보수성을 보장합니다.

시스템의 핵심 구조는 다음과 같은 계층으로 구성됩니다. 최상위 계층인 사용자 인터페이스 계층에서는 웹 기반 대시보드와 모바일 알림 시스템을 통해 사용자와의 상호작용을 담당합니다. 그 아래 비즈니스 로직 계층에서는 거래 전략, 리스크 관리, 포지션 관리 등의 핵심 비즈니스 로직을 처리합니다. 데이터 액세스 계층에서는 거래소 API와의 통신, 시장 데이터 수집, 주문 실행 등을 담당하며, 최하위 인프라 계층에서는 데이터베이스, 로깅, 모니터링 등의 기반 서비스를 제공합니다.

## 핵심 모듈 설계

**거래소 커넥터 모듈**은 바이낸스와 바이비트 API와의 통신을 추상화합니다. 이 모듈은 각 거래소의 API 차이점을 숨기고 통일된 인터페이스를 제공하여, 상위 계층에서는 거래소에 관계없이 동일한 방식으로 거래 기능을 사용할 수 있게 합니다. 커넥터는 REST API와 WebSocket 연결을 모두 지원하며, 자동 재연결, 에러 처리, 레이트 리미팅 등의 기능을 포함합니다.

**시장 데이터 수집 모듈**은 실시간 가격 정보, 오더북 데이터, 거래량 정보 등을 수집하고 처리합니다. 이 모듈은 WebSocket을 통해 실시간 데이터를 수신하고, 필요에 따라 REST API를 통해 과거 데이터를 보완합니다. 수집된 데이터는 메모리 캐시와 데이터베이스에 저장되어 빠른 접근과 영구 보관을 동시에 지원합니다.

**거래 전략 엔진**은 PRD에 정의된 다양한 진입 및 청산 전략을 구현합니다. Price Channel 분석, 호가 감지, PCS 청산 등의 알고리즘이 이 모듈에서 실행됩니다. 전략 엔진은 플러그인 아키텍처를 채택하여 새로운 전략을 쉽게 추가할 수 있도록 설계됩니다.

**리스크 관리 모듈**은 포지션 크기 계산, 손절/익절 설정, 최대 손실 제한 등의 리스크 관리 기능을 담당합니다. 이 모듈은 실시간으로 포트폴리오의 리스크를 모니터링하고, 설정된 한계를 초과할 경우 자동으로 보호 조치를 취합니다.

### 데이터 흐름 설계

시스템의 데이터 흐름은 실시간성과 정확성을 보장하도록 설계됩니다. 시장 데이터는 WebSocket을 통해 실시간으로 수집되어 메모리 기반 데이터 스토어에 저장됩니다. 이 데이터는 거래 전략 엔진에서 즉시 분석되어 거래 신호를 생성합니다.

거래 신호가 생성되면 리스크 관리 모듈에서 검증을 거친 후 주문 실행 모듈로 전달됩니다. 주문 실행 모듈은 거래소 API를 통해 실제 주문을 전송하고, 주문 상태를 실시간으로 추적합니다. 모든 거래 활동은 로깅 시스템을 통해 기록되며, 감사 추적과 성능 분석을 위해 데이터베이스에 영구 저장됩니다.

### 확장성 고려사항

시스템은 수평적 확장이 가능하도록 설계됩니다. 각 모듈은 독립적인 프로세스로 실행될 수 있으며, 메시지 큐를 통해 통신합니다. 이를 통해 트래픽 증가나 처리 요구사항 변화에 따라 특정 모듈만 확장할 수 있습니다.

데이터베이스는 읽기와 쓰기를 분리하여 성능을 최적화합니다. 실시간 데이터는 인메모리 데이터베이스에 저장하고, 과거 데이터와 분석 결과는 관계형 데이터베이스에 저장합니다. 대용량 시계열 데이터는 전용 시계열 데이터베이스를 사용하여 효율적으로 처리합니다.

### 보안 아키텍처

보안은 시스템 설계의 핵심 요소입니다. API 키와 같은 민감한 정보는 환경변수나 전용 시크릿 관리 시스템을 통해 관리됩니다. 모든 외부 통신은 TLS 암호화를 사용하며, 내부 모듈 간 통신도 필요에 따라 암호화됩니다.

접근 제어는 역할 기반 접근 제어(RBAC) 모델을 따릅니다. 사용자는 역할에 따라 시스템의 특정 기능에만 접근할 수 있으며, 모든 접근 시도는 로깅되어 보안 감사에 활용됩니다. 거래 실행과 같은 중요한 작업은 다중 인증을 요구하여 보안을 강화합니다.

## 핵심 거래 로직 구현

### 거래 엔진 기본 구조

거래 엔진은 시스템의 핵심 구성요소로서 모든 거래 결정과 실행을 담당합니다. 이 엔진은 상태 기반 설계를 채택하여 각 거래 세션의 상태를 명확히 관리하고, 예외 상황에서도 일관된 동작을 보장합니다.

거래 엔진의 기본 구조는 다음과 같은 핵심 컴포넌트로 구성됩니다. 신호 생성기는 시장 데이터를 분석하여 진입 및 청산 신호를 생성합니다. 주문 관리자는 생성된 신호를 바탕으로 실제 주문을 생성하고 관리합니다. 포지션 추적기는 현재 보유 포지션의 상태를 실시간으로 모니터링합니다. 리스크 검증기는 모든 거래 결정이 설정된 리스크 한계 내에서 이루어지도록 보장합니다.

```python
class TradingEngine:
    def __init__(self, config):
        self.config = config
        self.signal_generator = SignalGenerator(config)
        self.order_manager = OrderManager(config)
        self.position_tracker = PositionTracker(config)
        self.risk_validator = RiskValidator(config)
        self.state = TradingState.IDLE

    async def run(self):
        while self.is_running:
            try:
                # 시장 데이터 수집
                market_data = await self.collect_market_data()

                # 신호 생성
                signals = self.signal_generator.generate_signals(market_data)

                # 리스크 검증
                validated_signals = self.risk_validator.validate(signals)

                # 주문 실행
                for signal in validated_signals:
                    await self.execute_signal(signal)

                # 포지션 모니터링
                await self.monitor_positions()

            except Exception as e:
                logger.error(f"거래 엔진 실행 오류: {e}")
                await self.handle_error(e)
```

## 신호 생성 시스템

신호 생성 시스템은 PRD에 정의된 다양한 진입 조건을 실시간으로 모니터링하고 거래 신호를 생성합니다. 이 시스템은 모듈화된 설계를 통해 각 전략을 독립적으로 구현하고 조합할 수 있도록 합니다.

Price Channel 신호 생성기는 설정된 기간 동안의 최고가와 최저가를 기반으로 채널을 형성하고, 현재 가격이 이 채널을 돌파할 때 신호를 생성합니다. 호가 감지 신호 생성기는 매수/매도 호가의 변화를 실시간으로 모니터링하여 설정된 조건에 부합할 때 신호를 발생시킵니다.

```python
class SignalGenerator:
    def __init__(self, config):
        self.price_channel_analyzer = PriceChannelAnalyzer(config)
        self.orderbook_analyzer = OrderbookAnalyzer(config)
        self.moving_average_analyzer = MovingAverageAnalyzer(config)

    def generate_signals(self, market_data):
        signals = []

        # Price Channel 분석
        pc_signal = self.price_channel_analyzer.analyze(market_data)
        if pc_signal:
            signals.append(pc_signal)

        # 호가 분석
        ob_signal = self.orderbook_analyzer.analyze(market_data)
        if ob_signal:
            signals.append(ob_signal)

        # 이동평균선 분석
        ma_signal = self.moving_average_analyzer.analyze(market_data)
        if ma_signal:
            signals.append(ma_signal)

        return self.filter_signals(signals)
```

### 주문 실행 시스템

주문 실행 시스템은 생성된 신호를 바탕으로 실제 거래소에 주문을 전송하고 관리합니다. 이 시스템은 거래소별 API 차이점을 추상화하고, 주문 실행의 안정성과 정확성을 보장합니다.

주문 실행 과정은 여러 단계로 구성됩니다. 먼저 신호 검증 단계에서 신호의 유효성과 실행 가능성을 확인합니다. 다음으로 주문 파라미터 계산 단계에서 포지션 크기, 가격, 주문 타입 등을 결정합니다. 주문 전송 단계에서 실제 거래소 API를 호출하여 주문을 전송하고, 주문 추적 단계에서 주문 상태를 실시간으로 모니터링합니다.

```python
class OrderManager:
    def __init__(self, config):
        self.exchange_connector = ExchangeConnector(config)
        self.position_calculator = PositionCalculator(config)
        self.order_tracker = OrderTracker()

    async def execute_signal(self, signal):
        try:
            # 주문 파라미터 계산
            order_params = self.calculate_order_params(signal)

            # 주문 전송
            order_result = await
self.exchange_connector.place_order(order_params)

            # 주문 추적 시작
            self.order_tracker.add_order(order_result)

            return order_result

        except Exception as e:
            logger.error(f"주문 실행 실패: {e}")
            raise

    def calculate_order_params(self, signal):
        # 포지션 크기 계산
        position_size = self.position_calculator.calculate_size(
            signal.symbol, signal.direction, signal.risk_level
        )

        # 주문 타입 결정
        order_type = self.determine_order_type(signal)

        # 가격 계산
        price = self.calculate_price(signal, order_type)

        return {
            'symbol': signal.symbol,
            'side': signal.direction,
            'type': order_type,
            'quantity': position_size,
            'price': price
        }
```

## 포지션 관리 시스템

포지션 관리 시스템은 현재 보유 중인 모든 포지션의 상태를 실시간으로 추적하고 관리합니다. 이 시스템은 포지션의 진입가, 현재 손익, 보유 기간 등을 모니터링하여 청산 조건을 판단합니다.

포지션 추적은 거래소에서 제공하는 포지션 정보와 시스템 내부에서 관리하는 포지션 정보를 동기화하여 정확성을 보장합니다. 불일치가 발견될 경우 자동으로 조정하거나 관리자에게 알림을 전송합니다.

```python
class PositionTracker:
    def __init__(self, config):
        self.positions = {}
        self.exchange_connector = ExchangeConnector(config)

    async def update_positions(self):
        # 거래소에서 포지션 정보 조회
        exchange_positions = await self.exchange_connector.get_positions()

        # 내부 포지션 정보와 동기화
        for position in exchange_positions:
            symbol = position['symbol']
            if symbol in self.positions:
                self.positions[symbol].update_from_exchange(position)
            else:
                self.positions[symbol] = Position.from_exchange_data(position)

    def get_position_pnl(self, symbol):
        if symbol in self.positions:
            return self.positions[symbol].unrealized_pnl
        return 0

    def should_close_position(self, symbol):
        if symbol not in self.positions:
            return False

        position = self.positions[symbol]

        # 손절 조건 확인
        if position.unrealized_pnl_percent < -position.stop_loss_percent:
            return True, "stop_loss"

        # 익절 조건 확인
        if position.unrealized_pnl_percent > position.take_profit_percent:
            return True, "take_profit"

        # 시간 기반 청산 조건 확인
        if position.holding_time > position.max_holding_time:
            return True, "time_limit"

        return False, None
```

## 에러 처리 및 복구 메커니즘

거래 시스템에서 에러 처리는 매우 중요한 요소입니다. 네트워크 연결 실패, API 응답 지연, 주문 실행 실패 등 다양한 예외 상황에 대비한 견고한 에러 처리 메커니즘을 구현해야 합니다.

에러 처리는 계층적 접근 방식을 사용합니다. 낮은 수준의 에러는 자동으로 재시도하거나 대체 방법을 사용하여 처리합니다. 높은 수준의 에러는 시스템 상태를 안전 모드로 전환하고 관리자에게 알림을 전송합니다. 모든 에러는 상세히 로깅되어 추후 분석과 시스템 개선에 활용됩니다.

```python
class ErrorHandler:
    def __init__(self, config):
        self.config = config
        self.retry_config = config.retry_config
        self.notification_service = NotificationService(config)

    async def handle_api_error(self, error, context):
        if isinstance(error, NetworkError):
            return await self.handle_network_error(error, context)
        elif isinstance(error, RateLimitError):
            return await self.handle_rate_limit_error(error, context)
        elif isinstance(error, OrderError):
            return await self.handle_order_error(error, context)
        else:
            return await self.handle_unknown_error(error, context)

    async def handle_network_error(self, error, context):
        retry_count = context.get('retry_count', 0)
        max_retries = self.retry_config.max_retries

        if retry_count < max_retries:
            delay = self.calculate_backoff_delay(retry_count)
            await asyncio.sleep(delay)
            context['retry_count'] = retry_count + 1
            return True  # 재시도
        else:
            await self.notification_service.send_alert(
                "네트워크 오류로 인한 연결 실패", error, context
            )
            return False  # 재시도 포기
```

이러한 핵심 거래 로직의 구현을 통해 안정적이고 효율적인 자동거래 시스템의 기반을 마련할 수 있습니다. 각 컴포넌트는 독립적으로 테스트 가능하며, 필요에 따라 개별적으로 최적화할 수 있도록 설계됩니다.

# Price Channel 시스템 구현

## Price Channel 이론적 배경

Price Channel은 기술적 분석에서 사용되는 핵심 지표 중 하나로, 특정 기간 동안의 최고가와 최저가를 연결하여 형성되는 채널을 의미합니다. PRD에서 정의된 Price Channel 시스템은 Donchian Channel을 기반으로 하며, 설정 가능한 기간 동안의 가격 범위를 분석하여 돌파 신호를 생성합니다.

Price Channel의 핵심 개념은 가격이 일정 기간 동안 형성된 범위를 벗어날 때 새로운 추세가 시작될 가능성이 높다는 것입니다. 상단선(Upper Channel) 돌파는 상승 추세의 시작을, 하단선(Lower Channel) 돌파는 하락 추세의 시작을 의미할 수 있습니다. 이러한 돌파는 강력한 모멘텀을 동반하는 경우가 많아 자동거래 시스템의 진입 신호로 활용됩니다.

## Price Channel 계산 알고리즘

Price Channel의 계산은 비교적 단순하지만 정확한 구현이 중요합니다. 상단선은 설정된 기간 동안의 최고가로, 하단선은 같은 기간 동안의 최저가로 계산됩니다. 중간선은 상단선과 하단선의 평균값으로 구할 수 있습니다.

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

@dataclass
class PriceChannelConfig:
    period: int = 20  # 채널 계산 기간
    breakout_threshold: float = 0.001  # 돌파 임계값 (0.1%)
    confirmation_candles: int = 2  # 돌파 확인 캔들 수
    min_channel_width: float = 0.005  # 최소 채널 폭 (0.5%)

class PriceChannelCalculator:
    def __init__(self, config: PriceChannelConfig):
        self.config = config

    def calculate_channel(self, price_data: pd.DataFrame) -> Dict:
        """
        Price Channel 계산

        Args:
            price_data: OHLCV 데이터프레임

        Returns:
            채널 정보 딕셔너리
        """
        if len(price_data) < self.config.period:
            raise ValueError(f"데이터 길이가 부족합니다. 최소 {self.config.period}개
필요")

        # 롤링 최고가/최저가 계산
        price_data['upper_channel'] = price_data['high'].rolling(
            window=self.config.period
        ).max()

        price_data['lower_channel'] = price_data['low'].rolling(
            window=self.config.period
        ).min()

        # 중간선 계산
        price_data['middle_channel'] = (
            price_data['upper_channel'] + price_data['lower_channel']
        ) / 2

        # 채널 폭 계산
        price_data['channel_width'] = (
            price_data['upper_channel'] - price_data['lower_channel']
        ) / price_data['middle_channel']

        # 최신 채널 정보 반환
        latest = price_data.iloc[-1]

        return {
            'upper_channel': latest['upper_channel'],
            'lower_channel': latest['lower_channel'],
            'middle_channel': latest['middle_channel'],
            'channel_width': latest['channel_width'],
            'current_price': latest['close'],
```

```
            'timestamp': latest.name
        }
```

## 돌파 신호 감지 시스템

Price Channel 돌파 신호의 감지는 단순히 가격이 채널을 넘어서는 것만으로는 충분하지 않습니다. 거짓 신호를 줄이기 위해 여러 확인 조건을 적용해야 합니다. 돌파의 강도, 거래량 증가, 연속적인 확인 캔들 등을 종합적으로 고려하여 신호의 신뢰도를 높입니다.

```python
class PriceChannelBreakoutDetector:
    def __init__(self, config: PriceChannelConfig):
        self.config = config
        self.calculator = PriceChannelCalculator(config)
        self.breakout_history = []

    def detect_breakout(self, price_data: pd.DataFrame) -> Dict:
        """
        Price Channel 돌파 감지

        Returns:
            돌파 신호 정보
        """
        channel_info = self.calculator.calculate_channel(price_data)

        # 채널 폭이 최소 기준 이상인지 확인
        if channel_info['channel_width'] < self.config.min_channel_width:
            return {'signal': None, 'reason': 'channel_too_narrow'}

        current_price = channel_info['current_price']
        upper_channel = channel_info['upper_channel']
        lower_channel = channel_info['lower_channel']

        # 상단 돌파 확인
        upper_breakout_price = upper_channel * (1 +
self.config.breakout_threshold)
        if current_price > upper_breakout_price:
            if self._confirm_breakout(price_data, 'upper'):
                return self._create_breakout_signal('BUY', channel_info)

        # 하단 돌파 확인
        lower_breakout_price = lower_channel * (1 -
self.config.breakout_threshold)
        if current_price < lower_breakout_price:
            if self._confirm_breakout(price_data, 'lower'):
                return self._create_breakout_signal('SELL', channel_info)

        return {'signal': None, 'reason': 'no_breakout'}

    def _confirm_breakout(self, price_data: pd.DataFrame, direction: str) ->
bool:
        """돌파 확인"""
        recent_candles = price_data.tail(self.config.confirmation_candles)

        if direction == 'upper':
            # 연속적인 상승 확인
            return all(
                candle['close'] > candle['open']
                for _, candle in recent_candles.iterrows()
            )
        else:
            # 연속적인 하락 확인
            return all(
                candle['close'] < candle['open']
                for _, candle in recent_candles.iterrows()
            )

    def _create_breakout_signal(self, direction: str, channel_info: Dict) ->
Dict:
        """돌파 신호 생성"""
        signal = {
```

```
            'signal': 'BREAKOUT',
            'direction': direction,
            'entry_price': channel_info['current_price'],
            'upper_channel': channel_info['upper_channel'],
            'lower_channel': channel_info['lower_channel'],
            'channel_width': channel_info['channel_width'],
            'timestamp': datetime.now(),
            'confidence': self._calculate_confidence(channel_info)
        }

        self.breakout_history.append(signal)
        return signal

    def _calculate_confidence(self, channel_info: Dict) -> float:
        """신호 신뢰도 계산"""
        # 채널 폭이 클수록 신뢰도 높음
        width_score = min(channel_info['channel_width'] / 0.02, 1.0)

        # 기본 신뢰도 + 채널 폭 보너스
        confidence = 0.7 + (width_score * 0.3)

        return round(confidence, 2)
```

## 다중 시간대 분석

PRD에서 요구하는 차트 주기별 독립 설정을 구현하기 위해 다중 시간대 분석 시스템을 구축합니다. 이 시스템은 1분, 5분, 15분, 1시간 등 다양한 시간대에서 동시에 Price Channel을 분석하고, 각 시간대의 신호를 종합하여 최종 거래 결정을 내립니다.

```python
class MultiTimeframePriceChannel:
    def __init__(self, timeframes: List[str]):
        self.timeframes = timeframes
        self.detectors = {}

        # 각 시간대별 감지기 초기화
        for tf in timeframes:
            config = self._get_timeframe_config(tf)
            self.detectors[tf] = PriceChannelBreakoutDetector(config)

    def _get_timeframe_config(self, timeframe: str) -> PriceChannelConfig:
        """시간대별 설정 반환"""
        configs = {
            '1m': PriceChannelConfig(period=20, breakout_threshold=0.001),
            '5m': PriceChannelConfig(period=20, breakout_threshold=0.002),
            '15m': PriceChannelConfig(period=14, breakout_threshold=0.003),
            '1h': PriceChannelConfig(period=10, breakout_threshold=0.005),
            '4h': PriceChannelConfig(period=7, breakout_threshold=0.008),
            '1d': PriceChannelConfig(period=5, breakout_threshold=0.01)
        }
        return configs.get(timeframe, PriceChannelConfig())

    def analyze_all_timeframes(self, price_data_dict: Dict[str, pd.DataFrame])
-> Dict:
        """모든 시간대 분석"""
        results = {}

        for timeframe in self.timeframes:
            if timeframe in price_data_dict:
                detector = self.detectors[timeframe]
                result = detector.detect_breakout(price_data_dict[timeframe])
                results[timeframe] = result

        return self._synthesize_signals(results)

    def _synthesize_signals(self, results: Dict) -> Dict:
        """다중 시간대 신호 종합"""
        signals = [r for r in results.values() if r.get('signal') ==
'BREAKOUT']

        if not signals:
            return {'signal': None, 'reason': 'no_signals'}

        # 같은 방향 신호 개수 계산
        buy_signals = [s for s in signals if s['direction'] == 'BUY']
        sell_signals = [s for s in signals if s['direction'] == 'SELL']

        # 신호 강도 계산
        if len(buy_signals) >= 2:
            avg_confidence = sum(s['confidence'] for s in buy_signals) /
len(buy_signals)
            return {
                'signal': 'STRONG_BUY',
                'direction': 'BUY',
                'confidence': avg_confidence,
                'supporting_timeframes': list(results.keys()),
                'signal_count': len(buy_signals)
            }
        elif len(sell_signals) >= 2:
            avg_confidence = sum(s['confidence'] for s in sell_signals) /
len(sell_signals)
```

```
            return {
                'signal': 'STRONG_SELL',
                'direction': 'SELL',
                'confidence': avg_confidence,
                'supporting_timeframes': list(results.keys()),
                'signal_count': len(sell_signals)
            }
        else:
            return {'signal': 'WEAK', 'reason': 'conflicting_signals'}
```

## Price Channel 기반 진입 전략

Price Channel 돌파 신호가 확인되면 실제 진입 전략을 실행합니다. 이 과정에서는 진입 가격, 포지션 크기, 초기 손절가 등을 계산하고 주문을 실행합니다.

```python
class PriceChannelEntryStrategy:
    def __init__(self, config):
        self.config = config
        self.risk_manager = RiskManager(config)

    def execute_entry(self, signal: Dict, account_balance: float) -> Dict:
        """Price Channel 돌파 진입 실행"""

        # 포지션 크기 계산
        position_size = self.risk_manager.calculate_position_size(
            balance=account_balance,
            entry_price=signal['entry_price'],
            stop_loss_price=self._calculate_stop_loss(signal),
            risk_percent=self.config.risk_percent_per_trade
        )

        # 진입 주문 파라미터 생성
        entry_order = {
            'symbol': signal['symbol'],
            'side': signal['direction'],
            'type': 'MARKET',  # 돌파 시 즉시 진입
            'quantity': position_size,
            'timestamp': datetime.now()
        }

        # 손절 주문 파라미터 생성
        stop_loss_order = {
            'symbol': signal['symbol'],
            'side': 'SELL' if signal['direction'] == 'BUY' else 'BUY',
            'type': 'STOP_LOSS_LIMIT',
            'quantity': position_size,
            'stop_price': self._calculate_stop_loss(signal),
            'price': self._calculate_stop_loss(signal) * 0.995  # 슬리피지 고려
        }

        return {
            'entry_order': entry_order,
            'stop_loss_order': stop_loss_order,
            'expected_risk': self._calculate_expected_risk(signal,
position_size),
            'channel_info': signal
        }

    def _calculate_stop_loss(self, signal: Dict) -> float:
        """손절가 계산"""
        if signal['direction'] == 'BUY':
            # 매수 시 하단 채널 아래로 손절
            return signal['lower_channel'] * 0.995
        else:
            # 매도 시 상단 채널 위로 손절
            return signal['upper_channel'] * 1.005

    def _calculate_expected_risk(self, signal: Dict, position_size: float) ->
float:
        """예상 리스크 계산"""
        entry_price = signal['entry_price']
        stop_loss_price = self._calculate_stop_loss(signal)

        risk_per_unit = abs(entry_price - stop_loss_price)
        total_risk = risk_per_unit * position_size
```

```
        return total_risk
```

이러한 Price Channel 시스템의 구현을 통해 PRD에서 요구하는 체계적이고 신뢰할 수 있는 진입 신호 생성이 가능합니다. 다중 시간대 분석과 엄격한 확인 절차를 통해 거짓 신호를 최소화하고 수익성 있는 거래 기회를 포착할 수 있습니다.

# 호가 감지 진입 로직 구현

## 호가 감지 시스템 개요

호가 감지 진입 로직은 PRD에서 정의된 핵심 진입 방법 중 하나로, 매수/매도 호가의 변화를 실시간으로 모니터링하여 시장의 미세한 변화를 포착합니다. 이 시스템은 오더북의 깊이와 스프레드 변화를 분석하여 가격 움직임을 예측하고 적절한 진입 타이밍을 결정합니다.

호가 감지 시스템의 핵심은 매수 호가와 매도 호가 간의 차이(스프레드)와 각 호가 레벨의 물량 변화를 실시간으로 추적하는 것입니다. 큰 주문이 들어오거나 호가 스프레드가 급격히 변화할 때, 이는 가격 움직임의 전조가 될 수 있습니다. 특히 PRD에서 명시된 "설정된 호가 차이 발생 시 진입" 조건을 구현하기 위해서는 정밀한 호가 분석 알고리즘이 필요합니다.

## 실시간 오더북 데이터 수집

호가 감지를 위해서는 먼저 실시간 오더북 데이터를 안정적으로 수집해야 합니다. WebSocket을 통해 거래소로부터 실시간 오더북 업데이트를 받아 메모리에 최신 상태를 유지합니다.

```python
 import asyncio
import websockets
import json
from collections import deque
from typing import Dict, List, Optional
from dataclasses import dataclass, field
from datetime import datetime, timedelta

@dataclass
class OrderbookLevel:
    price: float
    quantity: float
    timestamp: datetime

@dataclass
class OrderbookSnapshot:
    symbol: str
    bids: List[OrderbookLevel] = field(default_factory=list)
    asks: List[OrderbookLevel] = field(default_factory=list)
    timestamp: datetime = field(default_factory=datetime.now)

    @property
    def best_bid(self) -> Optional[OrderbookLevel]:
        return self.bids[0] if self.bids else None

    @property
    def best_ask(self) -> Optional[OrderbookLevel]:
        return self.asks[0] if self.asks else None

    @property
    def spread(self) -> float:
        if self.best_bid and self.best_ask:
            return self.best_ask.price - self.best_bid.price
        return 0.0

    @property
    def spread_percentage(self) -> float:
        if self.best_bid and self.best_ask:
            mid_price = (self.best_bid.price + self.best_ask.price) / 2
            return (self.spread / mid_price) * 100
        return 0.0

class OrderbookManager:
    def __init__(self, symbol: str, depth: int = 20):
        self.symbol = symbol
        self.depth = depth
        self.current_orderbook = None
        self.orderbook_history = deque(maxlen=1000)
        self.update_callbacks = []

    async def start_websocket_feed(self, exchange_config):
        """WebSocket을 통한 실시간 오더북 수집 시작"""
        if exchange_config.exchange == 'binance':
            await self._start_binance_feed(exchange_config)
        elif exchange_config.exchange == 'bybit':
            await self._start_bybit_feed(exchange_config)

    async def _start_binance_feed(self, config):
        """바이낸스 WebSocket 피드"""
        uri =
f"wss://stream.binance.com:9443/ws/{self.symbol.lower()}@depth{self.depth}@100ms"
```

```
        async with websockets.connect(uri) as websocket:
            async for message in websocket:
                try:
                    data = json.loads(message)
                    orderbook = self._parse_binance_orderbook(data)
                    await self._update_orderbook(orderbook)
                except Exception as e:
                    logger.error(f"오더북 파싱 오류: {e}")

    def _parse_binance_orderbook(self, data: Dict) -> OrderbookSnapshot:
        """바이낸스 오더북 데이터 파싱"""
        bids = [
            OrderbookLevel(float(price), float(qty), datetime.now())
            for price, qty in data['bids']
        ]
        asks = [
            OrderbookLevel(float(price), float(qty), datetime.now())
            for price, qty in data['asks']
        ]

        return OrderbookSnapshot(
            symbol=self.symbol,
            bids=sorted(bids, key=lambda x: x.price, reverse=True),
            asks=sorted(asks, key=lambda x: x.price),
            timestamp=datetime.now()
        )

    async def _update_orderbook(self, orderbook: OrderbookSnapshot):
        """오더북 업데이트 및 콜백 실행"""
        self.current_orderbook = orderbook
        self.orderbook_history.append(orderbook)

        # 등록된 콜백 함수들 실행
        for callback in self.update_callbacks:
            try:
                await callback(orderbook)
            except Exception as e:
                logger.error(f"오더북 콜백 실행 오류: {e}")

    def add_update_callback(self, callback):
        """오더북 업데이트 콜백 등록"""
        self.update_callbacks.append(callback)
```

## 호가 변화 분석 알고리즘

호가 감지 시스템의 핵심은 오더북의 변화를 분석하여 의미 있는 신호를 추출하는 것입니다. 이를 위해 여러 지표를 동시에 모니터링합니다.

```python
@dataclass
class OrderbookAnalysisConfig:
    spread_threshold_percent: float = 0.1  # 스프레드 임계값 (%)
    volume_imbalance_threshold: float = 2.0  # 물량 불균형 임계값
    large_order_threshold: float = 10000  # 대량 주문 임계값 (USDT)
    price_level_count: int = 5  # 분석할 호가 레벨 수
    analysis_window_seconds: int = 10  # 분석 윈도우 (초)

class OrderbookAnalyzer:
    def __init__(self, config: OrderbookAnalysisConfig):
        self.config = config
        self.analysis_history = deque(maxlen=100)

    async def analyze_orderbook_change(self, orderbook: OrderbookSnapshot) ->
Dict:
        """오더북 변화 분석"""
        analysis_result = {
            'timestamp': orderbook.timestamp,
            'symbol': orderbook.symbol,
            'signals': []
        }

        # 스프레드 분석
        spread_signal = self._analyze_spread_change(orderbook)
        if spread_signal:
            analysis_result['signals'].append(spread_signal)

        # 물량 불균형 분석
        imbalance_signal = self._analyze_volume_imbalance(orderbook)
        if imbalance_signal:
            analysis_result['signals'].append(imbalance_signal)

        # 대량 주문 감지
        large_order_signal = self._detect_large_orders(orderbook)
        if large_order_signal:
            analysis_result['signals'].append(large_order_signal)

        # 호가 레벨 변화 분석
        level_change_signal = self._analyze_level_changes(orderbook)
        if level_change_signal:
            analysis_result['signals'].append(level_change_signal)

        self.analysis_history.append(analysis_result)
        return analysis_result

    def _analyze_spread_change(self, orderbook: OrderbookSnapshot) ->
Optional[Dict]:
        """스프레드 변화 분석"""
        if not orderbook.best_bid or not orderbook.best_ask:
            return None

        current_spread_pct = orderbook.spread_percentage

        # 과거 스프레드와 비교
        if len(self.analysis_history) > 0:
            recent_spreads = [
                self._get_spread_from_history(h)
                for h in list(self.analysis_history)[-10:]
            ]
            avg_spread = sum(s for s in recent_spreads if s) /
len(recent_spreads) if recent_spreads else 0
```

```
            # 스프레드가 급격히 확대된 경우
            if current_spread_pct > avg_spread * 2 and current_spread_pct >
self.config.spread_threshold_percent:
                return {
                    'type': 'SPREAD_WIDENING',
                    'current_spread': current_spread_pct,
                    'average_spread': avg_spread,
                    'severity': 'HIGH' if current_spread_pct > avg_spread * 3
else 'MEDIUM'
                }

        return None

    def _analyze_volume_imbalance(self, orderbook: OrderbookSnapshot) ->
Optional[Dict]:
        """매수/매도 물량 불균형 분석"""
        if not orderbook.bids or not orderbook.asks:
            return None

        # 상위 N개 레벨의 총 물량 계산
        bid_volume = sum(
            level.quantity for level in
orderbook.bids[:self.config.price_level_count]
        )
        ask_volume = sum(
            level.quantity for level in
orderbook.asks[:self.config.price_level_count]
        )

        if ask_volume == 0:
            return None

        imbalance_ratio = bid_volume / ask_volume

        # 매수 우세
        if imbalance_ratio > self.config.volume_imbalance_threshold:
            return {
                'type': 'BUY_VOLUME_DOMINANCE',
                'imbalance_ratio': imbalance_ratio,
                'bid_volume': bid_volume,
                'ask_volume': ask_volume,
                'strength': 'STRONG' if imbalance_ratio > 3.0 else 'MODERATE'
            }

        # 매도 우세
        elif imbalance_ratio < (1 / self.config.volume_imbalance_threshold):
            return {
                'type': 'SELL_VOLUME_DOMINANCE',
                'imbalance_ratio': imbalance_ratio,
                'bid_volume': bid_volume,
                'ask_volume': ask_volume,
                'strength': 'STRONG' if imbalance_ratio < 0.33 else 'MODERATE'
            }

        return None

    def _detect_large_orders(self, orderbook: OrderbookSnapshot) ->
Optional[Dict]:
        """대량 주문 감지"""
        large_orders = []
```

```
        # 매수 호가에서 대량 주문 찾기
        for level in orderbook.bids[:self.config.price_level_count]:
            order_value = level.price * level.quantity
            if order_value > self.config.large_order_threshold:
                large_orders.append({
                    'side': 'BUY',
                    'price': level.price,
                    'quantity': level.quantity,
                    'value': order_value
                })

        # 매도 호가에서 대량 주문 찾기
        for level in orderbook.asks[:self.config.price_level_count]:
            order_value = level.price * level.quantity
            if order_value > self.config.large_order_threshold:
                large_orders.append({
                    'side': 'SELL',
                    'price': level.price,
                    'quantity': level.quantity,
                    'value': order_value
                })

        if large_orders:
            return {
                'type': 'LARGE_ORDERS_DETECTED',
                'orders': large_orders,
                'total_value': sum(order['value'] for order in large_orders)
            }

        return None

    def _get_spread_from_history(self, history_item: Dict) -> Optional[float]:
        """과거 기록에서 스프레드 추출"""
        # 구현 세부사항...
        return None
```

## 호가 기반 진입 신호 생성

오더북 분석 결과를 바탕으로 실제 진입 신호를 생성합니다. 여러 신호가 동시에 발생할 때의 우선순위와 신호 강도를 고려하여 최종 진입 결정을 내립니다.

```python
class OrderbookEntrySignalGenerator:
    def __init__(self, config):
        self.config = config
        self.signal_history = deque(maxlen=100)

    def generate_entry_signal(self, analysis_result: Dict) -> Optional[Dict]:
        """호가 분석 결과를 바탕으로 진입 신호 생성"""
        signals = analysis_result.get('signals', [])

        if not signals:
            return None

        # 신호 강도 계산
        signal_strength = self._calculate_signal_strength(signals)

        # 진입 방향 결정
        entry_direction = self._determine_entry_direction(signals)

        if signal_strength < 0.6:  # 최소 신뢰도 임계값
            return None

        # 진입 신호 생성
        entry_signal = {
            'type': 'ORDERBOOK_ENTRY',
            'direction': entry_direction,
            'strength': signal_strength,
            'timestamp': analysis_result['timestamp'],
            'symbol': analysis_result['symbol'],
            'supporting_signals': signals,
            'entry_method': 'LIMIT',  # 호가 기반은 지정가 주문 선호
            'urgency': self._calculate_urgency(signals)
        }

        self.signal_history.append(entry_signal)
        return entry_signal

    def _calculate_signal_strength(self, signals: List[Dict]) -> float:
        """신호 강도 계산"""
        strength_weights = {
            'LARGE_ORDERS_DETECTED': 0.4,
            'BUY_VOLUME_DOMINANCE': 0.3,
            'SELL_VOLUME_DOMINANCE': 0.3,
            'SPREAD_WIDENING': 0.2
        }

        total_strength = 0.0
        total_weight = 0.0

        for signal in signals:
            signal_type = signal['type']
            if signal_type in strength_weights:
                weight = strength_weights[signal_type]

                # 신호별 강도 계산
                if signal_type == 'LARGE_ORDERS_DETECTED':
                    signal_strength = min(signal['total_value'] / 100000, 1.0)
                elif signal_type in ['BUY_VOLUME_DOMINANCE',
'SELL_VOLUME_DOMINANCE']:
                    signal_strength = min(signal['imbalance_ratio'] / 5.0, 1.0)
                elif signal_type == 'SPREAD_WIDENING':
                    signal_strength = 0.5 if signal['severity'] == 'MEDIUM'
```

```
else 0.8
                else:
                    signal_strength = 0.5

                total_strength += signal_strength * weight
                total_weight += weight

        return total_strength / total_weight if total_weight > 0 else 0.0

    def _determine_entry_direction(self, signals: List[Dict]) -> str:
        """진입 방향 결정"""
        buy_score = 0
        sell_score = 0

        for signal in signals:
            if signal['type'] == 'BUY_VOLUME_DOMINANCE':
                buy_score += 1
            elif signal['type'] == 'SELL_VOLUME_DOMINANCE':
                sell_score += 1
            elif signal['type'] == 'LARGE_ORDERS_DETECTED':
                # 대량 주문의 방향에 따라 점수 부여
                for order in signal['orders']:
                    if order['side'] == 'BUY':
                        buy_score += 0.5
                    else:
                        sell_score += 0.5

        return 'BUY' if buy_score > sell_score else 'SELL'

    def _calculate_urgency(self, signals: List[Dict]) -> str:
        """진입 긴급도 계산"""
        has_large_orders = any(s['type'] == 'LARGE_ORDERS_DETECTED' for s in
signals)
        has_strong_imbalance = any(
            s.get('strength') == 'STRONG'
            for s in signals
            if s['type'] in ['BUY_VOLUME_DOMINANCE', 'SELL_VOLUME_DOMINANCE']
        )

        if has_large_orders and has_strong_imbalance:
            return 'HIGH'
        elif has_large_orders or has_strong_imbalance:
            return 'MEDIUM'
        else:
            return 'LOW'
```

### 호가 기반 주문 실행 전략

호가 감지 신호가 생성되면 적절한 주문 실행 전략을 선택합니다. 호가 기반 진입은 일반적으로 지정가 주문을 사용하여 더 유리한 가격에 진입하려고 시도합니다.

```python
class OrderbookEntryExecutor:
    def __init__(self, exchange_connector, config):
        self.exchange_connector = exchange_connector
        self.config = config

    async def execute_orderbook_entry(self, signal: Dict, orderbook:
OrderbookSnapshot) -> Dict:
        """호가 기반 진입 실행"""

        # 진입 가격 계산
        entry_price = self._calculate_entry_price(signal, orderbook)

        # 포지션 크기 계산
        position_size = self._calculate_position_size(signal, entry_price)

        # 주문 타입 결정
        order_type = self._determine_order_type(signal)

        # 주문 실행
        if order_type == 'LIMIT':
            order_result = await self._execute_limit_order(
                signal, entry_price, position_size
            )
        else:
            order_result = await self._execute_market_order(
                signal, position_size
            )

        return {
            'order_result': order_result,
            'entry_price': entry_price,
            'position_size': position_size,
            'signal': signal
        }

    def _calculate_entry_price(self, signal: Dict, orderbook:
OrderbookSnapshot) -> float:
        """진입 가격 계산"""
        if signal['direction'] == 'BUY':
            # 매수 시 현재 매수 호가보다 약간 높은 가격
            best_bid = orderbook.best_bid.price
            tick_size = self._get_tick_size(signal['symbol'])
            return best_bid + (tick_size * 2)
        else:
            # 매도 시 현재 매도 호가보다 약간 낮은 가격
            best_ask = orderbook.best_ask.price
            tick_size = self._get_tick_size(signal['symbol'])
            return best_ask - (tick_size * 2)

    def _determine_order_type(self, signal: Dict) -> str:
        """주문 타입 결정"""
        if signal['urgency'] == 'HIGH':
            return 'MARKET'  # 긴급한 경우 시장가
        else:
            return 'LIMIT'   # 일반적으로 지정가

    async def _execute_limit_order(self, signal: Dict, price: float, quantity:
float) -> Dict:
        """지정가 주문 실행"""
        order_params = {
            'symbol': signal['symbol'],
```

```
            'side': signal['direction'],
            'type': 'LIMIT',
            'quantity': quantity,
            'price': price,
            'timeInForce': 'GTC'
        }

        return await self.exchange_connector.place_order(order_params)

    def _get_tick_size(self, symbol: str) -> float:
        """심볼별 틱 사이즈 조회"""
        # 거래소별 심볼 정보에서 틱 사이즈 조회
        # 구현 세부사항...
        return 0.01  # 기본값
```

이러한 호가 감지 진입 로직의 구현을 통해 시장의 미세한 변화를 포착하고 유리한 진입 기회를 활용할 수 있습니다. 실시간 오더북 분석과 정교한 신호 생성 알고리즘을 통해 PRD에서 요구하는 호가 기반 진입 전략을 효과적으로 구현할 수 있습니다.

# PCS 청산 시스템 구현

## PCS(Price Channel System) 청산 개념

PCS 청산 시스템은 PRD에서 정의된 핵심 청산 메커니즘으로, Price Channel의 상태 변화를 기반으로 단계적 청산을 수행합니다. 이 시스템은 1단, 2단, 3단의 단계별 청산 구조를 통해 리스크를 점진적으로 줄이면서 수익을 확보하는 것을 목표로 합니다.

PCS 청산의 핵심 아이디어는 Price Channel의 상단선이나 하단선의 움직임과 캔들 패턴을 종합적으로 분석하여 추세의 약화나 반전 신호를 조기에 감지하는 것입니다. 각 단계별로 포지션의 일정 비율을 청산함으로써 전체 포지션의 리스크를 관리하면서도 추가적인 수익 기회를 놓치지 않도록 설계됩니다.

## PCS 단계별 청산 로직

PCS 청산 시스템은 세 단계로 구성되며, 각 단계는 서로 다른 조건과 청산 비율을 가집니다. 1단 청산은 초기 수익 확보를 목적으로 하며, 2단과 3단 청산은 추세 약화 신호에 따라 점진적으로 포지션을 줄여나갑니다.

```python
 from enum import Enum
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple
import pandas as pd

class PCSStage(Enum):
    STAGE_1 = "1단"
    STAGE_2 = "2단"
    STAGE_3 = "3단"
    COMPLETED = "완료"

@dataclass
class PCSConfig:
    # 1단 청산 설정
    stage1_profit_threshold: float = 0.02  # 2% 수익 시 1단 청산
    stage1_exit_ratio: float = 0.3  # 30% 청산

    # 2단 청산 설정
    stage2_channel_break_threshold: float = 0.005  # 채널 이탈 임계값
    stage2_exit_ratio: float = 0.5  # 50% 청산 (잔여 포지션 기준)

    # 3단 청산 설정
    stage3_reversal_candles: int = 2  # 반전 캔들 개수
    stage3_exit_ratio: float = 1.0  # 100% 청산

    # 공통 설정
    channel_period: int = 20  # 채널 계산 기간
    min_holding_time_minutes: int = 5  # 최소 보유 시간

@dataclass
class PCSPosition:
    symbol: str
    entry_price: float
    current_quantity: float
    original_quantity: float
    entry_time: datetime
    direction: str  # 'BUY' or 'SELL'
    current_stage: PCSStage = PCSStage.STAGE_1
    stage1_executed: bool = False
    stage2_executed: bool = False
    stage3_executed: bool = False

    @property
    def remaining_ratio(self) -> float:
        return self.current_quantity / self.original_quantity

    @property
    def holding_time_minutes(self) -> float:
        return (datetime.now() - self.entry_time).total_seconds() / 60

class PCSExitSystem:
    def __init__(self, config: PCSConfig):
        self.config = config
        self.positions = {}  # symbol -> PCSPosition
        self.price_channel_calculator = PriceChannelCalculator(
            PriceChannelConfig(period=config.channel_period)
        )

    def add_position(self, symbol: str, entry_price: float, quantity: float,
direction: str):
        """새 포지션 추가"""
```

```
        position = PCSPosition(
            symbol=symbol,
            entry_price=entry_price,
            current_quantity=quantity,
            original_quantity=quantity,
            entry_time=datetime.now(),
            direction=direction
        )
        self.positions[symbol] = position
        logger.info(f"PCS 포지션 추가: {symbol} {direction} {quantity} @
{entry_price}")

    async def evaluate_exit_conditions(self, symbol: str, market_data:
pd.DataFrame) -> List[Dict]:
        """청산 조건 평가"""
        if symbol not in self.positions:
            return []

        position = self.positions[symbol]
        exit_signals = []

        # 최소 보유 시간 확인
        if position.holding_time_minutes <
self.config.min_holding_time_minutes:
            return []

        # 현재 가격 및 수익률 계산
        current_price = market_data.iloc[-1]['close']
        profit_ratio = self._calculate_profit_ratio(position, current_price)

        # 1단 청산 조건 확인
        if not position.stage1_executed:
            stage1_signal = self._evaluate_stage1_exit(position, profit_ratio)
            if stage1_signal:
                exit_signals.append(stage1_signal)

        # 2단 청산 조건 확인
        if position.stage1_executed and not position.stage2_executed:
            stage2_signal = await self._evaluate_stage2_exit(position,
market_data)
            if stage2_signal:
                exit_signals.append(stage2_signal)

        # 3단 청산 조건 확인
        if position.stage2_executed and not position.stage3_executed:
            stage3_signal = self._evaluate_stage3_exit(position, market_data)
            if stage3_signal:
                exit_signals.append(stage3_signal)

        return exit_signals

    def _calculate_profit_ratio(self, position: PCSPosition, current_price:
float) -> float:
        """수익률 계산"""
        if position.direction == 'BUY':
            return (current_price - position.entry_price) /
position.entry_price
        else:
            return (position.entry_price - current_price) /
position.entry_price

    def _evaluate_stage1_exit(self, position: PCSPosition, profit_ratio: float)
```

```
-> Optional[Dict]:
        """1단 청산 조건 평가"""
        if profit_ratio >= self.config.stage1_profit_threshold:
            exit_quantity = position.original_quantity *
self.config.stage1_exit_ratio

            return {
                'stage': PCSStage.STAGE_1,
                'symbol': position.symbol,
                'exit_quantity': exit_quantity,
                'exit_ratio': self.config.stage1_exit_ratio,
                'reason': f'{self.config.stage1_profit_threshold*100}% 수익 달
성',
                'profit_ratio': profit_ratio,
                'urgency': 'MEDIUM'
            }
        return None

    async def _evaluate_stage2_exit(self, position: PCSPosition, market_data:
pd.DataFrame) -> Optional[Dict]:
        """2단 청산 조건 평가"""
        try:
            # Price Channel 계산
            channel_info =
self.price_channel_calculator.calculate_channel(market_data)
            current_price = market_data.iloc[-1]['close']

            # 채널 이탈 확인
            if position.direction == 'BUY':
                # 매수 포지션: 상단 채널 아래로 이탈
                channel_break = current_price < (channel_info['upper_channel']
*
                                                 (1 -
self.config.stage2_channel_break_threshold))
            else:
                # 매도 포지션: 하단 채널 위로 이탈
                channel_break = current_price > (channel_info['lower_channel']
*
                                                 (1 +
self.config.stage2_channel_break_threshold))

            if channel_break:
                exit_quantity = position.current_quantity *
self.config.stage2_exit_ratio

                return {
                    'stage': PCSStage.STAGE_2,
                    'symbol': position.symbol,
                    'exit_quantity': exit_quantity,
                    'exit_ratio': self.config.stage2_exit_ratio,
                    'reason': 'Price Channel 이탈 감지',
                    'channel_info': channel_info,
                    'current_price': current_price,
                    'urgency': 'HIGH'
                }

        except Exception as e:
            logger.error(f"2단 청산 조건 평가 오류: {e}")

        return None

    def _evaluate_stage3_exit(self, position: PCSPosition, market_data:
```

```
pd.DataFrame) -> Optional[Dict]:
        """3단 청산 조건 평가"""
        if len(market_data) < self.config.stage3_reversal_candles:
            return None

        # 최근 캔들들 분석
        recent_candles = market_data.tail(self.config.stage3_reversal_candles)

        # 반전 패턴 확인
        reversal_detected = self._detect_reversal_pattern(position,
recent_candles)

        if reversal_detected:
            return {
                'stage': PCSStage.STAGE_3,
                'symbol': position.symbol,
                'exit_quantity': position.current_quantity,
                'exit_ratio': self.config.stage3_exit_ratio,
                'reason': '추세 반전 패턴 감지',
                'reversal_pattern': reversal_detected,
                'urgency': 'CRITICAL'
            }

        return None

    def _detect_reversal_pattern(self, position: PCSPosition, candles:
pd.DataFrame) -> Optional[str]:
        """추세 반전 패턴 감지"""
        if position.direction == 'BUY':
            # 매수 포지션: 연속 음봉 확인
            bearish_candles = all(
                candle['close'] < candle['open']
                for _, candle in candles.iterrows()
            )
            if bearish_candles:
                return "연속_음봉"

            # 상승 추세에서 긴 위꼬리 출현
            last_candle = candles.iloc[-1]
            upper_shadow = last_candle['high'] - max(last_candle['open'],
last_candle['close'])
            body_size = abs(last_candle['close'] - last_candle['open'])

            if upper_shadow > body_size * 2:
                return "긴_위꼬리"

        else:
            # 매도 포지션: 연속 양봉 확인
            bullish_candles = all(
                candle['close'] > candle['open']
                for _, candle in candles.iterrows()
            )
            if bullish_candles:
                return "연속_양봉"

            # 하락 추세에서 긴 아래꼬리 출현
            last_candle = candles.iloc[-1]
            lower_shadow = min(last_candle['open'], last_candle['close']) -
last_candle['low']
            body_size = abs(last_candle['close'] - last_candle['open'])

            if lower_shadow > body_size * 2:
```

```
                return "긴_아래꼬리"

        return None
```

## PCS 청산 실행 시스템

PCS 청산 신호가 생성되면 실제 청산 주문을 실행하고 포지션 상태를 업데이트합니다. 각 단계별 청산 후에는 남은 포지션에 대한 새로운 리스크 관리 설정을 적용합니다.

```python
class PCSExitExecutor:
    def __init__(self, exchange_connector, config):
        self.exchange_connector = exchange_connector
        self.config = config
        self.execution_history = []

    async def execute_pcs_exit(self, exit_signal: Dict, position: PCSPosition)
-> Dict:
        """PCS 청산 실행"""
        try:
            # 청산 주문 실행
            exit_order = await self._place_exit_order(exit_signal, position)

            # 포지션 상태 업데이트
            self._update_position_after_exit(position, exit_signal, exit_order)

            # 남은 포지션에 대한 새로운 설정 적용
            if position.current_quantity > 0:
                await self._apply_post_exit_settings(position, exit_signal)

            # 실행 기록 저장
            execution_record = {
                'timestamp': datetime.now(),
                'symbol': position.symbol,
                'stage': exit_signal['stage'],
                'exit_quantity': exit_signal['exit_quantity'],
                'exit_price': exit_order.get('price'),
                'reason': exit_signal['reason'],
                'remaining_quantity': position.current_quantity
            }
            self.execution_history.append(execution_record)

            logger.info(f"PCS {exit_signal['stage'].value} 청산 완료: "
                        f"{position.symbol} {exit_signal['exit_quantity']}")

            return {
                'success': True,
                'exit_order': exit_order,
                'execution_record': execution_record,
                'updated_position': position
            }

        except Exception as e:
            logger.error(f"PCS 청산 실행 실패: {e}")
            return {'success': False, 'error': str(e)}

    async def _place_exit_order(self, exit_signal: Dict, position: PCSPosition)
-> Dict:
        """청산 주문 실행"""
        # 주문 타입 결정 (긴급도에 따라)
        order_type = 'MARKET' if exit_signal['urgency'] == 'CRITICAL' else
'LIMIT'

        # 청산 방향 결정
        exit_side = 'SELL' if position.direction == 'BUY' else 'BUY'

        order_params = {
            'symbol': position.symbol,
            'side': exit_side,
            'type': order_type,
            'quantity': exit_signal['exit_quantity']
```

```
        }

        # 지정가 주문인 경우 가격 설정
        if order_type == 'LIMIT':
            order_params['price'] = self._calculate_exit_price(position,
exit_signal)
            order_params['timeInForce'] = 'IOC'  # 즉시 체결 또는 취소

        return await self.exchange_connector.place_order(order_params)

    def _calculate_exit_price(self, position: PCSPosition, exit_signal: Dict) -
> float:
        """청산 가격 계산"""
        # 현재 시장가 기준으로 약간 불리한 가격 설정 (빠른 체결을 위해)
        current_price = exit_signal.get('current_price', position.entry_price)

        if position.direction == 'BUY':
            # 매수 포지션 청산 시 현재가보다 약간 낮게
            return current_price * 0.999
        else:
            # 매도 포지션 청산 시 현재가보다 약간 높게
            return current_price * 1.001

    def _update_position_after_exit(self, position: PCSPosition, exit_signal:
Dict, exit_order: Dict):
        """청산 후 포지션 상태 업데이트"""
        # 포지션 수량 업데이트
        position.current_quantity -= exit_signal['exit_quantity']

        # 단계 상태 업데이트
        if exit_signal['stage'] == PCSStage.STAGE_1:
            position.stage1_executed = True
            position.current_stage = PCSStage.STAGE_2
        elif exit_signal['stage'] == PCSStage.STAGE_2:
            position.stage2_executed = True
            position.current_stage = PCSStage.STAGE_3
        elif exit_signal['stage'] == PCSStage.STAGE_3:
            position.stage3_executed = True
            position.current_stage = PCSStage.COMPLETED

    async def _apply_post_exit_settings(self, position: PCSPosition,
exit_signal: Dict):
        """청산 후 남은 포지션에 대한 새로운 설정 적용"""
        if position.current_stage == PCSStage.STAGE_2:
            # 1단 청산 후: 손절가를 진입가로 이동 (무손실 구간)
            await self._set_breakeven_stop_loss(position)

        elif position.current_stage == PCSStage.STAGE_3:
            # 2단 청산 후: 트레일링 스톱 적용
            await self._set_trailing_stop(position)

    async def _set_breakeven_stop_loss(self, position: PCSPosition):
        """무손실 손절가 설정"""
        stop_price = position.entry_price

        # 기존 손절 주문 취소 후 새로운 손절 주문 설정
        await self._update_stop_loss_order(position, stop_price)

        logger.info(f"무손실 손절가 설정: {position.symbol} @ {stop_price}")

    async def _set_trailing_stop(self, position: PCSPosition):
        """트레일링 스톱 설정"""
```

```
        # 트레일링 스톱 로직 구현
        # 현재가에서 일정 비율 아래로 손절가 설정
        trailing_percent = 0.02  # 2% 트레일링

        # 구현 세부사항...
        logger.info(f"트레일링 스톱 설정: {position.symbol}
{trailing_percent*100}%")

    async def _update_stop_loss_order(self, position: PCSPosition,
new_stop_price: float):
        """손절 주문 업데이트"""
        # 기존 손절 주문 조회 및 취소
        # 새로운 손절 주문 생성
        # 구현 세부사항...
        pass
```

## PCS 성과 분석 및 최적화

PCS 청산 시스템의 성과를 지속적으로 모니터링하고 최적화하기 위한 분석 도구를 구현합니다.

```python
class PCSPerformanceAnalyzer:
    def __init__(self):
        self.performance_data = []

    def analyze_pcs_performance(self, execution_history: List[Dict]) -> Dict:
        """PCS 성과 분석"""
        if not execution_history:
            return {}

        # 단계별 성과 분석
        stage_performance = self._analyze_stage_performance(execution_history)

        # 전체 수익률 분석
        total_performance = self._analyze_total_performance(execution_history)

        # 최적화 제안
        optimization_suggestions = self._generate_optimization_suggestions(
            stage_performance, total_performance
        )

        return {
            'stage_performance': stage_performance,
            'total_performance': total_performance,
            'optimization_suggestions': optimization_suggestions,
            'analysis_timestamp': datetime.now()
        }

    def _analyze_stage_performance(self, history: List[Dict]) -> Dict:
        """단계별 성과 분석"""
        stage_stats = {
            PCSStage.STAGE_1: {'count': 0, 'avg_profit': 0, 'success_rate': 0},
            PCSStage.STAGE_2: {'count': 0, 'avg_profit': 0, 'success_rate': 0},
            PCSStage.STAGE_3: {'count': 0, 'avg_profit': 0, 'success_rate': 0}
        }

        for record in history:
            stage = record['stage']
            if stage in stage_stats:
                stage_stats[stage]['count'] += 1
                # 추가 분석 로직...

        return stage_stats

    def _generate_optimization_suggestions(self, stage_perf: Dict, total_perf:
Dict) -> List[str]:
        """최적화 제안 생성"""
        suggestions = []

        # 1단 청산 최적화
        if stage_perf[PCSStage.STAGE_1]['success_rate'] < 0.7:
            suggestions.append("1단 청산 임계값 조정 검토 필요")

        # 2단 청산 최적화
        if stage_perf[PCSStage.STAGE_2]['count'] < stage_perf[PCSStage.STAGE_1]
['count'] * 0.5:
            suggestions.append("2단 청산 조건 완화 검토 필요")

        return suggestions
```

이러한 PCS 청산 시스템의 구현을 통해 PRD에서 요구하는 체계적이고 단계적인 청산 전략을 효과적으로 실현할 수 있습니다. 각 단계별 청산을 통해 리스크를 점진적으로 줄이면서도 추가 수익 기회를 놓치지 않는 균형잡힌 청산 전략을 제공합니다.

# 리스크 관리 시스템 구현

## 종합적 리스크 관리 프레임워크

리스크 관리는 자동거래 시스템의 생존과 직결되는 핵심 요소입니다. PRD에서 정의된 리스크 관리 요구사항을 바탕으로 다층적이고 포괄적인 리스크 관리 시스템을 구현합니다. 이 시스템은 포지션 레벨, 심볼 레벨, 포트폴리오 레벨에서 각각 다른 리스크 통제 메커니즘을 적용합니다.

리스크 관리 시스템의 핵심 원칙은 예방적 접근과 반응적 대응의 조합입니다. 예방적 접근은 거래 전 리스크 평가와 포지션 크기 제한을 통해 과도한 리스크 노출을 방지합니다. 반응적 대응은 실시간 모니터링을 통해 리스크 한계 초과 시 즉시 보호 조치를 취합니다.

```python
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Tuple
from enum import Enum
import asyncio
from datetime import datetime, timedelta

class RiskLevel(Enum):
    LOW = "낮음"
    MEDIUM = "보통"
    HIGH = "높음"
    CRITICAL = "위험"

@dataclass
class RiskLimits:
    # 포지션별 리스크 한계
    max_position_size_percent: float = 5.0  # 계좌 대비 최대 포지션 크기 (%)
    max_loss_per_trade_percent: float = 2.0  # 거래당 최대 손실 (%)

    # 심볼별 리스크 한계
    max_positions_per_symbol: int = 3  # 심볼당 최대 포지션 수
    max_exposure_per_symbol_percent: float = 10.0  # 심볼당 최대 노출 (%)

    # 포트폴리오 리스크 한계
    max_total_exposure_percent: float = 30.0  # 총 노출 한계 (%)
    max_daily_loss_percent: float = 5.0  # 일일 최대 손실 (%)
    max_drawdown_percent: float = 15.0  # 최대 드로우다운 (%)

    # 시간 기반 리스크 한계
    max_trades_per_hour: int = 10  # 시간당 최대 거래 수
    max_trades_per_day: int = 50  # 일일 최대 거래 수

    # 연속 손실 제한
    max_consecutive_losses: int = 5  # 최대 연속 손실 횟수
    consecutive_loss_cooldown_hours: int = 2  # 연속 손실 후 대기 시간

@dataclass
class RiskMetrics:
    current_exposure_percent: float = 0.0
    daily_pnl_percent: float = 0.0
    max_drawdown_percent: float = 0.0
    consecutive_losses: int = 0
    trades_today: int = 0
    trades_this_hour: int = 0
    last_trade_time: Optional[datetime] = None
    risk_level: RiskLevel = RiskLevel.LOW

class RiskManager:
    def __init__(self, limits: RiskLimits, initial_balance: float):
        self.limits = limits
        self.initial_balance = initial_balance
        self.current_balance = initial_balance
        self.positions = {}  # symbol -> position info
        self.trade_history = []
        self.risk_metrics = RiskMetrics()
        self.emergency_stop = False

    async def validate_new_trade(self, trade_request: Dict) -> Tuple[bool,
str]:
        """새로운 거래 요청 검증"""

        # 긴급 정지 상태 확인
```

```
        if self.emergency_stop:
            return False, "긴급 정지 상태"

        # 거래 빈도 제한 확인
        if not self._check_trade_frequency():
            return False, "거래 빈도 한계 초과"

        # 연속 손실 제한 확인
        if not self._check_consecutive_losses():
            return False, "연속 손실 한계 초과"

        # 포지션 크기 검증
        if not self._validate_position_size(trade_request):
            return False, "포지션 크기 한계 초과"

        # 심볼별 노출 한계 확인
        if not self._check_symbol_exposure(trade_request):
            return False, "심볼별 노출 한계 초과"

        # 포트폴리오 노출 한계 확인
        if not self._check_portfolio_exposure(trade_request):
            return False, "포트폴리오 노출 한계 초과"

        # 일일 손실 한계 확인
        if not self._check_daily_loss_limit():
            return False, "일일 손실 한계 초과"

        return True, "거래 승인"

    def _check_trade_frequency(self) -> bool:
        """거래 빈도 제한 확인"""
        now = datetime.now()

        # 시간당 거래 수 확인
        hour_ago = now - timedelta(hours=1)
        trades_this_hour = sum(
            1 for trade in self.trade_history
            if trade['timestamp'] > hour_ago
        )

        if trades_this_hour >= self.limits.max_trades_per_hour:
            return False

        # 일일 거래 수 확인
        today_start = now.replace(hour=0, minute=0, second=0, microsecond=0)
        trades_today = sum(
            1 for trade in self.trade_history
            if trade['timestamp'] > today_start
        )

        return trades_today < self.limits.max_trades_per_day

    def _check_consecutive_losses(self) -> bool:
        """연속 손실 제한 확인"""
        if self.risk_metrics.consecutive_losses >=
self.limits.max_consecutive_losses:
            # 마지막 손실 거래 시간 확인
            last_loss_time = self._get_last_loss_time()
            if last_loss_time:
                cooldown_end = last_loss_time + timedelta(
                    hours=self.limits.consecutive_loss_cooldown_hours
                )
```

```
                if datetime.now() < cooldown_end:
                    return False
                else:
                    # 쿨다운 기간 종료, 연속 손실 카운터 리셋
                    self.risk_metrics.consecutive_losses = 0

        return True

    def _validate_position_size(self, trade_request: Dict) -> bool:
        """포지션 크기 검증"""
        position_value = trade_request['quantity'] * trade_request['price']
        position_percent = (position_value / self.current_balance) * 100

        return position_percent <= self.limits.max_position_size_percent

    def _check_symbol_exposure(self, trade_request: Dict) -> bool:
        """심볼별 노출 한계 확인"""
        symbol = trade_request['symbol']

        # 현재 심볼의 포지션 수 확인
        current_positions = len([
            pos for pos in self.positions.values()
            if pos['symbol'] == symbol and pos['quantity'] > 0
        ])

        if current_positions >= self.limits.max_positions_per_symbol:
            return False

        # 심볼별 총 노출 확인
        current_exposure = self._calculate_symbol_exposure(symbol)
        new_position_value = trade_request['quantity'] * trade_request['price']
        total_exposure = current_exposure + new_position_value
        exposure_percent = (total_exposure / self.current_balance) * 100

        return exposure_percent <= self.limits.max_exposure_per_symbol_percent

    def _check_portfolio_exposure(self, trade_request: Dict) -> bool:
        """포트폴리오 노출 한계 확인"""
        current_total_exposure = self._calculate_total_exposure()
        new_position_value = trade_request['quantity'] * trade_request['price']
        total_exposure = current_total_exposure + new_position_value
        exposure_percent = (total_exposure / self.current_balance) * 100

        return exposure_percent <= self.limits.max_total_exposure_percent

    def _check_daily_loss_limit(self) -> bool:
        """일일 손실 한계 확인"""
        return abs(self.risk_metrics.daily_pnl_percent) <=
self.limits.max_daily_loss_percent
```

## 동적 포지션 크기 계산

리스크 기반 포지션 크기 계산은 Kelly Criterion과 Fixed Fractional 방법을 조합하여 구현합니다. 이 시스템은 현재 계좌 상태, 거래 성과, 시장 변동성을 종합적으로 고려하여 최적의 포지션 크기를 결정합니다.

```python
import numpy as np
from scipy import stats

class DynamicPositionSizer:
    def __init__(self, risk_manager: RiskManager):
        self.risk_manager = risk_manager
        self.performance_history = []
        self.volatility_cache = {}

    def calculate_position_size(self, trade_signal: Dict, market_data:
pd.DataFrame) -> float:
        """동적 포지션 크기 계산"""

        # 기본 리스크 기반 크기 계산
        base_size = self._calculate_base_position_size(trade_signal)

        # Kelly Criterion 적용
        kelly_multiplier =
self._calculate_kelly_multiplier(trade_signal['symbol'])

        # 변동성 조정
        volatility_multiplier = self._calculate_volatility_multiplier(
            trade_signal['symbol'], market_data
        )

        # 성과 기반 조정
        performance_multiplier = self._calculate_performance_multiplier()

        # 최종 포지션 크기 계산
        final_size = base_size * kelly_multiplier * volatility_multiplier *
performance_multiplier

        # 리스크 한계 적용
        final_size = self._apply_risk_limits(final_size, trade_signal)

        return final_size

    def _calculate_base_position_size(self, trade_signal: Dict) -> float:
        """기본 포지션 크기 계산 (Fixed Fractional)"""
        account_balance = self.risk_manager.current_balance
        risk_per_trade = account_balance *
(self.risk_manager.limits.max_loss_per_trade_percent / 100)

        entry_price = trade_signal['entry_price']
        stop_loss_price = trade_signal.get('stop_loss_price', entry_price *
0.98)

        price_risk = abs(entry_price - stop_loss_price)

        if price_risk == 0:
            return 0

        position_size = risk_per_trade / price_risk
        return position_size

    def _calculate_kelly_multiplier(self, symbol: str) -> float:
        """Kelly Criterion 기반 승수 계산"""
        # 최근 거래 성과 분석
        recent_trades = [
            trade for trade in self.performance_history
            if trade['symbol'] == symbol and len(self.performance_history) >=
```

```
20
        ]

        if len(recent_trades) < 10:
            return 0.5  # 충분한 데이터가 없으면 보수적 접근

        wins = [trade['pnl'] for trade in recent_trades if trade['pnl'] > 0]
        losses = [abs(trade['pnl']) for trade in recent_trades if trade['pnl']
< 0]

        if not wins or not losses:
            return 0.5

        win_rate = len(wins) / len(recent_trades)
        avg_win = np.mean(wins)
        avg_loss = np.mean(losses)

        if avg_loss == 0:
            return 0.5

        # Kelly Criterion: f = (bp - q) / b
        # b = avg_win / avg_loss, p = win_rate, q = 1 - win_rate
        b = avg_win / avg_loss
        kelly_fraction = (b * win_rate - (1 - win_rate)) / b

        # Kelly 비율을 0.1 ~ 1.0 범위로 제한
        kelly_multiplier = max(0.1, min(1.0, kelly_fraction))

        return kelly_multiplier

    def _calculate_volatility_multiplier(self, symbol: str, market_data:
pd.DataFrame) -> float:
        """변동성 기반 승수 계산"""
        if len(market_data) < 20:
            return 0.8  # 데이터 부족 시 보수적 접근

        # ATR (Average True Range) 계산
        atr = self._calculate_atr(market_data, period=14)
        current_price = market_data.iloc[-1]['close']
        volatility_percent = (atr / current_price) * 100

        # 변동성이 높을수록 포지션 크기 감소
        if volatility_percent > 5.0:
            return 0.5
        elif volatility_percent > 3.0:
            return 0.7
        elif volatility_percent > 2.0:
            return 0.9
        else:
            return 1.0

    def _calculate_atr(self, data: pd.DataFrame, period: int = 14) -> float:
        """Average True Range 계산"""
        high = data['high']
        low = data['low']
        close = data['close'].shift(1)

        tr1 = high - low
        tr2 = abs(high - close)
        tr3 = abs(low - close)

        true_range = pd.concat([tr1, tr2, tr3], axis=1).max(axis=1)
```

```
        atr = true_range.rolling(window=period).mean().iloc[-1]

        return atr

    def _calculate_performance_multiplier(self) -> float:
        """최근 성과 기반 승수 계산"""
        if len(self.performance_history) < 10:
            return 1.0

        recent_trades = self.performance_history[-20:]  # 최근 20거래
        total_pnl = sum(trade['pnl'] for trade in recent_trades)

        # 최근 성과가 좋으면 포지션 크기 증가, 나쁘면 감소
        if total_pnl > 0:
            return min(1.2, 1.0 + (total_pnl /
self.risk_manager.current_balance))
        else:
            return max(0.5, 1.0 + (total_pnl /
self.risk_manager.current_balance))
```

## 실시간 리스크 모니터링

실시간 리스크 모니터링 시스템은 포지션과 시장 상황을 지속적으로 감시하여 리스크 한계 초과 시 즉시 대응합니다.

```python
class RealTimeRiskMonitor:
    def __init__(self, risk_manager: RiskManager):
        self.risk_manager = risk_manager
        self.monitoring_active = False
        self.alert_callbacks = []
        self.emergency_callbacks = []

    async def start_monitoring(self):
        """실시간 모니터링 시작"""
        self.monitoring_active = True

        # 모니터링 태스크들 시작
        tasks = [
            asyncio.create_task(self._monitor_portfolio_risk()),
            asyncio.create_task(self._monitor_position_risk()),
            asyncio.create_task(self._monitor_market_conditions()),
            asyncio.create_task(self._monitor_system_health())
        ]

        try:
            await asyncio.gather(*tasks)
        except Exception as e:
            logger.error(f"리스크 모니터링 오류: {e}")
            await self._trigger_emergency_stop()

    async def _monitor_portfolio_risk(self):
        """포트폴리오 리스크 모니터링"""
        while self.monitoring_active:
            try:
                # 현재 포트폴리오 상태 계산
                portfolio_metrics = self._calculate_portfolio_metrics()

                # 드로우다운 확인
                if portfolio_metrics['drawdown_percent'] >
self.risk_manager.limits.max_drawdown_percent:
                    await self._handle_drawdown_breach(portfolio_metrics)

                # 일일 손실 한계 확인
                if abs(portfolio_metrics['daily_pnl_percent']) >
self.risk_manager.limits.max_daily_loss_percent:
                    await self._handle_daily_loss_breach(portfolio_metrics)

                # 총 노출 한계 확인
                if portfolio_metrics['total_exposure_percent'] >
self.risk_manager.limits.max_total_exposure_percent:
                    await self._handle_exposure_breach(portfolio_metrics)

                await asyncio.sleep(10)  # 10초마다 확인

            except Exception as e:
                logger.error(f"포트폴리오 리스크 모니터링 오류: {e}")
                await asyncio.sleep(30)

    async def _monitor_position_risk(self):
        """개별 포지션 리스크 모니터링"""
        while self.monitoring_active:
            try:
                for symbol, position in self.risk_manager.positions.items():
                    if position['quantity'] == 0:
                        continue
```

```
                    # 포지션별 손실 확인
                    current_pnl_percent =
self._calculate_position_pnl_percent(position)

                    if current_pnl_percent < -
self.risk_manager.limits.max_loss_per_trade_percent:
                        await self._handle_position_loss_breach(symbol,
position, current_pnl_percent)

                    # 포지션 보유 시간 확인
                    holding_time = datetime.now() - position['entry_time']
                    if holding_time > timedelta(hours=24):  # 24시간 초과 보유
                        await self._handle_long_holding_position(symbol,
position)

                await asyncio.sleep(5)  # 5초마다 확인

            except Exception as e:
                logger.error(f"포지션 리스크 모니터링 오류: {e}")
                await asyncio.sleep(15)

    async def _handle_drawdown_breach(self, metrics: Dict):
        """드로우다운 한계 초과 처리"""
        alert_message = f"드로우다운 한계 초과:
{metrics['drawdown_percent']:.2f}%"

        # 모든 포지션 긴급 청산
        await self._emergency_close_all_positions()

        # 거래 일시 정지
        self.risk_manager.emergency_stop = True

        # 알림 전송
        await self._send_critical_alert(alert_message, metrics)

    async def _handle_daily_loss_breach(self, metrics: Dict):
        """일일 손실 한계 초과 처리"""
        alert_message = f"일일 손실 한계 초과:
{metrics['daily_pnl_percent']:.2f}%"

        # 신규 거래 중단
        self.risk_manager.emergency_stop = True

        # 기존 포지션 점진적 청산
        await self._gradual_position_reduction()

        # 알림 전송
        await self._send_critical_alert(alert_message, metrics)

    async def _emergency_close_all_positions(self):
        """모든 포지션 긴급 청산"""
        for symbol, position in self.risk_manager.positions.items():
            if position['quantity'] > 0:
                try:
                    # 시장가 청산 주문
                    await self._place_emergency_exit_order(symbol, position)
                except Exception as e:
                    logger.error(f"긴급 청산 실패 {symbol}: {e}")

    async def _send_critical_alert(self, message: str, data: Dict):
        """중요 알림 전송"""
        alert = {
```

```
            'level': 'CRITICAL',
            'message': message,
            'timestamp': datetime.now(),
            'data': data
        }

        # 등록된 알림 콜백 실행
        for callback in self.emergency_callbacks:
            try:
                await callback(alert)
            except Exception as e:
                logger.error(f"알림 콜백 실행 오류: {e}")
```

### 적응형 리스크 조정

시장 상황과 시스템 성과에 따라 리스크 파라미터를 동적으로 조정하는 적응형 시스템을 구현합니다.

```python
class AdaptiveRiskAdjuster:
    def __init__(self, risk_manager: RiskManager):
        self.risk_manager = risk_manager
        self.adjustment_history = []
        self.market_regime_detector = MarketRegimeDetector()

    async def adjust_risk_parameters(self, market_data: Dict, performance_data:
Dict):
        """리스크 파라미터 적응형 조정"""

        # 시장 상황 분석
        market_regime = self.market_regime_detector.detect_regime(market_data)

        # 성과 분석
        performance_score = self._calculate_performance_score(performance_data)

        # 조정 필요성 판단
        adjustment_needed = self._should_adjust_parameters(market_regime,
performance_score)

        if adjustment_needed:
            # 새로운 리스크 파라미터 계산
            new_limits = self._calculate_adjusted_limits(market_regime,
performance_score)

            # 파라미터 업데이트
            await self._update_risk_limits(new_limits)

            # 조정 기록 저장
            self._record_adjustment(market_regime, performance_score,
new_limits)

    def _calculate_performance_score(self, performance_data: Dict) -> float:
        """성과 점수 계산"""
        # Sharpe Ratio, Win Rate, Profit Factor 등을 종합한 점수
        sharpe_ratio = performance_data.get('sharpe_ratio', 0)
        win_rate = performance_data.get('win_rate', 0.5)
        profit_factor = performance_data.get('profit_factor', 1.0)

        # 정규화된 점수 계산 (0-1 범위)
        score = (
            min(sharpe_ratio / 2.0, 1.0) * 0.4 +
            win_rate * 0.3 +
            min(profit_factor / 2.0, 1.0) * 0.3
        )

        return score

    def _calculate_adjusted_limits(self, market_regime: str, performance_score:
float) -> RiskLimits:
        """조정된 리스크 한계 계산"""
        base_limits = self.risk_manager.limits

        # 시장 상황에 따른 조정
        market_multiplier = {
            'trending': 1.1,      # 추세 시장: 리스크 증가
            'ranging': 0.9,       # 횡보 시장: 리스크 감소
            'volatile': 0.7,      # 변동성 시장: 리스크 대폭 감소
            'crisis': 0.5         # 위기 상황: 최소 리스크
        }.get(market_regime, 1.0)
```

```
        # 성과에 따른 조정
        performance_multiplier = 0.8 + (performance_score * 0.4)  # 0.8 ~ 1.2
범위

        # 최종 승수
        final_multiplier = market_multiplier * performance_multiplier

        # 새로운 한계 계산
        adjusted_limits = RiskLimits(
            max_position_size_percent=base_limits.max_position_size_percent *
final_multiplier,
            max_loss_per_trade_percent=base_limits.max_loss_per_trade_percent *
final_multiplier,
            max_total_exposure_percent=base_limits.max_total_exposure_percent *
final_multiplier,
            max_daily_loss_percent=base_limits.max_daily_loss_percent *
final_multiplier
        )

        return adjusted_limits
```

이러한 종합적인 리스크 관리 시스템의 구현을 통해 PRD에서 요구하는 안전하고 지속 가능한 자동 거래 시스템을 구축할 수 있습니다. 다층적 리스크 통제와 적응형 조정 메커니즘을 통해 시장 변화에 유연하게 대응하면서도 자본을 보호하는 견고한 시스템을 제공합니다.

## 실시간 모니터링 및 알림

### 통합 모니터링 대시보드

실시간 모니터링 시스템은 거래 시스템의 모든 측면을 실시간으로 감시하고 시각화합니다. 웹 기반 대시보드를 통해 포지션 상태, 수익률, 리스크 지표, 시스템 성능 등을 한눈에 파악할 수 있습니다.

```
 from flask import Flask, render_template, jsonify
import plotly.graph_objs as go
import plotly.utils

class MonitoringDashboard:
    def __init__(self, trading_system):
        self.app = Flask(__name__)
        self.trading_system = trading_system
        self.setup_routes()

    def setup_routes(self):
        @self.app.route('/')
        def dashboard():
            return render_template('dashboard.html')

        @self.app.route('/api/portfolio_status')
        def portfolio_status():
            return jsonify(self._get_portfolio_status())

        @self.app.route('/api/active_positions')
        def active_positions():
            return jsonify(self._get_active_positions())

        @self.app.route('/api/performance_chart')
        def performance_chart():
            return jsonify(self._generate_performance_chart())

    def _get_portfolio_status(self):
        return {
            'total_balance': self.trading_system.get_total_balance(),
            'daily_pnl': self.trading_system.get_daily_pnl(),
            'total_positions': len(self.trading_system.get_active_positions()),
            'risk_level': self.trading_system.get_current_risk_level(),
            'system_status': 'ACTIVE' if self.trading_system.is_running else
'STOPPED'
        }
```

### 다중 채널 알림 시스템

알림 시스템은 이메일, SMS, 텔레그램, 디스코드 등 다양한 채널을 통해 중요한 이벤트를 실시간으로 전달합니다.

```python
class NotificationSystem:
    def __init__(self, config):
        self.config = config
        self.channels = {
            'email': EmailNotifier(config.email),
            'telegram': TelegramNotifier(config.telegram),
            'discord': DiscordNotifier(config.discord),
            'sms': SMSNotifier(config.sms)
        }

    async def send_alert(self, level: str, message: str, data: Dict = None):
        """알림 전송"""
        alert = {
            'level': level,
            'message': message,
            'timestamp': datetime.now(),
            'data': data or {}
        }

        # 알림 레벨에 따른 채널 선택
        channels_to_use = self._get_channels_for_level(level)

        # 병렬로 알림 전송
        tasks = [
            self.channels[channel].send(alert)
            for channel in channels_to_use
        ]

        await asyncio.gather(*tasks, return_exceptions=True)
```

## 백테스팅 및 성능 최적화

### 백테스팅 엔진

백테스팅 엔진은 과거 데이터를 사용하여 거래 전략의 성능을 검증합니다. 실제 거래 환경을 최대한 정확히 시뮬레이션하여 신뢰할 수 있는 결과를 제공합니다.

```python
class BacktestEngine:
    def __init__(self, strategy_config, market_data):
        self.strategy_config = strategy_config
        self.market_data = market_data
        self.portfolio = BacktestPortfolio(initial_balance=100000)
        self.trade_executor = BacktestExecutor(self.portfolio)

    def run_backtest(self, start_date: str, end_date: str) -> Dict:
        """백테스트 실행"""
        results = {
            'trades': [],
            'portfolio_value': [],
            'metrics': {}
        }

        # 전략 초기화
        strategy = self._initialize_strategy()

        # 데이터 순회하며 백테스트 실행
        for timestamp, market_snapshot in self._iterate_market_data(start_date,
end_date):
            # 신호 생성
            signals = strategy.generate_signals(market_snapshot)

            # 거래 실행
            for signal in signals:
                trade_result = self.trade_executor.execute_signal(signal)
                if trade_result:
                    results['trades'].append(trade_result)

            # 포트폴리오 가치 기록
            portfolio_value = self.portfolio.get_total_value(market_snapshot)
            results['portfolio_value'].append({
                'timestamp': timestamp,
                'value': portfolio_value
            })

        # 성과 지표 계산
        results['metrics'] = self._calculate_performance_metrics(results)

        return results

    def _calculate_performance_metrics(self, results: Dict) -> Dict:
        """성과 지표 계산"""
        trades = results['trades']
        portfolio_values = [pv['value'] for pv in results['portfolio_value']]

        if not trades or not portfolio_values:
            return {}

        # 기본 지표
        total_return = (portfolio_values[-1] - portfolio_values[0]) /
portfolio_values[0]
        total_trades = len(trades)
        winning_trades = len([t for t in trades if t['pnl'] > 0])
        win_rate = winning_trades / total_trades if total_trades > 0 else 0

        # 샤프 비율
        returns = np.diff(portfolio_values) / portfolio_values[:-1]
        sharpe_ratio = np.mean(returns) / np.std(returns) * np.sqrt(252) if
np.std(returns) > 0 else 0
```

```
        # 최대 드로우다운
        peak = np.maximum.accumulate(portfolio_values)
        drawdown = (portfolio_values - peak) / peak
        max_drawdown = np.min(drawdown)

        return {
            'total_return': total_return,
            'total_trades': total_trades,
            'win_rate': win_rate,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': abs(max_drawdown),
            'profit_factor': self._calculate_profit_factor(trades)
        }
```

## 파라미터 최적화

유전 알고리즘과 그리드 서치를 활용하여 전략 파라미터를 최적화합니다.

```python
 from scipy.optimize import differential_evolution
import itertools

class ParameterOptimizer:
    def __init__(self, backtest_engine):
        self.backtest_engine = backtest_engine

    def optimize_parameters(self, parameter_ranges: Dict, optimization_target:
str = 'sharpe_ratio') -> Dict:
        """파라미터 최적화"""

        def objective_function(params):
            # 파라미터 설정
            param_dict = self._params_array_to_dict(params, parameter_ranges)

            # 백테스트 실행
            self.backtest_engine.update_parameters(param_dict)
            results = self.backtest_engine.run_backtest()

            # 목표 지표 반환 (최대화를 위해 음수 반환)
            target_value = results['metrics'].get(optimization_target, 0)
            return -target_value

        # 파라미터 범위 설정
        bounds = [(r['min'], r['max']) for r in parameter_ranges.values()]

        # 최적화 실행
        result = differential_evolution(
            objective_function,
            bounds,
            maxiter=100,
            popsize=15,
            seed=42
        )

        # 최적 파라미터 반환
        optimal_params = self._params_array_to_dict(result.x, parameter_ranges)

        return {
            'optimal_parameters': optimal_params,
            'optimal_value': -result.fun,
            'optimization_success': result.success
        }
```

# 배포 및 운영 가이드

## 프로덕션 배포

프로덕션 환경에서의 안전한 배포를 위한 단계별 가이드입니다.

```yaml
# docker-compose.yml
version: '3.8'
services:
  trading-system:
    build: .
    environment:
      - ENVIRONMENT=production
      - REDIS_URL=redis://redis:6379
      - DATABASE_URL=postgresql://user:pass@db:5432/trading
    depends_on:
      - redis
      - db
    volumes:
      - ./logs:/app/logs
      - ./config:/app/config
    restart: unless-stopped

  redis:
    image: redis:alpine
    restart: unless-stopped

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: trading
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped

  monitoring:
    image: grafana/grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
    restart: unless-stopped

volumes:
  postgres_data:
  grafana_data:
```

## 운영 모니터링

```python
class OperationalMonitoring:
    def __init__(self):
        self.metrics_collector = MetricsCollector()
        self.health_checker = HealthChecker()
        self.log_analyzer = LogAnalyzer()

    async def start_monitoring(self):
        """운영 모니터링 시작"""
        tasks = [
            self._monitor_system_health(),
            self._monitor_performance_metrics(),
            self._monitor_error_rates(),
            self._monitor_resource_usage()
        ]

        await asyncio.gather(*tasks)

    async def _monitor_system_health(self):
        """시스템 건강 상태 모니터링"""
        while True:
            health_status = await self.health_checker.check_all_components()

            if not health_status['overall_healthy']:
                await self._handle_health_issue(health_status)

            await asyncio.sleep(30)
```

## 보안 및 규정 준수

```python
class SecurityManager:
    def __init__(self):
        self.encryption_key = self._load_encryption_key()
        self.audit_logger = AuditLogger()

    def encrypt_sensitive_data(self, data: str) -> str:
        """민감한 데이터 암호화"""
        # 구현 세부사항...
        pass

    def log_security_event(self, event_type: str, details: Dict):
        """보안 이벤트 로깅"""
        self.audit_logger.log({
            'event_type': event_type,
            'timestamp': datetime.now(),
            'details': details,
            'user_id': self._get_current_user_id()
        })
```

# 결론

본 가이드에서 제시한 구현 방법을 통해 PRD에서 정의된 모든 요구사항을 충족하는 완전한 암호화폐 자동거래 시스템을 구축할 수 있습니다. 각 모듈은 독립적으로 개발하고 테스트할 수 있도록 설계되어 있으며, 실제 운영 환경에서의 안정성과 확장성을 고려하여 구현되었습니다.

성공적인 자동거래 시스템 구축을 위해서는 충분한 백테스팅, 점진적 배포, 지속적인 모니터링이 필수적입니다. 또한 시장 상황 변화에 따른 전략 조정과 리스크 관리 파라미터 최적화를 통해 시스템의 성능을 지속적으로 개선해야 합니다.

이 가이드가 실제 거래 시스템 개발에 도움이 되기를 바라며, 안전하고 수익성 있는 자동거래 시스템 구축에 성공하시기를 기원합니다.